

# **G R A F I K**

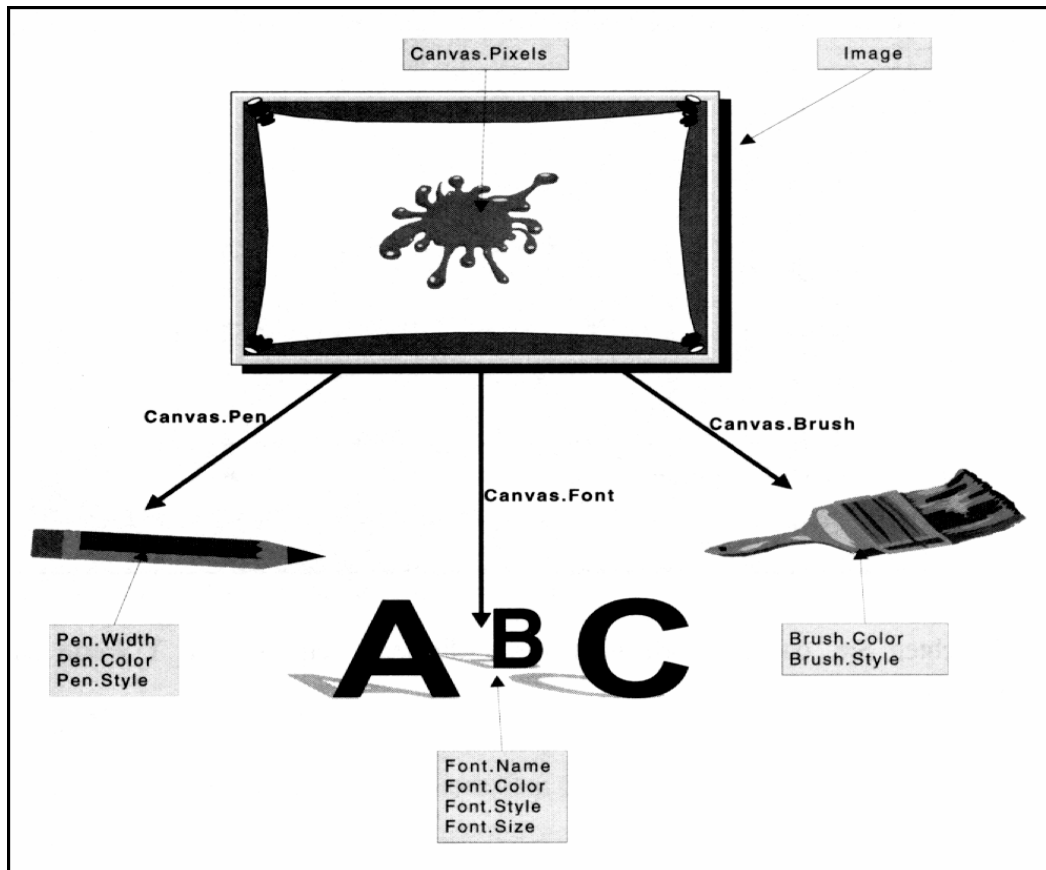
## **Grafik-Programmierung**

**© Herbert Paukert**

<b>[01] Grundlagen der Grafik-Programmierung</b>	<b>(- 02 -)</b>
<b>[02] Grafikdateien im BMP- und JPG-Format</b>	<b>(- 07 -)</b>
<b>[03] Koordinatentransformation von Welt zu Bild</b>	<b>(- 27 -)</b>
<b>[04] Einfache geometrische Abbildungen</b>	<b>(- 36 -)</b>
<b>[05] Sammlung von Grafikroutinen "<i>gtools_u</i>"</b>	<b>(- 41 -)</b>

## [01] Grundlagen der Grafik-Programmierung

Unter DELPHI haben wir es mit Grafikoperationen zu tun, die sich immer nur auf die Zeichenoberfläche (Leinwand, *Canvas*) eines bestimmten visuellen Objektes beziehen. So besitzen Formulare, Image- und Paintbox-Komponenten die Eigenschaft *TCanvas*. Die wichtigste Komponente zur grafischen Bilddarstellung ist sicherlich *TImage*.



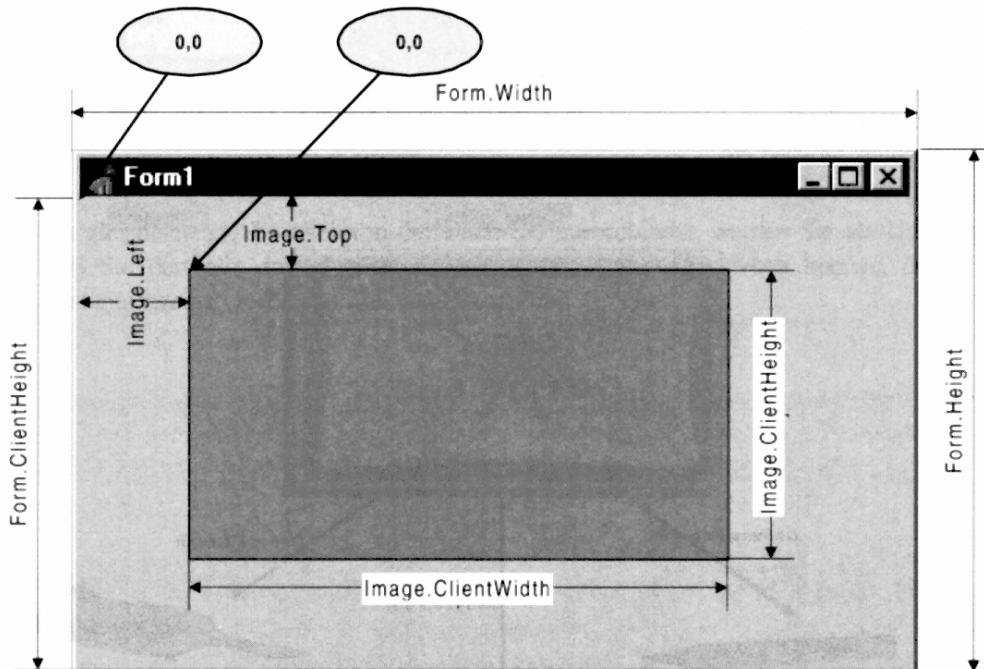
Ein *Canvas*-Objekt kann seinerseits weitere Objekte enthalten. Dazu zählen vor allem der Zeichenstift (**Pen**), die Farbbürste (**Brush**) und der Schriftsatz (**Font**). Während der Pen ausschließlich dem Zeichnen von Linien dient, wird die Brush zum Ausfüllen geschlossener Figuren (Rechteck, Ellipse usw.) verwendet.

Für den **Pen** gibt es nur vier Eigenschaften: *Pen.Width* (Linienbreite in Pixeln), *Pen.Style* (Art der Linie: *psSolid*, *psDash*, *psClear* usw.), *Pen.Color* (Zeichenfarbe: diese ist vom Objekttyp *TColor* abgeleitet und kann mit Konstanten wie *clBlack*, *clWhite*, *clRed*, *clGreen*, *clBlue* usw. bezeichnet werden) und *Pen.Mode* (Zeichenmodus im Verhältnis zum Hintergrund).

Für das Werkzeug **Brush** sind folgende Eigenschaften vorgesehen: *Brush.Style* (Art der Farbmusterfüllung einer geschlossenen Figur: *bsSolid*, *bsHorizontal*, *bsVertical*, *bsFDiagonal*, *bsCross* und *bsClear*) und *Brush.Color* (Füllfarbe: sie ist genauso definiert wie *Pen.Color*).

Die horizontale und vertikale Erstreckung eines Bildes (*Image*) innerhalb der grafischen Auflösung des physischen Bildschirms (*Screen*) kann in einem Formular (*Form*) individuell festgelegt werden. Zu beachten ist, dass sich der Koordinatenursprung immer in der linken, oberen Ecke des entsprechenden *Canvas* befindet, welche auch *Homeposition* genannt wird. Maßeinheit ist dabei immer ein Bildpunkt (*Pixel*).

Die Eigenschaften *Left* und *Top* bezeichnen den Abstand der linken bzw. oberen Kante des Objekts zur linken bzw. oberen Kante des umgebenden Objekts, in unserem Fall des Formulars. *Width* und *Height* bezeichnen die äußeren Abmessungen eines Objektes. Im vorliegenden Koordinatensystem werden die positiven X-Werte rechts von der Homeposition, die positiven Y-Werte hingegen nach unten abgetragen.



Die grundlegenden Zeichenfunktionen innerhalb einer Leinwand (Form.Image.Canvas) sind:

- (1) Einen Bildpunkt (**Pixel**) mit den Koordinaten *x*, *y* und der Zeichenfarbe **Farbe** setzen: Dabei ist *Pixels* ein zweidimensionales Array von Objekten des Typs *TColor*. Ein solches Farbobjekt besteht intern aus einem Steuerbyte und drei Farbbytes für Rot (*r*), Grün (*g*) und Blau (*b*) und kann mit der Funktion *RGB(r,g,b)* gesetzt werden. Dabei sind *r*, *g* und *b* die Bytewerte (0 bis 255) der drei Grundfarben. Somit sind insgesamt  $256^3 = 16\,777\,216$  Farbkombinationen möglich (TrueColor).

```
var Farbe : TColor;
```

```
Farbe := clRed; // entspricht RGB(255,0,0);
Canvas.Pixels[x,y] := Farbe;
```

- (2) Die Zeichenfarbe eines Bildpunktes ermitteln: *Farbe := Canvas.Pixels[x,y];*
- (3) Den unsichtbaren Grafikkursor (Zeiger) positionieren: *Canvas.MoveTo(x,y);*
- (4) Eine Linie zwischen (*x1,y1*) und (*x2,y2*) zeichnen:

```
Canvas.MoveTo(x1,y1); // Zum Anfangspunkt der Linie
Canvas.LineTo(x2,y2); // Zum Endpunkt der Linie
```

- (5) Ein Rechteck mit den Diagonalen-Eckpunkten (*x1,y1*) und (*x2,y2*) zeichnen:

```
Canvas.Rectangle(x1,y1,x2,y2);
```

Der gesamte Bildbereich (*Image*) kann folgendermaßen gelöscht werden: Zunächst *Brush.Style* auf *bsSolid* und *Brush.Color* auf *clWhite* setzen und danach ein Rechteck über den ganzen Bildbereich zeichnen: *Rectangle(0,0,ClientWidth,ClientHeight)*.

- (6) Eine Ellipse in einen rechteckigen Bereich von (x1,y1) bis (x2,y2) einpassen:

**Canvas.Ellipse(x1,y1,x2,y2);**

- (7) Einen Ellipsensektor in einen rechteckigen Bereich von (x1,y1) bis (x2,y2) einpassen, wobei die Punkte (x3,y3) und (x4,y4) auf den Begrenzungslinien des Sektors liegen.

**Canvas.Pie(x1,y1,x2,y2,x3,y3,x4,y4);**

Ganzen Sektor zeichnen

**Canvas.Arc(x1,y1,x2,y2,x3,y3,x4,y4);**

Nur den Bogen zeichnen

- (8) Ein Vieleck (Polygon) zeichnen.

```
var P: TPoint;    { Vordefinierter Datentyp:  type TPoint = record    }
                {                               X : Integer;          }
                {                               Y : Integer;          }
                {                               end;                    }
```

```
var A : Array[1 .. 8] of TPoint;
```

```
A[1].x := 15; A[1].y := 100;
```

```
.....
```

```
A[7].x := 85; A[7].y := 180 ;
```

```
A[8] := A[1];
```

```
Canvas.Polygon(A); // Verbindet alle Punkte
```

```
Canvas.Polygon.Slice(A,5); // Verbindet nur die ersten 5 Punkte
```

- (9) Einen mit einer Randfarbe (*TColor*) begrenzten Bereich, ausgehend vom Punkt (*x,y*), mit bestimmtem Muster und Farbe ausfüllen. Füllmuster und Füllfarbe sind durch die entsprechenden *Brush*-Eigenschaften *Style* und *Color* zu definieren. Beispielsweise:  
*Canvas.FloodFill(x,y,Randfarbe,bsBorder);*

```
Canvas.Pen.Color := clBlack;
```

```
Canvas.Rectangle(0,0,200,200);
```

```
Canvas.Brush.Style := bsSolid;
```

```
Canvas.Brush.Color := clRed;
```

```
Canvas.FloodFill(10,10,clBlack,fsBorder);
```

Der geschlossene Begrenzungsrand muss aber nicht unbedingt rechteckig sein.

- (10) Einen Text *S* (*String*) mit bestimmter Größe und Farbe an der Position (*x,y*) ausgeben:

```
Canvas.Font.Size := 15;
```

```
Canvas.Font.Color := clRed;
```

```
Canvas.TextOut(x,y,S);
```

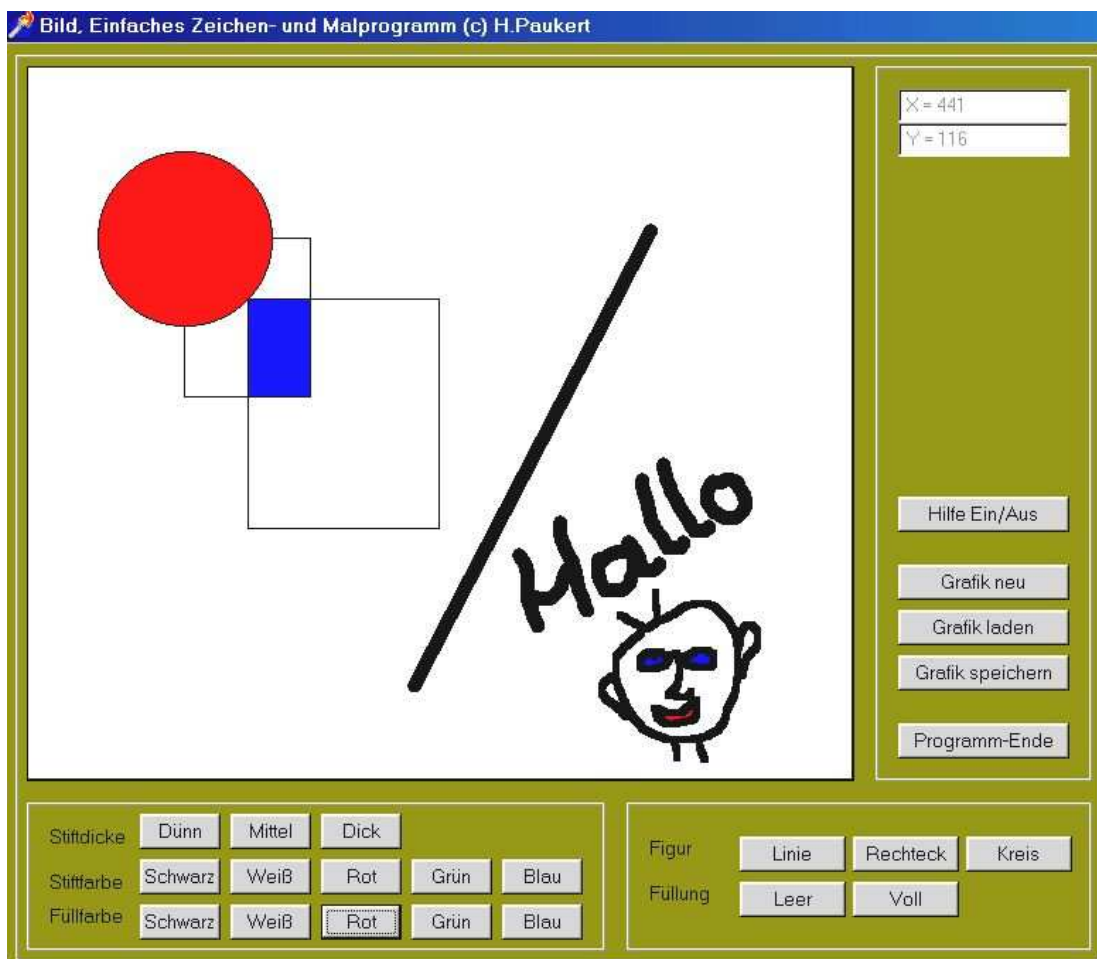
- (11) Das Bild in eine externe Grafikdatei speichern: *Image.Picture.SaveToFile('Pict.bmp')*.  
Das Bild aus einer Grafikdatei laden: *Image.Picture.LoadFromFile('Pict.bmp')*.  
Der Parameter ist dabei der Name der Grafikdatei im Bitmap-Format.
- (12) Das Bild in die Zwischenablage kopieren: *Clipboard.Assign(Image.Picture.Bitmap)*.  
Das Bild aus der Zwischenablage einfügen: *Image.Picture.Bitmap.Assign(ClipBoard)*.
- (13) Das Bild und das ganze Formular ausdrucken: *Form.Print*.  
Vorher sollte *Form.PrintScale := poProportional* gesetzt werden.

Wichtige Objekteigenschaften von *TImage* sind *TPicture* und *TGraphic* bzw. *TBitmap*. Diese spielen eine wesentliche Rolle bei der Darstellung von Grafikdateien mittels *Image*-Komponente und werden nachfolgend genauer besprochen.

Es soll ein Programm "*grafik1*" zur Erzeugung von Strichgrafiken geschrieben werden. Jede Mausbewegung mit gedrückter linker Maustaste soll dazu führen, dass eine Linie zur aktuellen Mausposition gezeichnet wird. Außerdem ist es möglich, einfache geometrische Figuren (Linie, Rechteck und Ellipse) mit Hilfe entsprechender Schaltknöpfe zu zeichnen. Dazu wird zuerst immer mit der linken Maustaste auf den ersten Eckpunkt und dann mit der rechten Maustaste auf den zweiten Eckpunkt der Diagonale des Figurenbereiches gedrückt; zuletzt muss der entsprechende Schaltknopf betätigt werden. Auch Farbe und Zeichenstiftdicke können verändert werden. Schließlich besteht noch die Möglichkeit, die produzierte Zeichnung in einer externen Grafikdatei (Bitmap-Datei) in das aktuelle Verzeichnis abzuspeichern und von dort wieder zu laden.

Wir beginnen wieder mit dem visuellen Entwurf der Bedienungsoberfläche und dem Zuweisen der Objekteigenschaften. Bei unserem einfachen Programm brauchen wir nur eine *Image*-Komponente (im Register "*Zusätzlich*" der Komponentenpalette) in das Formular einzufügen.

Durch Anklicken eines entsprechenden Schaltknopfes soll ein Memofeld mit Hilfsinformationen wechselweise eingeblendet und wieder ausgeblendet werden. Die Funktionsweisen der Schaltknöpfe sind aus der nachfolgenden Abbildung des Formulars direkt ersichtlich. Die entsprechenden ProgrammROUTINEN können im Listing genau studiert werden. Der Programmkern besteht aus drei Ereignisbehandlungsroutinen. Beim Drücken der linken Maustaste (*mbLeft*) soll die Anfangsposition der Linie festgelegt werden, indem der unsichtbare Grafikcursor mittels *MoveTo* an die aktuelle Mausposition gestellt wird. Außerdem wird diese Position in der Punktvariablen *P1* gemerkt. Bei einer Mausbewegung mit gehaltener linker Maustaste wird dann eine Linie zur neuen Mausposition gezeichnet. Wird hingegen die rechte Maustaste (*mbRight*) gedrückt, dann erfolgt eine Ausgabe der aktuellen Koordinaten (*x*, *y*) in den zwei Editierfeldern *Edit1*, *Edit2*. Außerdem wird diese Position in der Punktvariablen *P2* gemerkt. Die zwei Punkte *P1* und *P2* dienen dann als Diagonalen-Eckpunkte für Linien, Rechtecke und Kreisdefinitionen.



Die **erste Ereignisbehandlungsroutine** wird durch das *MouseDown*-Ereignis ausgelöst, wobei mit Hilfe des Parameters *Button* die niedergedrückte Maustaste festgestellt wird. Die MengenvARIABLE *Shift* gibt den Mausstatus beim Eintreffen des Ereignisses wieder - sie kann also [*ssLeft*] oder [*ssRight*] sein. Bei gedrückter linker Maustaste wird der unsichtbare Grafikcursor an die aktuelle Mausposition (*X,Y*) platziert und diese Position im Punkt **P1** gemerkt. Mit der rechten Maustaste wird die Mausposition am Bildschirm angezeigt und im Punkt **P2** gemerkt.

```
var P1, P2 : TPoint;
    Xmax, Ymax: Integer;

procedure TForm1.Image1MouseDown(Sender: TObject; Button: TMouseButton;
    Shift: TShiftState; X,Y: Integer);
begin
  if Button = mbLeft then begin
    Image1.Canvas.MoveTo(X,Y);
    P1.X := X;
    P1.Y := Y;
  end;
  if Button = mbRight then begin
    Edit1.Text := ' X = ' + IntToStr(X);
    Edit2.Text := ' Y = ' + IntToStr(Y);
    P2.X := X;
    P2.Y := Y;
  end;
end;
```

Die **zweite Ereignisbehandlungsroutine** wird durch das Ereignis *MouseMove* ausgelöst. Jedesmal wenn die Maus bewegt und gleichzeitig die linke Maustaste gedrückt wird (was mit Hilfe des Parameters *Shift* festgestellt werden kann), wird eine Linie zur neuen Position der Maus gezogen. Die Parameter *X, Y* liefern die Mausposition.

```
procedure TForm1.Image1MouseMove(Sender: TObject;
    Shift: TShiftState; X,Y: Integer);
begin
  if Shift = [ssLeft] then Image1.Canvas.LineTo(X,Y);
end;
```

Die **dritte Ereignisbehandlungsroutine** wird durch das Ereignis *DblClick* ausgelöst. Jedesmal, wenn mit der Maus an einer bestimmten Position ein Doppelklick erfolgt, soll von diesem Punkt **P1** aus der umgebende Bereich mit einer vorher gewählten Füllfarbe (*Brush.Color*) soweit ausgefüllt werden, bis eine Randlinie in der aktuellen Stiftfarbe (*Pen.Color*) erreicht wird.

```
procedure TForm1.Image1DblClick(Sender: TObject);
begin
  with Image1.Canvas do begin
    Brush.Style := bsSolid;
    FloodFill(P1.X,P1.Y,Pen.Color,bsBorder);
  end;
end;
```

Die Routinen der **weiteren Schaltknöpfe** sind sehr einfach konzipiert. Auszugsweise seien einige Beispiele angeführt.

```
procedure TForm1.Button17Click(Sender: TObject);
{ Stiftdicke auf 5 setzen }
begin
  Image1.Canvas.Pen.Width := 5;
end;

procedure TForm1.Button13Click(Sender: TObject);
{ Stiftfarbe auf ROT setzen }
begin
  Image1.Canvas.Pen.Color := clRed;
end;
```

```

procedure TForm1.Button6Click(Sender: TObject);
{ Linie zeichnen von P1 zu P2 }
begin
  with Image1.Canvas do begin
    MoveTo(p1.X,p1.Y);
    LineTo(p2.X,p2.Y);
  end;
end;

procedure TForm1.Button7Click(Sender: TObject);
{ Rechteck zeichnen mit der Diagonale von P1 zu P2 }
begin
  Image1.Canvas.Rectangle(p1.X,p1.Y,p2.X,p2.Y);
end;

procedure TForm1.Button8Click(Sender: TObject);
{ Kreis durch Punkt P2 mit Zentrum P1 zeichnen }
var r : Integer;
begin
  r := Round(Sqrt((p2.X-p1.X)*(p2.X-p1.X)+(p2.Y-p1.Y)*(p2.Y-p1.Y)));
  Image1.Canvas.Ellipse(p1.X-r,p1.Y-r,p1.X+r,p1.Y+r);
end;

```

## [02] Grafikdateien im BMP- und JPG-Format

Im Programm "*grafik2*" werden Grafikdateien im Bitmap- und im JPEG-Format mit Hilfe einer Image-Komponente dargestellt. Folgende vier grundlegende Themen werden besprochen:

- **Bitmaps und JPEG-Objekte**  
(wichtige Eigenschaften und Methoden)
- **Die Bilddarstellung großer Grafikdateien**  
(mit Hilfe einer Scrollbox)
- **Verkleinern und Vergrößern der Bilddarstellung**
- **Skaliertes Ausdrucken einer Grafik**

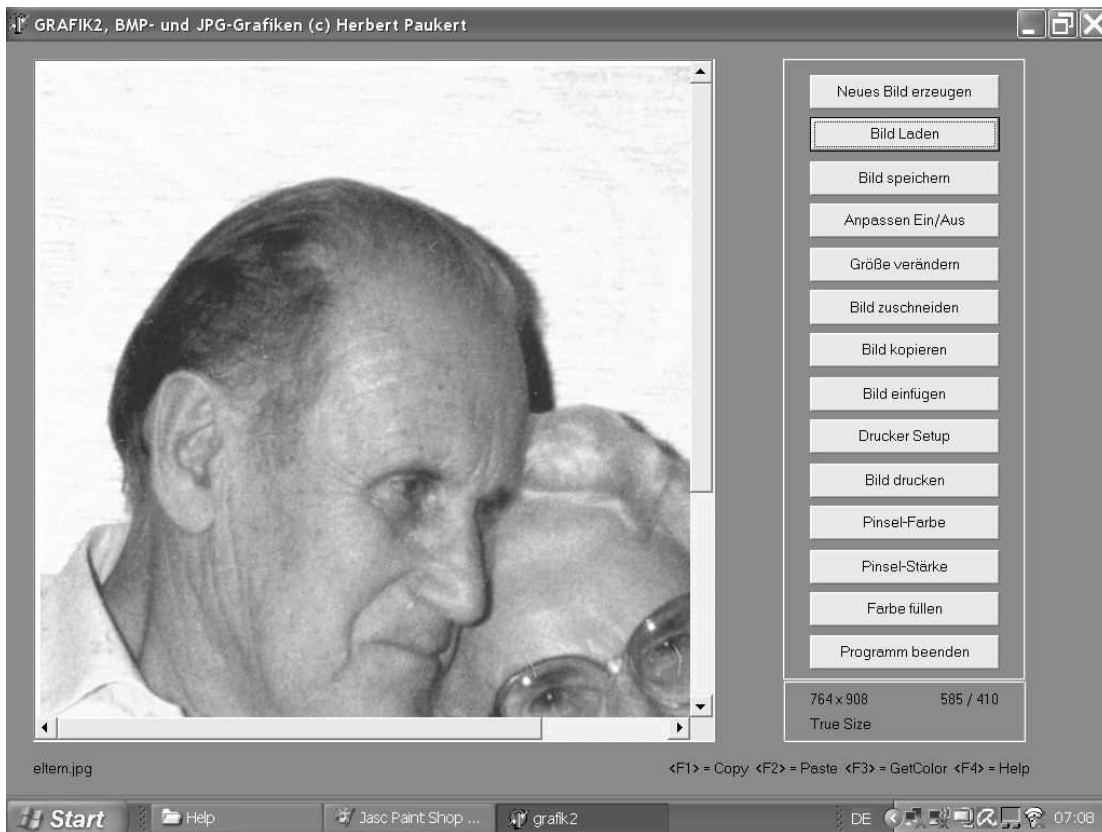
Die Komponente *TImage* dient der Darstellung von Grafikobjekten. Die Property *TPicture* ist der Container (Behälter) für diese Grafikobjekte und besitzt mächtige polymorphe Methoden zum Speichern, Laden und Bearbeiten verschiedener Grafikdateien, insbesondere Bitmaps (Pixelgrafiken), Icons (kleine Pixelgrafiken) und Metafiles (Vektorgrafiken). Die Eigenschaften von *TPicture* bestimmen Typ und Größe dieser Grafikobjekte. Soll eine Grafik auf einer Zeichenfläche (*Canvas*) ausgegeben werden, so kann das mit den Methoden *Draw* oder *StretchDraw* geschehen. Im ersten Fall wird die Grafik in unveränderter Originalgröße angezeigt, im zweiten Fall können horizontale und vertikale Abmessungen verändert werden. Das Zahlenverhältnis von Bildhöhe zur Bildbreite wird *AspectRatio* genannt.

Zwei wichtige *Image*-Eigenschaften, die erwähnt werden müssen, sind ***AutoSize*** und ***Stretch***:

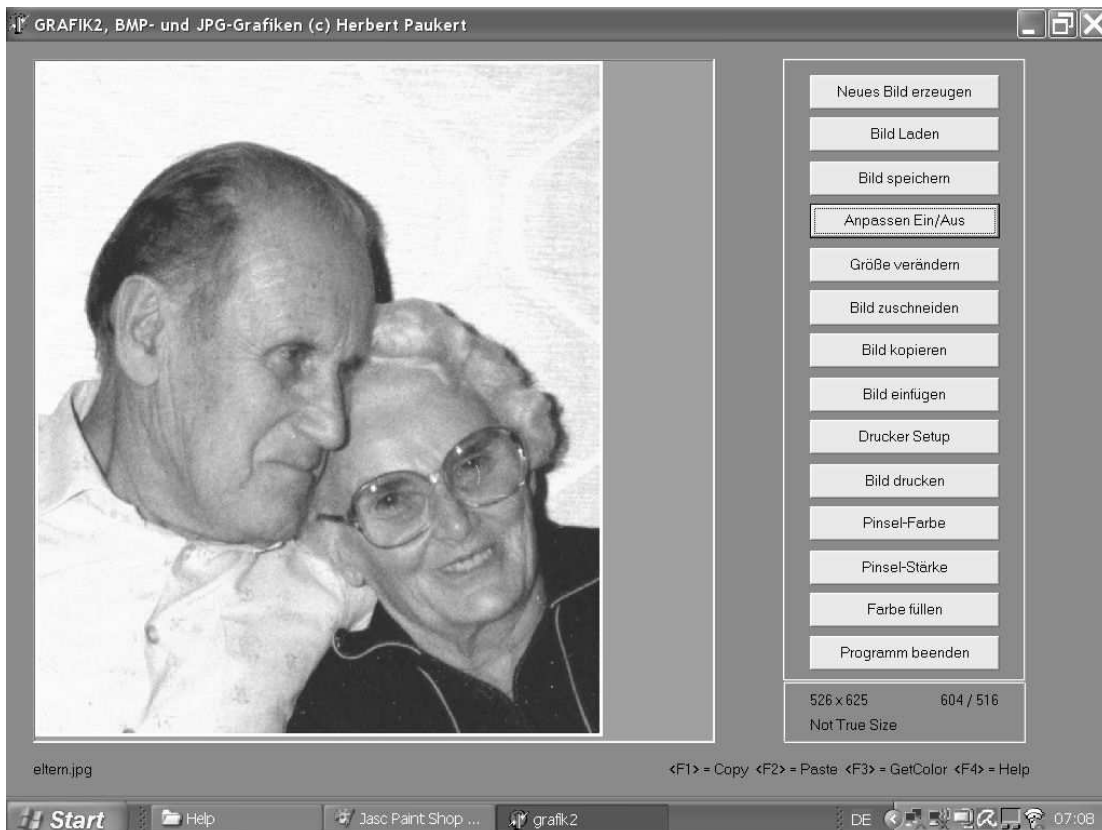
Wenn ***AutoSize*** auf *True* gesetzt ist, dann werden die Dimensionen (*Width* und *Height*) der *Image*-Komponente an die Abmessungen der geladenen Grafikdatei angepasst. Dabei kann es passieren, dass nur ein Ausschnitt einer zu großen Grafik sichtbar dargestellt wird.

Wenn ***Stretch*** auf *True* gesetzt ist, dann werden die Abmessungen der geladenen Grafikdatei an die Dimensionen der *Image*-Komponente angepasst. Dabei wird zwar das ganze Bild dargestellt, aber es kann zu erheblichen Verzerrungen der Darstellung kommen. Wie diese vermieden werden, also eine verhältnistreue Anpassung erfolgen kann, wird in der Prozedur ***FitImage*** gezeigt.

## Darstellung einer Grafikdatei OHNE Bildanpassung:



## Darstellung einer Grafikdatei MIT Bildanpassung:



## • Bitmaps und JPEG-Objekte

Die wichtigsten Grafikobjekte, welche mit Hilfe von *TImage* angezeigt werden können, sind sicherlich **Bitmaps**. Diese bestehen aus zwei Teilen: Erstens aus Strukturinformationen über die Abmessungen und über die verwendete Farbpalette, zweitens aus einem Array mit der Farbinformation für die einzelnen Bildpunkte (Pixel). Im TrueColor-Format (Farbtyp *TColor*) wird die Farbe eines Pixels in vier Bytes (ein Steuer-Byte, drei RGB-Bytes) verschlüsselt. Bei der Farb-reduktion auf nur 256 Farben wird eine Farbpalette von  $256 * 4$  Bytes (1 Kilobyte) definiert, was durch das Steuer-Byte angezeigt wird. In dem Array der Pixeldaten stehen dann nicht mehr die TrueColor-Farbwerte, sondern bloß Indizes, welche auf die entsprechenden Einträge in der Farbpalette verweisen. Weil diese Indizes auf ganzzahlige Werte zwischen 0 und 255 beschränkt sind, kann man sie durch ein Byte codieren. Dadurch schrumpft die Größe einer Bitmap erheblich. Bitmap-Dateien haben immer die Namensweiterung "**bmp**".

Die Bitmap-Eigenschaft *PixelFormat* bestimmt das interne Speicherformat der Pixelfarben (beispielsweise *pf8bit* oder *pf24bit*). Eine weitere wichtige Eigenschaft ist *Transparent*. Wird diese auf *True* gesetzt, dann ist die in der Eigenschaft *TransparentColor* definierte Farbe durchsichtig, d.h., diese Farbe wird zum transparenten Hintergrund des Bildes.

Unter **JPEG** versteht man eigentlich ein Verfahren, mit dessen Hilfe die Bilddaten komprimiert werden. Die Unit *JPEG* enthält eine Objektklasse *TJPEGImage*, welche für die Darstellung von JPEG-codierten Grafikdateien (Namenserweiterung "**jpg**") vorgesehen ist und mächtige Eigenschaften und Methoden zur deren Bearbeitung enthält.

Die Eigenschaft *TJPEGImage.PixelFormat* entspricht der gleichnamigen Property von Bitmaps. Wird *TJPEGImage.Grayscale* auf *True* gesetzt, dann erfolgt eine Umwandlung der Farbwerte in 256 Graustufen beim Laden der Bilddatei. Die Eigenschaft *TJPEGImage.CompressionQuality* bestimmt einen ganzzahligen Wert von 1 bis 100, welcher über die Bildqualität bei der Daten-Komprimierung entscheidet. Je höher dieser Wert ist, umso besser ist die Bildqualität. Entsprechend größer wird dann aber die Bilddatei bei ihrer Speicherung.

Wichtig ist noch die Tatsache, dass JPEG-Images keine Zeichenfläche (*Canvas*) besitzen. Will man in JPEG-Grafiken zeichnen und malen, dann muss die JPEG-Grafik in eine Bitmap umgewandelt werden. Das folgende Listing zeigt das Laden, Umwandeln und Speichern von Grafikdateien. Dabei werden zwei Image-Objekte dynamisch zur Laufzeit erzeugt.

```
var Image1: TImage; // Standard-Image
    Image2: TJPEGImage; // JPEG-Image
    FName1, FName2: String; // Datei-Namen
    K: Integer; // Kompressions-Faktor

begin
    Image1 := TImage.Create; // Bitmap-Objekt-Erzeugung
    Image1.AutoSize := True;
    Image1.Stretch := False;

    Image2 := TJPEGImage.Create; // JPEG-Objekt-Erzeugung

    Image1.Picture.Bitmap.LoadFromFile(FName1); // Bitmap-Datei laden
    Image1.Picture.Bitmap.PixelFormat := pf24bit;

    Image2.Assign(Image1.Picture.Bitmap); // Bitmap in JPEG umwandeln
    Image2.CompressionQuality := K; // Datenkomprimierung
    Image2.SaveToFile(FName2); // JPEG-Datei speichern

    Image2.LoadFromFile(FName2); // JPEG-Datei laden
    Image2.PixelFormat := jf24bit;
    Image1.Picture.Bitmap.Assign(Image2); // JPEG in Bitmap umwandeln
    Image1.Picture.Bitmap.SaveToFile(FName1); // Bitmap-Datei speichern
end;
```

## • Die Bilddarstellung großer Grafikdateien

Wie werden große Grafikdateien am Bildschirm dargestellt? Da gibt es verschiedene Möglichkeiten - die einfachste besteht in der Einrichtung einer *Scrollbox* auf dem Formular. Diese dient als Container (Behälter) für eine *Image*-Komponente, welche in der *Scrollbox* platziert wird. Dann stellt man die Eigenschaft *TImage.AutoSize* auf *True*. Dadurch wird die *Image*-Komponente in ihren Dimensionen an die Abmessungen der geladenen Grafikdatei automatisch angepasst. Mit Hilfe der *Scrollbox* ist nun ein Scrollen des unverzerrten Bildes möglich, denn alles, was innerhalb der Box liegt, wird beim Bildlauf mitbewegt. Alle Objekte außerhalb der Box bleiben auf ihren Positionen.

## • Verkleinern und Vergrößern der Bilddarstellung

Will man eine große Grafikdatei in eine *Image*-Komponente mit fixen Dimensionen unverzerrt und optimal einpassen, so muss die Grafik so verkleinert werden, dass sich ihr Seitenverhältnis (das so genannte *AspectRatio* = Bildhöhe : Bildbreite) nicht verändert und ihre größte Seite mit der entsprechenden Imageseite übereinstimmt. Dazu wird zunächst die Grafikbreite auf die fixe Imagebreite gesetzt und die Grafikhöhe ist gleich dem Produkt aus Grafikbreite mal *AspectRatio*. Wenn diese so berechnete Grafikhöhe nicht zur Gänze in die fixe Imagehöhe hineinpasst, dann verfährt man in umgekehrter Weise und setzt die Grafikhöhe auf die fixe Imagehöhe und die Grafikbreite ist nun gleich dem Produkt aus Grafikhöhe mal dem Kehrwert vom *AspectRatio*. Dadurch ist eine unverzerrte und optimale Anpassung erreicht. Natürlich muss die Eigenschaft *TImage.AutoSize* auf *False* gestellt sein, damit sich die Dimensionen der *Image*-Komponente beim Laden der Grafikdatei nicht verändern. Die Eigenschaft *TImage.Stretch* muss auf *True* gestellt sein, damit sich die Abmessungen der geladenen Grafik an die fixen Dimensionen der *Image*-Komponente anpassen.

Will man die beschriebenen Anpassungen wieder rückgängig machen und die dargestellte Grafik auf ihre ursprünglichen Abmessungen zurücksetzen, dann müssen *TImage.AutoSize* auf *True* und *TImage.Stretch* auf *False* gestellt werden. Ob eine Anpassung erfolgen soll oder nicht, wird in der Prozedur ***FitImage*** durch die boolesche Steuervariable ***Fit*** entschieden.

In der Prozedur ***FitImage*** wird noch ein zusätzlicher Streckungsfaktor *f* eingeführt, mit dem Bildbreite und Bildhöhe multipliziert werden, was eine Verkleinerung ( $f < 1$ ) oder Vergrößerung ( $f > 1$ ) bewirkt. Bei  $f = 1$  liegt eine genaue Grafikanpassung an die fixen Imagedimensionen vor.

```
var Fit: Boolean;           // Steuervariable für die Grafikanpassung
    Xmax, YMax: Integer;   // Dimensionen der Image-Komponente
    f: Real;               // Streckungsfaktor
    FName: String;        // Name der Grafikdatei

procedure FitImage(I: TImage; IW,IH: Integer; f: Real; Fit: Boolean);
var w,h: Integer;
    k: Real;
begin
  I.Visible := False;
  if Fit then begin
    k := I.Height / I.Width;
    w := Round(IW * f);
    h := Round(w * k);
    if h > IH then begin
      k := 1/k;
      h := Round(IH * f);
      w := Round(h * k);
    end;
    I.AutoSize := False;
    I.Stretch := True;
    I.Width := w;
    I.Height := h;
  end;
end;
```

```

    if not Fit then begin
        I.AutoSize := True;
        I.Stretch := False;
        I.Width := Round(IW * f);
        I.Height := Round(IH * f);
    end;
    I.Visible := True;
end;

begin
    with Form1 do begin
        Image1.AutoSize := True;
        Image1.Stretch := False;
        Xmax := Image1.Width;
        Ymax := Image1.Height;
        Image1.Picture.Bitmap.LoadFromFile(FName);
        Fit := True;
        f := 1.00;
        FitImage(Image1, Xmax, Ymax, f, Fit);
    end;
end;

```

### • Skaliertes Ausdrucken einer Grafik

Wie wird die Grafik einer *Image*-Komponente skaliert ausgedruckt? Dazu wird auf der Zeichenfläche (*Canvas*) des Druckers ein passender rechteckiger Bereich bestimmt, in welchem die *Image*-Komponente unverzerrt und optimal abgebildet und dann ausgedruckt wird. Zuerst wird das Verhältnis von Bildhöhe zur Bildbreite bestimmt (*AspectRatio*). Dann wird die Bereichsbreite auf die Breite des ganzen Druckerblattes gesetzt und die Bereichshöhe ist gleich dem Produkt aus der Bereichsbreite mal dem *AspectRatio*. Wenn diese so berechnete Bereichshöhe nicht zur Gänze in die Höhe des Druckerblattes hineinpasst, dann verfährt man in umgekehrter Weise und setzt die Bereichshöhe auf die Höhe des Druckerblattes und die Bereichsbreite ist nun gleich dem Produkt aus Bereichshöhe mal dem Kehrwert vom *AspectRatio*. Dadurch ist eine unverzerrte und optimale Abbildung der Grafik in den rechteckigen Druckbereich erreicht. Zusätzlich werden noch Breite und Höhe dieses Bereichs mit einem Skalierungsfaktor multipliziert, wodurch ein Verkleinern oder Vergrößern des Druckbereiches bezogen auf das ganze Druckerblatt möglich wird. Die eigentliche Ausgabe des Druckbereiches leistet die Methode *StretchDraw* der Zeichenfläche des Druckers.

In der folgenden Prozedur ***PrintImage*** wird das beschriebene Verfahren codiert. Voraussetzung ist, dass die Unit *Printers* in das Programm eingebunden ist.

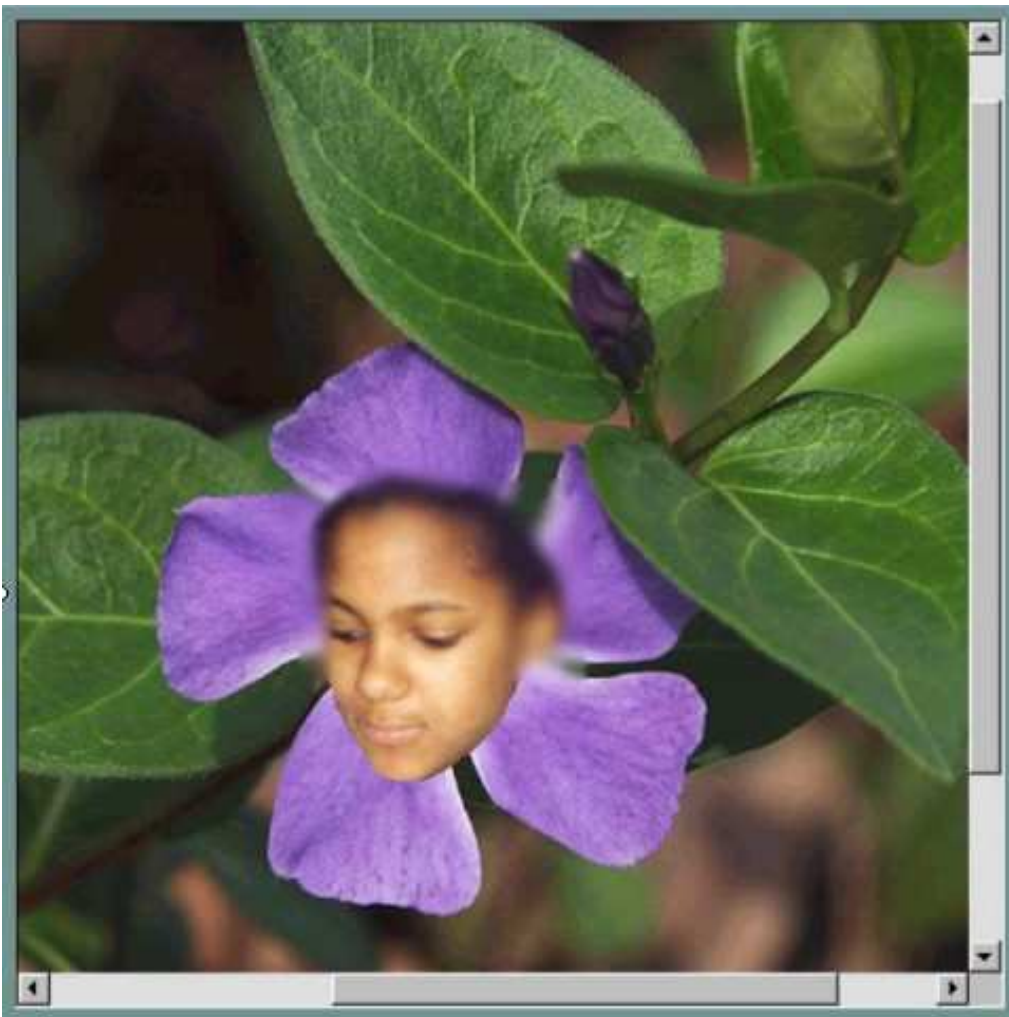
```

procedure PrintImage(I:TImage; f: Real);
var w,h : Integer;
    k : Real;
    Bereich : TRect;
begin
    Printer.BeginDoc;
    k := I.Height / I.Width;
    w := Round(Printer.PageWidth * f);
    h := Round(w * k);
    if h > Printer.PageHeight then begin
        k := 1/k;
        h := Round(Printer.PageHeight * f);
        w := Round(h * k);
    end;
    Bereich := Rect(0,0,w,h);
    Printer.Canvas.StretchDraw(Bereich,I.Picture.Graphic);
    Printer.EndDoc;
end;

```

## Fortgeschrittene Grafikverarbeitung

Das Programm "*grafik2*" demonstriert auch einige fortgeschrittene Techniken, welche in jeder professionellen Bildbearbeitung vorkommen. Als Beispiel wird ein Ausschnitt des zweiten Bildes herauskopiert, entsprechend verkleinert und dann mit Hintergrund-Transparenz in das erste Bild eingefügt (Clonen).



Als erstes Beispiel soll das "**Clonen eines Bildes**" besprochen werden. Kernstück des Programmcodes sind die beiden im System vordefinierten Canvas-Methoden *CopyRect* und *StretchDraw*. Bei der ersten Methode wird ein rechteckiger Ausschnitt aus einer Zeichenfläche (Quelle) in eine andere Zeichenfläche (Ziel) kopiert. Bei der zweiten Methode wird das ganze Grafikobjekt einer Bildquelle in einen rechteckigen Bereich einer anderen Zeichenfläche kopiert. Beide Methoden werden in den folgenden Routinen *CopyImage* und *PasteImage* verwendet.

```

var Image1: TImage;           // Standard-Image für Bitmaps
    Image3: TImage;           // Puffer-Image für Zwischenspeicherungen
    x1,y1,x2,y2: Integer;     // Diagonalen-Eckpunkte eines Rechteck-Bereichs
    x,y: Integer;             // Zielposition in einem Image

procedure CopyImage(Quelle,Puffer:TImage; x1,y1,x2,y2: Integer; HT: Boolean);
// kopiert einen rechteckigen Bereich eines Quellen-Images
// in einen Zwischenpuffer mit oder ohne Hintergrundtransparenz (HT)
var W,H : Integer;
begin
    W := x2 - x1;
    H := y2 - y1;
    Puffer.Picture := NIL;
    Puffer.AutoSize := False;
    Puffer.Stretch := False;
    Puffer.Width := W;
    Puffer.Height := H;
    Puffer.Canvas.CopyRect(Rect(0,0,W,H),Quelle.Canvas,Rect(x1,y1,x2,y2));
    Puffer.AutoSize := True;
    Puffer.Picture.Bitmap.PixelFormat := pf24bit;
    Puffer.Picture.Bitmap.TransparentColor := clBlack;
    Puffer.Transparent := HT;
end;

procedure PasteImage(Puffer,Ziel: TImage; x,y: Integer);
// Fügt einen zwischengepufferten Bereich
// an die Position (x,y) in ein Ziel-Image ein
var W,H : Integer;
begin
    W := Puffer.Width;
    H := Puffer.Height;
    Ziel.Canvas.StretchDraw(Rect(x,y,x+W,y+H),Puffer.Picture.Graphic);
    Ziel.Picture.Bitmap.PixelFormat := pf24bit;
end;

begin
    ..... // Rechteck-Bereich für das Kopieren festlegen (x1,y1,x2,y2)
    .....
    CopyImage(Image1,Image3,x1,y1,x2,y2,True);
    .....
    ..... // Zielposition für das Einfügen festlegen (x,y)
    .....
    PasteImage(Image3,Image1,x,y);
    .....
end;

```

Um einen beliebigen, nicht rechteckigen Bereich eines Bildes (Quelle) an eine bestimmte Position eines anderen Bildes (Ziel) mit Hintergrund-Transparenz einzufügen (Clonen), muss in der Praxis im Programm "*grafik2*" folgendermaßen vorgegangen werden:

- (1) Laden der ersten Bilddatei (Quelle).
- (2) Die Bildgröße entsprechend verändern und das Bild entsprechend zuschneiden.
- (3) Zeichnen einer geschlossenen Randlinie um den gewünschten Bildbereich mit Hilfe der Maus. Das entspricht der Funktion einer Freihand-Schere. Als Zeichenfarbe dabei schwarz wählen.
- (4) Schalter <Füllen Ein/Aus> betätigen und Füllfarbe schwarz wählen.
- (5) Mausclick auf einen nicht schwarzen Punkt außerhalb des Bereichs. Dadurch wird das Äußere des Bereichs mit schwarzer Hintergrund-Farbe gefüllt.

- (6) Den Bildbereich mit Hilfe der Maus markieren (Erklärung siehe weiter unten).
- (7) Schalter *<Bild kopieren>* betätigen und dabei Hintergrund-Transparenz einstellen.
- (8) Laden der zweiten Bilddatei (Ziel).  
(Dieses originale Bild ev. mit *<F1>* in das Windows-Clipboard speichern.)
- (9) Schalter *<Bild einfügen>* betätigen.
- (10) Linker Mausklick auf jene Bildposition, wo der kopierte Bereich eingefügt werden soll.  
(Die Einfügung kann ev. mit *<F2>* wieder rückgängig gemacht werden.)

Bei diesem Verfahren sind alle schwarzen Bildpunkte des eingefügten rechteckigen Bereichs transparent, d.h., dort sind die Bildpunkte des darunter liegenden Bildes sichtbar. Dadurch ist es möglich, den ursprünglich nicht rechteckigen Bildausschnitt exakt einzufügen.

Als zweites Beispiel soll das "**Ändern der Größe eines Bildes**" besprochen werden. Kernstück des Programmcodes ist die nachfolgende Prozedur. Der Parameter *k* ist der Streckungsfaktor des Bildes. Damit wird das Bild in ein ähnliches (verkleinertes oder vergrößertes) Bild umgewandelt, wobei das Verhältnis von Bildhöhe zur Bildbreite (aspect ratio) unverändert bleibt.

```

procedure StretchImage(Quelle,Puffer: TImage; k: Real);
// Verkleinerung oder Vergrößerung eines Images über einen Puffer
// mit k als Streckungsfaktor
var W,H: Integer;
begin
  k := abs(k);
  W := Quelle.Width;
  H := Quelle.Height;
  W := Round(W*k);
  H := Round(H*k);

  Puffer.Picture := NIL;
  Puffer.AutoSize := False;
  Puffer.Stretch := False;
  Puffer.Width := W;
  Puffer.Height := H;
  Puffer.Canvas.StretchDraw(Rect(0,0,W,H),Quelle.Picture.Graphic);
  Puffer.AutoSize := True;

  Quelle.Picture := NIL;
  Quelle.AutoSize := False;
  Quelle.Stretch := False;
  Quelle.Width := W;
  Quelle.Height := H;
  Quelle.Canvas.Draw(0,0,Puffer.Picture.Bitmap);
  Quelle.AutoSize := True;
  Quelle.Stretch := True;
  Quelle.Picture.Bitmap.PixelFormat := pf24bit;

  Puffer.Picture := NIL;
end;

```

Als drittes Beispiel soll das "**Zuschneiden eines Bildes**" auf einen bestimmten rechteckigen Bildausschnitt besprochen werden. Kernstück des Programmcodes ist die nachfolgende Prozedur. Die Parameter  $x_1, y_1, x_2, y_2$  geben den rechteckigen Bildausschnitt an. Wie ein rechteckiger Bildausschnitt zu markieren ist, wird dann weiter unten erklärt.

```

procedure CropImage(var Quelle, Puffer: TImage; x1, y1, x2, y2: Integer);
// Zuschneiden eines Images auf einen rechteckigen Bildausschnitt
var W, H : Integer;
begin
  W := x2 - x1;
  H := y2 - y1;
  Puffer.Picture := NIL;
  Puffer.Transparent := False;
  Puffer.AutoSize := False;
  Puffer.Stretch := False;
  Puffer.Width := W;
  Puffer.Height := H;
  Puffer.Canvas.CopyRect(Rect(0, 0, W, H), Quelle.Canvas, Rect(x1, y1, x2, y2));
  Puffer.AutoSize := True;

  Quelle.Picture := NIL;
  Quelle.AutoSize := False;
  Quelle.Stretch := False;
  Quelle.Width := W;
  Quelle.Height := H;
  Quelle.Canvas.Draw(0, 0, Puffer.Picture.Bitmap);
  Quelle.AutoSize := True;
  Quelle.Stretch := True;
  Quelle.Picture.Bitmap.PixelFormat := pf24bit;
  Puffer.Picture := NIL;
end;

```

Ein rechteckiger Bildausschnitt wird im Programm "grafik2" mit Hilfe der nachfolgenden Prozedur **MarkFrame** markiert:

```

procedure MarkFrame(I: TImage; x1, y1, x2, y2: Integer);
// zieht einen Markierungsrahmen im NOT-XOR-Modus
var PCol : TColor;
    PWid : Integer;
begin
  with I.Canvas do begin
    PCol := Pen.Color;
    PWid := Pen.Width;
    Brush.Style := bsClear;
    Pen.Width := 1;
    Pen.Color := clBlack;
    Pen.Style := psDot;
    Pen.Mode := pmNotXor; // Punktfarben mit NOT und XOR manipulieren
    Rectangle(x1, y1, x2, y2);
    Pen.Mode := pmCopy;
    Pen.Style := psSolid;
    Pen.Color := PCol;
    Pen.Width := PWid;
  end;
end;

```

Dazu müssen im Hauptprogramm zwei globale Variable **A** und **B** vom Typ **TPoint** definiert sein. Bei einem linken Mausklick wird die aktuelle Bildposition in A gespeichert. Bei einem rechten Mausklick wird die aktuelle Bildposition in B gespeichert und mit Hilfe der oben stehenden Prozedur **MarkFrame**(Image1, A.x, A.y, B.x, B.y) ein Markierungsrahmen gezeichnet. Bei einem neuerlichen rechten Mausklick wird **MarkFrame** zweimal aufgerufen. Dadurch wird der alte Markierungsrahmen gelöscht, das originale Bild wiederhergestellt und ein neuer Markierungsrahmen gezeichnet. Falls ein Markierungsrahmen existiert, dann wird bei einem neuerlichen linken Mausklick **MarkFrame** nur einmal aufgerufen und somit gelöscht. Dabei wird die logische Steuervariable **MarkFlag** verwendet, welche nur dann wahr ist, wenn ein Markierungsrahmen existiert.

Auf den folgenden Seiten ist das komplette Listing des Programms “grafik2“ abgedruckt. Das Programm “paupixel“ ist eine Erweiterung des Programmes “grafik2“ und stellt bereits eine semiprofessionelle Grafikverarbeitung dar.

```

unit grafik2_u;
// BMP- und JPG-Grafikdateien darstellen und manipulieren (c) H.Paukert

interface
uses
  Windows, Messages, SysUtils, Classes, Graphics,
  Controls, Forms, Dialogs, ExtCtrls, StdCtrls, ExtDlgs,
  Printers, ClipBrd, JPEG;

type
  TForm1 = class(TForm)
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    Label4: TLabel;
    Label5: TLabel;
    Memo1 : TMemo;
    ScrollBox1: TScrollBox;
    Image1: TImage;
    Bevel1: TBevel;
    Bevel2: TBevel;

    Button1: TButton;
    Button2: TButton;
    Button3: TButton;
    Button4: TButton;
    Button5: TButton;
    Button6: TButton;
    Button7: TButton;
    Button8: TButton;
    Button9: TButton;
    Button10: TButton;
    Button11: TButton;
    Button12: TButton;
    Button13: TButton;
    Button14: TButton;

    PrinterSetupDialog1: TPrinterSetupDialog;
    OpenPictureDialog1: TOpenPictureDialog;
    SavePictureDialog1: TSavePictureDialog;
    ColorDialog1: TColorDialog;

    procedure FormActivate(Sender: TObject);
    procedure FormKeyUp(Sender: TObject; var Key: Word;
      Shift: TShiftState);
    procedure Image1MouseDown(Sender: TObject; Button: TMouseButton;
      Shift: TShiftState; X, Y: Integer);

    procedure Image1MouseMove(Sender: TObject; Shift: TShiftState;
      X,Y: Integer);

    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
    procedure Button3Click(Sender: TObject);
    procedure Button4Click(Sender: TObject);
    procedure Button5Click(Sender: TObject);
    procedure Button6Click(Sender: TObject);
    procedure Button7Click(Sender: TObject);
    procedure Button8Click(Sender: TObject);
    procedure Button9Click(Sender: TObject);
    procedure Button10Click(Sender: TObject);
    procedure Button11Click(Sender: TObject);
    procedure Button12Click(Sender: TObject);
    procedure Button13Click(Sender: TObject);
    procedure Button14Click(Sender: TObject);
    private { Private-Deklarationen }
    public { Public-Deklarationen }
  end;

var Form1: TForm1;

```

```

implementation
{$R *.DFM}

var Image2: TJPEGImage; // JPEGG-Bild
    Image3: TImage; // Bitmap-Bild als Zwischenpuffer
    Verz, // Aktuelles Verzeichnis
    FExt, // Dateierweiterung
    Fname : String; // Dateiname
    Xmax, // Bildbreite
    Ymax : Integer; // Bildhöhe
    pict : Boolean; // Steuervariable für "Bild geladen"
    Fit : Boolean; // Steuervariable für Bildanpassung
    InsertFlag : Boolean; // Steuervariable für "Bild einfügen"
    MarkFlag : Boolean; // Steuervariable für "Markierungsrahmen"
    FillFlag : Boolean; // Steuervariable für "Farbe füllen"
    FillColor : TColor; // Gewählte Füllfarbe
    A,B : TPoint; // Eckpunkte des markierten Bildbereiches
    P : TPoint; // Aktuelle Mausposition

procedure FitForm(F : TForm);
// Anpassung des Formulars an die Monitorauflösung
const SW: Integer = 1024;
    SH: Integer = 768;
    FS: Integer = 96;
    FL: Integer = 120;
var X,Y,K: Integer;
    Z: Real;
begin
    with F do begin
        X := Screen.Width;
        Y := Screen.Height;
        K := Font.PixelsPerInch;
        Scaled := True;
        Z := Y/X;
        if (Z >= 0.75) then ScaleBy(X,SW) else ScaleBy(Y,SH);
        if (K <> FS) then ScaleBy(FS,K);
        WindowState := wsMaximized;
    end;
end;

procedure PrintMemo(M: TMemo);
// Memotext ausdrucken
// Ein Punkt am Zeilenanfang bewirkt einen Seitenvorschub
const LR = ' ';
    FF = '.';
var S : String;
    N : Integer;
    Datei: TextFile;
begin
    AssignPrn(Datei);
    {$I-} Rewrite(Datei); {$I+}
    if IOResult <> 0 then begin
        MessageBox(0, ' KEIN Druckerzugriff ! ', 'Problem', 16);
        Exit;
    end;
    Printer.Canvas.Font := M.Font;
    Printer.Canvas.Font.Size := 11;
    Writeln(Datei,LR);
    Writeln(Datei,LR);
    For N := 0 to M.Lines.Count-1 do begin
        S := M.Lines[N];
        if Pos(FF,Trim(S)) = 1 then begin
            Writeln(Datei,#12);
            Writeln(Datei,LR);
        end
        else Writeln(Datei,LR + S);
    end;
    CloseFile(Datei);
end;

```

```

procedure MarkFrame(I: TImage; x1,y1,x2,y2: Integer);
// zieht einen Markierungsrahmen im NOT-XOR-Modus
var PCol : TColor;
    PWid : Integer;
begin
  with I.Canvas do begin
    PCol := Pen.Color;
    PWid := Pen.Width;
    Brush.Style := bsClear;
    Pen.Width := 1;
    Pen.Color := clBlack;
    Pen.Style := psDot;
    Pen.Mode := pmNotXor;
    Rectangle(x1,y1,x2,y2);
    Pen.Mode := pmCopy;
    Pen.Style := psSolid;
    Pen.Color := PCol;
    Pen.Width := PWid;
  end;
end;

procedure PaintImage(I: TImage; F: TColor; X1,Y1,X2,Y2 : Integer);
// Füllt einen rechteckigen Image-Bereich mit der Farbe F
var PCol, BCol: TColor;
begin
  with I.Canvas do begin
    PCol := Pen.Color;
    BCol := Brush.Color;
    Pen.Color := F;
    Brush.Color := F;
    Brush.Style := bsSolid;
    FillRect(Rect(X1,Y1,X2,Y2));
    Brush.Style := bsClear;
    Brush.Color := BCol;
    Pen.Color := PCol;
  end;
end;

procedure NewImage(I:TImage; F:TColor; IW,IH: Integer);
// Löscht das alte Image und initialisiert ein neues Image
// mit der Breite IW und der Höhe IH und
// mit der Farbe F (0 = Schwarz, 1 = Weiß)
begin
  with I do begin
    Picture := NIL;
    AutoSize := False;
    Stretch := False;
    Width := IW;
    Height := IH;
    PaintImage(I,F,0,0,IW,IH);
    Picture.Bitmap.PixelFormat := pf24bit;
    Picture.Bitmap.TransparentColor := clBlack;
    AutoSize := True;
    Stretch := True;
    Fit := False;
  end;
end;

procedure FitImage(I: TImage; IW,IH: Integer; f: Real; Fit: Boolean);
// Anpassen einer geladenen Grafikdatei in ein vorgegebenes Image
// mit den Abmessungen IW und IH. Zusätzlich wird noch ein Faktor f
// zur Vergrößerung oder Verkleinerung des Bildes eingegeben.
// Wenn Fit = TRUE, dann Anpassung - andernfalls keine Anpassung.
var w,h: Integer;
    k : Real;
begin
  I.Visible := False;
  if Fit then begin
    k := I.Height / I.Width;
    w := Round(IW * f);
    h := Round(w * k);
    if h > IH then begin
      k := 1/k;
      h := Round(IH * f);
      w := Round(h * k);
    end;
  end;

```

```

    I.AutoSize := False;
    I.Stretch := True;
    I.Width := w;
    I.Height := h;
end;
if not Fit then begin
    I.AutoSize := True;
    I.Stretch := False;
    I.Width := Round(IW * f);
    I.Height := Round(IH * f);
end;
I.Stretch := True;
I.Visible := True;
end;

procedure CopyImage(Quelle, Puffer: TImage; x1, y1, x2, y2: Integer; HT: Boolean);
// kopiert einen Bereich des Quellen-Images in einen Zwischenpuffer
// mit oder ohne Hintergrundtransparenz
var W, H : Integer;
begin
    W := x2 - x1;
    H := y2 - y1;
    Puffer.Picture := NIL;
    Puffer.AutoSize := False;
    Puffer.Stretch := False;
    Puffer.Width := W;
    Puffer.Height := H;
    Puffer.Canvas.CopyRect(Rect(0, 0, W, H), Quelle.Canvas, Rect(x1, y1, x2, y2));
    Puffer.AutoSize := True;
    Puffer.Picture.Bitmap.PixelFormat := pf24bit;
    Puffer.Picture.Bitmap.TransparentColor := clBlack;
    Puffer.Transparent := HT;
end;

procedure PasteImage(Puffer, Ziel: TImage; x, y: Integer);
// Fügt einen zwischengepufferten Bereich
// in ein Ziel-Image ein - an die Position x, y
var W, H : Integer;
begin
    W := Puffer.Width;
    H := Puffer.Height;
    Ziel.Canvas.StretchDraw(Rect(x, y, x+W, y+H), Puffer.Picture.Graphic);
    Ziel.Picture.Bitmap.PixelFormat := pf24bit;
end;

procedure StretchImage(Quelle, Puffer: TImage; k: Real);
// Verkleinerung oder Vergrößerung eines Images über einen Puffer
var W, H: Integer;
begin
    k := abs(k);
    W := Quelle.Width;
    H := Quelle.Height;
    W := Round(W*k);
    H := Round(H*k);
    Puffer.Picture := NIL;
    Puffer.AutoSize := False;
    Puffer.Stretch := False;
    Puffer.Width := W;
    Puffer.Height := H;
    Puffer.Canvas.StretchDraw(Rect(0, 0, W, H), Quelle.Picture.Graphic);
    Puffer.AutoSize := True;

    Quelle.Picture := NIL;
    Quelle.AutoSize := False;
    Quelle.Stretch := False;
    Quelle.Width := W;
    Quelle.Height := H;
    Quelle.Canvas.Draw(0, 0, Puffer.Picture.Bitmap);
    Quelle.AutoSize := True;
    Quelle.Stretch := True;
    Quelle.Picture.Bitmap.PixelFormat := pf24bit;
    Puffer.Picture := NIL;
end;

```

```

procedure CropImage(var Quelle,Puffer: TImage; x1,y1,x2,y2: Integer);
// Zuschneiden eines Images auf einen markierten Bereich
var W,H : Integer;
begin
  W := x2 - x1;
  H := y2 - y1;
  Puffer.Picture := NIL;
  Puffer.Transparent := False;
  Puffer.AutoSize := False;
  Puffer.Stretch := False;
  Puffer.Width := W;
  Puffer.Height := H;
  Puffer.Canvas.CopyRect(Rect(0,0,W,H),Quelle.Canvas,Rect(x1,y1,x2,y2));
  Puffer.AutoSize := True;

  Quelle.Picture := NIL;
  Quelle.AutoSize := False;
  Quelle.Stretch := False;
  Quelle.Width := W;
  Quelle.Height := H;
  Quelle.Canvas.Draw(0,0,Puffer.Picture.Bitmap);
  Quelle.AutoSize := True;
  Quelle.Stretch := True;
  Quelle.Picture.Bitmap.PixelFormat := pf24bit;
  Puffer.Picture := NIL;
end;

procedure LoadImage;
// Bild laden
var JPGGray : Boolean;
begin
  with Form1 do begin
    Image1.AutoSize := True;
    Image1.Stretch := False;
    Fit := True;
    with OpenPictureDialog1 do begin
      InitialDir := Verz;
      Filter := 'JPG-Dateien |*.jpg|' +
        'BMP-Dateien |*.bmp' ;
      DefaultExt := 'jpg';
      Options := [ofFileMustExist];
      FileName := '';
      if Execute then begin
        Verz := Trim(LowerCase(ExtractFilePath(FileName)));
        FName := Trim(LowerCase(ExtractFileName(FileName)));
        FExt := Trim(LowerCase(ExtractFileExt(FileName)));
        try
          if FExt = '.jpg' then begin
            if MessageBox(0,'Bild in Farben anzeigen (wenn möglich) ?',
              'Frage',36) = 6 then JPGGray := False
              else JPGGray := True;

            if JPGGray then Image2.GrayScale := True
              else Image2.GrayScale := False;
            Image2.LoadFromFile(FileName);
            Image2.PixelFormat := jf24bit;
            Image1.Picture.Bitmap.Assign(Image2);
            Image1.Picture.Bitmap.PixelFormat := pf32bit;
          end;
          if FExt = '.bmp' then begin
            Image1.Picture.Bitmap.LoadFromFile(FileName);
            Image1.Picture.Bitmap.PixelFormat := pf32bit;
            ShowMessage('Bitmap-Grafik erfolgreich geladen !');
          end;
          Label1.Caption := FName;
        except
          MessageBox(0,'Daten-Fehler', 'Problem',16);
        end;
        Image1.Stretch := True;
        Image1.Picture.Bitmap.TransparentColor := clBlack;
        Fit := False;
        Pict := True;
      end;
    end;
  end;
end;
end;
end;

```

```

procedure SaveImage;
// Bild speichern
var JPGBild : Boolean;
    S : String;
    k,Code : Integer;
begin
with Form1 do begin
    if MessageBox(0,'Bild als JPG-Datei speichern',
        'Frage',36) = 6 then JPGBild := True
        else JPGBild := False;

with SavePictureDialog1 do begin
    InitialDir := Verz;
    Options := [ofOverWritePrompt];
    if JPGBild then begin
        S := InputBox('Komprimierungs-Qualität (0 bis 100)','','80');
        Val(S,k,Code);
        if (Code <> 0) or (k < 0) or (k > 100) then begin
            ShowMessage('Falsche Werteingabe !');
            k := 1;
        end;
        Filter := 'JPG-Dateien |*.jpg';
        DefaultExt := '.jpg';
        if FName <> '' then
            FName := Copy(FName,1,Pos('.',FName)-1) + '.jpg';
            FileName := Verz + FName;
        end
    else begin
        Filter := 'BMP-Dateien |*.bmp';
        DefaultExt := '.bmp';
        if FName <> '' then
            FName := Copy(FName,1,Pos('.',FName)-1) + '.bmp';
            FileName := Verz + FName;
        end
    end;
    if Execute then begin
        Verz := Trim(LowerCase(ExtractFilePath(FileName)));
        FName := Trim(LowerCase(ExtractFileName(FileName)));
        FExt := Trim(LowerCase(ExtractFileExt(FileName)));
        try
            if JPGBild then begin
                Image2.Assign(Image1.Picture.Bitmap);
                Image2.CompressionQuality := k;
                Image2.SaveToFile(FileName);
            end;
            if not JPGBild then begin
                Image1.Picture.Bitmap.SaveToFile(FileName);
            end;
            Label1.Caption := FName;
        except
            MessageBox(0,'Daten-Fehler','Problem',16);
        end;
    end;
end;
end;
end;

procedure PrintImage(I:TImage; f: Real);
{ Imageausdruck mit Skalierungsfaktor f bezogen auf die ganze Druckseite }
var w,h : Integer;
    k : Real;
    Bereich : TRect;
begin
    k := I.Height / I.Width;
    Printer.BeginDoc;
    w := Round(Printer.PageWidth * f);
    h := Round(w * k);
    if h > Printer.PageHeight then begin
        k := 1/k;
        h := Round(Printer.PageHeight * f);
        w := Round(h * k);
    end;
    Bereich := Rect(0,0,w,h);
    Printer.Canvas.StretchDraw(Bereich,I.Picture.Graphic);
    Printer.EndDoc;
end;

```

```

procedure ImageInfo;
// Bildabmessungen anzeigen
begin
  Form1.Label2.Caption := IntToStr(Form1.Image1.Width) + ' x ' +
    IntToStr(Form1.Image1.Height);
  if Fit then Form1.Label3.Caption := 'Not True Size';
  if Not Fit then Form1.Label3.Caption := 'True Size';
end;

procedure TForm1.FormActivate(Sender: TObject);
// Initialisieren
begin
  FitForm(Form1);
  Color := RGB(120,140,160);
  Image2 := TJPEGImage.Create;
  Image2.PixelFormat := jf24bit;
  Image3 := TImage.Create(Form1);
  Image3.Picture.Bitmap.PixelFormat := pf32bit;
  Image1.Picture.Bitmap.PixelFormat := pf32bit;
  GetDir(0,Verz);
  Xmax := Image1.Width;
  Ymax := Image1.Height;
  InsertFlag := False;
  Fit := False;
  Pict := False;
  MarkFlag := False;
  A.X := 0; A.Y := 0;
  B.X := 0; B.Y := 0;
  Button13.SetFocus;
end;

procedure TForm1.FormKeyUp(Sender: TObject; var Key: Word;
Shift: TShiftState);
// Hilfetext ein- oder ausblenden (F4)
// Bild in Zwischenablage kopieren bzw. einfügen (F1 bzw. F2)
// Farbe des aktuellen bildpunktes holen (F3)
begin
  if key = vk_F4 then Memo1.Visible := NOT Memo1.Visible;
  if NOT Pict then Exit;
  if Key = vk_F1 then begin
    Clipboard.Assign(Image1.Picture.Bitmap);
  end;
  if key = vk_F2 then begin
    try
      Image1.Picture.Bitmap.Assign(Clipboard);
      Fit := False;
      FitImage(Image1,Xmax,Ymax,1,Fit);
      Image1.AutoSize := False;
      Image1.Stretch := True;
    except
      ShowMessage('Kein Bild in der Zwischenablage');
    end;
  end;
  if key = vk_F3 then Image1.Canvas.Pen.Color := Image1.Canvas.Pixels[P.X,P.Y];
  Application.ProcessMessages;
  ImageInfo;
end;

procedure TForm1.Image1MouseDown(Sender: TObject; Button: TMouseButton;
Shift: TShiftState; X, Y: Integer);
// Mit der linken Maustaste den Grafikkursor bewegen
// und alten Markierungsrahmen löschen
// und gepuffertes Bild einfügen
// und einen Bildbereich mit Farbe füllen.
// Mit der rechten Maustaste einen Markierungsrahmen setzen.
var S,T: String;
begin
  if Fit then Exit;
  if Button = mbLeft then begin
    Image1.Canvas.MoveTo(X,Y);
    if MarkFlag then begin // Alten Markierungsrahmen löschen
      MarkFrame(Image1,A.X,A.Y,B.X,B.Y);
      MarkFlag := False;
    end;
    A.X := X;
    A.Y := Y;
  end;
end;

```

```

    if InsertFlag then begin                                // Gepuffertes Bild einfügen
        InsertFlag := False;
        PasteImage(Image3,Image1,X,Y);
        Image1.AutoSize := False;
        Image1.Stretch := True;
        ImageInfo;
    end;
    if FillFlag then begin                                // Bildbereich mit Farbe füllen
        FillFlag := False;
        Image1.Canvas.Brush.Color := FillColor;
        Image1.Canvas.Brush.Style := bsSolid;
        Image1.Canvas.FloodFill(P.X,P.Y,Image1.Canvas.Pen.Color,fsBorder);
        Image1.Canvas.Brush.Style := bsClear;
    end;
    end;                                                // Neuen Markierungsrahmen zeichnen
    if Button = mbRight then begin
        if MarkFlag then MarkFrame(Image1,A.X,A.Y,B.X,B.Y);
        B.X := X;
        B.Y := Y;
        MarkFrame(Image1,A.X,A.Y,B.X,B.Y);
        MarkFlag := True;
    end;
end;

procedure TForm1.Image1MouseMove(Sender: TObject; Shift: TShiftState;
                                X,Y: Integer);
// Mit der linken Maustaste ein Linie zur letzten Cursor-Position ziehen
begin
    if Fit then Exit;
    Label15.Caption := IntToStr(X) + ' / ' + IntToStr(Y);
    P.X := X;
    P.Y := Y;
    if Shift = [ssLeft] then Image1.Canvas.LineTo(X,Y);
end;

procedure TForm1.Button1Click(Sender: TObject);
// Bild laden
begin
    if MarkFlag then begin
        MarkFrame(Image1,A.X,A.Y,B.X,B.Y);
        MarkFlag := False;
    end;
    LoadImage;
    ImageInfo;
end;

procedure TForm1.Button2Click(Sender: TObject);
// Bild speichern
begin
    if NOT Pict then Exit;
    if MarkFlag then begin
        MarkFrame(Image1,A.X,A.Y,B.X,B.Y);
        MarkFlag := False;
    end;
    SaveImage;
end;

procedure TForm1.Button3Click(Sender: TObject);
// Markierten Bildbereich in Puffer kopieren
var Transp: Boolean;
begin
    if NOT Pict then Exit;
    if MarkFlag then begin
        MarkFrame(Image1,A.X,A.Y,B.X,B.Y);
        MarkFlag := False;
        if (B.X < A.X) or (B.Y < A.Y) then begin
            ShowMessage('Markierungsrichtung falsch !');
            Exit;
        end;
        if MessageBox(0,'Kopieren mit schwarzer Hintergrundtransparenz ?',
            'Frage',36) = 6 then Transp := True
            else Transp := False;
        CopyImage(Image1,Image3,A.X,A.Y,B.X,B.Y,Transp);
    end
    else begin
        ShowMessage('Bitte zuerst einen Bereich markieren !');
    end;
end;

```

```
procedure TForm1.Button4Click(Sender: TObject);
// Bildeinfügung aus Puffer ermöglichen
begin
  if NOT Pict then Exit;
  if MarkFlag then begin
    MarkFrame(Image1,A.X,A.Y,B.X,B.Y);
    MarkFlag := False;
  end;
  InsertFlag := True;
end;

procedure TForm1.Button5Click(Sender: TObject);
// Drucker-SetUp
begin
  if NOT Pict then Exit;
  PrinterSetupDialog1.Execute;
end;

procedure TForm1.Button6Click(Sender: TObject);
// Nur Image mit Skalierungsfaktor f ausdrucken
var s : String;
    f : Real;
    Code : Integer;
begin
  if NOT Pict then Exit;
  if MarkFlag then begin
    MarkFrame(Image1,A.X,A.Y,B.X,B.Y);
    MarkFlag := False;
  end;
  if Memo1.Visible then begin
    PrintMemo(Memo1);
    Exit;
  end;
  s := InputBox('Skalierter Bildausdruck','1.00 = ganze Druckseite','0.50');
  val(s,f,Code);
  if (Code<>0) then f := 1;
  PrintImage(Image1,f);
end;

procedure TForm1.Button7Click(Sender: TObject);
// Bildanpassung Ein/Aus-Schalten
begin
  if NOT Pict then Exit;
  if MarkFlag then begin
    MarkFrame(Image1,A.X,A.Y,B.X,B.Y);
    MarkFlag := False;
  end;
  Fit := NOT Fit;
  FitImage(Image1,Xmax,Ymax,1,Fit);
  ImageInfo;
end;

procedure TForm1.Button8Click(Sender: TObject);
// Bild auf markierten Bereich zuschneiden
begin
  if NOT Pict then Exit;
  if MarkFlag then begin
    MarkFrame(Image1,A.X,A.Y,B.X,B.Y);
    MarkFlag := False;
    CropImage(Image1,Image3,A.X,A.Y,B.X,B.Y);
    ImageInfo;
  end
  else begin
    ShowMessage('Bitte zuerst einen Bereich markieren !');
  end;
end;

procedure TForm1.Button9Click(Sender: TObject);
// Pinselfarbe
begin
  if NOT Pict then Exit;
  if Fit then begin
    ShowMessage('Bitte "Anpassen" ausschalten !');
    Exit;
  end;
  if ColorDialog1.Execute then
    Image1.Canvas.Pen.Color := ColorDialog1.Color;
end;
```

```

procedure TForm1.Button10Click(Sender: TObject);
// Pinselstärke
var s : String;
    p,Code : Integer;
begin
  if NOT Pict then Exit;
  if Fit then begin
    ShowMessage('Bitte "Anpassen" ausschalten !');
    Exit;
  end;
  s := InputBox('Pinselstärke zum Freihand-Zeichnen','1 ... 100','1');
  val(s,p,Code);
  if (Code<>0) or (p<1) or (p>100) then p := 1;
  Image1.Canvas.Pen.Width := p;
end;

procedure TForm1.Button11Click(Sender: TObject);
// Programm beenden
begin
  Application.Terminate;
end;

procedure TForm1.Button12Click(Sender: TObject);
// Bildgröße verändern
var S: String;
    f: Real;
    Code: Integer;
begin
  if NOT Pict then Exit;
  if MarkFlag then begin
    MarkFrame(Image1,A.X,A.Y,B.X,B.Y);
    MarkFlag := False;
  end;
  S := InputBox('Veränderung der Bildgröße',' 0.10 bis 10.00 ','1');
  Val(S,f,Code);
  if (Code <> 0) or (f < 0.10 ) or (f > 10.00) then f := 1;
  Fit := False;
  FitImage(Image1,Xmax,Ymax,f,Fit);
  StretchImage(Image1,Image3,f);
  ImageInfo;
end;

procedure TForm1.Button13Click(Sender: TObject);
// Neue Grafik erzeugen
var C,D,S: String;
    W,H,FA,P,Code: Integer;
    F: TColor;
    OK: Boolean;
begin
  if MarkFlag then begin
    MarkFrame(Image1,A.X,A.Y,B.X,B.Y);
    MarkFlag := False;
  end;

  S := InputBox('Bildfarbe (0,1), Breite, Höhe','Eingabe','1,600,600');
  S := Trim(S);
  if S = '' then Exit;
  P := Pos(', ',S);
  if (S = '') or (P = 0) then Exit;
  OK := False;
  C := Trim(Copy(S,1,P-1));
  S := Trim(Copy(S,P+1,Length(S)));
  val(C,FA,Code);

  if (Code = 0) and ((FA = 0) or (FA = 1)) then begin
    P := Pos(', ',S);
    if NOT (S = '') or (P = 0) then begin
      D := Trim(Copy(S,1,P-1));
      S := Trim(Copy(S,P+1,Length(S)));
      val(D,W,Code);
      if Code = 0 then begin
        val(S,H,Code);
        if Code = 0 then OK := True;
      end;
    end;
  end;
end;

```

```
if NOT OK then begin
    ShowMessage('Eingabefehler !');
    Exit;
end;
if FA = 0 then F := clBlack;
if FA = 1 then F := clWhite;
Image1.Canvas.Pen.Width := 1;
NewImage(Form1.Image1,F,W,H);
if FA = 0 then Image1.Canvas.Pen.Color := clWhite;
if FA = 1 then Image1.Canvas.Pen.Color := clBlack;
ImageInfo;
Pict := True;
end;

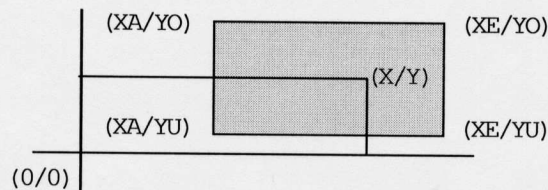
procedure TForm1.Button14Click(Sender: TObject);
// Füllfarbe auswählen und Farbfüllung ermöglichen
begin
    if NOT Pict then Exit;
    if MarkFlag then begin
        MarkFrame(Image1,A.X,A.Y,B.X,B.Y);
        MarkFlag := False;
    end;
    if ColorDialog1.Execute then FillColor := ColorDialog1.Color;
    FillFlag := True;
end;

end.
```

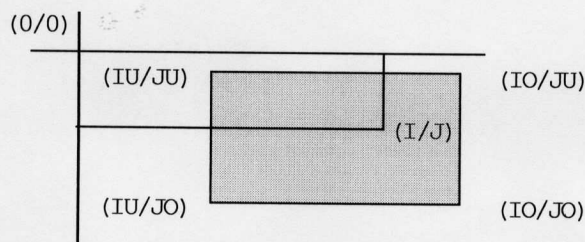
### [03] Koordinatentransformation von Welt zu Bild

Im Programm "*trans1*" soll ein Weltbereich auf einen Bildbereich abgebildet werden. Der Weltbereich ist durch das Rechteck mit der Diagonale  $(X_A/Y_O)-(X_E,Y_U)$  begrenzt. Der Bildbereich ist durch das Rechteck mit der Diagonale  $(I_U/J_U)-(I_O,J_O)$  begrenzt.

- (1) Der Weltbereich mit Diagonale  $(X_A/Y_O)-(X_E,Y_U)$   
Ein Weltpunkt  $(X/Y)$



- (2) Das Bildfenster mit Diagonale  $(I_U/J_U)-(I_O,J_O)$   
Ein Bildpunkt  $(I/J)$



Es seien  $DX = X_E - X_A$ ,  $DY = Y_O - Y_U$ ,  $DI = I_O - I_U$ ,  $DJ = J_O - J_U$

Dann gelten folgende einfache proportionale Beziehungen:

$$\begin{aligned} \text{(I)} \quad & (I - I_U) : +(X - X_A) = DI : DX \\ \text{(II)} \quad & (J - J_O) : -(Y - Y_U) = DJ : DY \end{aligned}$$

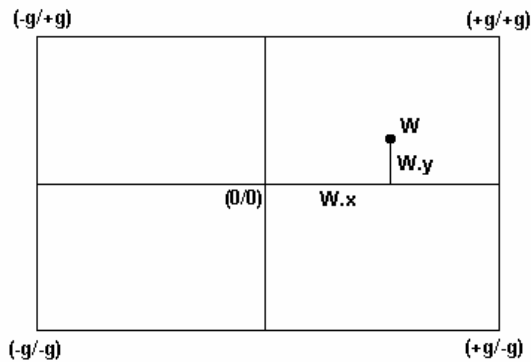
Durch Umformung bekommt man folgende Transformationsformeln:

$$\begin{aligned} \text{(I)} \quad & I = +(X - X_A) * DI/DX + I_U \\ \text{(II)} \quad & J = -(Y - Y_U) * DJ/DY + J_O \end{aligned}$$

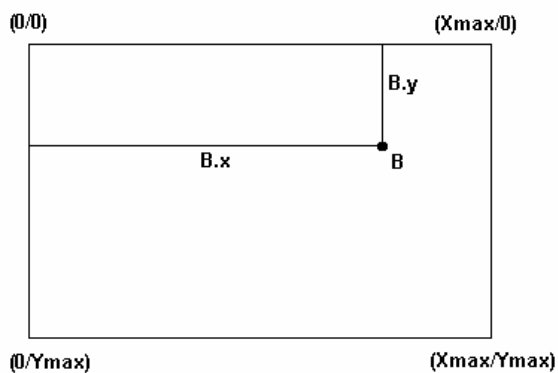
Die Abmessungen in der Welt sind reelle Zahlen, während die Abmessungen im Bildbereich ganzzahlige Werte sind, welche die Anzahlen von Bildpunkten darstellen. Der Ursprung des Bildbereiches entspricht der linken oberen Ecke einer Image-Komponente.

Der Sachverhalt vereinfacht sich wesentlich, wenn man als Weltbereich ein Quadrat mit der halben Seitenlänge  $g$  nimmt, also mit der Diagonale  $(-g/+g) - (+g/-g)$  begrenzt. Der Bildbereich ist durch die eingestellten Abmessungen der Image-Komponente  $(0,0,X_{max},Y_{max})$  vorgegeben. Dabei ist  $X_{max}$  die Bildbreite (Image.Width) und  $Y_{max}$  die Bildhöhe (Image.Height).

Ein Weltpunkt  $W$  hat die Weltkoordinaten  $W.x, W.y$  und der zugehörige Bildpunkt  $B$  hat die Bildkoordinaten  $B.x, B.y$ . Dann gelten geometrische Verhältnisse, welche die folgende Abbildung veranschaulichen soll.

**Weltbereich(-g,+g,+g,-g)**

Der Koordinatenursprung (0/0) im Weltbereich liegt genau in der Mitte. Die positive X-Achse zeigt nach rechts und die positive Y-Achse hinauf. Die Diagonale geht von Punkt (-g/+g) nach (+g/-g).

**Bildbereich(0,0,Xmax,Ymax)**

Der Koordinatenursprung (0/0) im Bildbereich liegt in der linken, oberen Ecke. Die positive X-Achse zeigt nach rechts und die positive Y-Achse zeigt hinunter. Die Diagonale geht von Punkt (0/0) nach (Xmax/Ymax).

Ein Weltpunkt *W* ist durch den Record-Typ *WPunkt* definiert. Ein Bildpunkt *B* ist durch den im System vordefinierten Record-Typ *TPoint* deklariert.

```

type WPunkt = Record           type TPoint = Record           type BPunkt = TPoint;
  X : Real;                    X : Integer;
  Y : Real;                    Y : Integer;
end;                           end;

```

Die Weltbreite  $2g$  entspricht dabei der Bildbreite  $Xmax$  und die Welthöhe  $2g$  entspricht der Bildhöhe  $Ymax$ . Wie aus der obigen Abbildung ersichtlich ist, bestehen zwischen den Weltkoordinaten ( $W.x, W.y$ ) und den Bildkoordinaten ( $B.x, B.y$ ) eines Punktes sehr einfache, direkt proportionale Beziehungen:

$$\begin{aligned} (W.x + g) : 2g &= B.x : Xmax \\ (W.y + g) : 2g &= (Ymax - B.y) : Ymax \end{aligned}$$

Durch Umformung bekommt man schließlich folgende Transformationsformeln:

$$\begin{aligned} B.x &:= \text{Round}((W.x + g) * Xmax / (2 * g)); \\ B.y &:= \text{Round}(Ymax - (W.y + g) * Ymax / (2 * g)); \end{aligned}$$

Die Prozedur *WeltZuBild* berechnet zu gegebenen Weltkoordinaten ( $W.x, W.y$ ) die zugehörigen Bildkoordinaten ( $B.x, B.y$ ). Die Prozedur *BildZuWelt* leistet genau das Umgekehrte.

```
procedure WeltZuBild(W: WPunkt; var B: BPunkt);
{ Rechnet die übergebenen Weltkoordinaten in Bildkoordinaten um }
begin
  B.x := Round((W.x + g) * Xmax / (2*g));
  B.y := Round(Ymax - (W.y + g) * Ymax / (2*g));
end;
```

```
procedure BildZuWelt(B: BPunkt; var W: WPunkt);
{ Rechnet die übergebenen Bildkoordinaten in Weltkoordinaten um }
begin
  W.x := (2*g*B.x - g*Xmax) / Xmax;
  W.y := (-2*g*B.y + g*Ymax) / Ymax;
end;
```

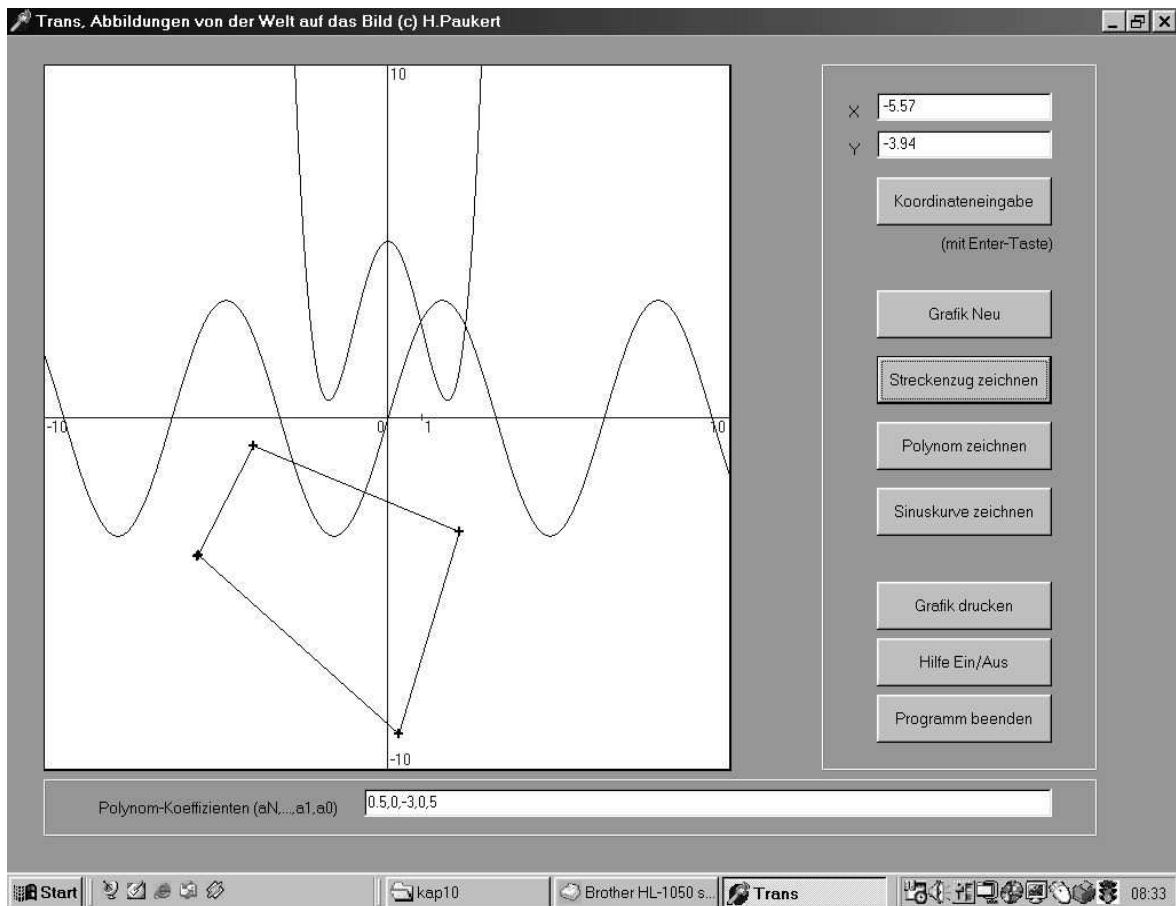
Mit Hilfe eines Schaltknopfs können die Koordinaten eines Punktes im Weltbereich eingegeben werden, worauf eine Umrechnung in Bildkoordinaten und eine Markierung des zugehörigen Bildpunktes erfolgt. Jede Koordinateneingabe muss mit der <Enter>-Taste abgeschlossen werden. Ein linker Mausklick führt zu einer Umrechnung der aktuellen Bildkoordinaten (der Mausposition im Bildbereich) in Weltkoordinaten und deren Anzeige. Mit einem rechten Mausklick wird der aktuelle Bildpunkt eingezeichnet und seine Koordinaten werden in einen Punktspeicher fortlaufend abgelegt. Über einen Schaltknopf können dann die abgespeicherten Punkte durch einen Streckenzug (Polygon) verbunden werden. Schließlich besteht noch die Möglichkeit, zwei zusätzliche Routinen aufzurufen. In der ersten werden mathematische Polynomfunktionen grafisch dargestellt, in der zweiten eine einfache Sinus-Schwingung. Über den Schaltknopf <Grafik Neu> wird das Bild gelöscht und mit einem Koordinatenraster neu aufgebaut. Dabei kann die Größe  $g$  des abgebildeten Weltbereichs individuell festgelegt werden.

Als Besonderheit bietet das Programm die Möglichkeit, eine Polynomfunktionen aus ihrem Weltbereich in den Bildbereich zu transformieren und dort zu zeichnen. Dabei werden zuerst die Koeffizienten, durch Kommas getrennt, in einem Editfeld eingegeben. Die Prozedur *ExtractValues* transferiert diese Koeffizienten in ein Zahlenarray *Koeff* und die Funktion *Poly* berechnet den Wert der Polynomfunktion an der Argumentstelle  $X$ . Die Funktion *Fun(Wahl, X)* schließlich entscheidet mittels Parameter *Wahl*, ob die Polynomfunktion ( $Wahl = 1$ ) oder die Sinusfunktion ( $Wahl = 2$ ) an der Argumentstelle  $X$  berechnet werden soll.

Die grafische Darstellung einer Funktion erfolgt dann in der Prozedur *KurveZeichnen(Wahl)*. Als lokale Variable werden *Anfa*, *Ende*, *Diff* und *A* definiert. *Anfa* ist der erste  $X$ -Wert der Funktionsberechnung und *Ende* der letzte. In diesem Intervall [*Anfa*, *Ende*] wird der  $X$ -Wert nun in einer Wiederholungsschleife um den Betrag *Diff* erhöht. *Diff* wird dadurch bestimmt, dass die Intervalllänge durch die Anzahl der horizontalen Pixel *Xmax* dividiert wird,  $Diff := (Ende - Anfa) / Xmax$ . Bei jedem Schritt muss zum  $X$ -Wert  $W.x$  eines Weltpunktes  $W$  der zugehörige Funktionswert  $W.y := Fun(Wahl, W.x)$  berechnet werden. Sodann werden die Weltkoordinaten des Punktes mit Hilfe der Prozedur *WeltZuBild* in die entsprechenden Bildkoordinaten transformiert. Das liefert zum Weltpunkt  $W$  den Bildpunkt  $B$ . Dieser muss vor jedem Schritt in den Hilfspunkt  $A$  umgespeichert werden. Beim Zeichnen der Funktion wird zunächst der Grafikkursor zu  $A$  bewegt, dann der neue Bildpunkt  $B$  ermittelt und eine Linie von  $A$  zu  $B$  gezeichnet. Danach wird wieder  $B$  in den Hilfspunkt  $A$  umgespeichert. Das Ganze wird so lange fortgesetzt, bis der Endpunkt des Intervalls erreicht ist, also  $W.x \geq Ende$ . Um störende vertikale Linien zu vermeiden, wird keine Verbindungsstrecke zu einem Punkt gezeichnet, falls seine  $y$ -Koordinate außerhalb des Weltbereichs liegt, d.h.  $abs(W.y) > g$  ist.

Das beschriebene Verfahren lässt sich zur grafischen Darstellung beliebiger mathematischer Funktionen verwenden. Dabei ist nur zu beachten, dass etwaige undefinierte Stellen (negative Zahlen unter der Wurzel, Divisionen durch Null) abgefangen werden müssen, was bei einfachen Polynomfunktionen aber nicht nötig ist.

Im Folgenden wird das Formular abgebildet und das Programm "*trans1*" aufgelistet.



### **unit trans1\_u;**

*// Trans1, Koordinaten-Transformation von Welt auf Bild (c) H.Paukert*

```
interface
uses Windows, Messages, SysUtils, Classes, Graphics,
    Controls, Forms, Dialogs, StdCtrls, ExtCtrls;
type
  TForm1 = class(TForm)
    Image1: TImage;
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    Label4: TLabel;
    Edit1: TEdit;
    Edit2: TEdit;
    Edit3: TEdit;
    Memo1: TMemo;
    Button1: TButton;
    Button2: TButton;
    Button3: TButton;
    Button4: TButton;
    Button5: TButton;
    Button6: TButton;
    Button7: TButton;
    Button8: TButton;
    Bevel1: TBevel;
    Bevel2: TBevel;
    procedure FormActivate(Sender: TObject);
    procedure Edit1KeyPress(Sender: TObject; var Key: Char);
    procedure Edit2KeyPress(Sender: TObject; var Key: Char);
    procedure Image1MouseDown(Sender: TObject; Button: TMouseButton;
      Shift: TShiftState; X, Y: Integer);
  end;
end;
```

```

    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
    procedure Button3Click(Sender: TObject);
    procedure Button4Click(Sender: TObject);
    procedure Button5Click(Sender: TObject);
    procedure Button6Click(Sender: TObject);
    procedure Button7Click(Sender: TObject);
    procedure Button8Click(Sender: TObject);
    private { Private declarations }
    public { Public declarations }
end;

var Form1: TForm1;

implementation
{$R *.DFM}

type WPunkt = record // Selbst definierter Typ
    X : Real; // eines Weltpunktes
    Y : Real;
end;

    BPunkt = TPoint; // Vordefinierter Bildpunkt-Typ

const g : Integer = 10; // Halbe Breite des Weltbereiches
    Anz : Integer = 0; // Aktuelle Anzahl an Punkten
    Max = 100; // Höchstanzahl an Punkten

type Feld = array[1..Max] of BPunkt; // Punktespeicher
    Koef = array[0..Max] of Real; // Koeffizientenspeicher

var P : Feld; // Fortlaufender Punktespeicher
    K : Koef; // Koeffizientenspeicher
    M : BPunkt; // Koordinatenursprung
    B : BPunkt; // Ein Bildpunkt
    W : WPunkt; // Ein Weltpunkt
    Xmax : Integer; // Bildbreite (Image.Width)
    Ymax : Integer; // Bildhöhe (Image.Height)

procedure FitForm(F :TForm);
{ Anpassung des Formulares an die jeweilige Monitorauflösung }
begin
    with F do begin
        if (Screen.Width<>1024) then ScaleBy(Screen.Width,1024);
        if (Font.PixelsPerInch<>120) then ScaleBy(120,Font.PixelsPerInch);
        WindowState := wsMaximized;
    end;
end;

procedure ExtractValues(S: String; SEP: Char; var K: Koef);
{ Kommaseparierte Zahlen aus einen String S extrahieren und in das Array K speichern. }
{ Die Anzahl der extrahierten Dezimalzahlen steht dann in der Zelle K[0]. }
var T : String;
    Z : Real;
    N,P,Code : Integer;
    Error : Boolean;
begin
    if S[Length(S)] <> SEP then S := S + SEP;
    Error := False;
    N := 0;
    Repeat
        P := Pos(SEP,S);
        if P > 0 then begin
            N := N + 1;
            T := Trim(Copy(S,1,P-1));
            Val(T,Z,Code);
            K[N] := Z;
            if Code <> 0 then Error := TRUE;
            S := Copy(S,P+1,Length(S));
        end;
    Until P = 0;
    if Error then K[0] := 0
        else K[0] := N;
end;

```

```

procedure WeltZuBild(W: WPunkt; var B: BPunkt);
{ Rechnet die übergebenen Weltkoordinaten in }
{ Bildkoordinaten um. }
begin
  B.X := Round((W.X + g) * Xmax / (2*g));
  B.Y := Round(Ymax - (W.Y + g) * Ymax / (2*g));
end;

procedure BildZuWelt(B: BPunkt; var W: WPunkt);
{ Rechnet die übergebenen Bildkoordinaten }
{ in Weltkoordinaten um. }
begin
  W.X := (2*g*B.X - g*Xmax) / Xmax;
  W.Y := (-2*g*B.Y + g*Ymax) / Ymax;
end;

function Poly(K:Koeff; x:Real): Real;
{ Polynomfunktion (n-1)-ten Grades mit dem Argument x }
{ und den n Koeffizienten K[1],...,K[n] }
{ und der Koeffizienten-Anzahl n in K[0] }
var n,i : Integer;
    sum : Real;
begin
  n := Round(K[0]);
  sum := 0;
  for i := 1 To n do sum := sum * x + K[i];
  Result := sum;
end;

procedure PunkteZuNull;
{ Setzt alle gespeicherten Punktkoordinaten auf Null }
var i : Integer;
begin
  Anz := 0;
  for i := 1 to Max do begin
    P[i].X := 0;
    P[i].Y := 0;
  end;
end;

procedure KoordinatenAnzeigen(W: WPunkt);
{ Zeigt die Koordinaten eines Weltpunkte W an }
var s : String;
begin
  str(W.X:6:2,s); Form1.Edit1.Text := s;
  str(W.Y:6:2,s); Form1.Edit2.Text := s;
end;

procedure PunktZeichnen(B: BPunkt);
{ Zeichnet einen Punkt als kleines Kreuz}
begin
  with Form1.Image1.Canvas do begin
    Pen.Width := 2;
    MoveTo(B.X,B.Y);
    LineTo(B.X-3,B.Y); LineTo(B.X+3,B.Y);
    MoveTo(B.X,B.Y);
    LineTo(B.X,B.Y-3); LineTo(B.X,B.Y+3);
    Pen.Width := 1;
  end;
end;

procedure StreckenZeichnen(p: Feld; n: Integer);
{ Streckenzug durch die eingegebenen Punkte p zeichnen. }
var i : Integer;
begin
  if n > 1 then begin
    with Form1.Image1.Canvas do begin
      MoveTo(P[1].X,P[1].Y);
      For i := 1 to n do begin
        LineTo(P[i].X,P[i].Y);
        MoveTo(P[i].X,P[i].Y);
      end;
      LineTo(P[1].X,P[1].Y);
    end;
  end;
end;
end;

```

```
function Fun(Wahl: Integer; X: Real): Real;
{ Auswahl und Berechnung einer Funktion an der Stelle X }
{ Polynom-Funktion bei Wahl = 1, Sinus-Funktion bei Wahl = 2 }
begin
  case Wahl of
    1: Fun := Poly(K,X);
    2: Fun := g/3 * sin(X);
  end;
end;
```

```
procedure KurveZeichnen(Wahl: Integer);
{ Zeichnet das Schaubild einer Funktion }
var Anfa, Ende, Diff : Real;
    A,B : BPunkt;
    W : WPunkt;
begin
  with Form1.Image1.Canvas do begin
    Anfa := -g;
    Ende := g;
    Diff := (Ende-Anfa) / Xmax;
    W.X := Anfa;
    W.Y := Fun(Wahl,W.X);
    WeltZuBild(W,B);
    A := B;

    repeat
      W.X := W.X + Diff;
      W.Y := Fun(Wahl,W.X);
      WeltZuBild(W,B);
      MoveTo(A.X,A.Y);
      if (abs(W.Y)) > g then MoveTo(B.X,B.Y)
        else LineTo(B.X,B.Y);
      A := B;
    until W.X >= Ende;

  end;
end;
```

```
procedure KoordinatenAchsen(g: Integer);
{ Koordinaten-Achsen zeichnen }
var S0,S1,S2 : String;
    dx : Integer;
begin
  S0 := '0';
  S1 := ' ' + IntToStr(g) + ' ';
  S2 := ' ' + IntToStr(-g) + ' ';
  dx := Xmax div (2*g);

  with Form1.Image1.Canvas do begin
    Font.Color:= clBlue;
    Pen.Color := clBlack;
    Pen.Width := 2;
    Pen.Style := psSolid;
    Brush.Color := clWhite;
    Brush.Style := bsSolid;
    Rectangle(0,0,Xmax,Ymax);
    Brush.Style := bsClear;
    Pen.Color := clBlue;
    Pen.Width := 1;
    MoveTo(0,M.Y); LineTo(Xmax,M.Y);
    MoveTo(M.X,0); LineTo(M.X,Ymax);
    TextOut(M.X-TextWidth(S0)-2,M.Y,S0);
    TextOut(M.X,0,S1);
    TextOut(M.X,Ymax-TextHeight(S2)-2,S2);
    TextOut(0,M.Y,S2);
    TextOut(Xmax-TextWidth(S1)-2,M.Y,S1);
    MoveTo(M.X+dx,M.Y-3); LineTo(M.X+dx,M.Y+3);
    TextOut(M.X+dx,M.Y,' 1 ');
    Pen.Color := clBlack;
    Font.Color:= clBlack;
  end;
end;
```

```

procedure TForm1.FormActivate(Sender: TObject);
{ Initialisierungen }
begin
  FitForm(Form1);
  Color := RGB(140,160,150);
  PunkteZuNull;
  Edit1.Text := 'X-Koordinate';
  Edit2.Text := 'Y-Koordinate';
  Xmax := Image1.ClientWidth;
  Ymax := Image1.ClientHeight;
  M.X := Round(Xmax/2);
  M.Y := Round(Ymax/2);
  g := 10;
  KoordinatenAchsen(g);
  Button1.SetFocus;
end;

procedure TForm1.Edit1KeyPress(Sender: TObject; var Key: Char);
{ Eingabe der Weltkoordinate X eines Punktes }
{ Eingabe-Ende mit der <Enter>-Taste }
var s : String;
    Code : Integer;
begin
  s := Edit1.Text;
  if Key = #13 then Begin
    val(s,W.X,Code);
    Edit2.SetFocus;
  end;
end;

procedure TForm1.Edit2KeyPress(Sender: TObject; var Key: Char);
{ Eingabe der Weltkoordinate Y eines Punktes. }
{ Eingabe-Ende mit der <Enter>-Taste. }
{ Fortlaufende Abspeicherung der Bildpunkte. }
var s : String;
    Code : Integer;
begin
  s := Edit2.Text;
  if Key = #13 then begin
    val(s,W.Y,Code);
    KoordinatenAnzeigen(W);
    WeltZuBild(W,B);
    PunktZeichnen(B);
    if Anz > Max then PunkteZuNull;
    Anz := Anz + 1;
    P[Anz].X := B.X;
    P[Anz].Y := B.Y;
    Edit1.SetFocus;
  end;
end;

procedure TForm1.Image1MouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
{ Umrechnung des, mit der rechten Maustaste angeklickten Bildpunktes }
{ in Weltkoordinaten. Ausgabe und Abspeicherung dieser Weltkoodinaten. }
{ Außerdem wird der angeklickte Bildpunkt gezeichnet. }
{ Das Gleiche erfolgt mit der linken Maustaste - nur OHNE Abspeichern }
{ und Zeichnen des angeklickten Punktes. }
begin
  if Button = mbRight then begin
    B.X := X;
    B.Y := Y;
    PunktZeichnen(B);
    BildZuWelt(B,W);
    KoordinatenAnzeigen(W);
    if Anz >= Max then PunkteZuNull;
    Anz := Anz + 1;
    P[Anz].X := B.X;
    P[Anz].Y := B.Y;
  end;
  if Button = mbLeft then begin
    B.X := X;
    B.Y := Y;
    BildZuWelt(B,W);
    KoordinatenAnzeigen(W);
  end;
end;
end;

```

```
procedure TForm1.Button1Click(Sender: TObject);
{ Bereitet die Eingabe von Weltkoordinaten vor }
begin
  Edit1.Text := '';
  Edit2.Text := '';
  Edit1.SetFocus;
end;

procedure TForm1.Button2Click(Sender: TObject);
{ Initialisiert das Grafikbild }
var s : String;
    Code : Integer;
begin
  PunkteZuNull;
  Edit1.Text := 'X-Koordinate';
  Edit2.Text := 'Y-Koordinate';
  s := InputBox('', 'Eingabe der geraden halben Weltbreite 2 ... 100', '10');
  val(s,g,Code);
  if (g<2) or (g>100) or odd(g) then g := 10;
  KoordinatenAchsen(g);
end;

procedure TForm1.Button3Click(Sender: TObject);
{ Streckenzug durch die eingegebenen Punkte p zeichnen }
begin
  StreckenZeichnen(P,Anz);
  PunkteZuNull;
end;

procedure TForm1.Button4Click(Sender: TObject);
{ Koeffizienten für Polynom-Funktion übernehmen und zeichnen }
var s : String;
begin
  s := Edit3.Text;
  if Trim(s) = '' then begin
    ShowMessage('Bitte Polynomkoeffizienten eingeben!');
    Edit3.SetFocus;
    Exit;
  end;
  ExtractValues(s, ', ', 'K');
  KurveZeichnen(1);
end;

procedure TForm1.Button5Click(Sender: TObject);
{ Sinus-Funktion zeichnen }
begin
  KurveZeichnen(2);
end;

procedure TForm1.Button6Click(Sender: TObject);
{ Ganzes Formular ausdrucken. }
begin
  PrintScale := poProportional;
  Print;
end;

procedure TForm1.Button7Click(Sender: TObject);
{ Memobox mit Hilfsinformationen einblenden }
begin
  Memo1.Visible := NOT Memo1.Visible;
end;

procedure TForm1.Button8Click(Sender: TObject);
{ Programm beenden }
begin
  Application.Terminate;
end;

end.
```

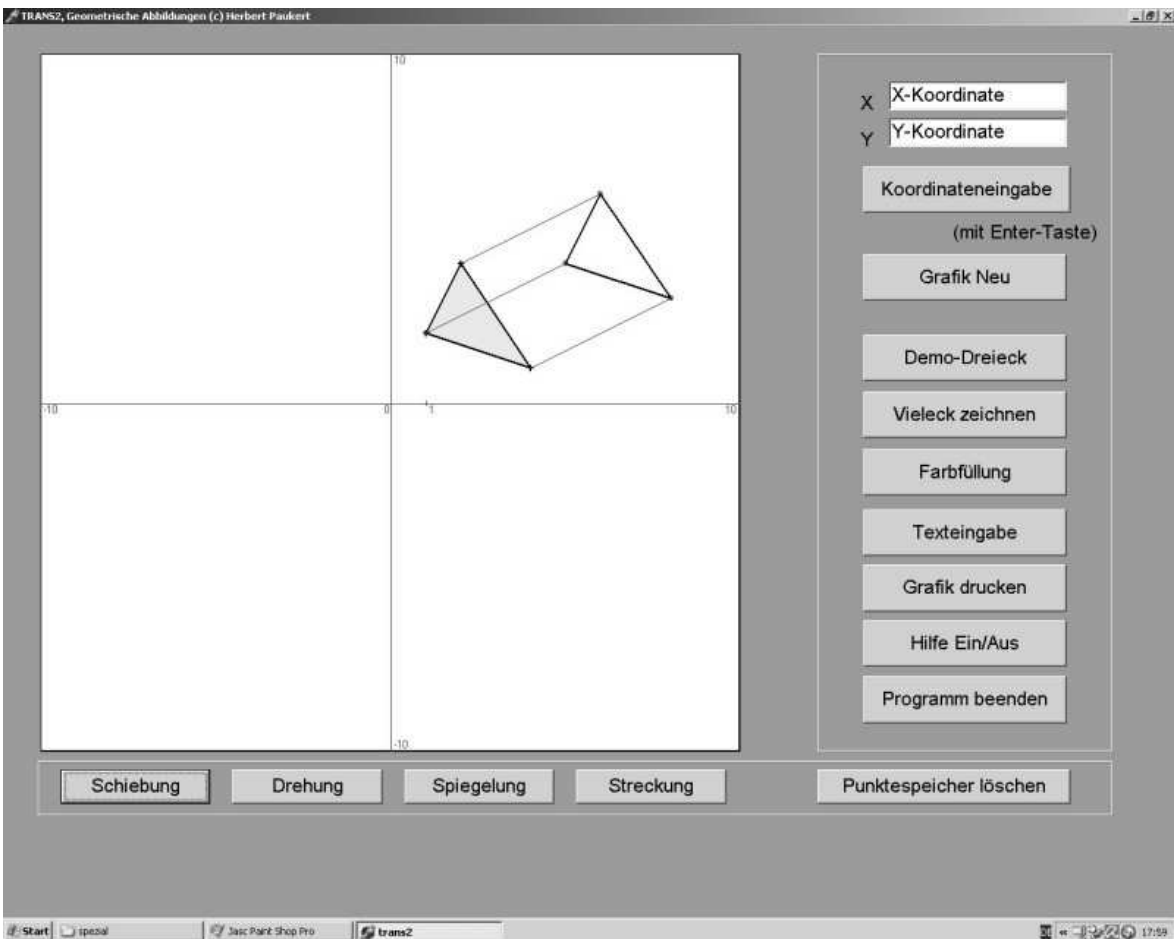
## [04] Einfache geometrische Abbildungen

Das Programm "*trans2*" ist grundsätzlich ähnlich aufgebaut wie das Programm "*trans1*". Folgende Änderungen gelten jedoch:

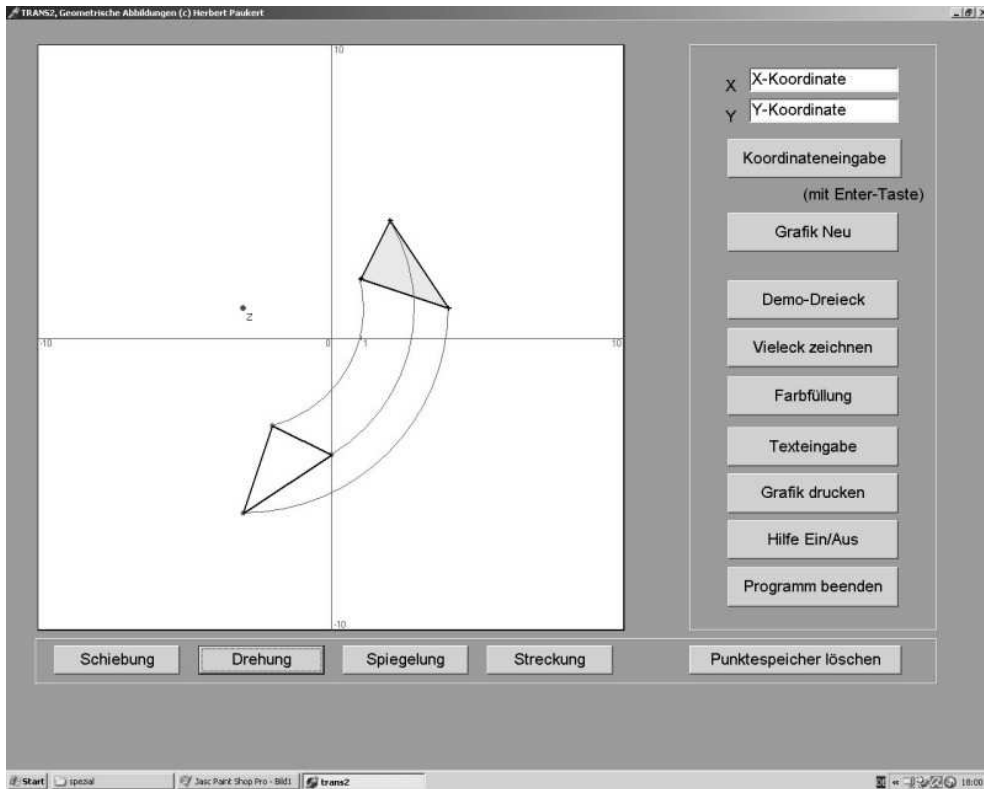
Erstens ist das Zeichnen von Polynomen und Sinuskurven hier nicht möglich.

Zweitens können auf die Punkte von Streckenzügen (Vielecken) einfache geometrische Abbildungen (Schiebung, Drehung, Spiegelung und Streckung) angewendet werden.

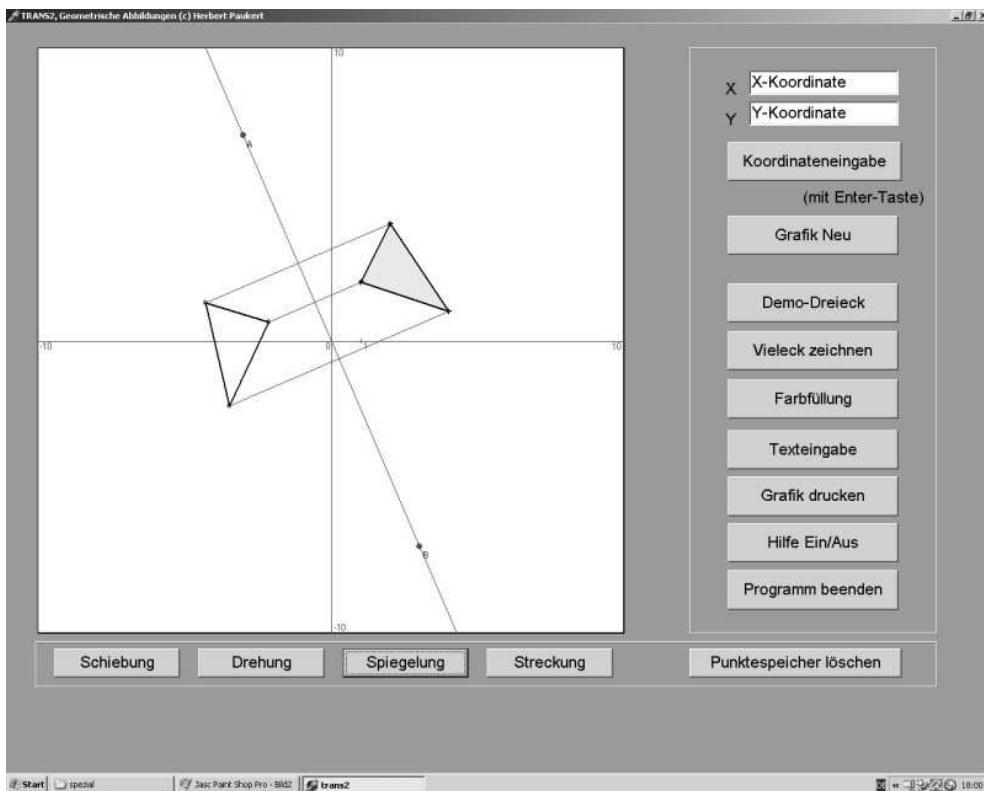
Die nachfolgenden Listings enthalten nur jene Unterprogramme, welche diese geometrischen Abbildungen realisieren. Die verwendeten mathematischen Transformations-Formeln ergeben sich aus der Vektorrechnung und der Trigonometrie. Wichtig ist dabei, dass die Systemunit *Math* in der *uses*-Anweisung eingebunden wird. Sie enthält den Programmcode der wichtigsten mathematischen Funktionen



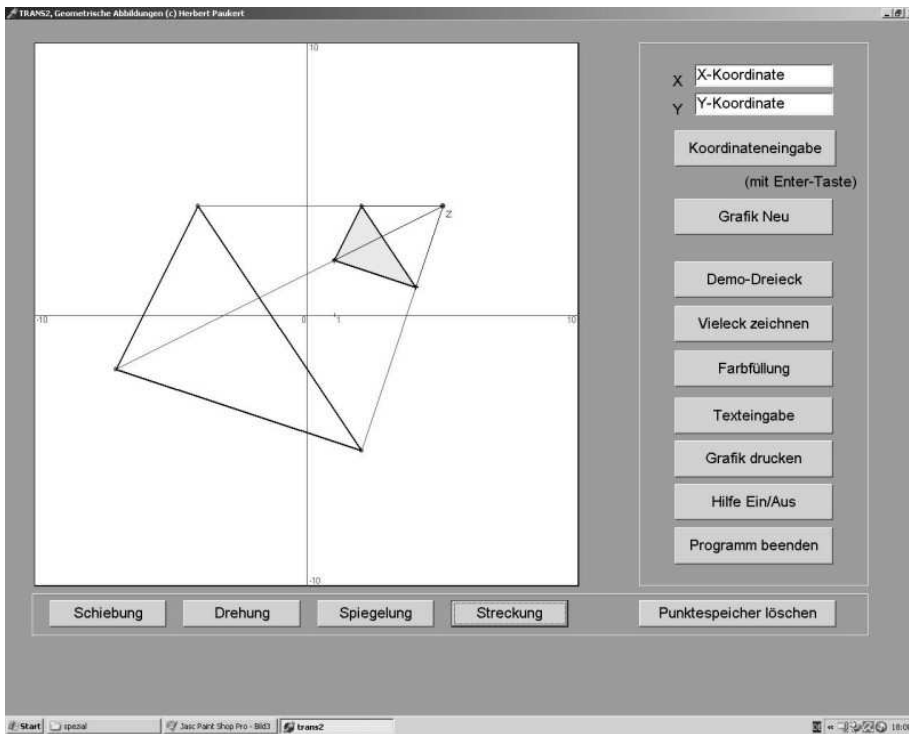
Oben stehende Grafik zeigt eine Schiebung des Dreiecks  $A(1/2)$ ,  $B(4/1)$ ,  $C(2/4)$  mit dem Schiebvektor  $V(4/2)$ .



Oben stehende Grafik zeigt eine Drehung des Dreiecks  $A(1/2)$ ,  $B(4/1)$ ,  $C(2/4)$  um das Zentrum  $Z(-3/1)$  um den Drehwinkel  $90^\circ$ .



Oben stehende Grafik zeigt eine Spiegelung des Dreiecks  $A(1/2)$ ,  $B(4/1)$ ,  $C(2/4)$  an der Achse, welche durch die Punkte  $P(-3/7)$  und  $Q(3,-7)$  geht.



Oben stehende Grafik zeigt eine Streckung des Dreiecks  $A(1/2)$ ,  $B(4/1)$ ,  $C(2/4)$  um das Zentrum  $Z(5/4)$  mit dem Streckungsfaktor 3.

```

type WPunkt = record
    X : Real;
    Y : Real;
end;
BPunkt = TPoint;
var g, Xmax, Ymax : Integer;
// Selbst definierter Typ
// eines Weltpunktes
// Vordefinierter Bildpunkt-Typ
// Abmessungen von Welt und Bild

procedure WeltZuBild(W: WPunkt; var B: BPunkt);
{ Rechnet die übergebenen Weltkoordinaten in }
{ Bildkoordinaten um; g ist die Halbbreite der Welt }
{ und Xmax und Ymax sind die Abmessungen des Bildes }
begin
    B.X := Round((W.X + g) * Xmax / (2*g));
    B.Y := Round(Ymax - (W.Y + g) * Ymax / (2*g));
end;

procedure BildZuWelt(B: BPunkt; var W: WPunkt);
{ Rechnet die übergebenen Bildkoordinaten }
{ in Weltkoordinaten um; g ist die Halbbreite der Welt }
{ und Xmax und Ymax sind die Abmessungen des Bildes }
begin
    W.X := (2*g*B.X - g*Xmax) / Xmax;
    W.Y := (-2*g*B.Y + g*Ymax) / Ymax;
end;

function Strecken(A,M: WPunkt; k: Real): WPunkt;
// Streckung von MA auf das k-Fache: MB = k * MA
var B: WPunkt;
begin
    B.X := M.X + k*(A.X - M.X);
    B.Y := M.Y + k*(A.Y - M.Y);
    Result := B;
end;

function Schieben(A,V: WPunkt): WPunkt;
// Schiebung von Punkt A um Vektor V: B = A + V
var B: WPunkt;
begin
    B.X := A.X + V.X;
    B.Y := A.Y + V.Y;
    Result := B;
end;

```

```

function Drehen(A,M: WPunkt; wi : Real): WPunkt;
// Drehen von Punkt A um M um Winkel wi, liefert Punkt B
var B,C,D : WPunkt;
begin
  wi := DegToRad(wi);
  C.X := A.X - M.X;
  C.Y := A.Y - M.Y;
  D.X := +C.X * cos(wi) + C.Y * sin(wi);
  D.Y := -C.X * sin(wi) + C.Y * cos(wi);
  B.X := D.X + M.X;
  B.Y := D.Y + M.Y;
  Result := B;
end;

function Spiegeln(A,B,C: WPunkt): WPunkt;
// Spiegelung von Punkt A an der Geraden durch B und C, liefert Punkt S
var k,d,u,xa,xg,ya,yg,f : Real;
var S : WPunkt;
begin
  k := (C.X - B.X);
  if k <> 0 then begin
    k := (C.Y - B.Y) / k;
    d := B.Y - k * B.X;
    ya := A.Y;
    yg := k*A.X + d;
    if ya > yg then f := +1 else f := -1;
    u := abs(k*A.X - A.Y + d) / (k*k + 1);
    S.X := A.X + 2*u*k*f;
    S.Y := A.Y - 2*u*f;
  end
  else begin
    xa := A.X;
    xg := B.X;
    if xa > xg then f := +1 else f := -1;
    u := abs(A.X - B.X);
    S.X := A.X - 2*u*f;
    S.Y := A.Y;
  end;
  Result := S;
end;

```

Die Prozedur **Gerade** zeichnet im Koordinatensystem eine Gerade durch die Punkte A und B.

```

procedure Gerade(A,B: WPunkt);
// Gerade durch A und B zeichnen
var k,d: Real;
    P,Q: WPunkt;
    P1,Q1: BPunkt;
begin
  if B.X = A.X then begin
    P.X := A.X;
    P.Y := -g;
    Q.X := A.X;
    Q.Y := -g;
  end
  else begin
    if B.y = A.y then begin
      P.x := -g;
      P.y := A.Y;
      Q.x := g;
      Q.y := A.Y;
    end
    else begin
      k := (B.Y-A.Y) / (B.X-A.X);
      d := A.Y - k * A.X;
      P.X := -(g+d)/k;
      P.Y := -g;
      Q.X := (g-d)/k;
      Q.Y := g;
    end;
  end;
  end;
  WeltZuBild(P,P1);
  WeltZuBild(Q,Q1);
  Form1.Image1.Canvas.MoveTo(P1.X,P1.Y);
  Form1.Image1.Canvas.LineTo(Q1.X,Q1.Y);
end;

```

Die Prozedur **Bogen** zeichnet einen Kreisbogen mit dem Mittelpunkt M, welcher durch die Punkte A und B verläuft.

```

procedure Bogen(M,A,B: WPunkt);
// Kreisbogen um M durch die Punkte A und B
var P,Q,C,D: WPunkt;
    A1,B1,P1,Q1,C1,D1: BPunkt;
    r,yA,yB,wA,wB: Real;
begin
  r := sqrt((A.X-M.X)*(A.X-M.X) + (A.Y-M.Y)*(A.Y-M.Y));
  if r = 0 then wA := 0 else wA := arccos((A.X-M.X)/r);
  if r = 0 then wB := 0 else wB := arccos((B.X-M.X)/r);

  yA := r;
  if A.Y < M.Y then ya := -r;
  C.y := M.y + yA;
  yB := r;
  if B.Y < M.Y then yB := -r;
  D.y := M.y + yB;

  if wA <> pi/2 then begin
    if wA <> 0 then C.x := M.x + r/tan(wA)
      else begin
        if A.x > M.x then C.x := 1000000
          else C.x := -1000000;
        end;
    end
  else C.x := M.x;

  if wB <> pi/2 then begin
    if wB <> 0 then D.x := M.x + r/tan(wB)
      else begin
        if B.x > M.x then D.x := 1000000
          else D.x := -1000000;
        end;
    end
  else D.x := M.x;

  P.x := M.x - r;
  P.y := M.y - r;
  Q.x := M.x + r;
  Q.y := M.y + r;

  WeltZuBild(P,P1);
  WeltZuBild(Q,Q1);
  WeltZuBild(C,C1);
  WeltZuBild(D,D1);

  Form1.Image1.Canvas.pen.Color := clBlue;
  Form1.Image1.Canvas.ARC(P1.X,P1.Y,Q1.X,Q1.Y,D1.X,D1.Y,C1.X,C1.Y);
  Form1.Image1.Canvas.pen.Color := clBlack;
end;

```

## [05] Sammlung von Grafikroutinen "gtools\_u"

Diese Sammlung enthält eine Vielzahl von verschiedenen wertvollen Grafikroutinen, wie sie beispielsweise in der Bildverarbeitung „*paupix*“ verwendet werden.

```

unit gtools_u;
// Graphic-Tools-Unit mit Grafik-Routinen (c) H. Paukert;

interface
uses Windows, Classes, Graphics, ExtCtrls, Printers;

procedure Rainbow(IM: TImage);
// Füllt ein Image mit Regenbogenfarben

procedure MarkFrame(I: TImage; x1,y1,x2,y2: Integer);
// Einen rechteckigen Markierungsrahmen zeichnen (mit Not Xor)

procedure PaintImage(I: TImage; F: TColor; X1,Y1,X2,Y2 : Integer);
// Einen rechteckigen Imagebereich mit Farbe füllen

procedure NewImage(I: TImage; F: TColor; IW,IH: Integer);
// Ein neues Image einrichten (mit Farbe, Breite und Höhe)

procedure FitImage(I: TImage; IW,IH: Integer; f: Real; Fit: Boolean);
// Ein Image verkleinern oder vergrößern (mit Streckungsfaktor f)

procedure PrintImage(I:TImage; f: Real);
// Ein Image skaliert ausdrucken (mit Skalierungsfaktor f)

procedure PrintPicture(I:TImage; a,b,w,h:Integer);
// Imageausdruck mit Versatz (a,b) und Druckgröße (w,h)

procedure ChangeImage(I: TImage; Nega,Grau,Kont,R,G,B : Integer);
// Farbnegativ (0,1), Graustufen (0,1), Kontrast (0..255)
// Farbwerte: Rot (0..255), Grün (0..255), Blau (0..255)

function MirrorImage(BM: TBitMap; Horiz: Boolean): TBitMap;
// Ein Image-Bitmap spiegeln (horizontal oder vertikal)

procedure RotateImageRight(Quelle: TImage);
// Ein Image-Bitmap um 90° nach rechts drehen

procedure RotateImageLeft(Quelle: TImage);
// Ein Image-Bitmap um 90° nach links drehen

procedure DistortImage(Quelle,Puffer: TImage; k: Real; Horiz: Boolean);
// Ein Image mit Hilfe eines Puffers verzerren (mit Verzerrungsfaktor k)

procedure StretchImage(Quelle,Puffer: TImage; k: Real);
// Ein Image in einen Puffer unverzerrt speichern (mit Streckungsfaktor k)

procedure CopyImage(Quelle,Puffer:TImage; x1,y1,x2,y2: Integer; HT: Boolean);
// Einen rechteckigen Bereich in einen Puffer unverzerrt speichern

procedure PasteImage(Puffer,Ziel: TImage; x,y: Integer);
// Einen gepufferten Bereich an einer Imageposition einfügen

```

```

procedure CropImage(var Quelle,Puffer: TImage; x1,y1,x2,y2: Integer);
// Ein Image auf einen rechteckigen Teil-Ausschnitt zuschneiden

procedure SmoothImage(IM: TImage; X1,Y1,X2,Y2: Integer);
// Einen rechteckigen Imagebereich weichzeichnen (Farbglättung)

procedure ImageBlob(Im0: TImage);
// Bildzerstörung mit angepassten Farbflecken

procedure ImageDecay(Im0: TImage);
// Bildzerstörung mit Random Image-Pixel

procedure ImageBlend(Im0,Im1: TImage; Mode: Integer);
// Bildüberblendung von Quelle (Im1) auf Ziel (Im0),
// mit Mode: 1 = ausgewogen, 2 = Ziel, 3,4 Quelle überwiegt

procedure ImageMove(Im0,Im1: TImage; Step: Integer);
// Bildüberblendung von Quelle (Im1) auf Ziel (Im0),
// mit Step als Geschwindigkeit, OHNE Transparenz

procedure ImageZoom(Im0,Im1,Im2: TImage; kmax: Real; n: Integer; Fit: Boolean);
// Zooming mit Im0 = Quelle, Im1 = erster Puffer, Im2 = zweiter Puffer
// und kmax = maximaler Zoomfaktor (%), n = Anzahl der Zoomschritte

procedure ImageSectionZoom(Im0,Im1,Im2: TImage; kmax: Real; n:Integer; PLO,PRU:
TPoint);
// Zooming mit Im0 = Quelle, Im1 = erster Puffer, Im2 = zweiter Puffer
// kmax = maximaler Zoomfaktor (%), n = Anzahl der Zoomschritte
// PLO und PUR sind die Eckpunkte des gezoomten Bildausschnittes

procedure ImageFish(Im0,Im1: TImage; FishSize,FishArea: Integer; PF: TPoint);
// Begrenzter Verzerrungs-Effekt, Im0 = Quelle, Im1 = Puffer für die Quelle

procedure ImageInsert(Im0,Im1,Im2: TImage; IB,IM,IX,IY: Integer);
// Einfügen von Im1 in Im0 mittels Im2, mit IB als Bildbreite
// an der Position (IX,IY) und mit IM als InsertModus:
// 0 = mit Schwarz als Transparenzfarbe, n = ohne Transparenz
// mit weißem Rahmen (+n) oder schwarzem Rahmen (-n)

procedure ImageFrame(Im0: TImage; IB,IM,IX1,IY1,IX2,IY2: Integer);
// Zeichnet im Bild Im0 einen rechteckigen Rahmen mit Breite IB
// und Farbe IM (1 = Schwarz, 2 = weiß) im Bereich (IX1,IY1,IX2,IY2)
// und bei IB < 0 wird der Bereich voll ausgefüllt

implementation

procedure Rainbow(IM: TImage);
// Füllt ein Image mit Regenbogenfarben
var i : Integer;
    d : Real;
    hb : HBrush;
    Rec : TRect;
begin
    IM.Picture.Bitmap.PixelFormat := pf24bit;
    d := IM.Width / (3*255);
    Rec.Top := 0;
    Rec.Bottom := IM.Height;

```

```
For i := 0 to 255 do begin
  hb := CreateSolidBrush(RGB(i,0,255-i));
  Rec.Left := Round(i*d);
  Rec.Right := Round((i+1)*d);
  FillRect(IM.Canvas.Handle,Rec,hb);
  DeleteObject(hb);
end;
For i := 0 to 255 do begin
  hb := CreateSolidBrush(RGB(255,i,0));
  Rec.Left := Round((i+255)*d);
  Rec.Right := Round((i+256)*d);
  FillRect(IM.Canvas.Handle,Rec,hb);
  DeleteObject(hb);
end;
For i := 0 to 255 do begin
  hb := CreateSolidBrush(RGB(255-i,255,0));
  Rec.Left := Round((i+510)*d);
  Rec.Right := Round((i+511)*d);
  FillRect(IM.Canvas.Handle,Rec,hb);
  DeleteObject(hb);
end;
end;

procedure MarkFrame(I: TImage; x1,y1,x2,y2: Integer);
// zieht einen Markierungsrahmen im NOT-XOR-Modus
var PCol : TColor;
    PWid : Integer;
begin
  with I.Canvas do begin
    PCol := Pen.Color;
    PWid := Pen.Width;
    Brush.Style := bsClear;
    Pen.Width := 1;
    Pen.Color := clBlack;
    Pen.Style := psDot;
    Pen.Mode := pmNotXor;
    Rectangle(x1,y1,x2,y2);
    Pen.Mode := pmCopy;
    Pen.Style := psSolid;
    Pen.Color := PCol;
    Pen.Width := PWid;
  end;
end;

procedure PaintImage(I: TImage; F: TColor; X1,Y1,X2,Y2 : Integer);
// Füllt ein Image-Rechteck mit einer Farbe
begin
  with I.Canvas do begin
    Pen.Color := F;
    Brush.Color := F;
    Brush.Style := bsSolid;
    FillRect(Rect(X1,Y1,X2,Y2));
    Brush.Style := bsClear;
  end;
end;
end;
```

```
procedure NewImage(I:TImage; F:TColor; IW,IH: Integer);
// Löscht das alte Image und initialisiert ein neues Image
// mit der Farbe F (0 = Schwarz, 1 = Weiß)
begin
  with I do begin
    Picture := NIL;
    AutoSize := False;
    Stretch := False;
    Width := IW;
    Height := IH;
    PaintImage(I,F,0,0,IW,IH);
    Picture.Bitmap.PixelFormat := pf24bit;
    AutoSize := True;
  end;
end;

procedure FitImage(I: TImage; IW,IH: Integer; f: Real; Fit: Boolean);
// Anpassen einer geladenen Grafikdatei in ein vorgegebenes Image
// mit den Abmessungen IW und IH. Zusätzlich wird noch ein Faktor f
// zur Vergrößerung oder Verkleinerung des Bildes eingegeben.
// Wenn Fit = TRUE, dann Anpassung - andernfalls keine Anpassung.
var w,h: Integer;
    k : Real;
begin
  I.Visible := False;
  if Fit then begin
    k := I.Height / I.Width;
    w := Round(IW * f);
    h := Round(w * k);
    if h > IH then begin
      k := 1/k;
      h := Round(IH * f);
      w := Round(h * k);
    end;
    I.AutoSize := False;
    I.Stretch := True;
    I.Width := w;
    I.Height := h;
  end;
  if not Fit then begin
    I.AutoSize := True;
    I.Stretch := False;
    I.Width := Round(IW * f);
    I.Height := Round(IH * f);
    I.AutoSize := False;
    I.Stretch := True;
  end;
  I.Visible := True;
end;
```

```

procedure PrintImage(I: TImage; f: Real);
// Ausdruck eines Images mit Skalierungsfaktor f (A4-Format mit f = 1)
var Breite, Hoehe: Integer;
    Faktor: Real;
    Bereich: TRect;
    Bild: TBitmap;
begin
  with I do begin
    if f = 0 then Exit;
    f := abs(f);
    Bild := TBitmap.Create;
    Bild.Width := ClientWidth;
    Bild.Height := ClientHeight;
    Bild.PixelFormat := pf24Bit;
    Bild.Canvas.CopyRect(Rect(0,0,ClientWidth,ClientHeight),
                        Canvas,Rect(0,0,ClientWidth,ClientHeight));
    Faktor := ClientHeight / ClientWidth;
    Printer.BeginDoc;
    Breite := Round(Printer.Canvas.ClipRect.Right * f);
    Hoehe := Round(Breite * Faktor);
    if Hoehe > Printer.Canvas.ClipRect.Bottom then begin
      Hoehe := Round(Printer.Canvas.ClipRect.Bottom * f);
      Breite := Round(Hoehe / Faktor);
    end;
    Bereich:= Rect(0,0,Breite,Hoehe);
    Printer.Canvas.StretchDraw(Bereich,Bild);
    Printer.EndDoc;
    Bild.Free;
  end;
end;

procedure PrintPicture(I:TImage; a,b,w,h:Integer);
// Imageausdruck mit Versatz (a,b) und Druckgröße (w,h)
var Bereich : TRect;
begin
  Printer.BeginDoc;
  Bereich := Rect(a,b,a+w,b+h);
  Printer.Canvas.StretchDraw(Bereich,I.Picture.Graphic);
  Printer.EndDoc;
end;

procedure ChangeImage(I: TImage; Nega,Grau,Kont,R,G,B : Integer);
// Veränderung der RGB-Werte einer Bitmap-Grafik
// Nega = 0,1 und Grau = 0,1 und
// Kont,R,G,B = 0,...,255

type TRGBValue = packed record
    Blue : Byte;
    Green: Byte;
    Red : Byte;
end;

var x,y : Integer;
    Pixel,Ziel,Quelle : ^TRGBValue;
    f : Byte;
    ar : array[0..255] of Byte;
    k : Integer;
    fak: Single;
    Col: Boolean;

```

```
begin
{
  If (I.Picture.Bitmap.PixelFormat <> pf24bit) or
    (I.Picture.Bitmap.PixelFormat <> pf32bit) then begin
    ShowMessage('Bild hat keine 24-Bitfarben');
    Exit;
  end;
end;
}
if Kont <> 0 then begin
  fak := 1 + kont/100;
  for x := 0 to 255 do begin
    k := Round((Integer(x)-128)*fak) + 128;
    if k > 255 then ar[x] := 255
      else if k < 0 then ar[x] := 0
        else ar[x] := k;
  end;
end;

if (R<>0) or (G<>0) or (B<>0) then Col := True
  else Col := False;

for y := 0 to (I.Picture.Bitmap.Height-1) do begin

  Pixel := I.Picture.Bitmap.Scanline[y];
  Ziel := I.Picture.Bitmap.Scanline[y];
  Quelle := I.Picture.Bitmap.Scanline[y];

  for x := 0 to (I.Picture.Bitmap.Width-1) do begin

    if Col then begin
      k := Pixel.Blue + B;
      if k > 255 then k := 255;
      if k < 0 then k := 0;
      Pixel.Blue := LoByte(k);

      k := Pixel.Green + G;
      if k > 255 then k := 255;
      if k < 0 then k := 0;
      Pixel.Green := LoByte(k);

      k := Pixel.Red + R;
      if k > 255 then k := 255;
      if k < 0 then k := 0;
      Pixel.Red := LoByte(k);
    end;

    if Grau <> 0 then begin
      F := HiByte(Pixel.Red*77+Pixel.Green*151+pixel.Blue*28);
      Pixel.Blue := F;
      Pixel.Green := F;
      Pixel.Red := F;
    end;

    if Nega <> 0 then begin
      Pixel.Blue := Pixel.Blue XOR $FF;
      Pixel.Green := Pixel.Green XOR $FF;
      Pixel.Red := Pixel.Red XOR $FF;
    end;
  end;
end;
```

```

        if Kont <> 0 then begin
            Ziel^.Red := ar[Quelle^.Red];
            Ziel^.Blue := ar[Quelle^.Blue];
            Ziel^.Green := ar[Quelle^.Green];
            inc(Ziel);
            inc(Quelle);
        end;
        Inc(Pixel);
    end;
end;
I.Refresh;
end;

function MirrorImage(BM: TBitmap; Horiz: Boolean): TBitmap;
// horizontale oder vertikale Spiegelung
begin
    Result := TBitmap.Create;
    Result.Width := BM.Width;
    Result.Height := BM.Height;
    if Horiz then
        StretchBlt(Result.Canvas.Handle,0,0,
            Result.Width,Result.Height,
            BM.Canvas.Handle,BM.Width,0,
            -BM.Width,BM.Height,srcCopy)
    else
        StretchBlt(Result.Canvas.Handle,0,0,
            Result.Width,Result.Height,
            BM.Canvas.Handle,0,BM.Height,
            BM.Width,-BM.Height,srcCopy);
end;

procedure RotateImageRight(Quelle: TImage);
// Ein Image-Bitmap um 90° nach links drehen
type TMyHelp = array[0..0] of TRGBQuad;
var P : PRGBQuad;
    x,y,w,h : Integer;
    RowOut : ^TMyHelp;
    help : TBitmap;
begin
    Quelle.Picture.Bitmap.PixelFormat := pf32bit;
    H := Quelle.Picture.Bitmap.Width;
    W := Quelle.Picture.Bitmap.Height;
    help := TBitmap.Create;
    help.PixelFormat := pf32bit;
    help.Width := W; help.Height := H;
    for Y := 0 to (H-1) do begin
        RowOut := help.ScanLine[Y];
        P := Quelle.Picture.Bitmap.ScanLine[Quelle.Picture.Bitmap.Height-1];
        inc(P,Y);
        For x := 0 to (W-1) do begin
            RowOut[X] := P^;
            inc(P,H);
        end;
    end;
    Quelle.Picture.Bitmap.Assign(help);
    Quelle.Picture.Bitmap.PixelFormat := pf24bit;
    help.Free;
end;

```

```

procedure RotateImageLeft(Quelle: TImage);
// Ein Image-Bitmap um 90° nach links drehen
type TMyHelp = array[0..0] of TRGBQuad;
var P      : PRGBQuad;
    x,y,w,h : Integer;
    RowOut  : ^TMyHelp;
    help    : TBitmap;
begin
  Quelle.Picture.Bitmap.PixelFormat := pf32bit;
  H := Quelle.Picture.Bitmap.Width;
  W := Quelle.Picture.Bitmap.Height;
  help := TBitmap.Create;
  help.PixelFormat := pf32bit;
  help.Width := W;
  help.Height := H;
  for Y := 0 to (H-1) do begin
    RowOut := help.ScanLine[Y];
    P := Quelle.Picture.Bitmap.ScanLine[Quelle.Picture.Bitmap.Height-1];
    inc(P,Y);
    For x := (W-1) downto 0 do begin
      RowOut[X] := P^;
      inc(P,H);
    end;
  end;
  help := MirrorImage(help,False);
  Quelle.Picture.Bitmap.Assign(help);
  Quelle.Picture.Bitmap.PixelFormat := pf24bit;
  help.Free;
end;

```

```

procedure DistortImage(Quelle,Puffer: TImage; k: Real; Horiz: Boolean);
// horizontale oder vertikale Verzerrung eines Images über einen Puffer
var W,H : Integer;
    k1,k2 : Real;
begin
  W := Quelle.Width;
  H := Quelle.Height;
  k1 := 1; k2 := 1;
  if Horiz then k1 := k else k2 := k;
  W := Round(W*k1);
  H := Round(H*k2);
  Puffer.Picture := NIL;
  Puffer.AutoSize := False;
  Puffer.Stretch := False;
  Puffer.Width := W;
  Puffer.Height := H;
  Puffer.Canvas.StretchDraw(Rect(0,0,W,H),Quelle.Picture.Graphic);
  Puffer.AutoSize := True;
  Quelle.Picture := NIL;
  Quelle.AutoSize := False;
  Quelle.Stretch := False;
  Quelle.Width := W;
  Quelle.Height := H;
  Quelle.Canvas.Draw(0,0,Puffer.Picture.Bitmap);
  Quelle.AutoSize := True;
  Quelle.Picture.Bitmap.PixelFormat := pf24bit;
  Puffer.Picture := NIL;
end;

```

```
procedure StretchImage(Quelle,Puffer: TImage; k: Real);
// Verkleinerung oder Vergrößerung eines Images über einen Puffer
var W,H,L: Integer;
begin
  W := Quelle.Width;
  H := Quelle.Height;
  L := 1;
  if k < 0 then begin
    L := -1;
    k := -k;
  end;
  W := Round(W*k);
  H := Round(H*k);
  if (L < 1) then begin
    if (H <= Round(W*4/5)) then H := Round(W*4/5);
    if (H > Round(W*4/5)) and (H < W) then H := W;
    if (H > 115) then H := 115;
  end;
  Puffer.Picture := NIL;
  Puffer.AutoSize := False;
  Puffer.Stretch := False;
  Puffer.Width := W;
  Puffer.Height := H;
  Puffer.Canvas.StretchDraw(Rect(0,0,W,H),Quelle.Picture.Graphic);
  Puffer.AutoSize := True;

  Quelle.Picture := NIL;
  Quelle.AutoSize := False;
  Quelle.Stretch := False;
  Quelle.Width := W;
  Quelle.Height := H;
  Quelle.Canvas.Draw(0,0,Puffer.Picture.Bitmap);
  Quelle.AutoSize := True;
  Quelle.Picture.Bitmap.PixelFormat := pf24bit;
  Puffer.Picture := NIL;
end;

procedure CopyImage(Quelle,Puffer:TImage; x1,y1,x2,y2: Integer; HT: Boolean);
// kopiert einen Bereich des Quellen-Images in einen Zwischenpuffer
// mit oder ohne Hintergrundtransparenz
var W,H : Integer;
begin
  W := x2 - x1;
  H := y2 - y1;
  Puffer.Picture := NIL;
  Puffer.AutoSize := False;
  Puffer.Stretch := False;
  Puffer.Width := W;
  Puffer.Height := H;
  Puffer.Canvas.CopyRect(Rect(0,0,W,H),Quelle.Canvas,Rect(x1,y1,x2,y2));
  Puffer.AutoSize := True;
  Puffer.Picture.Bitmap.PixelFormat := pf24bit;
  Puffer.Picture.Bitmap.TransparentColor := clBlack;
  Puffer.Transparent := HT;
end;
```

```

procedure PasteImage(Puffer,Ziel: TImage; x,y: Integer);
// Fügt einen zwischengepufferten Bereich
// in ein Ziel-Image ein - an die Position x,y
var W,H : Integer;
begin
  W := Puffer.Width;
  H := Puffer.Height;
  Ziel.Canvas.StretchDraw(Rect(x,y,x+W,y+H),Puffer.Picture.Graphic);
  Ziel.Picture.Bitmap.PixelFormat := pf24bit;
end;

procedure CropImage(var Quelle,Puffer: TImage; x1,y1,x2,y2: Integer);
// Schneidet ein Image zu über einen zwischengepufferten Bereich
var W,H : Integer;
begin
  W := x2 - x1;
  H := y2 - y1;
  Puffer.Picture := NIL;
  Puffer.Transparent := False;
  Puffer.AutoSize := False;
  Puffer.Stretch := False;
  Puffer.Width := W;
  Puffer.Height := H;
  Puffer.Canvas.CopyRect(Rect(0,0,W,H),Quelle.Canvas,Rect(x1,y1,x2,y2));
  Puffer.AutoSize := True;

  Quelle.Picture := NIL;
  Quelle.AutoSize := False;
  Quelle.Stretch := False;
  Quelle.Width := W;
  Quelle.Height := H;
  Quelle.Canvas.Draw(0,0,Puffer.Picture.Bitmap);
  Quelle.AutoSize := True;
  Quelle.Picture.Bitmap.PixelFormat := pf24bit;
  Puffer.Picture := NIL;
end;

procedure SmoothImage(IM: TImage; X1,Y1,X2,Y2: Integer);
// Farbglättung im Rechtecksbereich (X1/Y1) - /X2/Y2)
var I,J: Integer;

function SmoothPixel(IM: TImage; X,Y: Integer): TColor;
// Passt eine Pixelfarbe an die Umgebung an
// Wichtig beim Weichzeichnen von Konturen
type TFB = record
  Red : Byte;
  Green: Byte;
  Blue : Byte;
  Palet: Byte;
end;
TAFB = array[1..9] of TFB;
var AFB : TAFB;
    OP,NP: TFB;
    I,S : Integer;
begin

```

```

with IM.Canvas do begin
  OP := TFB(Pixels[X,Y]);
  AFB[1] := TFB(Pixels[X-1,Y-1]);
  AFB[2] := TFB(Pixels[X,Y-1]);
  AFB[3] := TFB(Pixels[X+1,Y-1]);
  AFB[4] := TFB(Pixels[X-1,Y]);
  AFB[5] := TFB(Pixels[X,Y]);
  AFB[6] := TFB(Pixels[X+1,Y]);
  AFB[7] := TFB(Pixels[X-1,Y+1]);
  AFB[8] := TFB(Pixels[X,Y+1]);
  AFB[9] := TFB(Pixels[X+1,Y+1]);
  S := 0;
  For I := 1 to 9 do S := S + AFB[I].Red;
  NP.Red := LoByte(Round(S/9));
  S := 0;
  For I := 1 to 9 do S := S + AFB[I].Green;
  NP.Green := LoByte(Round(S/9));
  S := 0;
  For I := 1 to 9 do S := S + AFB[I].Blue;
  NP.Blue := LoByte(Round(S/9));
  NP.Palet := OP.Palet;
  Result := RGB(NP.Red, NP.Green, NP.Blue);
end;
end;

begin
  with IM do begin
    For I := X1 to X2 do
      For J := Y1 to Y2 do
        Canvas.Pixels[I,J] := SmoothPixel(IM,I,J);
      end;
    end;
  end;

  procedure ImageBlob(Im0: TImage);
  // Bildzerstörung mit angepassten Farbflecken
  const n = 20;
        k = 2 / 3;
        a = 500;
  Var x,y: Integer;
        p: Tcolor;
        r,g,b: Byte;
        sr,sg,sb: Integer;
        i,j: Integer;
        rx,ry,cnt,blb: Integer;
        w,h: Integer;
  begin
    Im0.Stretch := True;
    w := Im0.Width;
    h := Im0.Height;
    rx := w div n; // 30
    ry := Round(k*rx); // 20
    cnt := rx div 4; // 10
    blb := a; // 2000
    Im0.Canvas.Pen.Style := psClear;
    For I := 0 to blb do begin
      x := random(w);
      y := random(h);
      sr := 0; sg := 0; sb := 0;

```

```

For j := 1 to cnt do begin
  p := Im0.Canvas.Pixels[x-Rx+Random(2*rx+1),y-ry+Random(2*ry+1)];
  sr := sr + (p and $0000FF);
  sg := sg + (p and $00FF00) shr 8;
  sb := sb + (p and $FF0000) shr 16;
end;
r := sr div cnt;
g := sg div cnt;
b := sb div cnt;
Im0.Canvas.Brush.Color := r + g shl 8 + b shl 16;
Im0.Canvas.Pen.Color := r + g shl 8 + b shl 16;
Im0.Canvas.Rectangle(x-Rx, y-Ry, x+Rx, y+Ry);
// Im0.Canvas.Ellipse(x-Rx, y-Ry, x+Rx, y+Ry);
Im0.Refresh;
if (GetAsyncKeyState(VK_ESCAPE)<>0) then Break;
end;
end;

procedure ImageDecay(Im0: TImage);
// Random Image-Pixel
Const m = 20;
      n = 200000;
Const RP = 0.2989;
      GP = 0.5866;
      BP = 1-RP-GP;
Var x,y,x2,y2,i,j: integer;
    w,h: Integer;
    p1,p2: TColor;
    r,g,b: Byte;
    light1,light2: Integer;
    Im1: TBitmap;
begin
  Im0.Stretch := True;
  w := Im0.Width;
  h := Im0.Height;
  Im1 := TBitmap.Create;
  Im1.Width := w;
  Im1.Height := h;
  Im1.Assign(Im0.Picture.Bitmap);
  for j := 1 to m do begin
    for i := 1 to n do begin
      x := 1 + random(w-1);
      y := random(h-1);
      x2 := x - 1 + random(3);
      y2 := y + Random(10);
      p1 := Im1.Canvas.Pixels[x,y];
      r := (P1 and $0000FF);
      g := (P1 and $00FF00) shr 8;
      b := (P1 and $FF0000) shr 16;
      light1 := trunc (r * RP + g * GP + b*BP);
      p2 := Im1.Canvas.Pixels[x2,y2];
      r := (P2 and $0000FF);
      g := (P2 and $00FF00) shr 8;
      b := (P2 and $FF0000) shr 16;
      light2 := trunc (r * RP + g * GP + b*BP);
    end;
  end;
end;

```

```

    if light2 > light1 then begin
        Im1.Canvas.Pixels[x, y] := p2;
        Im1.Canvas.Pixels[x2, y2] := p1;
    end;
end;
with Im0 do
    Canvas.CopyRect(Rect(0,0,w-1,h-1),
        Im1.Canvas,Rect(0,0,w-1,h-1));
    Im0.Refresh;
    if (GetAsyncKeyState(VK_ESCAPE)<>0) then Break;
end;
Im1.Free;
end;

procedure ImageBlend(Im0,Im1: TImage; Mode: Integer);
// Bildüberblendung von Quelle (Im1) auf Ziel (Im0),
// mit Mode: 1 = ausgewogen, 2 = Ziel, 3,4 Quelle überwiegt
var x,y: integer;
    c: TColor;
    r,g,b: byte;
    r1,g1,b1: byte;
    r2,g2,b2: byte;
    t: real;
    w0,h0: Integer;
begin
    if Mode = 0 then Exit;

    w0 := Im0.Width;
    h0 := Im0.Height;
    Im1.Stretch := True;

    for x := 0 to w0-1 do begin
        case Mode of
            1: t := 1/2;
            2: t := x/w0;
            3: t := 1 - (x/w0);
            4: t := 1;
        end;
        for y := 0 to h0-1 do begin
            c := Im0.Canvas.Pixels[x,y];
            r1 := (c and $0000FF);
            g1 := (c and $00FF00) shr 8;
            b1 := (c and $FF0000) shr 16;
            c := Im1.Canvas.Pixels[x,y];
            r2 := (c and $0000FF);
            g2 := (c and $00FF00) shr 8;
            b2 := (c and $FF0000) shr 16;
            r := round( t*r1 + (1-t)*r2 );
            g := round( t*g1 + (1-t)*g2 );
            b := round( t*b1 + (1-t)*b2 );
            Im1.Canvas.Pixels[x,y] := r + g shl 8 + b shl 16;
        end;
        Im1.Refresh;
        if (GetAsyncKeyState(VK_ESCAPE)<>0) then Break;
    end;
    Im1.AutoSize := True;
    Im1.Picture.Bitmap.Assign(Im0.Picture.Bitmap);
end;

```

```

procedure ImageMove(Im0,Im1: TImage; Step: Integer);
// Bildüberblendung von Quelle (Im1) auf Ziel (Im0),
// mit Step als Geschwindigkeit, OHNE Transparenz
var w0,h0,w1,h1: Integer;
    x,y: Integer;
begin
    if (Step = 0) then Exit;
    w0 := Im0.Width;
    h0 := Im0.Height;
    w1 := Im1.Width;
    h1 := Im1.Height;
    Im1.Stretch := True;
    Im1.Canvas.CopyMode := cmSrcCopy;
    x := 0;
    repeat
        x := x + Step;
        Im1.Canvas.CopyRect(Rect(0,0,x,h0),Im0.Canvas,Rect(0,0,x,h0));
        Im1.Refresh;
        if (GetAsyncKeyState(VK_ESCAPE)<>0) then Break;
    until x >= w0;
    {
    y := 0;
    repeat
        y := y + Step;
        Im1.Canvas.CopyRect(Rect(0,0,w0,y),Im0.Canvas,Rect(0,0,w0,y));
        Im1.Refresh;
        if (GetAsyncKeyState(VK_ESCAPE)<>0) then Break;
    until y >= h0;
    }
    Im1.AutoSize := True;
    Im1.Picture.Bitmap.Assign(Im0.Picture.Bitmap);
end;

procedure ImageZoom(Im0,Im1,Im2: TImage; kmax: Real; n: Integer; Fit: Boolean);
// Zooming mit Im0 = Quelle, Im1 = erster Puffer, Im2 = zweiter Puffer
// und kmax = maximaler Zoomfaktor (%), n = Anzahl der Zoomschritte
var k,k0: Real;
    i,l,t: Integer;
    w0,h0,w2,h2: Integer;
    rep: Boolean;
begin
    rep := False;
    kmax := kmax / 100;
    if (kmax < 0) then begin
        rep := True;
        kmax := -kmax;
    end;
    k0 := (kmax-1) / n;
    w0 := Im0.width;
    h0 := Im0.height;

    Im1.Visible := False;
    Im1.AutoSize := False;
    Im1.Stretch := True;
    Im1.Picture.Bitmap.Assign(Im0.Picture.Bitmap);
    Im2.Visible := False;
    Im2.AutoSize := False;
    Im2.Stretch := True;

```

```

For i := 1 to n do begin
  k := 1 + i * k0;
  w2 := Round(w0*k);
  h2 := Round(h0*k);
  l:= (w2-w0) div 2;
  t:= (h2-h0) div 2;
  Im2.Picture := NIL;
  Im2.Width   := w2;
  Im2.Height  := h2;
  Im2.Canvas.StretchDraw(Rect(0,0,w2,h2),Im1.Picture.Bitmap);

  if Fit then begin
    Im0.Picture := NIL;
    Im0.AutoSize := False;
    Im0.Stretch := True;
    Im0.Width   := w0;
    Im0.Height  := h0;
  end;
//   if kmax < 1 then Im0.Canvas.FillRect(Rect(0,0,Im0.Width,Im0.Height));

  Im0.Canvas.CopyRect(Rect(0,0,w0,h0),Im2.Canvas,Rect(l,t,l+w0,t+h0));
  Im0.Refresh;
  if (GetAsyncKeyState(VK_ESCAPE)<>0) then Break;
end;

if rep then begin
  k0 := (kmax-1) / n;
  For i := n downto 1 do begin
    k := 1 + i * k0;
    w2 := Round(w0*k);
    h2 := Round(h0*k);
    l:= (w2-w0) div 2;
    t:= (h2-h0) div 2;
    Im2.Picture := NIL;
    Im2.Width   := w2;
    Im2.Height  := h2;
    Im2.Canvas.StretchDraw(Rect(0,0,w2,h2),Im1.Picture.Bitmap);

    if Fit then begin
      Im0.Picture := NIL;
      Im0.AutoSize := False;
      Im0.Stretch := True;
      Im0.Width   := w0;
      Im0.Height  := h0;
    end;
//   if kmax < 1 then Im0.Canvas.FillRect(Rect(0,0,Im0.Width,Im0.Height));

    Im0.Canvas.CopyRect(Rect(0,0,w0,h0),Im2.Canvas,Rect(l,t,l+w0,t+h0));
    Im0.Refresh;
    if (GetAsyncKeyState(VK_ESCAPE)<>0) then Break;
  end;
end;
end;

```

```

procedure ImageSectionZoom(Im0,Im1,Im2: TImage; kmax: Real; n:Integer; PLO,PRU:
TPoint);
// Zooming mit Im0 = Quelle, Im1 = erster Puffer, Im2 = zweiter Puffer
// kmax = maximaler Zoomfaktor (%), n = Anzahl der Zoomschritte
// PLO und PRU sind die Eckpunkte des gezoomten Bildausschnittes
var k,k0: Real;
    i,l,t, w0,h0,w2,h2: Integer;
    rep: Boolean;
begin
    rep := False;
    kmax := kmax / 100;
    if (kmax < 0) then begin
        rep := True;
        kmax := -kmax;
    end;
    k0 := (kmax-1) / n;
    w0 := PRU.X - PLO.X;
    h0 := PRU.Y - PLO.Y;
    Im1.Picture := NIL;
    Im1.Visible := False;
    Im1.AutoSize := False;
    Im1.Stretch := True;
    Im1.Width := w0;
    Im1.Height := h0;
    Im1.Canvas.CopyRect(Rect(0,0,w0,h0),Im0.Canvas,Rect(PLO,PRU));
    Im2.Visible := False;
    Im2.AutoSize := False;
    Im2.Stretch := True;
    For i := 0 to n do begin
        k := 1 + i * k0;
        w2 := Round(w0*k);
        h2 := Round(h0*k);
        l:= (w2-w0) div 2;
        t:= (h2-h0) div 2;
        Im2.Picture := NIL;
        Im2.Width := w2;
        Im2.Height := h2;
        Im2.Canvas.StretchDraw(Rect(0,0,w2,h2),Im1.Picture.Bitmap);
        Im0.Canvas.CopyRect(Rect(PLO,PRU),Im2.Canvas,Rect(l,t,l+w0,t+h0));
        Im0.Refresh;
    end;
    if rep then begin
        k0 := (kmax-1) / n;
        For i := n downto 1 do begin
            k := 1 + i * k0;
            w2 := Round(w0*k);
            h2 := Round(h0*k);
            l:= (w2-w0) div 2;
            t:= (h2-h0) div 2;
            Im2.Picture := NIL;
            Im2.Width := w2;
            Im2.Height := h2;
            Im2.Canvas.StretchDraw(Rect(0,0,w2,h2),Im1.Picture.Bitmap);
            Im0.Canvas.CopyRect(Rect(PLO,PRU),Im2.Canvas,Rect(l,t,l+w0,t+h0));
            Im0.Refresh;
        end;
    end;
end;
end;
end;

```

```

procedure ImageFish(Im0,Im1: TImage; FishSize,FishArea: Integer; PF: TPoint);
// Begrenzter Verzerrungs-Effekt, Im0 = Quelle, Im1 = Puffer für die Quelle
{
const p1: Integer = 100; // Kernbereich (FishSize)
      p2: Integer = 100; // Ausdehnung (FishArea)
}
const p3: Integer = 100; // x-Koordinate (PF.X)
      p4: Integer = 100; // y-Koordinate (PF.Y)
var p1,p2,x,y,x2,y2: integer;
    r: integer;
    f: real;
    x0: integer;
    y0: integer;
    m: single;
    d: single;
    p: TPoint;
begin
  p1 := FishSize;
  p2 := FishArea;
  Im0.Picture.Graphic.Assign(Im1.Picture.Graphic);
  Im0.Stretch := True;
  m := p1 * 100;
  d := p1 * p2;
  p := PF; // Im0.ScreenToClient(Mouse.CursorPos);
  if (p.x < 0) or (p.y < 0) then begin
    p.x := p3;
    p.y := p4;
  end;
  x0 := p.x;
  y0 := p.y;
  for x := 0 to Im0.Width do begin
    for y := 0 to Im0.Height do
      begin
        r := (x-x0)*(x-x0) + (y-y0)*(y-y0);
        if (r > d) or (r = 0) then f := 1
          else f := m / r;
        x2 := round(x0 + (x-x0)/f);
        y2 := round(y0 + (y-y0)/f);
        Im0.Canvas.Pixels[x,y] := Im1.Canvas.Pixels[x2,y2]
      end
    end;
  end;
end;

procedure ImageInsert(Im0,Im1,Im2: TImage; IB,IM,IX,IY: Integer);
// Einfügen von Im1 in Im0 mittels Im2, mit IB als Bildbreite
// an der Position (IX,IY) und mit IM als InsertModus:
// 0 = mit Schwarz als Transparenzfarbe, n = ohne Transparenz
// mit weißem Rahmen (+n) oder schwarzem Rahmen (-n)
var F: TColor;
    k: Real;
    IH: Integer;
begin
  Im2.Picture.Bitmap.TransparentColor := clBlack;
  Im2.Transparent := False;
  if IM = 0 then begin
    Im2.Transparent := True;
  end;
end;

```

```

if IM > 0 then F := clWhite;
if IM < 0 then begin
    F := clBlack;
    IM := - IM;
end;
k := Im1.Height / Im1.Width;
IH := Round(IB * k);

Im2.Picture := NIL;
Im2.Visible := False;
Im2.AutoSize := False;
Im2.Stretch := True;
Im2.Width := IB;
Im2.Height := IH;

Im2.Canvas.StretchDraw(Rect(0,0,IB,IH),Im1.Picture.Bitmap);
Im0.Canvas.StretchDraw(Rect(IX,IY,IX+IB,IY+IH),Im2.Picture.Bitmap);
if IM > 0 then begin
    Im0.Canvas.Pen.Width := IM;
    Im0.Canvas.Pen.Color := F;
    Im0.Canvas.Rectangle(Rect(IX,IY,IX+IB,IY+IH));
    Im0.Canvas.Pen.Width := 1;
    Im0.Canvas.Pen.Color := clBlack;
end;
Im2.Transparent := False;
end;

procedure ImageFrame(Im0: TImage; IB,IM,IX1,IY1,IX2,IY2: Integer);
// Zeichnet im Bild Im0 einen rechteckigen Rahmen mit Breite IB
// und Farbe IM (1 = Schwarz, 2 = weiß) im Bereich (IX1,IY1,IX2,IY2)
// und bei IB < 0 wird der Bereich voll ausgefüllt
var F: TColor;
    Fill: Boolean;
begin
    Fill := False;
    if IB < 0 then begin
        IB := -IB;
        Fill := True;
    end;
    if IB = 0 then IB := 1;
    if IM = 1 then F := clBlack;
    if IM <> 1 then F := clWhite;
    if Fill then begin
        Im0.Canvas.Brush.Color := F;
        Im0.Canvas.Brush.Style := bsSolid;
    end;
    Im0.Canvas.Pen.Width := IB;
    Im0.Canvas.Pen.Color := F;
    Im0.Canvas.Rectangle(Rect(IX1,IY1,IX2,IY2));
    Im0.Canvas.Pen.Width := 1;
    Im0.Canvas.Pen.Color := clBlack;
    if Fill then begin
        Im0.Canvas.Brush.Color := clBlack;
        Im0.Canvas.Brush.Style := bsClear;
    end;
end;

end.

```