Dynamische Erzeugung von Bildern im geräteabhängigen angepassten Formaten

Da es mir widerstrebt Bilder in unzähligen Formaten auf der Festplatte zu speichern und diese Sammlung dann jedes mal wenn ein Gerät mit anderen Abmessungen erscheint zu erweitern, ist eine Lösung von Nöten die Bilder in minimaler Zeit skaliert. Da das Laden eines kompletten Bildes in ein Image Objekt und es dann zu skalieren sehr zeitaufwändig ist, muss je nach Zielgröße entschieden werden.
Wird z.B. nur eine kleine Voransicht benötigt, kann sie aus einer internen Voransicht (EXIF Thumbnail) generiert werden. Gerade kleine Voransichten treten gehäuft auf (z.B. in Bildergalerien), größere eher in geringerer Anzahl pro Seitenaufruf, gerade darum ist ein spezielles Augenmerk auch darauf zu Richten, schon beim Entwurf gewisse Schwellwerte zu beachten. Im Regelfall hat so ein Thumbnail Abmessungen von 160x120 Pixel!
Wird eine Bild mit größeren Abmessungen benötigt aber dennoch viel kleiner als das Original, wird nicht die komplette Auflösung benötigt und so kann es auch reichen nur ein Teil der Auflösung in den Speicher zu laden und dann die Skalierung vorzunehmen.

Zur Zeit (Stand 01.11.2023) existiert meines Erachtens ein Fehler in der Datei fpreadjpeg.pas des Free Pascal Package fcl-image, der das fehlerfreie Scalieren verhindert, hier folgt nun ein Patch für das Problem.

```
Index: packages/fcl-image/src/fpreadjpeg.pas
===================================================================
--- packages/fcl-image/src/fpreadjpeg.pas (Revision 30e7bfb)
+++ packages/fcl-image/src/fpreadjpeg.pas (Arbeitskopie)
@@ -258,6 +258,19 @@
   c: word;
   Status,Scan: integer;
   ReturnValue,RestartLoop: Boolean;
+  S: TSize;
+
+  function TranslateSize(const Sz: TSize): TSize;
+  begin
+    case FOrientation of
+      eoUnknown, eoNormal, eoMirrorHor, eoMirrorVert, eoRotate180: Result
:= Sz;
+      eoMirrorHorRot270, eoRotate90, eoMirrorHorRot90, eoRotate270:
+      begin
+        Result.Width := Sz.Height;
+        Result.Height := Sz.Width;
+      end;
+    end;
+  end;

   procedure InitReadingPixels;
   var d1,d2:integer;
@@ -465,7 +478,9 @@

   jpeg_start_decompress(@FInfo);

-  Img.SetSize(FWidth,FHeight);
```

```
+//    Img.SetSize(FWidth,FHeight);
+    S :=
TranslateSize(TSize.Create(FInfo.output_width,FInfo.output_height));
+    Img.SetSize(S.Width, S.Height);

   GetMem(SampArray,SizeOf(JSAMPROW));
   GetMem(SampRow,FInfo.output_width*FInfo.output_components);
```

In der folgenden Unit ImgUtils habe ich nun die benötigten Funktionen zusammengefasst, bei der Entwicklung habe ich besonders darauf geachtet das die von den Browser unterstützten Formate (JPEG, PNG, GIF) berücksichtigt werden, aber auch am Beispiel von BMP eine Konvertierung eingebaut ist. Das Ermitteln der Abmessungen wurde bewusst in separate Funktionen auf der Basis einer Stream-Analyse ausgelagert, da das der Weg über die speziellen Bild-Klassen zu zeitaufwändig ist. Um ein Thumbnail aus einer JPEG-Datei zu lesen hätte sicherlich auch der Weg über eineEXIF Komponente gewählt werden können, jedoch muss ich auch an dieser Stelle wieder auf die Geschwindigkeit zurück kommen, das gezielte auslesen auf Stream-Basis ist natürlich um einiges schneller. Auch in der eigentlichen Funktion zum Skalieren der Bilder habe ich nur native Möglichkeiten genutzt nicht auf System eigene, ich nutze Free Pascal Standard Units aus dem Package **fcl-image** und eine eigene Unit FPWriteGIF die fehlt nämlich noch in diesem Package, die von mir geschriebene Unit veröffentliche ich natürlich auch noch, animierte GIFs werden jedoch z.Z. noch nicht unterstützt. Ein Geschwindigkeitsvergleich mit der GDI-Plus Variante unter Windows ergab sogar kleinere Zeitersparnisse, aber da die Bildskalierung in sehr kurzer Zeit geschieht ist es sehr schlecht messbar, eine Skalierung mit ImageMagick dauerte hingegen erheblich länger und macht eine "On-the-fly" Generierung, also ohne Caching quasi unmöglich.

imgutils.pas Pascal (32,63 kByte) 01.11.2023 08:26

```
//
*****************************************************************************

//  Title.............. :  Image utils
//
//  Modulname ......... :  imgutils.pas
//  Type .............. :  Unit
//  Author ............ :  Udo Schmal
//  Development Status  :  01.11.2023
//
*****************************************************************************

unit ImgUtils;
{$ifdef fpc}
  {$mode objfpc}
{$endif}
{$H+}

interface
uses
  {$ifdef windows}Windows{$else}BaseUnix, Unix, clocale{$endif},
  Classes, SysUtils, FPImage, FPImgCanv, FPReadBMP, FPReadJPEG, FPWriteJPEG,
  FPReadPNG, FPWritePNG, FPReadGIF, FPWriteGIF, FPReadTIFF, Clipping,
  {$ifdef USE_WebP}FPReadWebP, FPWriteWebP,{$endif}
```

```
  DateUtils, Math;

type
  TFPImageCanvasHelper = class helper for TFPImageCanvas
    procedure CopyRect90 (x,y:integer; canvas:TFPImageCanvas;
SourceRect:TRect);
    procedure CopyRect180 (x,y:integer; canvas:TFPImageCanvas;
SourceRect:TRect);
    procedure CopyRect270 (x,y:integer; canvas:TFPImageCanvas;
SourceRect:TRect);
    procedure FlipRectHorizontal (x,y:integer; canvas:TFPImageCanvas;
SourceRect:TRect);
    procedure FlipRectVertical (x,y:integer; canvas:TFPImageCanvas;
SourceRect:TRect);
    procedure TransposeRect (x,y:integer; canvas:TFPImageCanvas;
SourceRect:TRect);
    procedure TransverseRect (x,y:integer; canvas:TFPImageCanvas;
SourceRect:TRect);
  end;

function GetImageSize(const AFilename: string; out wWidth, wHeight: word):
boolean;
function GetImageDimensions(const AFilename: string): string;
function scaleImageToStream(const AFilename: string; var AMimeType: string;
var MemoryStream: TMemoryStream; const maxWidth, maxHeight: word; const
crop: boolean = false; const focusX: word = 50; const focusY: word = 50):
boolean;

implementation

procedure TFPImageCanvasHelper.CopyRect90 (x,y:integer;
canvas:TFPImageCanvas; SourceRect:TRect);
var xx,r,t : integer;
begin
  SortRect (SourceRect);
  with SourceRect do
  begin
    for t := top to bottom do
    begin
      xx := bottom-1 - t + x;
      for r := left to right do
        colors[xx, r - left + y] := canvas.colors[r, t];
    end;
  end;
end;

procedure TFPImageCanvasHelper.CopyRect180 (x,y:integer;
canvas:TFPImageCanvas; SourceRect:TRect);
var yy,r,t : integer;
begin
```

```
  SortRect (SourceRect);
  with SourceRect do
  begin
    for t := top to bottom do
    begin
      yy := bottom-1 - t + y;
      for r := left to right do
        colors[right-1 - r + x, yy] := canvas.colors[r, t];
    end;
  end;
end;

procedure TFPImageCanvasHelper.CopyRect270 (x,y:integer;
canvas:TFPImageCanvas; SourceRect:TRect);
var xx,r,t : integer;
begin
  SortRect (SourceRect);
  with SourceRect do
  begin
    for t := top to bottom do
    begin
      xx := t - top + x;
      for r := left to right do
        colors[xx, right-1 - r + y] := canvas.colors[r, t];
    end;
  end;
end;

procedure TFPImageCanvasHelper.FlipRectHorizontal (x,y:integer;
canvas:TFPImageCanvas; SourceRect:TRect);
var yy,r,t : integer;
begin
  SortRect (SourceRect);
  with SourceRect do
  begin
    for t := top to bottom do
    begin
      yy := t - top + y;
      for r := left to right do
        colors[right-1 - r + x, yy] := canvas.colors[r, t];
    end;
  end;
end;

procedure TFPImageCanvasHelper.FlipRectVertical (x,y:integer;
canvas:TFPImageCanvas; SourceRect:TRect);
var yy,r,t : integer;
begin
  SortRect (SourceRect);
  with SourceRect do
```

```
  begin
    for t := top to bottom do
    begin
      yy := bottom-1 - t + y;
      for r := left to right do
        colors[r - left + x, yy] := canvas.colors[r, t];
    end;
  end;
end;

procedure TFPImageCanvasHelper.TransposeRect (x,y:integer;
canvas:TFPImageCanvas; SourceRect:TRect);
var xx,r,t : integer;
begin
  SortRect (SourceRect);
  with SourceRect do
  begin
    for t := top to bottom do
    begin
      xx := bottom-1 - t + x;
      for r := left to right do
        colors[xx, right-1 - r + y] := canvas.colors[r, t];
    end;
  end;
end;

procedure TFPImageCanvasHelper.TransverseRect (x,y:integer;
canvas:TFPImageCanvas; SourceRect:TRect);
var xx,r,t : integer;
begin
  SortRect (SourceRect);
  with SourceRect do
  begin
    for t := top to bottom do
    begin
      xx := t - top + x;
      for r := left to right do
        colors[xx, r - left + y] := canvas.colors[r, t];
    end;
  end;
end;

type
  TSeg = packed record
    Prefix: byte; // $FF
    Marker: byte; // Marker Nr (1 byte)
    DataSize: word; // Data Size
  end;

  TFrame = packed record
```

```
    precision: byte;
    height: word;
    width: word;
    sampling: byte;
  end;

  TTIFFHeader = record
    ByteOrder: Word; // "II" ($4949, Little Endian) or  "MM" ($4D4D, Big
Endian)
    i42: Word; // $2A00 or $002A
  end;

  TTag = record
    TagID: Word; // Number
    TagType: Word; // Type
    Count: Cardinal; // Length
    Value: Cardinal; // Offset / Value
  end;

  TWordRec = record
    W1, W2:word;
  end;

function swap32(X:cardinal):cardinal;
begin
  TwordRec(Result).W2:=swap(TwordRec(X).W1);
  TwordRec(Result).W1:=swap(TwordRec(X).W2);
end;

procedure customizeTag(var ActTag: TTag);
var Totalbytesize: word;
begin
  with ActTag do
  begin
    TagID := swap(TagId);
    TagType := swap(TagType);
    Count := swap32(Count);
    Totalbytesize := 0;
    // 1 = unsigned byte
    // 2 = ascii string
    // 3 = unsigned short
    // 4 = unsigned long / cardinal
    // 5 = unsigned rational
    // 6 = signed byte
    // 7 = undefined
    // 8 = signed short
    // 9 = signed long / integer
    // 10 = signed rational / longint
    // 11 = signed float
    // 12 = double float
```

```
    // 13 =
    case byte(TagType) of
      1,2,6,7: Totalbytesize := ActTag.Count;
      3,8: Totalbytesize := ActTag.Count*2;
      4,9,11,13: Totalbytesize := ActTag.Count*4;
      5,10,12: Totalbytesize := ActTag.Count*8;
    end;
    case byte(TagType) of
      1,6: Value := byte(Value);
      3,8: Value := swap(TWordRec(Value).W1);
      4,9,11,13: Value := swap32(Value);
      5,10,12: Value := swap32(Value);
      2,7: if Totalbytesize>4 then
             Value := swap32(Value);
    end;
  end;
end;

function GetJPGSize(const AFilename: string; out wWidth, wHeight: word):
boolean;
var
  segment: TSeg;
  frame: TFrame;
  TIFFHeader: TTIFFHeader;
  tag: TTag;
  SOI, len, w: word;
  SegOffset, IFD0, EXIF_IFD: cardinal;
  Orientation: byte;
  i: integer;
  str: string;
  BigEndian, endloop: boolean;
  FileStream: TFileStream;
begin
  result := false;
  wWidth := 0;
  wHeight := 0;
  if FileExists(AFilename) then
  begin
    FileStream := TFileStream.Create(AFilename, fmOpenRead or
fmShareDenyNone);
    try
      wWidth := 0;
      wHeight := 0;
      FileStream.Seek(0, soFromBeginning);
      FileStream.Read(SOI, 2); // read start of image
      Orientation := $1;
      endloop := false;
      if SOI = $D8FF then // SOI marker FF D8 (Start Of Image) is JPEG
      while not endloop do
      begin
```

```
        FileStream.Read(segment, 4);
        endloop := segment.Prefix <> $FF;
        segment.DataSize := BEtoN(segment.DataSize);
        SegOffset := FileStream.Position;
        case segment.Marker of
        $C0, $C1, $C2, $C3, $C5, $C6, $C7, $C8, $C9, $CA, $CB, $CD, $CE, $CF:
// Start of Frame markers
          begin
            FileStream.Read(frame, sizeof(frame));
            frame.height := BEtoN(frame.height);
            frame.width := BEtoN(frame.width);
            if (frame.height <> wHeight) or (frame.width <> wWidth) then
            begin
              wHeight := frame.height;
              wWidth := frame.width;
            end
            else if Orientation in [$5, $6, $7, $8] then
            begin
              w := wWidth;
              wWidth := wHeight;
              wHeight := w;
            end;
            if (wHeight <> 0) and (wWidth <> 0) then
              endloop := true;
          end;
        $E1: // Application Marker APP1 Exif Section FF E1
          begin
            SetLength(str, 5);
            FileStream.Read(str[1], 5);
            if (str = 'Exif'#$0) then
            begin
              FileStream.Seek(1, soFromCurrent); // skip pad
              FileStream.Read(TIFFHeader, 4); // TIFFHeader
              BigEndian := (TIFFHeader.ByteOrder = $4D4D); // numeric data
stored in reverse order
              FileStream.Read(IFD0, 4); // Offset of IFD0
              if BigEndian then IFD0 := swap32(IFD0);

              EXIF_IFD := 0;
              FileStream.Position := SegOffset + 6 + IFD0; // start of IFD0
              FileStream.Read(len, 2); // number of Tags of IFD0
              if BigEndian then len := swap(len);
              for i:=1 to len do // read IFD0
              begin
                FileStream.Read(tag, sizeof(TTag));
                if BigEndian then customizeTag(tag);
                case tag.TagID of
                $0112: Orientation := tag.Value; // ExifOrientation
                $8769: EXIF_IFD := tag.Value; // Offset of Exif $8769
                end;
```

```
              end;

              if EXIF_IFD > 0 then // if Exif $8769 found
              begin
                FileStream.Position := SegOffset + 6 + EXIF_IFD;
                FileStream.Read(len, 2); // number of Tags of Exif
                if BigEndian then len := swap(len);
                for i:=1 to len do // read Exif
                begin
                  FileStream.Read(tag, sizeof(TTag));
                  if BigEndian then customizeTag(tag);
                  case tag.TagID of
                  $A002: wWidth := Tag.Value; // ExifImageWidth
                  $A003: wHeight := Tag.Value; // ExifImageHeight
                  end;
                end;
                if (wHeight <> 0) and (wWidth <> 0) then
                  endloop := true;
              end;
            end;
          end;
        $DA: endloop := true; // Start Of Scan (begins compressed data)
        end;
        FileStream.Position := SegOffset + segment.DataSize - 2; // skip to
next segment
      end;
      if (wWidth<>0) and (wHeight<>0) then
        result := true;
    finally
      FileStream.Free;
    end;
  end;
end;

function GetJPGThumbFromFile(const Filename: AnsiString; var RetImage:
TFPMemoryImage): boolean;
var
  segment: TSeg;
  TIFFHeader: TTIFFHeader;
  tag: TTag;
  SOI, len, version, ThumbLength, StripByteCounts, Xdensity, Ydensity: word;
  SegOffset, IFD0, IFD1, ThumbOffset, StripOffset, EXIF_IFD, Width, Height:
cardinal;
  Orientation, ThumbType, units, Xthumbnail, Ythumbnail: byte;
  i: integer;
  str: string;
  BigEndian, endloop: boolean;
  FileStream: TFileStream;
  MemoryStream: TMemoryStream;
  ReaderJPEG: TFPReaderJPEG;
```

```
  ReaderTIFF: TFPReaderTIFF;
  Image: TFPMemoryImage;
  RGBImage: TFPCompactImgRGB8Bit;
  Canvas, RetCanvas: TFPImageCanvas;
  copyRect: TRect;
begin
  result := false;
  Orientation := $1;
  if FileExists(FileName) then
  begin
    FileStream := TFileStream.Create(FileName, fmOpenRead or
fmShareDenyNone);
    try
//      try
        FileStream.Seek(0, soFromBeginning);
        FileStream.Read(SOI, 2); // read start of image
        endloop := false;
        if SOI = $D8FF then // SOI marker FF D8 (Start Of Image) is JPEG
        while not endloop do
        begin
          FileStream.Read(segment, 4);
          segment.DataSize := BEtoN(segment.DataSize);
          SegOffset := FileStream.Position;
          case segment.Marker of
          $E0: // Application Marker APP0 JFIF Marker
            begin
              SetLength(str,5);
              FileStream.Read(str[1], 5);
              if (str = 'JFIF'#$0) then
              begin // orientation top-down
                 // version (2 bytes) = X'0102'
                FileStream.Read(version, 2);
                version := BEtoN(version);
                // units (1 byte) Units for the X and Y densities.
                // units = 0:  no units, X and Y specify the pixel aspect
ratio
                // units = 1:  X and Y are dots per inch
                // units = 2:  X and Y are dots per cm
                FileStream.Read(units, 1);
                //case units of
                //0: ;// units: aspect
                //1: ;// units: pixels per inch
                //2: ;// units: pixels per cm
                //end;
                // Xdensity (2 bytes) Horizontal pixel density
                FileStream.Read(Xdensity, 2);
                Xdensity := BEtoN(Xdensity);
                // Ydensity (2 bytes) Vertical pixel density
                FileStream.Read(Ydensity, 2);
                Ydensity := BEtoN(Ydensity);
```

```
                // Xthumbnail (1 byte) Thumbnail horizontal pixel count
                FileStream.Read(Xthumbnail, 1);
                // Ythumbnail (1 byte) Thumbnail vertical pixel count
                FileStream.Read(Ythumbnail, 1);
                if (Xthumbnail > 0) and (Ythumbnail = 0) then
                  Ythumbnail := Xthumbnail;
                if (Ythumbnail > 0) and (Xthumbnail = 0) then
                  Xthumbnail := Ythumbnail;
                // (RGB)n (3n bytes)  Packed (24-bit) RGB values for the
thumbnail pixels, n = Xthumbnail * Ythumbnail
                if Ythumbnail > 0 then
                begin
                  MemoryStream := TMemoryStream.Create;
                  try
                    MemoryStream.CopyFrom(FileStream, (Xthumbnail*3) *
Ythumbnail); // copy from file Stream
                    MemoryStream.Position := 0;
                    ReaderTIFF := TFPReaderTIFF.Create;
                    try
                      RGBImage := TFPCompactImgRGB8Bit.Create(Xthumbnail,
Ythumbnail);
                      RGBImage.LoadFromStream(MemoryStream, ReaderTIFF); //
write Stream to Image
                    finally
                      ReaderTIFF.Free;
                    end;
                  finally
                    MemoryStream.Free;
                  end;
                end;
              end
              else if (str = 'JFXX'#$0) then
              begin
                // extension_code (1 byte) = Code which identifies the
extension.
                // In this version, the following extensions are defined:
                //   = X'10'   Thumbnail coded using JPEG
                //   = X'11'   Thumbnail stored using 1 byte/pixel
                //   = X'13'   Thumbnail stored using 3 bytes/pixel
                // extension_data (variable) = The specification of the
remainder of the JFIF
                // extension APP0 marker segment varies with the extension.

              end;
            end;
          $E1: // Application Marker APP1 Exif Section E1
            begin
              SetLength(str, 5);
              FileStream.Read(str[1], 5);
              if (str = 'Exif'#$0) then
```

```
              begin
                FileStream.Seek(1, soFromCurrent); // skip pad
                FileStream.Read(TIFFHeader, 4); // TIFFHeader
                BigEndian := (TIFFHeader.ByteOrder = $4D4D); // numeric data
stored in reverse order
                FileStream.Read(IFD0, 4); // Offset of IFD0
                if BigEndian then IFD0 := swap32(IFD0);

                EXIF_IFD := 0;
                FileStream.Position := SegOffset + 6 + IFD0; // start of IFD0
                FileStream.Read(len, 2); // number of Tags of IFD0
                if BigEndian then len := swap(len);
                for i:=1 to len do // read IFD0
                begin
                 FileStream.Read(tag, sizeof(TTag));
                 if BigEndian then customizeTag(tag);
                 case tag.TagID of
                 $0112: Orientation := Tag.Value; // ExifOrientation
                 $8769: EXIF_IFD := tag.Value; // Offset of Exif
                 end;
                end;

                FileStream.Read(IFD1, 4); // Offset of IFD1
                if BigEndian then IFD1 := swap32(IFD1);

                Width := 0;
                Height :=0;
                if EXIF_IFD>0 then // if Exif IFD found
                begin
                  FileStream.Position := SegOffset + 6 + EXIF_IFD;
                  FileStream.Read(len, 2); // number of Tags of Exif
                  if BigEndian then len := swap(len);
                  for i:=1 to len do // read Exif
                  begin
                    FileStream.Read(tag, sizeof(TTag));
                    if BigEndian then customizeTag(tag);
                    case tag.TagID of
                    $A002: Width := Tag.Value; // ExifImageWidth
                    $A003: Height := Tag.Value; // ExifImageHeight
                    end;
                  end;
                end;

                if IFD1>0 then // if IFD1 found
                begin
                  FileStream.Position := SegOffset + 6 + IFD1;
                  FileStream.Read(len, 2); // number of Tags of IFD1
                  if BigEndian then len := swap(len);
                  ThumbType := 6; // default JPEG
                  ThumbOffset := 0;
```

```
                    ThumbLength := 0;
                    for i:=1 to len do // read IFD1
                    begin
                      FileStream.Read(Tag, sizeof(TTag));
                      if BigEndian then customizeTag(Tag);
                      case Tag.TagID of
                      $0103: ThumbType := byte(Tag.Value); // 1 = TIFF, 6 =
JPEG
                      $0201: ThumbOffset := Tag.Value; // Thumbnail JPEG Offset
                      $0202: ThumbLength := word(Tag.Value); // Thumbnail JPEG
Length
                      $0111: StripOffset := Tag.Value;
                      $0117: StripByteCounts := word(Tag.Value);
                      end;
                    end;
                    if (ThumbOffset>0) and (ThumbLength>0) then // Thumbnail
found
                    begin
                      MemoryStream := TMemoryStream.Create;
                      try
                        if (ThumbType=1) then // TIFF
                        begin
                          FileStream.Position := SegOffset + 6 + StripOffset;
                          MemoryStream.CopyFrom(FileStream, StripByteCounts);
// copy from file Stream
                          MemoryStream.Position := 0;
                          ReaderTIFF := TFPReaderTIFF.Create;
                          try
                            Image := TFPMemoryImage.create(0,0);
                            Image.LoadFromStream(MemoryStream, ReaderTIFF); //
write Stream to Image
                          finally
                            ReaderTIFF.Free;
                          end;
                        end
                        else // if (ThumbType = 6) then // JPEG
                        begin
                          FileStream.Position := SegOffset + 6 + ThumbOffset;
                          MemoryStream.CopyFrom(FileStream, ThumbLength); //
copy from file Stream
                          MemoryStream.Position := 0;
                          ReaderJPEG := TFPReaderJPEG.Create;
                          try
                            ReaderJPEG.Performance := jpBestQuality; //
jpBestSpeed;
                            Image := TFPMemoryImage.create(0,0);
                            Image.LoadFromStream(MemoryStream, ReaderJPEG); //
write Stream to Image
                          finally
                            ReaderJPEG.Free;
```

```
                    end;
                  end;
                finally
                  MemoryStream.Free;
                end;
                if assigned(Image) then
                begin
                  if (Image.Width/Image.Height) = (Width/Height) then //
same ascpect ratio
                  begin
                    Width := Image.Width;
                    Height := Image.Height;
                  end
                  else if (Image.Width/Image.Height) > (Width/Height)
then // thumbnail wider
                  begin
                    Width := Ceil(Image.Height * (Width/Height));
                    Height := Image.Height;
                  end
                  else // thumbnail higher
                  begin
                    Height := Ceil(Image.Width / (Width/Height));
                    Width := Image.Width;
                  end;
                  if not((Abs(Abs(Image.Width)-Abs(Width))>1) or
(Abs(Abs(Image.Height)-Abs(Height))>1)) then
                  begin
                    Height := Image.Height;
                    Width := Image.Width;
                  end;

                  RetCanvas := TFPImageCanvas.Create(RetImage); // canvas
for dest image
                  Canvas := TFPImageCanvas.Create(Image); // canvas for
thumbnail without borders
                  try
                    copyRect.Top := (Image.Height - Height) DIV 2;
                    copyRect.Left := (Image.Width - Width) DIV 2;
                    copyRect.Right := Image.Width - copyRect.Left;
                    copyRect.Bottom := Image.Height - copyRect.Top;
                    case Orientation of
                    $1: begin // copy clip version
                          RetImage.Width := Width;
                          RetImage.Height := Height;
                          RetCanvas.CopyRect(0, 0, Canvas, copyRect); //
copy clip version
                        end;
                    $2: begin // flip horizontal
                          RetImage.Width := Width;
                          RetImage.Height := Height;
```

```
                        RetCanvas.FlipRectHorizontal(0, 0, Canvas,
copyRect);
                    end;
                  $3: begin // rotate 180
                        RetImage.Width := Width;
                        RetImage.Height := Height;
                        RetCanvas.CopyRect180(0, 0, Canvas, copyRect);
                    end;
                  $4: begin // flip vertical
                        RetImage.Width := Width;
                        RetImage.Height := Height;
                        RetCanvas.FlipRectVertical(0, 0, Canvas,
copyRect);
                    end;
                  $5: begin // transpose
                        RetImage.Width := Height;
                        RetImage.Height := Width;
                        RetCanvas.TransposeRect(0, 0, Canvas,
copyRect);
                    end;
                  $6: begin // rotate 90
                        RetImage.Width := Height;
                        RetImage.Height := Width;
                        RetCanvas.CopyRect90(0, 0, Canvas, copyRect);
                    end;
                  $7: begin // transverse
                        RetImage.Width := Height;
                        RetImage.Height := Width;
                        RetCanvas.TransverseRect(0, 0, Canvas,
copyRect);
                    end;
                  $8: begin // rotate 270
                        RetImage.Width := Height;
                        RetImage.Height := Width;
                        RetCanvas.CopyRect270(0, 0, Canvas, copyRect);
                    end;
                  else begin
                    RetImage.Width := Width;
                    RetImage.Height := Height;
                    RetCanvas.CopyRect(0, 0, Canvas, copyRect);
                  end;
                end;
                result := true;
              finally
                RetCanvas.Free;
                Canvas.Free
              end;
            end;
            Image.Free;
          end;
```

```
                end;
                endloop := true;
              end;
            end;
          $DA: endloop := true; // Start Of Scan (begins compressed data)
          $D9: endloop := true; // EOI - End of image (end of datastream);
          end;
          FileStream.Position := SegOffset + segment.DataSize - 2; // skip
to next segment
        end;
//      except on E: Exception do
//{$ifdef debug}writeln('GetJPGThumbFromFile: ' + E.Message + #$0D#$0A);
{$endif}
//      end;
    finally
      FileStream.Free;
    end;
  end;
end;

function GetPNGSize(const AFilename: string; out wWidth, wHeight: word):
boolean;
type TPNGSig = array[0..7] of byte;
const ValidSig: TPNGSig = ($89, $50, $4e, $47, $0d, $0a, $1a, $0a);
var
  FileStream: TFileStream;
  Sig: TPNGSig;
begin
  result := false;
  wWidth := 0;
  wHeight := 0;
  FileStream := TFileStream.Create(AFilename, fmOpenRead or
fmShareDenyNone);
  try
    FileStream.Seek(0, soFromBeginning);
    FillChar(Sig, SizeOf(TPNGSig), #0);
    FileStream.Read(Sig[0], SizeOf(TPNGSig));
    if CompareMem(@Sig[0], @ValidSig[0], SizeOf(TPNGSig)) then
    begin
      FileStream.Seek(18, soFromBeginning);
      FileStream.Read(wWidth, 2);
      wWidth := swap(wWidth);
      FileStream.Seek(22, soFromBeginning);
      FileStream.Read(wHeight, 2);
      wHeight := swap(wHeight);
      result := true;
    end;
  finally
    FileStream.Free;
  end;
```

```
end;

function GetGIFSize(const AFilename: string; out wWidth, wHeight: word):
boolean;
type
  TGifHeader = record
    Signature: array [0..5] of char;
    Width, Height: word;
    Flags, Background, Aspect: byte;
  end;
  TGIFImageBlock = record
    Left, Top, Width, Height: word;
    Flags: byte;
  end;
var
  FileStream: TFileStream;
  Header: TGifHeader;
  ImageBlock: TGifImageBlock;
  Seg: byte;
  i: integer;
begin
  result := false;
  wWidth := 0;
  wHeight := 0;
  FileStream := TFileStream.Create(AFilename, fmOpenRead or
fmShareDenyNone);
  try
    FillChar(Header, SizeOf(TGifHeader), #0);
    FileStream.Seek(0, soFromBeginning);
    FileStream.ReadBuffer(Header, SizeOf(TGifHeader));
    if (AnsiUpperCase(Header.Signature) = 'GIF89A') or
      (AnsiUpperCase(Header.Signature) = 'GIF87A') then
    begin
      wWidth  := Header.Width;
      wHeight := Header.Height;
      result := true;
    end;
    if not result and ((Header.Flags and $80) > 0) then
    begin
      i := 3 * (1 SHL ((Header.Flags and 7) + 1));
      FileStream.Seek(i, soFromBeginning);
      FillChar(ImageBlock, SizeOf(TGIFImageBlock), #0);
      while (FileStream.Position<FileStream.Size) and not result do
      begin
        Seg := FileStream.ReadByte();
        if Seg = $2c then
        begin
          FileStream.ReadBuffer(ImageBlock, SizeOf(TGIFImageBlock));
          wWidth  := ImageBlock.Width;
          wHeight := ImageBlock.Height;
```

```
          result := true;
        end;
      end;
    end;
  finally
    FileStream.Free;
  end;
end;

function GetBMPSize(const AFilename: string; out wWidth, wHeight: word):
boolean;
type
  TBitmapFileHeader = packed record
    ID: word;
    FileSize: dword;
    Reserved: dword;
    BitmapDataOffset: dword;
  end;

  TBitmapInfo = packed record
    BitmapHeaderSize: dword;
    Width: dword;
    Height: dword;
    Planes: word;
    BitsPerPixel: word;
    Compression: dword;
    BitmapDataSize: dword;
    XpelsPerMeter: dword;
    YPelsPerMeter: dword;
    ColorsUsed: dword;
    ColorsImportant: dword;
  end;
var
  FileStream: TFileStream;
  BitmapFileHeader: TBitmapFileHeader;
  BitmapInfo: TBitmapInfo;
  IDStr: String;
begin
  result := false;
  wWidth := 0;
  wHeight := 0;
  FileStream := TFileStream.Create(AFilename, fmOpenRead or
fmShareDenyNone);
  try
    FileStream.Seek(0, soFromBeginning);
    FileStream.ReadBuffer(BitmapFileHeader, SizeOf(TBitmapFileHeader));
    FileStream.ReadBuffer(BitmapInfo, SizeOf(TBitmapInfo));
    IDStr := Char(Lo(BitmapFileHeader.ID)) + Char(Hi(BitmapFileHeader.ID));
    if (not (IDStr = 'BM') or (IDStr = 'BA')) or
      (not (BitmapInfo.BitmapHeaderSize in [$28,$0c,$f0])) or
```

```
      (not (BitmapInfo.BitsPerPixel in [1,4,8,16,24,32])) then Exit;
    wWidth := BitmapInfo.Width;
    wHeight := BitmapInfo.Height;
    result := true;
  finally
    FileStream.Free;
  end;
end;

function GetImageSize(const AFilename: string; out wWidth, wHeight: word):
boolean;
var ext: string;
begin
  result := false;
  wWidth := 0;
  wHeight := 0;
  if FileExists(AFilename) then
  begin
    ext := LowerCase(ExtractFileExt(AFilename));
    if (ext='.jpg') or (ext='.jpeg') then result := GetJPGSize(AFilename,
wWidth, wHeight)
    else if ext='.png' then result := GetPNGSize(AFilename, wWidth, wHeight)
    else if ext='.gif' then result := GetGIFSize(AFilename, wWidth, wHeight)
    else if ext='.bmp' then result := GetBMPSize(AFilename, wWidth, wHeight)
  end;
end;

function GetImageDimensions(const AFilename: string): string;
var width, height: word;
begin
  result := '';
  try
    GetImageSize(AFilename, width, height);
    result := IntToStr(width) + ' x ' + IntToStr(height);
  except
    result := '? x ?';
  end;
end;

function scaleImageToStream(const AFilename: string; var AMimeType: string;
  var MemoryStream: TMemoryStream;
  const maxWidth, maxHeight: word; const crop: boolean = false;
  const focusX: word = 50; const focusY: word = 50): boolean;
type TImgType = (itJPEG, itGIF, itPNG {$ifdef USE_WebP}, itWebP{$endif});
var
  wWidth, wHeight, wcWidth, wcHeight: word;
  x, y: integer;
  sourceAspectRatio, destAspectRatio: real;
  it: TImgType;
  Image, DestImage: TFPMemoryImage;
```

```
  Canvas: TFPImageCanvas;
  Reader: TFPCustomImageReader;
  Writer: TFPCustomImageWriter;
  takeThumb: boolean;
  sExt: string;
begin
  result := false;
  case AMimeType of
    'image/jpeg': it := itJPEG;
    'image/png':  it := itPNG;
    'image/gif':  it := itGIF;
{$ifdef USE_WebP}
    'image/webp': it := itWebP;
{$endif}
    else it := itJPEG;
  end;
  wWidth := maxWidth;
  wHeight := maxHeight;
  Image := TFPMemoryImage.Create(0, 0);
  try
    sExt := LowerCase(ExtractFileExt(AFilename));
    case sExt of
    '.jpg', '.jpeg': Reader := TFPReaderJPEG.Create;
    '.png': Reader := TFPReaderPNG.Create;
    '.gif': Reader := TFPReaderGIF.Create;
    '.bmp': Reader := TFPReaderBMP.Create;
    '.tif': Reader := TFPReaderTIFF.Create;
{$ifdef USE_WebP}
    '.webp': Reader := TFPReaderWebP.Create;
{$endif}
    end;
    try
      takeThumb := false;
      if ((sExt = '.jpg') or (sExt = '.jpeg')) and (wWidth <= 160) and
(wHeight <= 160) then
        takeThumb := GetJPGThumbFromFile(AFilename, Image);
      if not takeThumb then
      begin
        if (sExt = '.jpg') or (sExt = '.jpeg') then
        begin
          TFPReaderJPEG(Reader).Performance := jpBestQuality; //
jpBestSpeed;
          TFPReaderJPEG(Reader).MinHeight := wHeight;
          TFPReaderJPEG(Reader).MinWidth := wWidth;
          Image.LoadFromFile(AFilename, Reader);
        end
        else
          Image.LoadFromFile(AFilename, Reader);
      end;
    finally
```

```
    Reader.Free;
  end;
  if (wWidth = 0) then wWidth := Image.Width;
  if (wHeight = 0) then wHeight := Image.Height;
  // Scale image whilst preserving aspect ratio
  sourceAspectRatio := Image.Width / Image.Height;
  destAspectRatio := wWidth / wHeight;
  wcWidth := wWidth;
  wcHeight := wHeight;
  x := 0;
  y := 0;
  if crop then
  begin
    if sourceAspectRatio > destAspectRatio then
    begin
      wcWidth := Round(wcHeight * sourceAspectRatio);
      x := Round((wWidth/2) - (wcWidth * focusX/100));
      if x > 0 then
        x := 0
      else if x < wWidth - wcWidth then
        x := wWidth - wcWidth;
    end
    else if sourceAspectRatio < destAspectRatio then
    begin
      wcHeight := Round(wWidth / sourceAspectRatio);
      y := Round((wHeight/2) - (wcHeight * focusY/100));
      if y > 0 then
        y := 0
      else if y < wHeight - wcHeight then
        y := wHeight - wcHeight;
    end;
  end
  else
  begin
    if sourceAspectRatio > destAspectRatio then
    begin
      wHeight := Round(wWidth / sourceAspectRatio);
      wcHeight := wHeight;
    end
    else if sourceAspectRatio < destAspectRatio then
    begin
      wWidth := Round(wHeight * sourceAspectRatio);
      wcWidth := wWidth;
    end;
  end;

  DestImage := TFPMemoryImage.Create(wWidth, wHeight);
  try
    Canvas := TFPImageCanvas.Create(DestImage);
    try
```

```
        if (wHeight = Image.Height) and (wWidth = Image.Width) then
          Canvas.Draw(x, y, Image)
        else
          Canvas.StretchDraw(x, y, wcWidth, wcHeight, Image);
      finally
        FreeAndNil(Canvas);
      end;
      case it of
      itJPEG: Writer := TFPWriterJPEG.Create;
      itPNG: Writer := TFPWriterPNG.Create;
      itGIF: Writer := TFPWriterGIF.Create;
{$ifdef USE_WebP}
      itWebP: Writer := TFPWriterWebP.Create;
{$endif}
      end;
      try
        case it of
          itJPEG: begin
                    TFPWriterJPEG(Writer).CompressionQuality := 95;
                    TFPWriterJPEG(Writer).ProgressiveEncoding := true;
                    AMimeType := 'image/jpeg';
                  end;
          itPNG: begin
                    TFPWriterPNG(Writer).UseAlpha := true;
                    TFPWriterPNG(Writer).WordSized := false;
                  end;
{$ifdef USE_WebP}
          itWebP: begin
                    TFPWriterWebP(Writer).QualityPercent := 75;
                    TFPWriterWebP(Writer).Lossless := false;
                  end;
{$endif}
        end;
        Writer.ImageWrite(MemoryStream, DestImage);
        result := true;
      finally
        Writer.Free;
      end;
    finally
      DestImage.Free;
    end;
  finally
    Image.Free;
  end;
end;

end.
```

Autor: Udo Schmal, veröffentlicht: 03.10.2013, letzte Änderung: 29.08.2024