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.

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 eine Exif 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 gesetzt nicht auf System eigene, ich nutze ich Free Pascal Standard Units aus dem Package**fcl-image** und eine eigene UnitFPWriteGIF (/FPWriteGIF) die fehlt nämlich noch in diesem Package, die von mir geschriebene Unit veröffentliche 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, unmöglich.

imgutils.pas (/code/imgutils.pas?mode=download) Pascal (17,42 kByte) 15.02.2014 14:19

```
// ****************************************************************************
// Title.....: Image utils
//
// Modulname .....: imgutils.pas
// Type .....: Unit
// Author .....: Udo Schmal
// Development Status : 15.02.2014
// Operating System ..: Win32/Win64
// IDE .....: Lazarus
// ****************************************************************************
unit ImgUtils;
{$ifdef fpc}
  {$mode objfpc}
{$endif}
{$H+}

interface
uses
  Classes, SysUtils, FPImage, FPImgCanv, FPReadBMP, FPReadJPEG, FPWriteJPEG,
  FPReadPNG, FPWritePNG, FPReadGIF, FPWriteGIF;

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): boolean;

implementation

function GetJPGSize(const AFilename: string; out wWidth, wHeight: word; out CMYK: boolean): boolean;
var
  FileStream: TFileStream;
  Seg, Sampling: byte;
  Dummy: array[0..2] of byte;
  Len, SOI: word;
begin
  result := false;
  wWidth := 0;
  wHeight := 0;
  CMYK := false;
  FileStream := TFileStream.Create(AFilename, fmOpenRead or fmShareDenyNone);
  try
    FileStream.Seek(0, soFromBeginning);
    FileStream.Read(SOI, 2);
    if SOI=$D8FF then
    while (FileStream.Position<FileStream.Size) and not result do
    begin
      FileStream.Read(Seg, 1);
      if Seg in [$C0, $C1, $C2] then //StartOfFrame
      begin
        FileStream.Read(Dummy[0], 3); // don't need these bytes
        FileStream.Read(wHeight, 2);
        wHeight := swap(wHeight);
        FileStream.Read(wWidth, 2);
        wWidth := swap(wWidth);
        FileStream.Read(Sampling, 1);
        case Sampling of
          3: CMYK := false; // RGB
          4: CMYK := true; // CMYK
        end;
```

```
      result := true;
    end
    else if not (Seg in [$01, $D0, $D1, $D2, $D3, $D4, $D5, $D6, $D7, $FF]) then
    begin
      FileStream.Read(Len, 2);
      Len := swap(Len);
      FileStream.Seek(Len-1, 1);
    end
  end;
 finally
  FileStream.Free;
 end;
end;

function GetPNGSize(const sFile: 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(sFile, 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 sFile: 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(sFile, 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 sFile: 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(sFile, 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;
 CMYK: boolean;
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, CMYK)
  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 ReadThumbFromFile(const FileName: AnsiString; var RetImage: TFPMemoryImage): boolean;
type
  TMarker = record
    Marker: Word; // Marker $FF + Nr (1 byte)
    DataSize: Word; // Data Size
    Data: Array [0..4] of Char; // "Exif" 00, "JFIF" 00 and ets
    Pad: Char; // $00
  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
      case byte(TagType) of
        1,2,6,7: Totalbytesize := ActTag.Count;
        3,8: Totalbytesize := ActTag.Count*2;
        4,9,11: 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: Value := swap32(Value);
        5,10,12: Value := swap32(Value);
        2,7: if Totalbytesize>4 then
               Value := swap32(Value);
      end;
```

```
    end;
  end;

var
  buffer: TMarker;
  TIFFHeader: TTIFFHeader;
  tag: TTag;
  StartOffset, IFD0offset, IFD1Offset, ThumbOffset, EXIFOffset: cardinal;
  SOI, IFD0Length, IFD1Length, EXIFLength: word;
  width, height: cardinal;
  i, ThumbLength: integer;
  PtrBuf: pchar;
  BigEndian: boolean;
  f: File;
  MemoryStream: TMemoryStream;
  Reader: TFPReaderJPEG;
  Image: TFPMemoryImage;
  Canvas, RetCanvas: TFPImageCanvas;
  copyRect : TRect;
begin
  result := false;
  if not FileExists(FileName) then exit;
  AssignFile(f, FileName);
  FileMode := fmOpenRead;
  reset(f, 1);

  BlockRead(f, SOI, 2); // read start of image
  if SOI = $D8FF then // SOI marker FF D8 (Start Of Image) is JPEG
  begin
    BlockRead(f, buffer, 10);
    if buffer.Marker = $E0FF then // JFIF Marker FF E0
    begin
      Seek(f, 20); // skip JFIF Header
      BlockRead(f, buffer, 10);
    end;
    if buffer.Marker = $E1FF then // Application Marker APP1 Exif Section FF E1
    begin
      StartOffset := FilePos(f); // Offset Exif header
      BlockRead(f, TIFFHeader, 4); // TIFFHeader
      BigEndian := (TIFFHeader.ByteOrder = $4D4D); // numeric data stored in reverse order

      BlockRead(f, IFD0Offset, 4); // Offset of IFD0
      if BigEndian then IFD0Offset := swap32(IFD0Offset);

      EXIFOffset := 0;
      Seek(f, StartOffset + IFD0Offset); // start of IFD0
      BlockRead(f, IFD0Length, 2); // number of Tags of IFD0
      if BigEndian then IFD0Length := swap(IFD0Length);
      for i:=1 to IFD0Length do // read IFD0
      begin
        BlockRead(f, tag, sizeof(TTag));
        if BigEndian then customizeTag(tag);
        if tag.TagID = $8769 then // Offset of Exif
          EXIFOffset := tag.Value;
      end;

      BlockRead(f, IFD1Offset, 4); // Offset of IFD1
      if BigEndian then IFD1Offset := swap32(IFD1Offset);

      Width := 0;
      Height :=0;
      if EXIFOffset>0 then // if Exif found
      begin
        Seek(f, StartOffset + EXIFOffset);
        BlockRead(f, EXIFLength, 2); // number of Tags of Exif
        if BigEndian then EXIFLength := swap(EXIFLength);
        for i:=1 to EXIFLength do // read Exif
        begin
          BlockRead(f, tag, sizeof(TTag));
          if BigEndian then customizeTag(tag);
          if tag.TagID = $A002 then // ExifImageWidth
            Width := Tag.Value;
          if tag.TagID = $A003 then // ExifImageHeight
            Height := Tag.Value;
        end;
      end;

      if IFD1Offset>0 then // if IFD1 found
      begin
        Seek(f, StartOffset + IFD1Offset);
        BlockRead(f, IFD1Length, 2); // number of Tags of IFD1
```

```
      if BigEndian then IFD1Length := swap(IFD1Length);
      ThumbOffset := 0;
      ThumbLength := 0;
      for i:=1 to IFD1Length do // read IFD1
      begin
        BlockRead(f, tag, sizeof(TTag));
        if BigEndian then customizeTag(tag);
        if tag.TagID = $0201 then // Thumbnail JPEG Offset
          ThumbOffset := Tag.Value;
        if tag.TagID = $0202 then // Thumbnail JPEG Length
          ThumbLength := Tag.Value;
      end;
      if (ThumbOffset>0) and (ThumbLength>0) then //Thumbnail found
      begin
        Seek(f, StartOffset + ThumbOffset);
        GetMem(PtrBuf, ThumbLength);
        BlockRead(f, PtrBuf^, ThumbLength); // write Thubnail to Buffer
        MemoryStream := TMemoryStream.Create;
        MemoryStream.WriteBuffer(PtrBuf^, ThumbLength); // write Buffer to Stream
        MemoryStream.Position := 0;
        FreeMem(PtrBuf);
        Reader := TFPReaderJPEG.Create;
        Reader.Performance := jpBestQuality;//jpBestSpeed;
        Image := TFPMemoryImage.create(0,0);
        Image.LoadFromStream(MemoryStream, Reader); // write Stream to Image
        MemoryStream.Free;
        Reader.Free;
        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 := trunc(Image.Height * (Width/Height));
            Height := Image.Height;
          end
          else // thumbnail higher
          begin
            Height := trunc(Image.Width / (Width/Height));
            Width := Image.Width;
          end;
          RetCanvas := TFPImageCanvas.Create(RetImage); // canvas for dest image
          RetImage.Width := Width;
          RetImage.Height := Height;
          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;
            RetCanvas.CopyRect(0, 0, Canvas, copyRect); // copy clip version
            result := true;
          finally
            RetCanvas.Free;
            Canvas.Free
          end;
        end;
        Image.Free;
      end;
    end;
  end;
  CloseFile(f);
end;

function scaleImageToStream(const AFilename: string; var AMimeType: string; var MemoryStream: TMemoryStream; const maxWidth, maxHeight: word;
const crop: boolean = false): boolean;
type TImgType = (itJPEG, itGIF, itPNG, itBMP);
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;
```

```
begin
  result := false;
  if (AMimeType = 'image/jpeg') then it := itJPEG
  else if (AMimeType = 'image/png') then it := itPNG
  else if (AMimeType = 'image/gif') then it := itGIF
  else if (AMimeType = 'image/bmp') then it := itBMP
  else it := itJPEG;
  wWidth := maxWidth;
  wHeight := maxHeight;
  Image := TFPMemoryImage.Create(0, 0);
  try
    case it of
    itJPEG: Reader := TFPReaderJPEG.Create;
    itPNG: Reader := TFPReaderPNG.Create;
    itGIF: Reader := TFPReaderGIF.Create;
    itBMP: Reader := TFPReaderBMP.Create;
    end;
    try
      takeThumb := false;
      if (wWidth<=160) and (wHeight<=160) and (it=itJPEG) then
        takeThumb := ReadThumbFromFile(AFilename, Image);
      if not takeThumb then
      begin
        if it = itJPEG then
        begin
          TFPReaderJPEG(Reader).Performance := jpBestQuality; // jpBestSpeed;
          TFPReaderJPEG(Reader).MinHeight := wHeight;
          TFPReaderJPEG(Reader).MinWidth := wWidth;
        end;
        Image.LoadFromFile(AFilename, Reader);
      end;
    finally
      Reader.Free;
    end;

    // no upscaling
    if (wWidth = 0) or (wWidth > Image.Width) then wWidth := Image.Width;
    if (wHeight = 0) or (wHeight > Image.Height) 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(wHeight * sourceAspectRatio);
        x := - Round((wcWidth - wWidth) / 2);
      end
      else if sourceAspectRatio < destAspectRatio then
      begin
        wcHeight := Round(wWidth / sourceAspectRatio);
        y := - Round((wcHeight - wHeight) / 2);
      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;
    itBMP: begin
             Writer := TFPWriterJPEG.Create;
             AMimeType := 'image/jpeg';
           end;
    end;
    try
      case it of
      itJPEG,
      itBMP: begin
               TFPWriterJPEG(Writer).CompressionQuality := 95;
               TFPWriterJPEG(Writer).ProgressiveEncoding := true;
             end;
      itPNG: begin
               TFPWriterPNG(Writer).UseAlpha := true;
               TFPWriterPNG(Writer).WordSized := false;
             end;
      end;
      Writer.ImageWrite(MemoryStream, DestImage);
      result := true;
    finally
      Writer.Free;
    end;
  finally
    DestImage.Free;
  end;
 finally
  Image.free;
 end;
end;

end.
```

Autor: Udo Schmal (https://plus.google.com/107378996518617284564?rel=author) , veröffentlicht: 03.10.2013, letzte Änderung: 03.02.2016