FPWriteGIF

In Free Pascal existiert die Unit FPReadGIF, aber die Unit FPWriteGIF fehlt, hier nun ein Anfang.

Animierte GIFs werden noch nicht unterstützt!

fpwritegif.pas Pascal (25,43 kByte) 06.11.2013 00:30

```
unit FPWriteGIF;
{$mode objfpc}{$H+}
interface

uses Classes, SysUtils, FPImage, FPReadGif;

type TColor = -$7FFFFFFF - 1..$7FFFFFFF;

const
// GIF record separators
  kGifImageSeparator: byte = $2c;
  kGifExtensionSeparator: byte = $21;
  kGifTerminator: byte = $3b;
  kGifLabelGraphic: byte = $f9;
  kGifBlockTerminator: byte = $00;
// LZW encode table sizes
  kGifCodeTableSize = 4096;
// Raw rgb value
  clNone = TColor($1FFFFFFF);
  AlphaOpaque = $FF;
  AlphaTransparent = 0;
  MaxArr = (MaxLongint div Sizeof(integer)) - 1;

type
  APixel8 = array[0..MaxArr] of Byte;
  PAPixel8 = ^APixel8;

  TRGBQuadArray256 = array[0..256] of TFPCompactImgRGBA8BitValue;
  TOpenColorTableArray = array of TColor;
  TColorTableArray = array[0..$FF] of TColor;

  TOctreeNode = class; // Forward definition so TReducibleNodes can be
declared
  TReducibleNodes = array[0..7] of TOctreeNode;
  TOctreeNode = class(TObject)
    IsLeaf: Boolean;
    PixelCount: Integer;
    RedSum, GreenSum, BlueSum: Integer;
    Next: TOctreeNode;
    Child: TReducibleNodes;
    constructor Create(const Level: Integer; var LeafCount: Integer; var
ReducibleNodes: TReducibleNodes);
    destructor Destroy; override;
  end;
```

```
  TFPWriterGIF = class(TFPCustomImageWriter)
  private
    fHeader: TGifHeader;
    fDescriptor: TGifImageDescriptor; // only one image supported
    fGraphicsCtrlExt: TGifGraphicsControlExtension;
    fTransparent: Boolean;
    fBackground: TColor;
    fPixels: PAPixel8;
    fPixelList: PChar; // decoded pixel indices
    fPixelCount: longint; // number of pixels
    fColorTable: TColorTableArray;
    fColorTableSize: integer;

    procedure SaveToStream(Destination: TStream);
  protected
    procedure InternalWrite(Stream: TStream; Img: TFPCustomImage); override;
  public
    constructor Create; override;
    destructor Destroy; override;
  end;

implementation
{$REGION ' - TOctreeNode - '}
constructor TOctreeNode.Create(const Level: Integer; var LeafCount: Integer;
var ReducibleNodes: TReducibleNodes);
var i: Integer;
begin
  PixelCount := 0;
  RedSum := 0;
  GreenSum := 0;
  BlueSum := 0;
  for i := Low(Child) to High(Child) do
    Child[i] := nil;
  IsLeaf := (Level = 8);
  if IsLeaf then
  begin
    Next := nil;
    Inc(LeafCount);
  end
  else
  begin
    Next := ReducibleNodes[Level];
    ReducibleNodes[Level] := Self;
  end
end;

destructor TOctreeNode.Destroy;
var i: Integer;
begin
```

```
  for i := Low(Child) to High(Child) do
    Child[i].Free
end;
{$ENDREGION}

{$REGION ' - TFPWriterGIF. - '}
constructor TFPWriterGIF.Create;
begin
  inherited Create;
end;

destructor TFPWriterGIF.Destroy;
begin
  inherited Destroy;
end;

// save the current GIF definition to a stream object
// at first, just write it to our memory stream fSOURCE
procedure TFPWriterGIF.SaveToStream(Destination: TStream);
var
  LZWStream: TMemoryStream; // temp storage for LZW
  LZWSize: integer; // LZW minimum code size

  // these LZW encode routines sqrunch a bitmap into a memory stream
  procedure LZWEncode();
  var
    rPrefix: array[0..kGifCodeTableSize-1] of integer; // string prefixes
    rSuffix: array[0..kGifCodeTableSize-1] of integer; // string suffixes
    rCodeStack: array[0..kGifCodeTableSize-1] of byte; // encoded pixels
    rSP: integer; // pointer into CodeStack
    rClearCode: integer; // reset decode params
    rEndCode: integer; // last code in input stream
    rCurSize: integer; // current code size
    rBitString: integer; // steady stream of bits to be decoded
    rBits: integer; // number of valid bits in BitString
    rMaxVal: boolean; // max code value found?
    rCurX: integer; // position of next pixel
    rCurY: integer; // position of next pixel
    rCurPass: integer; // pixel line pass 1..4
    rFirstSlot: integer; // for encoding an image
    rNextSlot: integer; // for encoding
    rCount: integer; // number of bytes read/written
    rLast: integer; // last byte read in
    rUnget: boolean; // read a new byte, or use zLast?

    procedure LZWReset;
    var i: integer;
    begin
      for i := 0 to (kGifCodeTableSize - 1) do
      begin
```

```
        rPrefix[i] := 0;
        rSuffix[i] := 0;
      end;
      rCurSize := LZWSize + 1;
      rClearCode := (1 shl LZWSize);
      rEndCode := rClearCode + 1;
      rFirstSlot := (1 shl (rCurSize - 1)) + 2;
      rNextSlot := rFirstSlot;
      rMaxVal := false;
    end;

    // save a code value on the code stack
    procedure LZWSaveCode(Code: integer);
    begin
      rCodeStack[rSP] := Code;
      inc(rSP);
    end;

    // save the code in the output data stream
    procedure LZWPutCode(code: integer);
    var
      n: integer;
      b: byte;
    begin
      // write out finished bytes
      // a literal "8" for 8 bits per byte
      while (rBits >= 8) do
      begin
        b := (rBitString and $ff);
        rBitString := (rBitString shr 8);
        rBits := rBits - 8;
        LZWStream.Write(b, 1);
      end;
      // make sure no junk bits left above the first byte
      rBitString := (rBitString and $ff);
      // and save out-going code
      n := (code shl rBits);
      rBitString := (rBitString or n);
      rBits := rBits + rCurSize;
    end;

    // get the next pixel from the bitmap, and return it as an index into
the colormap
    function LZWReadBitmap: integer;
    var
      n: integer;
      j: longint;
      p: PChar;
    begin
      if (rUnget) then
```

```
    begin
      n := rLast;
      rUnget := false;
    end
    else
    begin
      inc(rCount);
      j := (rCurY * fDescriptor.Width) + rCurX;
      if ((0 <= j) and (j < fPixelCount)) then
      begin
        p := fPixelList + j;
        n := ord(p^);
      end
      else
        n := 0;
      // if first pass, make sure CurPass was initialized
      if (rCurPass = 0) then rCurPass := 1;
      inc(rCurX); // inc X position
      if (rCurX >= fDescriptor.Width) then // bumping Y ?
      begin
        rCurX := 0;
        inc(rCurY);
      end;
    end;
    rLast := n;
    result := n;
  end;

var
  i,n,
  cc: integer; // current code to translate
  oc: integer; // last code encoded
  found: boolean; // decoded string in prefix table?
  pixel: byte; // lowest code to search for
  ldx: integer; // last index found
  fdx: integer; // current index found
  b: byte;
begin
  // init data block
  fillchar(rCodeStack, sizeof(rCodeStack), 0);
  rBitString := 0;
  rBits := 0;
  rCurX := 0;
  rCurY := 0;
  rCurPass := 0;
  rLast := 0;
  rUnget:= false;

  LZWReset;
  // all within the data record
```

```
  // always save the clear code first ...
  LZWPutCode(rClearCode);
  // and first pixel
  oc := LZWReadBitmap;
  LZWPutCode(oc);
  // nothing found yet (but then, we haven't searched)
  ldx := 0;
  fdx := 0;
  // and the rest of the pixels
  rCount := 1;
  while (rCount <= fPixelCount) do
  begin
    rSP := 0; // empty the stack of old data
    n := LZWReadBitmap; // next pixel from the bitmap
    LZWSaveCode(n);
    cc := rCodeStack[0]; // beginning of the string
    // add new encode table entry
    rPrefix[rNextSlot] := oc;
    rSuffix[rNextSlot] := cc;
    inc(rNextSlot);
    if (rNextSlot >= kGifCodeTableSize) then
      rMaxVal := true
    else if (rNextSlot > (1 shl rCurSize)) then
      inc(rCurSize);
    // find the running string of matching codes
    ldx := cc;
    found := true;
    while (found and (rCount <= fPixelCount)) do
    begin
      n := LZWReadBitmap;
      LZWSaveCode(n);
      cc := rCodeStack[0];
      if (ldx < rFirstSlot) then
        i := rFirstSlot
      else
        i := ldx + 1;
      pixel := rCodeStack[rSP - 1];
      found := false;
      while ((not found) and (i < rNextSlot)) do
      begin
        found := ((rPrefix[i] = ldx) and (rSuffix[i] = pixel));
        inc(i);
      end;
      if (found) then
      begin
        ldx := i - 1;
        fdx := i - 1;
      end;
    end;
    // if not found, save this index, and get the same code again
```

```
      if (not found) then
      begin
        rUnget := true;
        rLast := rCodeStack[rSP-1];
        dec(rSP);
        cc := ldx;
      end
      else
        cc := fdx;
      // whatever we got, write it out as current table entry
      LZWPutCode(cc);
      if ((rMaxVal) and (rCount <= fPixelCount)) then
      begin
        LZWPutCode(rClearCode);
        LZWReset;
        cc := LZWReadBitmap;
        LZWPutCode(cc);
      end;
      oc := cc;
    end;
    LZWPutCode(rEndCode);
    // write out the rest of the bit string
    while (rBits > 0) do
    begin
      b := (rBitString and $ff);
      rBitString := (rBitString shr 8);
      rBits := rBits - 8;
      LZWStream.Write(b, 1);
    end;
  end;

var i: integer;
begin
  Destination.Position := 0;
  with fHeader do
  begin
    // write the GIF signature
    // if only one image, and no image extensions, then GIF is GIF87a,
    // else use the updated version GIF98a
    // we just added an extension block; the signature must be version 89a
    Destination.Write(Signature, 3);
    Destination.Write(Version, 3);
    // write the overall GIF screen description to the source stream
    Destination.Write(ScreenWidth, 2); // logical screen width
    Destination.Write(ScreenHeight, 2); // logical screen height
    Destination.Write(Packedbit, 1); // packed bit fields (Global Color
valid, Global Color size, Sorted, Color Resolution)
    Destination.Write(BackgroundColor, 1); // background color
    Destination.Write(AspectRatio, 1); // pixel aspect ratio
    if (Packedbit and $80)>0 then //Global Color valid
```

```
      // write out color gobal table with RGB values
      for i := 0 to fColorTableSize-1 do
        Destination.Write(fColorTable[i], 3);
  end;
  // write out graphic extension for this image
  Destination.Write(kGifExtensionSeparator, 1); // write the extension
separator
  Destination.Write(kGifLabelGraphic, 1); // write the extension label
  Destination.Write(fGraphicsCtrlExt.BlockSize, 1); // block size (always 4)
  Destination.Write(fGraphicsCtrlExt.Packedbit, 1); // packed bit field
  Destination.Write(fGraphicsCtrlExt.DelayTime, 2); // delay time
  Destination.Write(fGraphicsCtrlExt.ColorIndex, 1); // transparent color
  Destination.Write(fGraphicsCtrlExt.Terminator, 1); // block terminator
  // write actual image data
  Destination.Write(kGifImageSeparator, 1);
  // write the next image descriptor shortcut to the record fields
  with fDescriptor do
  begin
    // write the basic descriptor record
    Destination.Write(Left, 2); // left position
    Destination.Write(Top, 2); // top position
    Destination.Write(Width, 2); // size of image
    Destination.Write(Height, 2); // size of image
    Destination.Write(Packedbit, 1); // packed bit field
    // there is no local color table defined we use global
    LZWSize := 8; // the LZW minimum code size
    Destination.Write(LZWSize, 1);
    LZWStream := TMemoryStream.Create; // init the storage for compressed
data
    try
      LZWEncode(); // encode the image and save it in LZWStream
      // write out the data stream as a series of data blocks
      LZWStream.Position := 0;
      while (LZWStream.Position < LZWStream.Size) do
      begin
        i := LZWStream.Size - LZWStream.Position;
        if (i > 255) then i := 255;
        Destination.Write(i, 1);
        Destination.CopyFrom(LZWStream, i);
      end;
    finally
      FreeAndNil(LZWStream);
    end;
    Destination.Write(kGifBlockTerminator, 1); // block terminator
  end;
  Destination.Write(kGifTerminator, 1); // done with writing
end;

procedure TFPWriterGIF.InternalWrite(Stream: TStream; Img: TFPCustomImage);
var
```

```
  CT: TOpenColorTableArray;
  Palette: TList;
  PaletteHasAllColours: Boolean;
  Mappings: array[BYTE, BYTE] of TList;
  Tree: TOctreeNode;
  LeafCount: Integer;
  ReducibleNodes: TReducibleNodes;
  LastColor: TColor;
  LastColorIndex: Byte;

  // convert TFPCustomImage TFPColor to TColor
  function FPColorToTColor(const FPColor: TFPColor): TColor;
  begin
    result := TColor(((FPColor.Red shr 8) and $ff) or (FPColor.Green and
$ff00) or ((FPColor.Blue shl 8) and $ff0000));
  end;

  // try to make color table of all colors
  function MakeColorTableOfAllColors(): Boolean;
  var
    Flags: array[Byte, Byte] of TBits;
    x, y, ci: Cardinal;
    Red, Green, Blue: Byte;
    Cnt: word;
  begin
    result := false;
    // init Flags
    for y := 0 to $FF do
      for x := 0 to $FF do
        Flags[x, y] := nil;
    try
      for ci := 0 to $ff do
        CT[ci] := 0;
      Cnt := 0;
      for y := 0 to Img.Height - 1 do
        for x := 0 to Img.Width - 1 do
        begin
          Red := Byte(Img.Colors[x, y].red shr 8);
          Green := Byte(Img.Colors[x, y].green shr 8);
          Blue := Byte(Img.Colors[x, y].blue shr 8);
          if (Flags[Red, Green]) = nil then
          begin
            Flags[Red, Green] := Classes.TBits.Create;
            Flags[Red, Green].Size := 256;
          end;
          if not Flags[Red, Green].Bits[Blue] then
          begin
            CT[Cnt] := FPColorToTColor(Img.Colors[x, y]);
            if Cnt = $ff then exit;
            inc(Cnt);
```

```
            Flags[Red, Green].Bits[Blue] := true;
          end;
        end;
      result := true;
      PaletteHasAllColours := true;
    finally // free Flags
      for y := 0 to $FF do
        for x := 0 to $FF do
          if Flags[x, y] <> nil then
            FreeAndNil(Flags[x, y]);
    end;
    fColorTableSize := High(CT) + 1;
    for x := 0 to fColorTableSize - 1 do
      fColorTable[x] := CT[x];
    LastColor := clNone;
  end;

  procedure MakeColorTableofReducedColors();
    procedure AddColor(var Node: TOctreeNode; const r, g, b: Byte; const
Level: Integer; var ReducibleNodes: TReducibleNodes);
    const mask: array[0..7] of Byte = ($80, $40, $20, $10, $08, $04, $02,
$01);
    var Index, Shift: Integer;
    begin
      if Node = nil then
        Node := TOctreeNode.Create(Level, LeafCount, ReducibleNodes);
      if Node.IsLeaf then
      begin
        Inc(Node.PixelCount);
        Inc(Node.RedSum, r);
        Inc(Node.GreenSum, g);
        Inc(Node.BlueSum, b)
      end
      else
      begin
        Shift := 7 - Level;
        Index := (((r and mask[Level]) shr Shift) shl 2) or (((g and
mask[Level]) shr Shift) shl 1) or
          ((b and mask[Level]) shr Shift);
        AddColor(Node.Child[Index], r, g, b, Level + 1, ReducibleNodes)
      end
    end;

    procedure ReduceTree(var LeafCount: Integer; var ReducibleNodes:
TReducibleNodes);
    var
      RedSum, BlueSum, GreenSum, Children, i: Integer;
      Node: TOctreeNode;
    begin
      i := 7;
```

```
      while (i > 0) and (ReducibleNodes[i] = nil) do
        dec(i);
      Node := ReducibleNodes[i];
      ReducibleNodes[i] := Node.Next;
      RedSum := 0;
      GreenSum := 0;
      BlueSum := 0;
      Children := 0;
      for i := Low(ReducibleNodes) to High(ReducibleNodes) do
        if Node.Child[i] <> nil then
        begin
          Inc(RedSum, Node.Child[i].RedSum);
          Inc(GreenSum, Node.Child[i].GreenSum);
          Inc(BlueSum, Node.Child[i].BlueSum);
          Inc(Node.PixelCount, Node.Child[i].PixelCount);
          Node.Child[i].Free;
          Node.Child[i] := nil;
          inc(Children)
        end;
      Node.IsLeaf := true;
      Node.RedSum := RedSum;
      Node.GreenSum := GreenSum;
      Node.BlueSum := BlueSum;
      Dec(LeafCount, Children - 1)
    end;

    procedure GetPaletteColors(const Node: TOctreeNode; var RGBQuadArray:
TRGBQuadArray256; var Index: integer);
    var i: integer;
    begin
      if Node.IsLeaf then
      begin
        with RGBQuadArray[Index] do
        begin
          try
            r := Byte(Node.RedSum div Node.PixelCount);
            g := Byte(Node.GreenSum div Node.PixelCount);
            b := Byte(Node.BlueSum div Node.PixelCount);
            a := 0;
          except
            r := 0;
            g := 0;
            b := 0;
            a := 0;
          end;
          a := 0
        end;
        inc(Index);
      end
      else
```

```
    for i := Low(Node.Child) to High(Node.Child) do
      if Node.Child[i] <> nil then
        GetPaletteColors(Node.Child[i], RGBQuadArray, Index)
end;

procedure SetPalette(Pal: array of TColor; Size: integer);
var
  PalSize, i: integer;
  Col: PFPCompactImgRGB8BitValue;
  x, y: Cardinal;
  Red, Green, Blue: Byte;
  Pcol: PInteger;
  DistanceSquared, SmallestDistanceSquared: integer;
  R1, G1, B1: Byte;
begin
  if Size <> -1 then PalSize := Size else PalSize := High(Pal) + 1;
  for i := 0 to PalSize - 1 do
  begin
    GetMem(Col, SizeOf(TFPCompactImgRGB8BitValue));
    Col^.r := Byte(Pal[i]);
    Col^.g := Byte(Pal[i] shr 8);
    Col^.b := Byte(Pal[i] shr 16);
    Palette.Add(Col);
  end;
  for y := 0 to $ff do
    for x := 0 to $ff do
      Mappings[y,x] := nil;
  for y := 0 to Img.Height - 1 do
    for x := 0 to Img.Width - 1 do
    begin
      Red := Byte(Img.Colors[x, y].red shr 8);
      Green := Byte(Img.Colors[x, y].green shr 8);
      Blue := Byte(Img.Colors[x, y].blue shr 8);
      //Small reduction of color space
      dec(Red, Red mod 3);
      dec(Green, Green mod 3);
      dec(Blue, Blue mod 3);
      if (Mappings[Red, Green]) = nil then
      begin
        Mappings[Red, Green] := TList.Create;
        Mappings[Red, Green].Count := 256;
      end;
      if (Mappings[Red, Green].Items[Blue] = nil) then
      begin
        GetMem(Pcol, SizeOf(integer));
        PCol^ := 0;
        SmallestDistanceSquared := $1000000;
        for i := 0 to Palette.Count - 1 do
        begin
          R1 := PFPCompactImgRGB8BitValue(Palette[i])^.r;
```

```
              G1 := PFPCompactImgRGB8BitValue(Palette[i])^.g;
              B1 := PFPCompactImgRGB8BitValue(Palette[i])^.b;
              DistanceSquared := (Red - R1) * (Red - R1) + (Green - G1) *
(Green - G1) + (Blue - B1) * (Blue - B1);
              if DistanceSquared < SmallestDistanceSquared then
              begin
                PCol^ := i;
                if (Red = R1) and (Green = G1) and (Blue = B1) then break;
                SmallestDistanceSquared := DistanceSquared;
              end
            end;
            Mappings[Red, Green].Items[Blue] := PCol;
          end;
        end;
    end;

    procedure DeleteTree(var Node: TOctreeNode);
    var i: integer;
    begin
      for i := Low(TReducibleNodes) to High(TReducibleNodes) do
        if Node.Child[i] <> nil then
          DeleteTree(Node.Child[i]);
      FreeAndNil(Node);
    end;

  var
    i, j, Index: integer;
    QArr: TRGBQuadArray256;
  begin
    PaletteHasAllColours := false;
    Tree := nil;
    LeafCount := 0;
    for i := Low(ReducibleNodes) to High(ReducibleNodes) do
      ReducibleNodes[i] := nil;
    if (Img.Height > 0) and (Img.Width > 0) then
      for j := 0 to Img.Height - 1 do
        for i := 0 to Img.Width - 1 do
        begin
          AddColor(Tree, Byte(Img.Colors[i,j].red shr 8),
Byte(Img.Colors[i,j].green shr 8), Byte(Img.Colors[i,j].blue shr 8), 0,
ReducibleNodes);
          while LeafCount > 256 do
            ReduceTree(LeafCount, ReducibleNodes)
        end;
    Index := 0;
    GetPaletteColors(Tree, QArr, Index);
    for i := 0 to LeafCount - 1 do
      CT[i] := (QArr[i].b shl 16) + (QArr[i].g shl 8) + QArr[i].r;
    fColorTableSize := LeafCount;
    for i := 0 to fColorTableSize - 1 do
```

```
      fColorTable[i] := CT[i];
    LastColor := clNone;
    SetPalette(fColorTable, LeafCount);
    if Tree <> nil then DeleteTree(Tree);
  end;

  procedure ClearMappings;
  var i, j, k: integer;
  begin
    for j := 0 to $FF do
      for i := 0 to $FF do
      begin
        if Assigned(Mappings[i, j]) then
        begin
          for k := 0 to $FF do
            FreeMem(Mappings[i, j].Items[k], SizeOf(TColor));
          Mappings[i, j].Free;
        end;
        Mappings[i, j] := nil;
      end;
  end;

  procedure SetPixel(X, Y: Integer; Value: TColor);
  var
    Val: integer;
    PCol: PInteger;
    R, G, B: byte;
  begin
    if not ((Img.Width >= X) and (Img.Height >= Y) and (X > -1) and (Y > -1))
then exit;
    Val := -1;
    if LastColor = Value then
      Val := LastColorIndex
    else
    begin
      if PaletteHasAllColours then
      begin
        TFPCompactImgRGBA8BitValue(Value).a := 0;
        for Val := 0 to fColorTableSize - 1 do
          if fColorTable[Val] = Value then break;
      end
      else
      begin
        B := Byte(Value shr 16);
        B := B - (B mod 3);
        G := Byte(Value shr 8);
        G := G - (G mod 3);
        R := Byte(Value);
        R := R - (R mod 3);
        Val := -1;
```

```
        if Mappings[R, G] <> nil then
        begin
          PCol := Mappings[R, G].Items[B];
          if PCol <> nil then Val := PCol^;
        end;
      end;
      LastColor := Value;
      LastColorIndex := Val;
    end;
    fPixels^[Y * Img.Width + X] := Val;
  end;

  // find the color within the color table; returns 0..255, -1 if color not
found
  function FindColorIndex(c: TColor): integer;
  var i: integer;
  begin
    i := 0;
    result := -1;
    while (i<fColorTableSize) and (result < 0) do
    begin
      if (fColorTable[i] = c) then result := i;
      inc(i);
    end;
  end;

  function lsb(w: word): byte;
  begin
    result := 0;
    while ((w shr result) and 1) = 0 do inc(result);
  end;

var
  x, y: cardinal;
  i, n, ci: integer;
  b: byte;
  pptr: PChar;
begin
  if not ((Img.Width < 1) or (Img.Height < 1)) then
  try
    fTransparent := false;
    // translate 64bit image to 8bit colortable image
    Palette := TList.Create;
    fColorTableSize := 0;
    SetLength(CT, 256);
    //try to make optimized palette on original Data.
    if not MakeColorTableOfAllColors() then
      MakeColorTableofReducedColors(); // to mutch colors, reduce colors
    GetMem(fPixels, Img.Height * Img.Width);
    for y := 0 to Img.Height - 1 do
```

```
      for x := 0 to Img.Width - 1 do
      begin
        SetPixel(x, y, FPColorToTColor(Img.Colors[x, y]));
        if not fTransparent then
          if Img.Colors[x, y].alpha = AlphaTransparent then
          begin
            fBackground := FPColorToTColor(Img.Colors[x, y]);
            fTransparent := true;
          end;
      end;
    // color count must be a power of 2
    if (fColorTableSize <= 2) then fColorTableSize := 2
    else if (fColorTableSize <= 4) then fColorTableSize := 4
    else if (fColorTableSize <= 8) then fColorTableSize := 8
    else if (fColorTableSize <= 16) then fColorTableSize := 16
    else if (fColorTableSize <= 32) then fColorTableSize := 32
    else if (fColorTableSize <= 64) then fColorTableSize := 64
    else if (fColorTableSize <= 128) then fColorTableSize := 128
    else fColorTableSize := 256;
  finally
    for i := 0 to Palette.Count - 1 do
      FreeMem(Palette[i], SizeOf(TFPCompactImgRGB8BitValue));
    Palette.Clear;
    ClearMappings;
    Palette.Free;
  end;

  // create a new gif image record from the given 8bit colortable image
  with fHeader do
  begin
    Signature := 'GIF';
    Version := '89a';
    ScreenWidth := Img.Width;
    ScreenHeight := Img.Height;
    b := lsb(fColorTableSize)-1;
    Packedbit := (Packedbit and $8F) or (b shl 4); // Color Resolution
    Packedbit := (Packedbit and $F7); // not sorted
    Packedbit := (Packedbit and $F8) or b;
    BackgroundColor := 0;
    Packedbit := Packedbit or $80; // Global Color valid
  end;

  // make a descriptor record, color map for this image, and space for a
pixel list
  with fDescriptor do
  begin
    Left := 0;
    Top := 0;
    Width := Img.Width;
    Height := Img.Height;
```

```
    Packedbit := 0; // or $80 = but non local Color Table; or $40 = but not
interlaced; or $20 but not sorted
  end;

  fPixelList := nil; // make empty pixel list
  fPixelCount := Img.Width * Img.Height;
  fPixelList := allocmem(fPixelCount);
  if (fPixelList = nil) then OutOfMemoryError;
  // and the color table
  // the first call attempts to use all colors in the bitmap
  // if too many colors, the 2nd call uses only most significat 8 bits of
color
  for ci:=0 to fPixelCount-1 do
  begin
    pptr := fPixelList + ci;
    pptr^ := Chr(fPixels^[ci]);
  end;

  // set transparency for this image
  with fGraphicsCtrlExt do
  begin
    BlockSize := 4;
    Packedbit := $00;
    ColorIndex := 0;
    if (fTransparent) then
    begin
      n := FindColorIndex(fBackground);
      if (n < 0) then n := FindColorIndex(fBackground and $00E0E0E0);
      if (n < 0) then n := FindColorIndex(fBackground and $00C0E0E0);
      if (n > -1) then
      begin
        Packedbit := Packedbit or $01; // transparent color given (Packedbit
or $01)
        ColorIndex := n; //transparent color index
      end;
    end;
    DelayTime := 0;
    Terminator := 0; // allways 0
  end;

  SaveToStream(Stream);

  if (fPixelList <> nil) then FreeMem(fPixelList);
  FreeMem(fPixels);
  fPixels := nil;
end;
{$ENDREGION}

initialization
  ImageHandlers.RegisterImageWriter ('GIF Graphics', 'gif', TFPWriterGif);
```

```
end.
```

Anwendungs-Beispiel: Konvertierung PNG nach GIF

WriteGifTest.lpr Pascal (990 Bytes) 31.03.2025 06:58

```
program WriteGifTest;
{$ifdef FPC}
  {$mode objfpc}
{$endif}
{$H+}

uses
  Classes, SysUtils, FPImage, FPReadPNG, FPWriteGIF;

procedure testGifWriter(png, gif: string);
var
  fs : TStream;
  ms: TMemoryStream;
  Image: TFPMemoryImage;
  Reader: TFPCustomImageReader;
  Writer: TFPCustomImageWriter;
begin
  Image := TFPMemoryImage.Create(0, 0);
  try
    fs := TFileStream.Create(png, fmOpenRead or fmShareDenyWrite);
    try
      Reader := TFPReaderPNG.Create();
      try
        Image.LoadFromStream(fs, Reader);
      finally
        Reader.Free;
      end;
    finally
      fs.Free;
    end;
    ms := TMemoryStream.Create();
    try
      Writer := TFPWriterGIF.Create;
      try
        Writer.ImageWrite(ms, Image);
        ms.SaveToFile(gif);
      finally
        Writer.Free;
      end;
    finally
      ms.Free;
    end;
  finally
    Image.Free;
```

```
  end;
end;

begin
  testGifWriter(
    'test/lat_54.93254_lon_8.86996.png',
    'test/lat_54.93254_lon_8.86996.gif'
  );
end.
```

Autor: Udo Schmal, veröffentlicht: 26.08.2014, letzte Änderung: 31.03.2025