The Graphics32 Library

Delphi Classes, Components and Functions for Fast 32-bit Graphics Programming
Version 0.91
March 10, 2000

Alex Denissov

Overview

Graphics32 is a set of classes, components and functions designed for Delphi 5. It allows for fast graphics programming using 32-bit DIBs. Being highly specialized for 32-bit pixel format, it provides fast operations with pixels and graphic primitives and in many cases Graphics32 outperforms the *TCanvas* class.

Some of its features include:

- Fast per-pixel access;
- Bitmap transparency;
- Bitmap alpha blending (including per-pixel alpha blending);
- Pixel and line antialiasing (combined with alpha blending);
- Locking the bitmap for safe multithreading;
- Enhanced scaling of bitmaps;
- Flicker-free image displaying component;
- Multiple transparent layers or sprites;
- A property editor for RGB and alpha channel loading.

Some functions require Windows 98 or Windows 2000 to be installed. Even with a latest version of Windows you may experience some problems with displaying of transparent bitmaps due to obsolete video drivers. It is recommended to update your drivers to the latest available.

There is no MMX, SSE or other processor-specific code included in the current version. Future versions of library probably will have some of that realized.

The library comes with a full source code and pre-compiled examples.

The latest version of Graphics32 may be found on the web site:

http://www.geocities.com/den_alex

License

Installation

Delphi 5 is required in order to install Graphics32 library.

- Unzip the files.
- Select **File** | **Open**... on the menu bar. Set **Files of type** to **Delphi package source**, locate and select the **G32.dpk** file, and click **Open**.
- Check the necessary file paths in **Tools** | **Environment Options** | **Library** | **Library Path**. They should include **G32** directory as well as **$(DELPHI)\Source\Toolsapi** and **$(DELPHI)\Source\Vcl**.
- A package editor window will appear. Click **Compile**, then click **Install**.

Introduction

Many features or Graphics32 are similar to those found in standard *TImage*, *TBitmap* and *TCanvas* classes, however they were rewritten to accelerate and optimize drawing on 32-bit DIBs.

Delphi	Graphics32
TColor	TColor32
TBitmap	TBitmap32
TImage	TImage32

TBitmap32 basically manages a 32-bit device independent bitmap (DIB). It is assignment-compatible with a standard *TBitmap* class, that is the *Assign* method of *TBitmap32* may be used to copy it from a *TBitmap* object and vice-versa. However, *TBitmap32* is not a descendant of *TBitmap* object. For example, it does not provide a canvas for drawing and has different implementation of transparency.

***Note:** The same handle may be employed to attach the internal DIB to the external TCanvas object, this allows using the standard TCanvas methods for drawing.*

If you want to use low-level Windows API functions in your application, the *Handle* property of *TBitmap32* may be of some help.

TImage32 may be considered as an alternative to *TImage* component. It displays a flicker-free graphics and manages sprites. There are several bitmap scaling options supported as well as a new property editor which handles loading of bitmaps with alpha channel.

This documentation includes description of the following items:

- Data types;
- Global procedures and functions;
- *TBitmap32* class;
- *TSprite* and *TSpriteSet* class;
- *TImage32* class.

The TColor32 Type

TColor32 represents an ARGB quad with color components in the following order:

Bits 31…24	Bits 23…16	Bits 15…8	Bits 7…0
A	R	G	B

***Note:** TColor32 has its own property editor, which is capable of displaying semi-transparent colors.*

This order is different from ABGR pixel format used by most Windows API functions and implemented in Delphi as *TColor* type. Several functions are provided to convert colors between the different standards (See *"Color Construction and Conversion"* on page 3).

The alpha channel is responsible for pixel's opacity: zero value corresponds to complete transparency, and the value of 255 corresponds to completely opaque pixels.

New color constants are similar to standard ones: *clBlack32*, *clWhite32*, etc. Do not use Delphi's standard color constants directly, convert them with the *Color32* function instead e.g.:

```
Bitmap32.SetPixel(10, 10, Color32(clBtnFace));
```

Note: There is an issue with color conversion: the appearance of *TBitmap32* filled with the color converted from, for example, *clGray* may be slightly different than Delphi's *TPanel* or some other control filled with the same *clGray* color. It seems to happen due to Windows color profiles which are not applied to DIBs.

Color Functions

This section describes color handling functions which are not members of any class.

Color Construction and Conversion

Color32

function Color32(R, G, B: Byte; A: Byte = $FF): TColor32; **overload**;

This function combines its arguments into a 4-byte *TColor32*.

function Color32(WinColor: TColor): TColor32; **overload**;

The pixel format of 32-bit DIBs (ARGB) is different from that used in standard *TColor* type (ABGR). Some standard windows colors are coded using special constants which should be converted into RGB form. This function provides conversion of *TColor* into *TColor32*.

function Color32(Index: Byte; Palette: PPalette32): TColor32; **overload**;
type PPalette32 = ^TPalette32;
type TPalette32 = **array** [0..255] **of** TColor32;

This function simply picks the color value from the palette.

SEE ALSO: *"The TColor32 Type"* on page 2.

Gray32

function Gray32(Intensity: Byte; Alpha: Byte): TColor32;

The action of Gray32(I, A) is the same as Color32(I, I, I, A). It just works faster.

SEE ALSO: *Color32*.

WinColor

function WinColor(Color32: TColor32): TColor;

Provides conversion of TColor32 value back into TColor. The highest-order byte (Alpha channel) of resulting color is assigned the $FF value.

HSLtoRGB

function HSLtoRGB(H, S, L: Single): TColor32;

Conversion from HSL color space. Each argument should normally be in 0...1 range.

RGBtoHSL

procedure RGBtoHSL(RGB: TColor32; **var** H, S, L : Single);

Conversion from RGB into HSL color space. The H, S and L components are returned in corresponding var parameters ranging from 0 to 1.

Component Access

RedComponent

function RedComponent(Color32: TColor32): Integer;

GreenComponent

function GreenComponent(Color32: TColor32): Integer;

BlueComponent

function BlueComponent(Color32: TColor32): Integer;

AlphaComponent

function AlphaComponent(Color32: TColor32): Integer;

These functions return the value of the corresponding color component ranging from 0 to 255.

Intensity

function Intensity(Color32: TColor32): Integer;

Returns the intensity of the color, which is calculated as

```
I = R * 0.21 + G * 0.71 +B * 0.08;
```

SetAlpha

function SetAlpha(Color32: TColor32; NewAlpha: Integer): TColor32;

Returns the RGB color of the argument with a new alpha.

Color Algebra

The following functions may be used for color combining.

ColorAdd

function ColorAdd(C1, C2: TColor32): TColor32;

Returns the sum of two colors. Each color component: red, green, blue and alpha is added separately and summation results are clamped to fit into [0...255] range.

ColorSub

function ColorSub(C1, C2: TColor32): TColor32;

Subtracts C2 from C1. The resulting color components are clamped to [0...255] range. This involves the alpha channel subtraction.

ColorModulate

function ColorModulate(C1, C2: TColor32): TColor32;

The resulting color is the product of C1 and C2 divided by $FF:

$C_R = C1_R * C2_R$ / $FF;
$C_G = C1_G * C2_G$ / $FF;
$C_B = C1_B * C2_B$ / $FF;
$C_A = C1_A * C2_A$ / $FF.

ColorMax

function ColorMax(C1, C2: TColor32): TColor32;

Returns the maximum of C1 and C2.

ColorMin

function ColorMin(C1, C2: TColor32): TColor32;

Returns the minimum of C1 and C2.

ColorMix

function ColorMix(C1, C2: TColor32; W1, W2: Integer): TColor32;

The resulting color is the blend of C1 and C2 with the applied contribution factors:
C = C1 * W1 / 100 + C2 * W2 / 100.

Gamma Correction for Antialiasing

Pixel and line antialiasing produces much better results with correction of opacities of partially covered pixels. The *SetGamma* procedure generates a lookup table for opacity correction:

SetGamma

procedure SetGamma(Gamma: Single = 0.7);

The default value of 0.7 works fine in most cases, but it may require some changes.

The TBitmap32 Class

TBitmap32 is the most important class in Graphics32 library. It manages a single 32-bit device-independent bitmap (DIB) and provides methods for drawing on it and combining it with other DIBs, device-dependent bitmaps or other objects with the device context (DC).

***Note:** Actually, TBitmap32 is actually not a direct descendant of TPersistent. See the source code for details.*

TBitmap32 is a descendant of the *TPersistent* class. It overrides the *Assign* and *AssignTo* methods to provide a compatibility with standard objects: *TBitmap*, *TPicture*, *TClipboard* in both directions. The design-time streaming to and from ***.dfm** files, inherited from *TPersistent*, is supported, but it is realized differently from streaming with other stream types (See the source code for details).

TBitmap32 does not implement low-level streaming or low level file loading/saving. Instead, it uses streaming methods of temporal *TBitmap* or *TPicture* buffers which are assigned to/from the bitmap. This is an obvious performance penalty, however such approach allows using third-party libraries, which extend *TGraphic* class for various image formats support (TGA, TIFF, GIF, etc.). If you install them, *TBitmap32* will automatically obtain support for new image file formats in design time and in run time.

In addition, *TBitmap32* provides functions to prevent a simultaneous access from several threads to the same data, which is done similar to the thread lock in *TCanvas*. For more information on multithreading and locking, see comments for *Lock* and *Unlock* methods as well as Delphi documentation.

TBitmap32 has several properties and methods which have similar action but may have different arguments or other realization details. They follow the simple naming convention:

Postfix	Details	Example
none	Property or method does not perform range checking of its arguments. All the coordinates should be valid.	DrawLine
S	‘Safe’ version. Validates coordinates. If necessary, clipping of lines etc. is performed.	DrawLineS
T	‘Transparent’ version of the method. Uses the alpha channel of the provided color to blend the drawn primitive with the background pixels. Does not validate coordinates.	DrawLineT
TS	Combines both transparency and coordinates validation.	DrawLineTS
F	Methods with ‘F’ postfix take the coordinates as floating point arguments and provide the antialiasing of the drawn primitive. Does not validate coordinates.	DrawLineF
FS	‘Safe’ version of antialiased methods which performs range checking of its arguments.	DrawLineFS

Properties

AutoAlphaMult

property AutoAlphaMult: Boolean;

When the bitmap has to be drawn with alpha blending (*UseAlpha* property is *True*), *AutoAlphaMult* specifies the format of the bitmap.

***Note:** Alpha blending requires Windows 98 or later to be installed. The latest video drivers are recommended as well.*

Windows API requires the blended bitmap to have premultiplied alpha, which means that the red, green and blue channel values in the bitmap must be premultiplied with the alpha channel value. That is, if the alpha value is *x*, the color channels must be multiplied by *x* and divided by $FF before call.

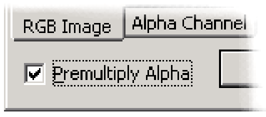
Normally, you operate bitmaps without premultiplied alpha, and perform alpha multiplication just before blending the bitmap to a screen, or to another bitmap. The problem is that premultiplied bitmap should be completely redrawn even if you need to make just a small change in it.

When *AutoAlphaMult* property is *True*, *TBitmap32* creates temporal copy of the bitmap in the memory, automatically premultiplies colors and uses it as a source each time the bitmap should be blended.

You should set *AutoAlphaMult* to *True* if you want to use per-pixel opacity and if colors of pixels were not premultiplied. One of the following strategies may be chosen to display a bitmap with alpha blending:

- Do not use per-pixel alpha blending. If the bitmap has a uniform transparency, reset all the pixels to opaque state with the *ResetAlpha* method and use the *MasterAlpha* property to set the overall bitmap transparency;

```
Bitmap32 := TBitmap32.Create;
... draw something ...
Bitmap32.UseAlpha := True;                    // use alpha channel then painting
Bitmap32.ResetAlpha;                          // set the alpha channel to $FF
Bitmap32.MasterAlpha := 127;                  // set the blending factor to 50%
Bitmap32.Draw(Form.Canvas.Handle, 10, 10);    // paint it to a form
```



- In design time, when loading a bitmap from file, select the ‘Premultiply Alpha’ check box. It will cause automatic conversion of the bitmap while importing it. This approach is useful when you use static images with per-pixel transparency which are not changed in run time;
- Operate the bitmap normally and change its *AutoAlphaMult* property to *True*. Each time the bitmap has to be drawn, its copy is created in memory, then colors of the copy are premultiplied and that data is actually used for drawing. This is the easiest way to implement the per-pixel opacity, but unfortunately the slowest. Each time you draw the bitmap, the buffer is recreated and recalculated;

```
Bitmap32 := TBitmap32.Create;
... draw something ...
Bitmap32.AutoAlphaMult := True;               // automatic pre-multiplication mode
Bitmap32.UseAlpha := True;                    // use alpha channel for painting
Bitmap32.MasterAlpha := 127;                  // set the blending factor to 50%
Bitmap32.Draw(Form.Canvas.Handle, 10, 10);    // paint it to a form
... make some changes to the bitmap ...
Bitmap32.Draw(Form.Canvas.Handle, 10, 10);    // update the image in a form
```

- Construct the bitmap, and before displaying it, call the *PreMultAlpha* method, which will convert it to a necessary state. If you want to make some changes into it, you have to redraw all over again.

```
Bitmap32 := TBitmap32.Create;
... draw something ...
Bitmap32.UseAlpha := True;                    // use alpha channel for painting
Bitmap32.MasterAlpha := 127;                  // set the blending factor to 50%
Bitmap32.PreMultAlpha;                        // pre-multiply opacities
Bitmap32.Draw(Form.Canvas.Handle, 10, 10);    // paint it to a form
... completely redraw the bitmap ...
Bitmap32.PreMultAlpha;                        // pre-multiply opacities
Bitmap32.Draw(Form.Canvas.Handle, 10, 10);    // update the image in a form
```

- Construct two bitmaps: one for drawing and another as a temporal buffer. After some changes have been made in the drawing bitmap, call the *PreMultAlpha* method to copy and convert the data into a temporal buffer, then use it for blending until some changes have to be made to a drawing bitmap. This will work faster than using *AutoAlphaMult* if the bitmap is blended more often than it is altered, because the automatic premultiplication with AutoAlphaMult recreates the buffer every time even if there were no changes made to the bitmap.

```
Bitmap32 := TBitmap32.Create;
Buf := TBitmap32.Create;                      // temporal buffer
... draw something in Bitmap32...
Buf.UseAlpha := True;                         // use alpha channel for painting
Buf.MasterAlpha := 127;                       // set the blending factor to 50%
Buf.PreMultAlpha(Bitmap32);                   // copy data and pre-multiply opacities
Buf.Draw(Form.Canvas.Handle, 10, 10);         // paint it to a form
... make changes to Bitmap32...
Buf.PreMultAlpha(Bitmap32);                   // copy data and pre-multiply opacities
Buf.Draw(Form.Canvas.Handle, 10, 10);         // paint it to a form
```

SEE ALSO: *Draw*, *DrawBlend*, *MasterAlpha*, *PreMultAlpha*, *ResetAlpha*.

Bits

property Bits: PColor32Array; *// read-only*
type PColor32Array = ^TColor32Arra;
type TColor32Array = **array**[0..0] of TColor32;

The bits property contains the address of the first pixel in a bitmap. If the bitmap is not allocated (has zero width or zero height), the returned address is *nil*.

SEE ALSO: *PixelPtr*, *ScanLine*.

Font

property Font: TFont32;

Specifies a current font used by text output functions. The *TFont32* class is the same as *TFont* but it additionally contains an integer *Escapement* property, which specifies the angle, in tenths of degrees, between each character's base line and the *x*-axis of the device.

SEE ALSO: *UpdateFont*.

Handle

property Handle: HDC; *// read-only*

Provides the device handle of the contained DIB. This handle may be used in low-level Windows API calls or, for example, to attach a *TCanvas* object to *TBitmap32*:

```
var
  Canvas: TCanvas;
begin
  Canvas := TCanvas.Create;                 // create a new independent TCanvas object
  try
    Canvas.Handle := Bitmap32.Handle;       // attach it to the Bitmap32 object
    Canvas.Pen.Color := clRed;              // use standard TCanvas methods for drawing
    Canvas.Brush.Color := clGreen;
    Canvas.Ellipse(10, 10, 60, 40);
  finally
    Canvas.Free;
  end;
end;
```

Handle contains zero, if the bitmap is empty (width or height is zero), and its value may be changed after resizing.

Height

property Height: Integer;

Specifies the height of the bitmap in pixels.

SEE ALSO: *Width*, *SetSize*, *Empty*.

LockCount

property LockCount: Integer; *// read-only*

Shows the nesting level of the thread lock. The bitmap is unlocked only when *LockCount* equals 0.

SEE ALSO: *Lock*, *Unlock*.

MasterAlpha

property MasterAlpha: Byte;

When blending a bitmap to the screen or to another bitmap, *MasterAlpha* controls the blending factor. The information stored in the blended bitmap is premultiplied with a *MasterAlpha* value. If the *MasterAlpha* property is $00, the bitmap will be fully transparent, if it is equal to $FF, only per-pixel opacity, stored in bitmap's alpha channel is used. The colors of the bitmap should not premultiplied with *MasterAlpha*.

SEE ALSO: *AutoAlphaMult*, *Draw*, *DrawBlend*.

OuterColor

property OuterColor: TColor32;

This property specifies the color returned by *PixelS* property when reading the pixel with coordinates that lie outside of the bitmap. The default value is $00000000 which corresponds to a fully transparent black.

SEE ALSO: *Pixel*.

Pixel

property Pixel[X, Y: Integer]: TColor32; **default**;
property PixelS[X, Y: Integer]: TColor32;

When writing, the *Pixel* property sets the value of the pixel in the bitmap. Reading it, will return the color value of the pixel located at specified coordinates. This property does not validate the specified coordinates, so use it only then you are completely sure that you are not trying to read from or write to the outside of the bitmap boundary. *Pixel* is declared as default property, you may use it as shown below:

```
Bitmap32[10, 20] := Bitmap32[20, 10];     // copy a pixel from (20, 10) to (10, 20) position
```

PixelS is a 'safe' version of the *Pixel* property. When reading pixels from the outside of the bitmap boundary, the value specified by *OuterColor* is returned. Writing with invalid coordinates will have no effect.

SEE ALSO: *OuterColor*, *SetPixel*.

PixelPtr

property PixelPtr[X, Y: Integer]: PColor32; *// read-only*

Converts coordinates of a pixel to its address in memory. Since *TBitmap32* uses 32-bit device independent bitmaps, its memory is allocated as continuous string of 4-byte TColor32 values, starting at the top left corner.

SEE ALSO: *SetPixel*, *Bits*, *ScanLine*.

ScanLine

property ScanLine[Y: Integer]: PColor32; *// read-only*

Provides indexed access to each line of pixels. Has the same result as *PixelPtr*[0, Y].

SEE ALSO: *Bits*, *PixelPtr*.

Transparent

property Transparent: Boolean;

Determines that pixels with color specified by *TransparentColor* propery has to be transparent when the bitmap is drawn with the *Draw* method. Setting *Transparent* to *True* will automatically reset the *UseAlpha* property. When drawing in transparent mode, the alpha channel of the bitmap is disregarded.

SEE ALSO: *TransparentColor*, *Draw*, *DrawTransparent*, *UseAlpha*.

TransparentColor

property TransparentColor: TColor32;

Specifies a key color used for transparent drawing when the *Transparent* property is set or than *DrawTransparent* method is used. The alpha channel of the color key is not compared with the bitmap pixels. That is colors $FFFFFFFF and $00FFFFFF, for example, are considered to be the same when drawing in transparent mode.

SEE ALSO: *DrawTransparent*, *Transparent*.

UpdateCount

property UpdateCount: Integer; *// read-only*

The current nesting level of the update block. It is increased each time you call the *BeginUpdate* method and is decremented with *EndUpdate* calls. Bitmap does not report changes to its parent (for example, *TImage32* component) as long as *UpdateCount* is greater than 0.

SEE ALSO: *BeginUpdate*, *EndUpdate*.

UseAlpha

property UseAlpha: Boolean;

Specifies that *Draw* method should use blending when drawing the bitmap. Setting this property will automatically reset the *Transparent* property.

SEE ALSO: *Draw*, *DrawBlend*, *AutoAlphaMult*.

Width

property Width: Integer;

Specifies the width of the bitmap in pixels.

SEE ALSO: *Height*, *SetSize*, *Empty*.

Methods

AlphaToGrayscale

procedure AlphaToGrayscale(AlphaSrc: TBitmap32 = nil);

This function converts *AlphaChannel* of an external bitmap referenced by *AlphaSrc* parameter into the grayscale image stored in RGB channels. If *AlphaSrc* is *nil*, the bitmap converts itself. The resulting bitmap has the alpha channel filled with $FF values. If necessary, the destination bitmap is resized to fit *AlphaSrc* dimensions.

This property is useful when you want visualize alpha channel or save it to disk.

SEE ALSO: *ColorToGrayscale*.

BeginUpdate

procedure BeginUpdate;

Increases the *UpdateCount* property and disables the generation of *OnChange* events. Calls to *BeginUpdate* method must be paired with *EndUpdate* calls and they may be safely nested.

SEE ALSO: *EndUpdate*, *UpdateCount*, *OnChange*.

Changed

procedure Changed(Sender: TObject); **virtual**;

Calls the *OnChange* event. If the code is outside the *BeginUpdate*...*EndUpdate* block.

SEE ALSO: *OnChange*, *BeginUpdate*, *EndUpdate*.

Clear

procedure Clear; **overload**;
procedure Clear(FillColor: TColor32); **overload**;

Fills the whole bitmap with *FillColor*. If no argument is specified, clBlack32 ($FF000000) is used.

ColorToGrayscale

procedure ColorToGrayscale(ColorSrc: TBitmap32 = **nil**);

This function converts the image given in *ColorSrc* into a grayscale image. If the source is *nil* (default) then the bitmap converts itself. If necessary, resizing is performed to fit the size of the source bitmap. For example:

```
Bitmap32.ColorToGrayScale(ColorBitmap32);
```

resizes *Bitmap32* to the size of *ColorBitmap32* and puts the grayscale version of *ColorBitmap32* into *Bitmap32*.

The after conversion, the alpha channel contains $FF values.

SEE ALSO: *AlphaToGrayscale*, *IntensityToAlpha*.

Draw

procedure Draw(hDst: HDC); **overload**;
procedure Draw(hDst: HDC; X, Y: Integer); **overload**;
procedure Draw(hDst: HDC; **const** Dst: TRect); **overload**;
procedure Draw(hDst: HDC; **const** Dst, Src: TRect); **overload**;

The *Draw* method draws the bitmap to any object that have the device context. Usually it will be *TCanvas*, or another *TBitmap32*.

If the *UseAlpha* property is *True*, it calls the *DrawBlend* method to blend the bitmap with a background.

If *Transparent* property is *True*, the *DrawTransparent* method will be called to draw the bitmap with some pixels being transparent.

Overwise the bitmap is drawn with *DrawOpaque*.

SEE ALSO: *DrawBlend*, *DrawOpaque*, *DrawTransparent*, *UseAlpha*, *MasterAlpha*, *Transparent*, *PreMultAlpha*.

DrawBlend

procedure DrawBlend(hDst: HDC); **overload**;
procedure DrawBlend(hDst: HDC; X, Y: Integer); **overload**;

procedure DrawBlend(hDst: HDC; **const** Dst: TRect); **overload**;
procedure DrawBlend(hDst: HDC; **const** Dst, Src: TRect); **overload**;

Blends the bitmap with a background, using its per-pixel transparency contained in its alpha channel and *MasterAlpha* to control the overall opacity.

The source format of the bitmap is specified by the *AutoAlphaMult* property. If it is set to *False*, all the bitmap should consist of pixels premultiplied with the alpha channel. If *AutoAlphaMult* is set to *True*, *DrawBlend* will copy and convert data into a temporal buffer which will be used for painting.

SEE ALSO: *Draw*, *UseAlpha*, *AutoAlphaMult*, *MasterAlpha*.

DrawHorzLine

procedure DrawHorzLine(X1, Y, X2: Integer; Value: TColor32);
procedure DrawHorzLineS(X1, Y, X2: Integer; Value: TColor32);
procedure DrawHorzLineT(X1, Y, X2: Integer; Value: TColor32);
procedure DrawHorzLineTS(X1, Y, X2: Integer; Value: TColor32);

Draws a horizontal line from (*X1*,*Y*) to (*X2*, *Y*). The last point is included. These functions works faster compared to *DrawLine*. In versions with ‘S’ postfix necessary clipping to a bitmap coordinate range is provided. The *X1* value should be less than or equal to *X2*.

SEE ALSO: *DrawLine*, *DrawVertLine*.

DrawLine

procedure DrawLine(X1, Y1, X2, Y2: Integer; Value: TColor32);
procedure DrawLineS(X1, Y1, X2, Y2: Integer; Value: TColor32);
procedure DrawLineT(X1, Y1, X2, Y2: Integer; Value: TColor32);
procedure DrawLineTS(X1, Y1, X2, Y2: Integer; Value: TColor32);
procedure DrawLineA(X1, Y1, X2, Y2: Integer; Value: TColor32);
procedure DrawLineAS(X1, Y1, X2, Y2: Integer; Value: TColor32);
procedure DrawLineF(X1, Y1, X2, Y2: Single; Value: TColor32);
procedure DrawLineFS(X1, Y1, X2, Y2: Single; Value: TColor32);

Draws a line from (*X1*,*Y1*) to (*X2*, *Y2*). In versions with ‘S’ postfix necessary line clipping to a bitmap boundary is provided.

DrawLineA and *DrawLineAS* use modified Bresenham’s algorithm for antialiasing, but do not support opacity of a color.

DrawLineF and *DrawLineFS* use my own algorithm for antialiasing. The end points may have floating point coordinates and the line opacity is supported. These methods work approximately 2.5 times slower than *DrawLineA* and *DrawLineAS*.

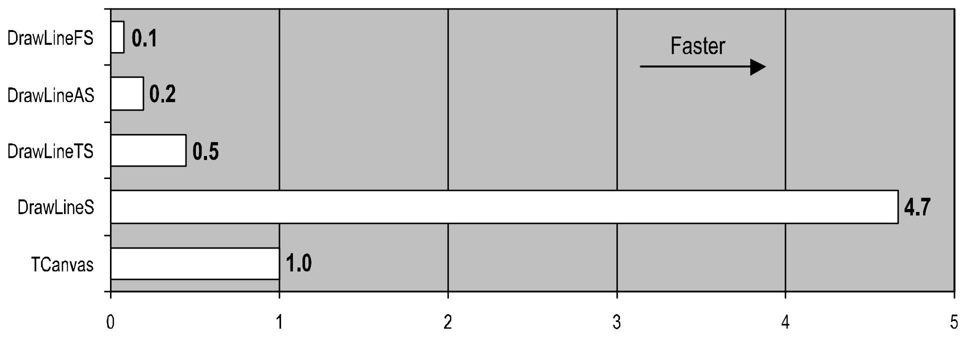
Note: *Delphi does not render the last point in a line.*

All the methods, except *DrawLineF* and *DrawLineFS*, include rendering of the last point with (*X2*, *Y2*) coordinates.

DrawLineF and *DrawLineFS* do not draw the last point in the line. That makes them suitable for drawing of series of connected lines with antialiasing and with floating point coordinates. For example, these functions are ideal for drawing of highly detailed graphs:

```
var
  I: Integer;
  X1, Y1, X2, Y2, H2: Single;
begin
  Bitmap32.Clear(clBlack32);
  H2 := Bitmap32.Height / 2;
  X1 := 0;                                        // store coordinates for the first point
  Y1 := H2 - Func(X1);
  for I := 1 to N - 1 do
  begin
    X2 := 1.5 * I;                                // increment X2 to a next point
    Y2 := H2 - Func(X2);                          // move the origin and invert the y coordinate
    Bitmap32.DrawLineFS(X1, Y1, X2, Y2, clWhite32);  // draw the line
    X1 := X2;
    Y1 := Y2;
  end;
end;
```

Note, that most common line antialiasing algorithms (like modified Bresenham’s) use integer end point coordinates, which would make the curve distorted at such small steps*.



Results of performance tests for different line drawing algorithms.

SEE ALSO: *DrawHorzLine*, *DrawVertLine*.

DrawOpaque

procedure DrawOpaque(hDst: HDC); **overload**;
procedure DrawOpaque(hDst: HDC; X, Y: Integer); **overload**;
procedure DrawOpaque(hDst: HDC; **const** Dst: TRect); **overload**;
procedure DrawOpaque(hDst: HDC; **const** Dst, Src: TRect); **overload**;

DrawOpaque copies the bitmap to the destination specified in *hDst* argument. If destination rectangle, given by *Dst* has different dimensions than size of the source bitmap (or from the rectangle given by *Src*), image will be stretched.

SEE ALSO: *Draw*.

DrawTransparent

procedure DrawTransparent(hDst: HDC); **overload**;
procedure DrawTransparent(hDst: HDC; X, Y: Integer); **overload**;
procedure DrawTransparent(hDst: HDC; **const** Dst: TRect); **overload**;
procedure DrawTransparent(hDst: HDC; **const** Dst, Src: TRect); **overload**;

Draws the bitmap with transparency. That is the pixels having the same color components as given by the *TransparentColor* property are fully transparent. The alpha channel is disregarded.

SEE ALSO: *Draw*, *Transparent*, *TransparentColor*.

DrawVertLine

procedure DrawVertLine(X, Y1, Y2: Integer; Value: TColor32);
procedure DrawVertLineS(X, Y1, Y2: Integer; Value: TColor32);
procedure DrawVertLineT(X, Y1, Y2: Integer; Value: TColor32);
procedure DrawVertLineTS(X, Y1, Y2: Integer; Value: TColor32);

Draws a vertical line from (*X*,*Y1*) to (*X*, *Y2*). The last point is included. These functions works faster compared to *DrawLine*. In versions with 'S' postfix necessary clipping to a bitmap coordinate range is provided. The *Y2* value should be greater or equal to *Y1*.

SEE ALSO: *DrawLine*, *DrawVertLine*.

Empty

function Empty: Boolean;

Returns true if the bitmap is empty, that is both *Width* and *Height* are equal to zero and there is no device context (*Handle* property) allocated.

SEE ALSO: *Width*, *Height*, *Handle*.

EndUpdate

procedure EndUpdate;

Decreases the *UpdateCount* property and enables the generation of *OnChange* events if *UpdateCount* reaches 0.

SEE ALSO: *BeginUpdate*, *UpdateCount*, *OnChange*.

FillRect

procedure FillRect(X1, Y1, X2, Y2: Integer; Value: TColor32);
procedure FillRectS(X1, Y1, X2, Y2: Integer; Value: TColor32);

* In fact, this was the original reason why I started making the Graphics32 library.

procedure FillRectT(X1, Y1, X2, Y2: Integer; Value: TColor32);
procedure FillRectTS(X1, Y1, X2, Y2: Integer; Value: TColor32);

Fills the rectangle with a specified color. Methods with 'S' postfix provide necessary clipping to bitmap boundaries. Unlike TCanvas, TBitmap32 fills the rectangle including the right column (*X2*) and the bottom row (*Y2*).

SEE ALSO: *FrameRect*.

FrameRect

procedure FrameRectS(X1, Y1, X2, Y2: Integer; Value: TColor32);
procedure FrameRectTS(X1, Y1, X2, Y2: Integer; Value: TColor32);

Draws a rectangle. Row with *X2* coordinate and column with *Y2* coordinate are included.

SEE ALSO: *FillRect*.

IntensityToAlpha

procedure IntensityToAlpha(IntensitySrc: TBitmap32 = **nil**);

This function converts intensity stored in IntensitySrc into opacities stored in alpha channel. If *IntensitySrc* is *nil*, the bitmap converts itself. If necessary, the destination bitmap is resized to fit *IntensitySrc* dimensions. Only the alpha channel is changed, RGB channels remain intact as long as the bitmap has the same dimensions as IntensitySrc.

SEE ALSO: *AlphaToGrayscale*, *ColorToGrayscale*.

Invert

procedure Invert(Src: TBitmap32 = **nil**);

This method copies inverted *Src* to the bitmap. If *Src* is *nil*, the the bitmap inverts itself. Inversion is not performed for alpha channel.

LoadFromFile

procedure LoadFromFile(**const** FileName: string);

Loads an image from a file. This method uses a temporal *TPicture* object to load data and will succeed with any format supported by *TPicture*.

SEE ALSO: *LoadFromStream*, *SaveToFile*, *SaveToStream*.

LoadFromStream

procedure LoadFromStream(Stream: TStream);

Loads an image from a stream. This method uses a temporal *TPicture* object to load data and will succeed with any format supported by *TPicture*.

SEE ALSO: *LoadFromFile*, *SaveToStream*, *SaveToFile*.

Lock

procedure Lock;

Blocks other execution threads from using the bitmap until the *Unlock* method is called. If another thread is trying to call a *Lock* method of an object which is already locked, its execution is stalled until the lock is released with *Unlock* method.

Once a thread has locked the object, it can make additional calls to *Lock* method without blocking its execution. This prevents the thread from deadlocking itself while waiting for releasing of a lock that it already owns.

The *LockCount* property is increased each time the *Lock* method is called.

SEE ALSO: *Unlock*, *LockCount*.

PreMultAlpha

procedure PreMultAlpha;

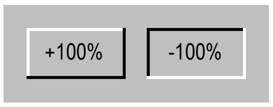
Prepares the bitmap for blending with *DrawBlend* function.

SEE ALSO: *AutoAlphaMult*, *DrawBlend*, *Draw*.

RaiseRectTS

procedure RaiseRectTS(X1, Y1, X2, Y2: Integer; Contrast: Integer);

This function draws a raised or recessed frame. The contrast property is an integer value ranging from –100 to +100.



ResetAlpha

procedure ResetAlpha;

Resets the alpha channel of the entire bitmap to $FF. Call this method after loading a bitmap from a file or after assigning it from a *TBitmap* object, if the *AutoAlphaMult* property is not used for blending.

SEE ALSO: *AutoAlphaMult*, *DrawBlend*.

Resize

procedure Resize(NewWidth, NewHeight: Integer); virtual;

The *Resize* method scales the bitmap to the specified size. In current version of *Graphics32*, it uses Windows API functions for bitmap resizing.

SEE ALSO: *SetSize*, *Height*, *Width*.

SetPixel

procedure SetPixelT(X, Y: Integer; Value: TColor32); **overload**;
procedure SetPixelT(**var** Ptr: PColor32; Value: TColor32); **overload**;
procedure SetPixelTS(X, Y: Integer; Value: TColor32);
procedure SetPixelF(X, Y: Single; Value: TColor32);
procedure SetPixelFS(X, Y: Single; Value: TColor32);

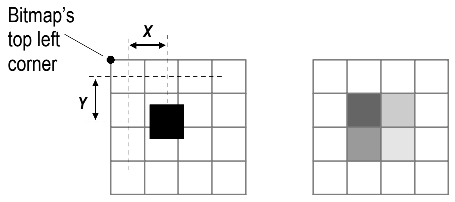
SetPixelT blends the pixel with a bitmap at specified coordinates using the specified color. The pixel's alpha channel is used, but coordinates are not validated.

The overloaded version of *SetPixelT* with a pixel pointer argument allows setting pixels addressed with the pointer rather than with coordinates. The pointer is automatically incremented to a next pixel position each time you call *SetPixelT*, for example:

```
var
  P: PColor32;
  I: Integer;
begin
  { Draw a fading white line from (10, 20) to (265, 20) }
  P := PixelPtr[10, 20];
  for I := 0 to 255 do
    SetPixelT(P, Color32(255, 255, 255, 255 - I));
end;
```

SetPixelTS is the *SetPixelT* method with added coordinate range checking. If pixel coordinates lie outside the bitmap area, *SetPixelTS* does nothing.

SetPixelF and *SetPixelFS* methods provide antialiased rendering of points.



SEE ALSO: *Pixel*, *PixelPtr*.

SaveToFile

procedure SaveToFile(**const** FileName: string);

Writes a bitmap image to disk. The format of the file is compatible with *TBitmap* and *TPicture* objects.

SEE ALSO: *SaveToStream*

SaveToStream

procedure SaveToStream(Stream: TStream);

Stores a bitmap image to a stream. The data in the stream is stored in a form compatible with *TBitmap* and *TPicture* objects.

SEE ALSO: *SaveToFile*

SetSize

procedure SetSize(NewWidth, NewHeight: Integer); **overload**; **virtual**;
procedure SetSize(Source: TBitmap32); **overload**; **virtual**;

Call *SetSize* to set a new width and height of the bitmap. If one of the arguments is zero, the bitmap is considered empty and its *Handle* property is set to zero. Calling *SetSize* works faster than consecutive changing of *Width* and *Height* properties.

If you use another bitmap as an argument, the bitmap will be sized to its dimensions.

If you have an external *TCanvas* attached, refresh it *Handle* property after the bitmap resizing:

```
Bitmap32.SetSize(100, 200);
Canvas.Handle := Bitmap32.Handle;
```

After the *SetSize* call the image will be corrupted and the bitmap should be completely redrawn.

SEE ALSO: *Height*, *Width*, *Resize*, *Handle*.

TextOut

procedure TextOut(X, Y: Integer; **const** Text: string); **overload**;
procedure TextOut(X, Y: Integer; **const** ClipRect: TRect; **const** Text: string); **overload**;
procedure TextOut(ClipRect: TRect; **const** Flags: Cardinal; **const** Text: string); **overload**;

Use *TextOut* to write a string onto the bitmap. The string will be written using the current value of *Font*. Use the *TextExtent* method to determine the space occupied by the text in the image.

TextOut does not support transparent text colors.

The second version performs clipping of a text to the *ClipRect* rectangle.

The last variant provides the most flexible text formatting. See description of *DrawText* function in '*Win32 Developer Reference*' help file for information on *Flags* and their function.

SEE ALSO: *TextExtent*, *TextWidth*, *TextHeight*.

TextExtent

function TextExtent(**const** Text: string): TSize;

Returns the width and height, in pixels, of a string rendered in the current font.

SEE ALSO: *TextHeight*, *TextWidth*, *TextOut*.

TextHeight

function TextHeight(**const** Text: string): Integer;

Returns the width, in pixels, of a string rendered in the current font.

SEE ALSO: *TextExtent*, *TextWidth*, *TextOut*.

TextWidth

function TextWidth(**const** Text: string): Integer;

Returns the height, in pixels, of a string rendered in the current font.

SEE ALSO: *TextExtent*, *TextHeight*, *TextOut*.

Unlock

procedure Unlock;

Decreases the *LockCount* property allowing other threads to access the object when *LockCount* reaches 0. The thread must call *Unlock* once for each time that it locked the object.

SEE ALSO: *Lock*, *LockCount*.

UpdateFont

procedure UpdateFont;

Use this method before calling the Windows API functions that handle text output. It will synchronize the device font object with the *Font* property. You do not have to call *UpdateFont* when using text output methods of *TBitmap32* since they call *UpdateFont* automatically.

SEE ALSO: *Font*.

Events

OnChange **property** OnChange: TNotifyEvent;

This event is called when the bitmap is changed. In order to prevent generation of multiple *OnChange* events than making a lot of changes simultaneously, put *BeginChange* before making any changes and call *EndChange* after it to generate a single *OnChange* event.

Due to performance reasons not all the methods, which change the bitmap, do in fact generate the *OnChange* event. In fact, most of the drawing functions do not.

The table below shows the list of methods and properties of *TBitmap32* which generate *OnChange* event.

OnChange event is generated	
Properties (writing):	Methods:
• Height	• AlphaToGrayScale
• Width	• Assign
	• Clear
	• ColorToGrayscale
	• IntensityToAlpha
	• Invert
	• LoadFromFile
	• LoadFromStream
	• Resize
	• SetSize

OnChange event is not generated	
Properties (writing):	Methods:
• AutoAlphaMult	• CheckerFillRectS
• MasterAlpha	• FillRect[S,T,TS]
• Pixel[S]	• FrameRect[S,T,TS]
• Transparent	• DrawHorzLine[S,T,TS]
• TransparentColor	• DrawLine[S,T,TS,F,FS]
• UseAlpha	• DrawVertLine[S,T,TS]
	• PreMultAlpha
	• RaiseRectTS
	• ResetAlpha
	• SetPixel[S,T,TS,F,FS]
	• TextOut

SEE ALSO: *Changed*, *BeginUpdate*, *EndUpdate*.

The TSprite Class

TSprite is a container for *TBitmap32* object inserted in a *TSpriteSet* structure. It is used in TImage32 component to draw sprites or layers over the background bitmap.

TSprite defines position and visibility of contained bitmap. The rest of the parameters, such as opacity, transparency, dimensions etc., are defined in the contained bitmap object.

TSprite is a descendant of *TCollectionItem* object.

Properties

Bitmap

property Bitmap: TBitmap32;

The *Bitmap* property specifies the bitmap object associated with the sprite.

SEE ALSO: *The TBitmap32 Class*.

Index

property Index: Integer; *// inherited from TCollectionItem*

Determines index of the sprite in a *TSpriteSet* collection as well as the order in which the sprites should be drawn. The sprites with greater *Index* are drawn after sprites with smaller *ones*.

SEE ALSO: *The TSpriteSet Class*.

Position

property Position: TPoint;

Holds coordinates of the sprite object. The location is stored either relative to the background bitmap or relative to the image displaying component.

SEE ALSO: *PositionX*, *PositionY*, *The TImage32 Component*.

PositionX

property PositionX: Integer;

Specifies the horizontal coordinate of the sprite.

SEE ALSO: *PositionY*, *Position*.

PositionY

property PositionY: Integer;

Specifies the vertical coordinate of the sprite.

SEE ALSO: *PositionX*, *Position*.

Visible

property Visible: Boolean;

Determines visibility of the sprite. Those sprites with *Visible* property set to False will not be displayed by the *TImage32* component.

SEE ALSO: *The TImage32 Component*.

Methods

Changed

procedure Changed(Sender: TObject);

When some of the properties of *TSprite* change, this method is called automatically to command to *TImage32*, that it has to be repainted.

The TSpriteSet Class

TSpriteSet is a descendant of Delphi's *TCollection*. It is used by *TImage32* to maintain a list of *TSprite* objects.

Properties

Count **property** Count: Integer; *// read-only; inherited from TCollection*

Returns the number of sprites in *TSpriteSet*.

SEE ALSO: *Items*.

Image **property** Image: TImage32; *// read-only*

Contains the reference to the owner.

SEE ALSO: *The TImage32 Component*.

Items **property** Items[Index: Integer]: TSprite; **default**;

Provides indexed access to stored sprites. The value of the *Index* parameter corresponds to the *Index* property of *TSprite*.

SEE ALSO: *Count*.

Add **function** Add: TSprite;

Creates a new TSprite instance and adds it to the Items array.

Use this method to create an item in the collection in run time. The new item is placed at the end of the Items array, for example:

```
var
  ASprite: TSprite;
  I: Integer;
begin
  Image32.BeginUpdate;                  // temporarily disable image repainting
  for I := 0 to 5 do
  begin
    ASprite := Image32.Sprites.Add;     // Image32.Sprites property holds a TSpriteSet object.
    ASprite.Bitmap.LoadFromFile('Sprite' + IntToStr(I) + '.bmp');
  end;
  Image32.EndUpdate;                    // enable image repainting
end;
```

SEE ALSO: *Items*.

The TImage32 Component

TImage32 is a VCL component which displays *TBitmap32* together with sprites on a form. *TImage32* introduces several properties to determine how the image is displayed within the boundaries of the *TImage32* object.

It is a descendant of the *TCustomControl* class which automatically provides double-buffering capabilities and flicker-free image.

Properties

Bitmap

property Bitmap: TBitmap32;

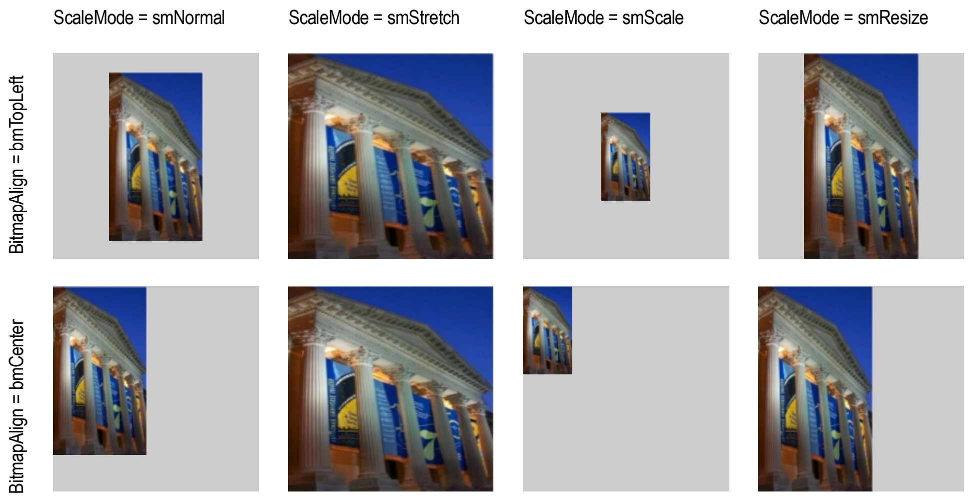
Specifies the bitmap which appears on the *TImage32* control. The same bitmap is used as a background when image contains sprites.

SEE ALSO: *BitmapAlign*, *Scale*, *ScaleMode*, *Sprites*, *SpriteOrigin*.

BitmapAlign

property BitmapAlign: TBitmapAlign;
type TBitmapAlign = (baTopLeft, baCenter);

Specifies *Bitmap* alignment if the image constrol has dimensions different from *Bitmap*. It may be centered (*baCenter*) or their top left corners may be aligned (*baTopLeft*).



Bitmap positioning with ScaleMode and BitmapAlign properties

SEE ALSO: *Bitmap*, *ScaleMode*, *Scale*.

DoubleBuffered

property DoubleBuffered: Boolean; *// inherited from TCustomControl*

Determines whether the image is rendered directly to the window or painted to an in-memory bitmap first. Double buffering reduces the amount of flicker when the control repaints, but is more memory intensive.

Scale

property Scale: Single;

Controls the bitmap scale when the *ScaleMode* is set to *smScale*

SEE ALSO: *ScaleMode*, *Bitmap*, *BitmapAlign*.

ScaleMode

property ScaleMode: TScaleMode;
type TScaleMode = (smNormal, smStretch, smScale, smResize);

Determines how the bitmap is scaled (See the image above).

SEE ALSO: *Scale*, *Bitmap*, *BitmapAlign*.

Sprites **property** Sprites: TSpriteSet;

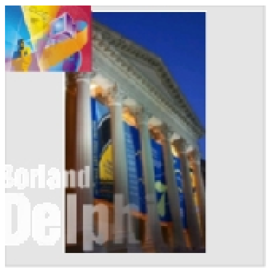
Contains a set of sprites which are drawn over the background image. Each sprite is rendered using the properties of its bitmap.

SEE ALSO: *SpriteOrigin*, *The TSprite Class*, *The TSpriteSet Class*.

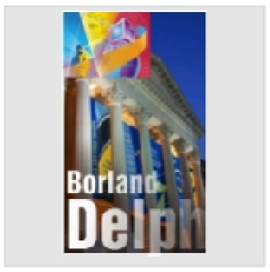
SpriteOrigin **property** SpritesOrigin: TSpriteOrigin;
type TSpriteOrigin = (soImage, soBitmap);

SpriteOrigin controls positioning of sprites inside the image. The sprite origin may be positioned at the top left corner of the *TImage32* component or it may coincide with the top left corner of the background bitmap after its scaling and aligning.

SpriteOrigin = soImage



SpriteOrigin = soBitmap



SEE ALSO: *Sprites*, *Bitmap*.

Methods

BeginUpdate **procedure** BeginUpdate;

Disables the image repainting until the *EndUpdate* method is called. Use *BeginUpdate* when making multiple changes to the image simultaneously, when call *EndUpdate* to repaint the image and to enable further repainting.

BeginUpdate...*EndUpdate* blocks may be nested, only the outermost one re-enables image repainting.

Note, that *TBitmap32* has the same type of update blocking. You may want to use it if you are making changes only to a bitmap itself. The *BeginUpdate* and *EndUpdate* methods of *TImage32* provide blocking of updates when *TImage* properties, such as scale, alignment or sprite positions are changed. The following example moves each sprite 2 pixels right and 1 pixel down:

```
dx := 2;
dy := 1;
Image32.BeginUpdate;                          // disable image repainting
try
  for I := 0 to Image32.Sprites.Count - 1 do
  begin
    Image32.Sprites[I].PositionX := Image32.Sprites[I].PositionX + dx;
    Image32.Sprites[I].PositionY := Image32.Sprites[I].PositionY + dy;
  end;
finally
  Image32.EndUpdate;                          // repaint image and allow further repainting
end;
```

SEE ALSO: *EndUpdate*.

EndUpdate **procedure** EndUpdate;

Re-enables image repainting when bitmap changes. The number of *EndUpdate* calls should match the number of *BeginUpdate* calls.

SEE ALSO: *BeginUpdate*.

SetupBitmap **procedure** SetupBitmap(DoClear: Boolean = False; ClearColor: TColor32 = $FF000000); **virtual**;

SetupBitmap simply sets the size of contained *Bitmap* (in pixels) to the size of an *Image* control, then it may optionally fill the bitmap with specified color. It has the same action as does the next pair or lines:

```
Image32.Bitmap.SetSize(Image32.Width, Image32.Height);
if DoClear then Image32.Bitmap.Clear(ClearColor);
```

Finalization

That's all for now.

Do not forget to send your comments and suggestions, and visit my web page for updated versions of Graphics32, as well as for some other freeware components.

Good luck,

Alex Denissov
denisso@uwindsor.ca
http://www.geocities.com/den_alex