The Graphics32 Library

Delphi Classes, Components and Functions for Fast 32-bit Graphics Programming
Version 0.98
May 17, 2000

Alex Denissov

Overview

Graphics32 is a set of classes, components and functions designed for Delphi. 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 most cases Graphics32 considerably outperforms the *TCanvas* class.

Some of its features include:

- Fast per-pixel access up to 100 times faster compared to *TCanvas*;
- Bitmap alpha blending (including per-pixel alpha blending);
- Pixel, line and polygon antialiasing (combined with alpha blending);
- Locking the bitmap for safe multithreading;
- Enhanced scaling of bitmaps;
- Linear transformations of bitmaps: rotations, scaling, etc with sub-pixel accuracy;
- Flicker-free image displaying components with optimized double buffering;
- Multiple customizible easy-to-use overlay layers;
- A property editor for RGB and alpha channel loading;
- Design-time loading of image formats supported by standard *TPicture*;

Graphics32 comes with a full source code and examples. The latest version of Graphics32 may be found on the web site **http://www.geocities.com/den_alex**

License

Installation

Graphics32 supports Delphi 4 and Delphi 5

Delphi 5

- 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 Graphics32 directory as well as **$(DELPHI)\Source\Toolsapi** and **$(DELPHI)\Source\Vcl**.
- A package editor window will appear. Click **Compile**, then click **Install**.

Delphi 4

- Unzip the files.
- Select **File** | **Open**... on the menu bar. Set **Files of type** to **Delphi package source**, locate and select the **G32_D4.dpk** file, and click **Open**.
- Check the necessary file paths in **Tools** | **Environment Options** | **Library** | **Library Path**. They should include Graphics32 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 in 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.

The current version includes the folloing units:

- Graphics32.pas – basic classes, defines and implements TBitmap32 class;
- Image32.pas – VCL wrappers (TImage32, TLayer32, etc.);
- Blend32.pas – color mixing, blending, multiplying, adding, etc..;
- ByteMaps.pas – multi-purpose 2D byte arrays;
- Transform32.pas – Scaling, linear transformations, etc.;
- Polygons32.pas – Polygon rasterization, etc.;
- Filter32.pas – Image processing engine and filters (this unit is not finished yet);
- Graphics32Reg.pas – design time classes and property/component editors;
- LowLevel32.pas – some low level routines, mainly in-line assembler code.

Except the extended features, Graphics32 has some important differences from standard *TBitmap*, *TImage* and other components. It does not rely on Windows GDI, most of the functions are reimplemented and optimized specifically for 32-bit pixel format. The major exception is *BitBlt* function, which is used to copy the portions of images in opaque mode. I left it since there is a good chance that graphic driver will use hardware acceleration for bitmap blitting, this might be changed in future versions. Another use of GDI functions is to transfer images to/from other Windows objects (screen, printer, non 32-bit bitmaps, device-dependant bitmaps etc).

This documentation includes the following topics:

- Color Types;
- Color Functions;
- TThreadPersistent;
- TCustomMap;
- TBitmap32;
- Polygons and PolyLines;
- TPaintBox32;
- TImage32;
- TLayer32 and TBitmapLayer32;
- TBitmapList32;
- TByteMap;
- Transformations;
- Filters.

Color Types

Color types are defined in Graphics32.pas as follows:

```
TColor32 = type Longword;
PColor32 = ^TColor32;
TColor32Array = array [0..0] of TColor32;
PColor32Array = ^TColor32Array;
TArrayOfColor32 = array of TColor32;
TPalette32 = array [Byte] of TColor32;
PPalette32 = ^TPalette32;
```

TColor32 represents an ARGB color 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 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.

Note: *Do not use standard Delphi color constants with Graphics32, they have different format and they don't support alpha channel.*

Color constants are similar to standard ones: *clBlack32*, *clWhite32*, etc. Do not use *TColor*-typed values with Graphics32 directly, use provided conversion functions, e.g.:

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

Do not confuse *PColor32Array* and *TArrayOfColor32* types while the first holds the pointer to a memory location, the second is a fully functional dynamic array.

TPalette32 types are mostly used to simulate palette-based operations.

Color Functions

This section describes color handling functions, that allow construct colors as well as convert between different color formats. These functions are implemented in Graphics32.pas unit and in Blend32.pas unit.

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: *"Color Types"*.

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 the *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, although the H value is automatically wrapped.

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.

Color 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 weighted 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 a color with the new alpha channel.

Color Blending

The following functions are defined in Blend32.pas. The library automatically determines if CPU supports MMX instructions, and uses MMX optimized routines whenever it is possible.

Note

Calls to the functions described here **must be followed by EMMS procedure**. This procedure restores the state of FPU flags which is altered by MMX instructions. When CPU does not support MMX, the EMMS function will do nothing.

An typical example of using blending functions:

```
try
  for i := X1 to X2 do
  begin
    BlendMem(Clr32, P^); // This function uses MMX, it must be followed by EMMS before using FPU
    Inc(P);
  end;
finally
  EMMS;  // EMMS is called only once, since there is no FPU code inside the loop
end;
```

***Note:** When you receive unexpected FPU exceptions, most likely it means that you forgot to call EMMS instruction after using blending/combining functions*

Color blending and combining functions come in two versions each. The ones with ‘Reg’ postfix take parameters and produce the result operating on CPU registers, while ‘Mem’ versions operate with the background color referenced by a memory address. Using ‘Mem’ functions is more efficient when blending/combining data to a bitmap since it excludes writing operation for transparent pixels.

The color blending functions (*Blend* and *BlendEx*) ignore the background alpha and the alpha of the result is not specified.

Blend

function BlendReg(F, B: TColor32): TColor32;
procedure BlendMem(F: TColor32; **var** B: TColor32);

Mixes a foregrownd (*F*) color with the background color (*B*) using alpha of the foreground color.

$S_{RGB} = F_A * F_{RGB} + (1 – F_A) * B_{RGB}$;

BlendEx

function BlendRegEx(F, B, M: TColor32): TColor32;
procedure BlendMemEx(F: TColor32; **var** B: TColor32; M: TColor32);

Mixes a foregrownd color with the background color using alpha of the foreground color scaled with master alpha value M.

$S_{RGB} = (M * F_A) * F_{RGB} + (1 – (M * F_A)) * B_{RGB}$;

M is defined as *TColor32* do avoid unnecessary type conversions, it should store only the value in [0..255] range, the function does not perform range checking and the result in case *M* > 255 is not specified.

Combine

function Combine(X, Y, W: TColor32): TColor32;
procedure Combine(F: TColor32; **var** B: TColor32; W: TColor32)

Returns the linear interpolation between two colors. The *W* parameter [0..255] specifies the weight of the first color. The alpha channel is interpolated as well.

$S_{RGBA} = W * X_{RGBA} + (1 – W) * Y_{RGBA}$;

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:

$C_{RGB} = C1_{RGB} * C2_{RGB}$;

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*.

Opacity Correction for Antialiasing

Pixel and line antialiasing produces much better results with the correction of opacities of partially covered pixels. This partially accounts for monitor gamma and, in part, for pixel shape correction. This is a simplified approach to antialiasing problem, but it works in most cases. The *SetGamma* procedure generates a lookup table for opacity correction:

SetGamma

procedure SetGamma(Gamma: Single = 0.7);

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

TThreadPersistent

TThreadPersistent extends standard *TPersistent* class with thread-safe locking and declares change notification events. Locking is provided to syncronize simultaneous access in applications with multiple thread and it works similar to that in *TCanvas* class. For additional information, see also Delphi documentation on *TCanva*s.

The class declares change notification abilities. That is, it provides methods and events allowing it descendants to issue notification on their changes. For example, *TBitmap32* uses *OnChange* to notify its container (usually *TImage32* or *TByteMap*) that it was modified and its data should be repainted. *TThreadPersistent*, however, does not implement automatic change notification. It is done in descendants.

Properties

LockCount

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

Shows the current nesting level of the thread lock. *TThreadPersistent* is unlocked only when *LockCount* is 0. Only one thread may lock the object at the time.

SEE ALSO: *Lock*, *Unlock*.

UpdateCount

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

The current nesting level of the update block. It is increased each time you call the *BeginUpdate* method and is decremented with *EndUpdate* calls. The object does not generate *OnChanging* and *OnChange* events as long as its *UpdateCount* is greater than 0.

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

Methods

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: *UpdateCount*.

Changed

procedure Changed; **virtual**;

Calls the *OnChange* event. Descendants of *TThreadPersistent* call *Changing* after making changes to their data or properties.

SEE ALSO: *Changing*, *OnChange*.

Changing

procedure Changing; **virtual**;

Calls the *OnChanging* event. Descendants of *TThreadPersistent* call *Changing* before making changes to their data or properties.

SEE ALSO: *Changed*, *OnChanging*.

EndUpdate

procedure EndUpdate;

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

SEE ALSO:

Lock

procedure Lock;

Blocks other execution threads from locking 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 own 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*.

Unlock **procedure** Unlock;

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

SEE ALSO: *Lock*, *LockCount*.

Events

OnChange **property** OnChange: TNotifyEvent;

Occurs immediately after the object changes. For example, the *TThreadPersistent*'s descendant *TBitmap32*, uses *OnChange* event to notify its parent that something was changed in a bitmap, and the screen image must be updated.

OnChanging **property** OnChanging: TNotifyEvent;

Occurs before the changes are made to the object. This event, for example, may be used to implement 'undo' buffers in your application.

TCustomMap

TCustomMap is a direct descendant of *TThreadPersistent*. It serves as a common ancestor for objects that hold 2D data arrays (*TBitmap32*, *TByteMap*).

SEE ALSO: *"TBitmap32"* on page 9, *"TByteMap"* on page 26.

Properties

Height

property Height: Integer;

Defines the height of the contained data array. Writing into the *Height* property will resize the data array. The exact behavior is defined in descendants. Use *SetSize* methos to change both width and height simultaneously.

SEE ALSO: *Width*, *SetSize*.

Width

property Width: Integer;

Defines the width of the contained data array. Writing into the *Width* property will resize the data array.

SEE ALSO: *Height*, *SetSize*.

Methods

Delete

procedure Delete; **virtual**;

Empty

function Empty: Boolean; **virtual**;

Returns *true* if data set is empty. Usually it means that either its width or its height is 0. Note that even if one of the dimensions is non-zero the bitmap may be

SetSize

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

Simultaneously changes both width and height of data.

I don't know why, but in D4 it is possible to use only the first version.

The second overloaded version 'knows' how to get the size from the following objects or their descendants: *TCustomMap*, *TGraphic*, *TControl* and *nil*. When another parameter is specified *TCustomMap* generates exception.

TBitmap32

TBitmap32 is the most important class in the 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 or other objects with the device context (DC).

TBitmap32 is a descendant of the *TCustomMap* class. It overrides the *Assign* and *AssignTo* methods (inherited from *TPersistent*) to provide 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 its realization is different from streaming with other stream types (See the source code for details).

Note: *References to third-party image import/export libraries may be obtained from my web site.*

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

Since *TBitmap32* is a descendant of *TThreadPersistent*, it inherits its locking mechanism and it may be safely used in multi-threaded applications.

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.	DrawLineT
A	Methods with ‘A’ postfix provide antialiasing of the drawn primitive.	DrawLineA
F	Methods with ‘F’ postfix take the coordinates as floating point arguments and perform antialiasing of the drawn primitive.	DrawLineF
TS, AS, FS	‘Transparent’, ‘antialiased’, and ‘float’ versions combined with clipping or coordinate validation.	DrawLineFS
P	‘Pattern’ version. Usually combined with TS or FS postfix, it may be used to implement various effects like gradient or dashed lines.	DrawLineFSP

Properties

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 (topmost, leftmost) pixel in a bitmap. If the bitmap is not allocated (has zero width or zero height), the returned address is *nil*. Data is continuously allocated in memory, row by row. You may safely access *Width * Height* elements, each of them is a 4-byte *TColor32* value. For example:

```
var
  P: PColor32Array;
begin
  P := Bitmap32.Bits;
  for I := 0 to Bitmap32.Width * Bitmap32.Height - 1 do
    P[I] := Gray32(Random(255));          // fill the bitmap with a random grayscale noise
end;
```

Note that in this code no size verification is required, if width or height is zero, their product is zero and the loop will never be executed.

SEE ALSO: *PixelPtr*, *ScanLine*.

DrawMode

property DrawMode: TDrawMode;
type TDrawMode = (dmOpaque, dmBlend);

Specifies how the bitmap should be combined with a background during pixel transfer and similar operations. In *dmOpaque* mode, new pixels replace the background pixels, in *dmBlend* mode, they are combined using the alpha blending operation. This property is used while copying one bitmap into another, scaling, performing linear transformations etc.

The blending in *dmBlend* mode, is performed using *Blend* or *BlendEx* functions from the Blend32.pas unit.

SEE ALSO: *"Blend"* on page 5.

Font

property Font: TFont;

Specifies a current font used by text output functions.

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; **override**;

Specifies the height of the bitmap in pixels. *Height* is inherited from *TCustomMap*.

SEE ALSO: *Width*, *SetSize*.

MasterAlpha

property MasterAlpha: Byte;

When blending a bitmap to the screen or to another bitmap, *MasterAlpha* controls the blending factor. The per-pixel opacity stored in the blended bitmap is premultiplied with *MasterAlpha*. 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. This property is usen only when *DrawMode = dmBlend* and only for bitmap blending, it does not affect pixel/line drawing and other similar routines.

SEE ALSO: *DrawMode*.

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. It is also used (in the current version) when performing linear transformations of a bitmap.

SEE ALSO: *Pixel*.

PenColor

property PenColor: TColor32;

Simulates *TCanvas.Pen.Color* property. *PenColor* is used exclusively in MoveTo/LineTo functions.

Pixel

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

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.

Benchmarking results (writing pixels, PIII 575MHz, TNT2)

Method	Pixels/second	Description
TBitmap32.Pixel	41 MPixels/s	No coordinate range checking
TBitmap32.PixelS	34 MPixels/s	With coordinate validation
TBitmap32.SetPixelTS	12 MPixels/s	Alpha-blended
TBitmap32.SetPixelFS	2.5 MPixels/s	Antialiased, with alpha blending and opacity correction
TCanvas.Pixels	0.44 MPixels/s	Attached to TBitmap with pf32bit pixel format

As you can see, *TBitmap32* provides access to pixels much faster than that in *TCanvas*. For more information on performance, see G32Bench.xls file.

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 DIBs, 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]: PColor32Array; *// read-only*
type PColor32Array = ^TColor32Array;
type TColor32Array = **array** [0..0] **of** TColor32;

Provides indexed access to each line of pixels. Returns the same address as *PixelPtr*[0, Y]. This property acts similar to *TBitmap's ScanLine*.

SEE ALSO: *Bits*, *PixelPtr*.

StretchFilter

property StretchFilter: TStretchFilter;
type TStretchFilter = (sfNearest, sfLinear, sfSpline);

StretchFilter specifies color interpolation method for image stretching as well as for some other operations, like linear transformations. Some functions (linear transformations, for example) do not distinguish *sfSpline* filter, even if it is specified, instead they will use *sfLinear*.

SEE ALSO: *"StretchTransfer"* on page 28.

Width

property Width: Integer;

Specifies the width of the bitmap in pixels. *Width* is inherited from *TCustomMap*.

SEE ALSO: *Height*, *SetSize*, *"TCustomMap"* on page 8.

Methods

BeginUpdate

procedure BeginUpdate; **override**;

Temporarily prevents generation of *OnChanging* and *OnChange* events. The method is inherited from the *BeginUpdate* method declared in *TThreadPersistent*. In order to re-enable event generation, call *EndUpdate*.

SEE ALSO: *EndUpdate*, *OnChanging*, *OnChange*, *"TThreadPersistent"* on page 6.

Changed

procedure Changed; **override**;

The *Changed* method is called automatically after every change in the bitmap is about to change with a few exceptions (See remarks in *OnChange* description). It is inherited from the *Changed* method of *TThreadPersistent* and provides the same responce – calls the *OnChange* event that is used, for example, by *TImage32* to repaint the bitmap to the screen. In order to prevent multiple repainting of the bitmap while making several changes to it simultaneously, use *BeginUpdate* and *EndUpdate* methods.

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

Changing

procedure Changing; **override**;

Calls the *OnChanging* event when the bitmap is about to be changed.

SEE ALSO: *Changed*, *OnChanging*.

Clear

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

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

Delete

procedure Delete; **override**;

Call *Delete* to free up to destroy the allocated DIB, after the *Delete* call, the bitmap is considered empty, both *Width* and *Height* properties become zeroes.

Draw

procedure Draw(DstX, DstY: Integer; Src: TBitmap32); **overload**;
procedure Draw(DstRect, SrcRect: TRect; Src: TBitmap32); **overload**;
procedure Draw(DstRect, SrcRect: TRect; hSrc: HDC); **overload**;

Renders the image specified by *Src/hSrc* parameter at the location given by the coordinates (*DstX*, *DstY*) or the *DstRect* rectangle.

The method provides both: block transfer (versions with *DstX*, *DstY* parameters) and stretching (versions with *DstRect* parameter).

When the source is another *TBitmap32* object (*Src* parameter), the method uses *Src.DrawMode* do determine how it should be blended with the background, and if stretching, *Src.StretchFilter* specifies how the image should be stretched (For more information see *“BlockTransfer“* and *”StretchTransfer”* on page 28)

The version with *hSrc parameter*, is introduced mainly for compatibility reasons. You may use it to transfer data from bitmaps with other formats, or any other windows objects that have device handle (DC). It is based on *StretchDIBits* GDI call, it does not support transparency and always uses nearest neighbor interpolation when stretching.

The *Dst* parameter must not be necessarily some other bitmap. In fact, it is possible to copy/stretch areas inside the same bitmap that calls the *Draw* method. However, in this case, if source and destination areas intersect, the result is not specified (this is a limitation of the current version).

SEE ALSO: *DrawMode*, *StretchFilter*, *DrawTo*, *“BlockTransfer”* and *”StretchTransfer”* on page 28.

DrawTo

procedure DrawTo(Dst: TBitmap32); **overload**;
procedure DrawTo(Dst: TBitmap32; DstX, DstY: Integer); **overload**;
procedure DrawTo(Dst: TBitmap32; DstRect: TRect); **overload**;
procedure DrawTo(Dst: TBitmap32; DstRect, SrcRect: TRect); **overload**;
procedure DrawTo(hDst: HDC; DstX, DstY: Integer); **overload**;
procedure DrawTo(hDst: HDC; DstRect, SrcRect: TRect); **overload**;

The *DrawTo* method renders the bitmap (or part of it specified bu *SrcRect* parameter) onto another bitmap specified by *Dst/hDst* parameter. It works similar to Draw method but instead of copying data from some other source, the bitmap is renders itselt to destination object. See the *Draw* method description for details.

SEE ALSO: *Draw*.

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);
procedure DrawHorzLineTSP(X1, Y, X2: Integer);

Draws a horizontal line from (*X1*,*Y*) to (*X2*, *Y*). The last point is included. These functions works faster compared to *DrawLine* (In fact, *DrawHorzLine* is the fastest line drawing function in the world :) . In versions with 'S' postfix necessary clipping to a bitmap coordinate range is provided. In versions without 'S' postfix, the *X1* value should be less than or equal to *X2*. The 'TSP' version uses a stipple pattern to vary the color along the line.

SEE ALSO: *DrawLine*, *DrawVertLine*, *"Line Patterns"* on page 17.

DrawLine

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

Draws a line from (*X1*,*Y1*) to (*X2*, *Y2*). Methods with 'S' postfix perform necessary clipping to a bitmap boundary.

DrawLineA and *DrawLineAS* use modified Bresenham's algorithm (also known as Wu's antialiasing), modified to support line transparency and pixel shape/gamma correction.

DrawLineF and *DrawLineFS* use my own algorithm for antialiasing. Line ends have floating point coordinates. These methods work approximately 2–2.5 times slower than *DrawLineA* and *DrawLineAS.*

The last parameter, *L*, determines if the last point (X2,Y2) has to be drawn. It is useful in some cases to leave that point empty, especially when drawing sequences of transparent and/or antialiased lines. By default, *DrawLine* methods don't render the last point.

DrawLineFSP is a version of *DrawLineFS*, which supports color patterns, for more information see *"Line Patterns"* on page 17.

Fore information on performance, see the included G32Bench.xls file. Drawing lines with *TCanvas* is very inefficient for short lines. In fact for lines about 3–4 pixels long even *DrawLineFS* outperforms *TCanvas.*

SEE ALSO: *DrawHorzLine*, *DrawVertLine*.

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);
procedure DrawVertLineTSP(X, Y1, Y2: Integer);

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. In versions without 'S', the *Y2* value should be greater or equal to *Y1*. 'FSP' version supports line patterns.

SEE ALSO: *DrawLine*, *DrawVertLine*, *"Line Patterns"* on page 17.

EndUpdate

procedure EndUpdate; **override**;

EndUpdate re-enables generation of events suppressed by *BeginUpdate* call.

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

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*.

FillRect

procedure FillRect(X1, Y1, X2, Y2: Integer; Value: TColor32);
procedure FillRectS(X1, Y1, X2, Y2: Integer; Value: TColor32);
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, versions without 'S' must be supplied with valid parameters and *X2* >= *X1*; *Y2* >= *Y1*. 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);
procedure FrameRectTSP(X1, Y1, X2, Y2: Integer);

Draws a rectangle. Row with *X2* coordinate and column with *Y2* coordinate are included. 'TSP' version supports line patterns.

SEE ALSO: *FillRect*, *"Line Patterns"* on page 17.

LineTo

procedure LineToS(X, Y: Integer);
procedure LineToTS(X, Y: Integer);
procedure LineToAS(X, Y: Integer);
procedure LineToFS(X, Y: Single);

*LineTo** methods are similar to *TCanvas.LineTo*. The line is drawn from the current raster position, to the position specified in *X* and *Y* parameters **excluding** the last point. Then raster position is shifted to (*X*, *Y*) point. The line is drawn with the color specified in *PenColor* property. '*S*', '*TS*', and '*AS*' versions use and update integer integer raster position, while '*FS*' version uses and updates independent floating point raster position.

To start a new line or sequence of lines, use *MoveTo* methods.

These methods use *DrawLine* functions and are introduced mainly to facilitate porting the *TCanvas*-oriented code.

SEE ALSO: *MoveTo*, *PenColor*, *DrawLine*.

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;

This property is inherited from *TThreadPersistent* class.

SEE ALSO: *"TThreadPersistent"* on page 6.

MoveTo

procedure MoveTo(X, Y: Integer);
procedure MoveToF(X, Y: Single);

Shifts current raster to specified position, analogous to *MoveTo* method of standard *TCanvas*. Each bitmap maintains separate raster positions for integer and for floating point coordinates. Raster position is not affected neither by *DrawLine* methods nor by any other method except *MoveTo* and *LineTo*.

SEE ALSO: *LineTo*, *PenColor*, *DrawLine*.

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.

RenderText

procedure RenderText(X, Y: Integer; **const** Text: string; AALevel: Integer; Color: TColor32);

RenderText method draws a string of characters. This method is much slower compared to *TextOut* functions, however it supports antialiasing and transparency. *AALevel* specifies how the text is antialiased. If it is zero, no antialiasing is performed, the value of 4 corresponds to a maximum quality.

The method draws a string using current *Font*, but it ignores the *Font.Color* property, substituting it with the *Color* parameter.

SEE ALSO: *Font*, *TextOut*.

ResetAlpha

procedure ResetAlpha;

Resets the alpha channel of the entire bitmap to $FF.

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);
```

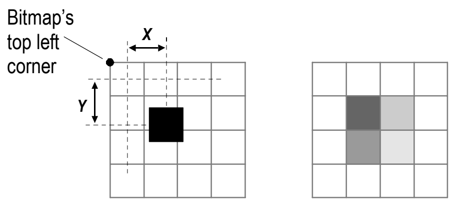
SetPixelT blends the pixel with a bitmap at specified coordinates using the specified color. The pixel's alpha channel is used, but the coordinates are not validated.

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 verification. 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**;
procedure SetSize(Source: TPersistent; **overload**;

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 or control 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*, *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. Note, that the size returned by this function may differ from the actual width of the text produced by *RenderText* function, especially when using raster fonts.

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

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;

Unlock is inherited from the *TThreadPersistent* ancestor.

SEE ALSO: *"TThreadPersistent"* on page 6.

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;

OnChange occurs after changes were made to the bitmap. Or then the program implicitly calls the *Changed* method.

OnChanging

property OnChanging: TNotifyEvent;

OnChange occurs immediately before changes are made to the bitmap. Or then the program implicitly calls the *Changing* method.

Both events are inherited from *TThreadPersistent* ancestor.

Most functions that alter the bitmap do generate *OnChanging* and *OnChange* events. However due to performance considerations some methods and properties do not generate events. Namely:

- Pixel-based operations (*SetPixelT*, *SetPixelF*...);
- DrawHorzLine* and DrawVertLine* functions;

In this case, if necessary, *OnChanging/OnChange* events may be generated by manually calling *Changing* and *Changed* (See *"TThreadPersistent"* on page 6).

If the bitmap is a part of *TPaintBox32*, *TImage32* or some other container, these events are linked to the container to notify it on data changes, so that it 'knows' then to update the screen image. When making several simultaneous changes it may be beneficial to enclose them in *BeginUpdate*...*EndUpdate* block followed by the *Changed* call (and optionally preceeded by *Changing*) so that container repaints only one instead of updating with every change.

SEE ALSO: *Changing*, *Changed*, *"TThreadPersistent"* on page 6.

Line Patterns

Graphics32 defines several functions to support non-uniform lines. That includes gradient lines, dashed lines etc. The idea is pretty simple: *TBitmap32* holds a dynamic array of colors, and a counter which 'crowls' along that array and reads colors. The line drawing algorithm queries the color value from the current counter position at each point, when the counter is automatically incremented to get ready to supply next value to line drawing routine.

The dynamic array of colors is set with *SetStipple*:.

SetStipple

procedure SetStipple(NewStipple: TArrayOfColor32); **overload**;
procedure SetStipple(NewStipple: **array of** TColor32); **overload**;

The counter, referred here as stipple counter, wraps itself automatically on the line edges so that the pattern is a loop from the point of view of a drawing function. Some time it may be useful to be able to reset the counter to its initial position (0), for example, when starting a new gradient line. This is done with *ResetStippleCounter* function:

ResetStippleCounter

procedure ResetStippleCounter;

There is more, the counter is actually not an integer value, it may have a fractional step, and even a negative step:

SetStippleStep **procedure** SetStippleStep(Value: Extended);

The default value for counter step is 1.0. It is possible to change the step dynamically while drawing the line (can’t imagin the reason for that, but why not?).

In order to get the color from the current counter position in the pattern, use *GetStipple-Color* function:

GetStippleColor **function** GetStippleColor: TColor32;

If the counter step is fractional, it interpolates the color between two closest colors in the pattern. *GetStippleColor* automatically changes the counter value by the stipple step.

Line patterns are supported by the following functions:

- *DrawHorzLineTSP;*
- *DrawLineFSP;*
- *DrawVertLineTSP;*
- *FrameRectTSP.*

Polygons and PolyLines

Polygons32.pas unit defines functions for drawing polygons and polylines. Also see the demo project in **Examples\Polygons** directory.

All the functions support transparency and automatically perform clipping.

PolyLine

```
procedure PolyLineTS(Bitmap: TBitmap32; const Points: TArrayOfPoint; Color: TColor32);
procedure PolyLineAS(Bitmap: TBitmap32; const Points: TArrayOfPoint; Color: TColor32);
procedure PolyLineFS(Bitmap: TBitmap32; const Points: TArrayOfPointF; Color: TColor32);
type TArrayOfPoint = array of TPoint;
type TArrayOfPointF = array of TPointF;
type TPointF = record
   X, Y: Single;
end;
```

Draws a series of lines, connecting vertices passed in. The polygon is closed automatically by drawing a line from the last vertex to the first.

Polygon

```
procedure PolygonTS(Bitmap: TBitmap32; const Points: TArrayOfPoint; Color: TColor32);
procedure PolygonAS(Bitmap: TBitmap32; const Points: TArrayOfPoint; Color: TColor32);
procedure PolygonFS(Bitmap: TBitmap32; const Points: TArrayOfPointF; Color: TColor32);
```

Fills the shape defined by *Points* parameter with specified color. Filling works similar to *ALTERNATE* mode (See *SetPolyFillMode* in Windows SDK help).

TPaintBox32

TPaintBox32 is a descendant of the *TWinControl* class with optimized double-buffering. The back buffer is a *TBitmap32* object which stores the image data before flushing it to a screen.

This control is capable of accepting overlay layers (*"TLayer32"* on page 24). However, *TImage32* might be more suitable for rendering layers, if you need to have some background for them.

Since *TPaintBox32* is double buffered, you don't need to draw it every time the control receives WM_PAINT message, as with standard *TPaintBox*. The repainting is still required when it is resized.

The source code for *TPaintBox32* is located in Image32.pas.

Properties

Buffer

property Buffer: TBitmap32

Provides direct access to the back buffer.

SEE ALSO: *"TBitmap32"* on page 9.

SmartResize

property SmartResize: Boolean;

SmartResize specifies how the back buffer is reallocated when the control is resized. When equal to *false* (Default), the back buffer is reallocated each time to fit the new control dimensions.

When set to *true*, the back buffer is allowed to be up to 40 pixels larger then the control in each direction. The buffer reallocation strategy tracks the size changes and generally, it reallocates the buffer 40 times less when the control resizes. You should keep in mind, however, that the buffer may be larger than the control is.

Methods

BeginUpdate

procedure BeginUpdate; **virtual**;

See description of *BeginUpdate* method of *TThreadPersistent*, this method has the same function.

SEE ALSO: *"TThreadPersistent"* on page 6.

ClearBuffer

procedure ClearBuffer; **virtual**;

Cleans the buffer with the color specified by *Color* property, which is inherited from *TWinControl*. Since *Color* is of type *TColor*, not *TColor32*, method uses the *Color32* function to convert it.

If *OnCleanBuffer* event is not empty, the method does not perform the buffer cleaning, it just calls the *OnCleanBuffer* event.

Changed

procedure Changed; **virtual**;

Calls the *OnChange* event immediately before meking changes to a buffer. Works similar to the *Changed* method of *TThreadPersistent*.

SEE ALSO: *"TThreadPersistent"* on page 6.

Changing

procedure Changing; **virtual**;

Calls the *OnChanging* event immediately before making changes to a buffer. Works similar to the *Changing* method of *TThreadPersistent*.

SEE ALSO: *"TThreadPersistent"* on page 6.

EndUpdate **procedure** EndUpdate; **virtual**;

See description of *BeginUpdate* method of *TThreadPersistent*, this method has the same function.

"TThreadPersistent" on page 6.

Events

OnCleanBuffer **property** OnClearBuffer: TNotifyEvent;

OnChanging **property** OnChanging: TNotifyEvent;

OnChange **property** OnChange: TNotifyEvent;

OnPaint **property** OnPaint: TNotifyEvent;

OnPaint is issued every time the control receives WM_PAINT event.

OnResize **property** OnResize: TNotifyEvent;

The resizing of the control is accompanied with the following sequence of events:

- Control changes its size;
- OnChanging;
- Reallocation of the buffer bitmap;
- OnChanged;
- OnResize;
- OnClearBuffer;
- OnPaint.

See the source for details, I'll document the rest of events later.

TImage32

TImage32 is a descendant of *TPaintBox32*, which holds an easy manageable bitmap together with optional overlay layers on a form. It introduces several properties to determine how the image is displayed within the boundaries of the *TImage32* object.

Though it is still possible to interact with the back buffer dirrectly through the inherited *Buffer* property, I wouldn't recommend doing that, since the *TImage32* implements its own change notification strategy, that is different from that in *TPaintBox32*. By default, *TImage32* sets its *SmartResize* property to *true*.

Properties

AutoSize

property AutoSize: Boolean;

Determines if *TImage32* automatically resizes to fit the contained bitmap.

Bitmap

property Bitmap: TBitmap32;

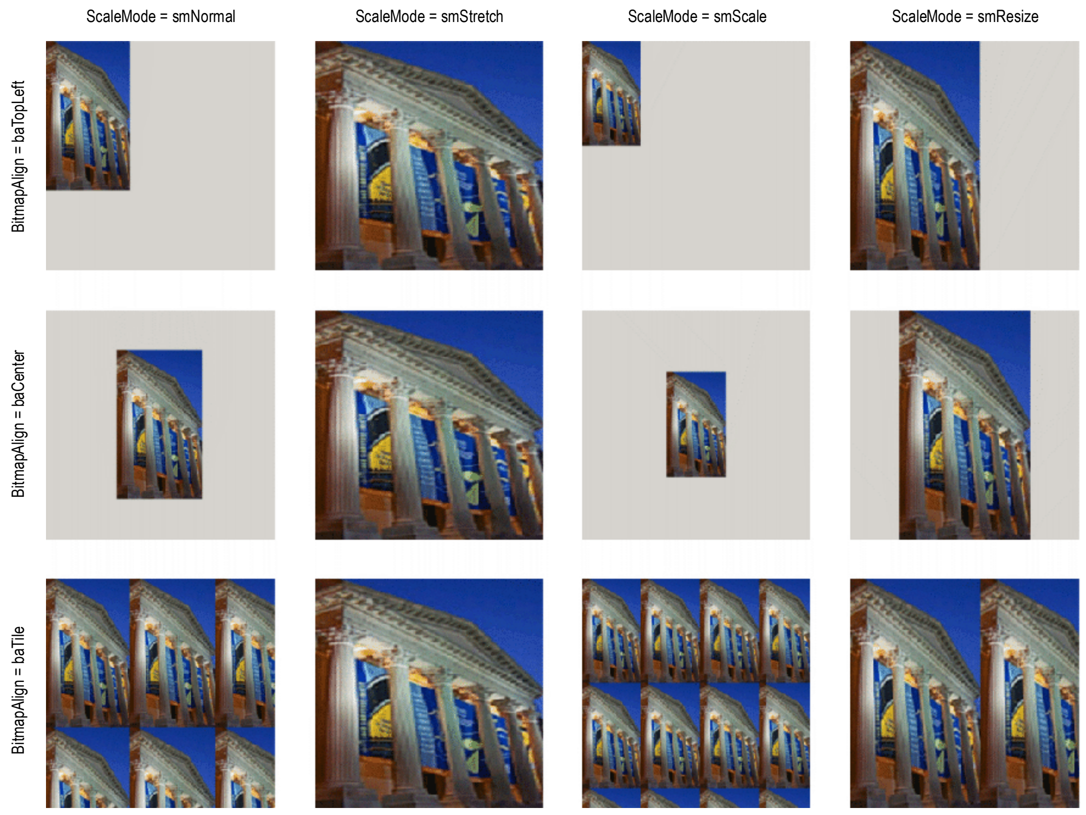
Specifies the bitmap which appears on the *TImage32* control. The actual position of the bitmap within the *TImage32* boundaries may be acquired with the *GetPictureRect* method.

SEE ALSO: *"TBitmap32"* on page 9, *BitmapAlign*, *Scale*, *ScaleMode*, *GetPictureRect*.

BitmapAlign

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

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*) or the bitmap may be tiled (*baTile)*.



Bitmap positioning with ScaleMode and BitmapAlign properties

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

GetPictureRect

function GetPictureRect: TRect;

GetPictureRect returns the boundaries of the picture after scaling and aligning. When the bitmap is aligned in *baTile* mode, the function returns boundaries of the top-left tile.

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

Scale

property Scale: Single;

Controls the bitmap scale when the *ScaleMode* is set to *smScale*. Be carefull when setting *ScaleMode* in tile mode. If it is too small, there will be too much tiles and you may end up waiting for ages while the control repaints each tile.

New bitmap coordinates are converted to integer values after scaling, if you need a sub-pixel accuracy, use linear transformations, described in *"Transformations"* on page 28.

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

ScaleMode

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

Determines how the bitmap is scaled (See the image on the previous page). If the control is in *AutoSize* mode, its width and height will match the original width and height of contained bitmap, except to smScale mode, when both picture and control are resized with account to *Scale* property.

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

Methods

BeginUpdate

procedure BeginUpdate;

Disables the image repainting until the *EndUpdate* method is called. Use *BeginUpdate* when making multiple changes to the image simultaneously, then call *EndUpdate* followed by the *Changed* call, 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 properties, such as scale, alignment or contained overlay layers are changed.

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);
```

TLayer32

TLayer32 is an overlay layer (sprite), which is in essense a non-windowed *TControl* descendant that 'knows' how to paint itself to a backbuffer of *TPaintBox32* or *TImage32* when it is inserted there as a child.

Since it descends from *TControl*, you may use all the standard events of Delphi to manipulate layer's positioning in both design- and run-time. For example, you may find it useful to change the z-order of sprites with Delphi's **EDIT** | **Bring to Front** / **Send to Back** menu commands, etc.

All the standard properties, like Align, Constraints, Anchors, Cursor, PopupMenu, Hint, Visible, etc are compatible with *TLayer32* and its descendants.

TLayer32 is not capable of drawing itself, instead it calls the *OnPaint* event. Similarly, it just declares *OnChanging* on *OnChange* events but never calls them, that is overriden in descendants.

The source of *TLayer32* is located in Image32.pas.

Methods

Changed

procedure Changed; **virtual**;

Updates the screen image and calls *OnChange* event.

Changing

procedure Changing; **virtual**;

Calls the *OnChanging* event.

Events

OnChange

property OnChange: TNotifyEvent;

OnChanging

property OnChanging: TNotifyEvent;

OnPaint

property OnPaint: TPaintEvent;
type TPaintEvent = **procedure**(Sender: TObject; BackBuffer: TBitmap32) **of object**;

The *BackBuffer* parameter references the back buffer of the parent in case the parent is *TPaintBox32* or its descendant.

It is your responsibility to perform drawing within the rectangle defined by *BoundsRect* property inherited from *TControl*. *TLayer32* does not perform neither clipping nor origin shifting.

TBitmapLayer32

The descendant of *TLayer32*, which holds a *TBitmap32* and automatically paints it within the rectangle specified by the inherited from *TControl*'s *BoundsRect* property.

It 'knows' when and how to repaint itself, and, unlike TLayer32, does generate *OnChanging* and *OnChange* events.

Due to performance considerations I would recommend changing *BoundsRect* property directly, when you wand to reposition the layer instead of consecutively changing its *Left*, *Top*, *Width* and *Height* properties.

Layer's scaling and opacity options are controlled by corresponding properties of contained bitmap.

Properties

AutoSize **property** AutoSize: Boolean;

Inherited from *TControl*, this property defines the sizing of the layer. If set to true, the width and height of the layer are automatically set to dimensions of contained bitmap, otherwise, the contained bitmap is stretched to fit in dimensions specified by *Width* and *Height* properties.

Bitmap **property** Bitmap: TBitmap32;

References the contained bitmap.

TBitmapList32

This object contains a collection of bitmaps accessible in design time. See the source for details.

TByteMap

The *TByteMap* class defined in *ByteMaps.pas*, may be used to simulate palette-based operations or to access separate color layers of *TBitmap32*. *TByteMap* is an ancestor of *TCustomMap* and is assignment compatible with *TBitmap32*. It uses the same thread-safe locking and change notification mechanism as *TBitmap32*.

Change notifications work similar to those in *TBitmap32*.

Properties

Bytes

property Bytes: TArrayOfBytes;
type TArrayOfBytes = **array of** Byte;

Returns the pointer to the internal array of bytes. Row-major storage order, top line comes first.

Height

property Height: Integer;

The height of the stored byte map in pixels.

ValPtr

property ValPtr[X, Y: Integer]: PByte;

Returns a pointer to the specific byte in the array.

Value

property Value[X, Y: Integer]: Byte; **default**;

Provides coordinate-based access to stored bytes. This function does not perform range checking of its arguments. Be sure, that byte map is not empty and both *X* and *Y* lie in a valid range.

Width

property Width: Integer;

The width of the stored byte map in pixels.

Methods

Empty

function Empty: Boolean; **virtual**; *// read-only*

Returns true if the byte map contains no data, that is when both width and height equal 0.

Clear

procedure Clear(FillValue: Byte);

Fills the entire byte map with specified value;

ReadFrom

procedure ReadFrom(Source: TBitmap32; Conversion: TConversionType);
type TConversionType = (ctRed, ctGreen, ctBlue, ctAlpha, ctUniformRGB, ctWeightedRGB);

The *ReadFrom* method allows reading of color layers from *Bitmap32* as well as filling the byte map with a grayscale version of Bitmap32. The byte map is automatically resized to *Source* dimensions. When *Conversion* parameter is *ctUniformRGB*, the byte value is written as the average value of red, green and blue components of corresponding pixel in the source. If it is equal to *ctWeightedRGB*, the *Intensity* function is used instead of averaging.

SEE ALSO: *WriteTo*, *"Intensity"* on page 4.

SetSize

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

Similar to TBitmap32.SetSize, this function sets the byte map dimensions according to specified parameters. The following TPersistent descendants may be used as the Source parameter: *TByteMap*, *TBitmap32*, *TGraphic*, *TControl*, *nil*;

WriteTo **procedure** WriteTo(Dest: TBitmap32; Conversion: TConversionType); **overload**;
procedure WriteTo(Dest: TBitmap32; **const** Palette: TPalette32); **overload**;
type TConversionType = (ctRed, ctGreen, ctBlue, ctAlpha, ctUniformRGB, ctWeightedRGB);
type TPalette32 = **array** [0..255] **of** TColor32;

WriteTo fills the *Dest* bitmap using the values stored in the byte map. For *ctRed*, *ctGreen*, *ctBlue* and *ctAlpha* values of the *Conversion* parameter, this method fills the corresponding color layer in the destination, while the rest color components remain intact. stUniformRGB and ctWeightedRGB have the same action, they fill the bitmap with corresponding grayscale values (the alpha channel is filled with $FF).

The second overloaded version of *WriteTo* uses *TPalette32* to map a byte value to a color in destination.

Transformations

Graphics32 supports scaling and other linear transformations of bitmaps with sub pixel accuracy they are implemented in Transform32.pas unit.

All the transformation functions have two common parameters: *Src* and *Dst* that specify the source and destination bitmaps respectively. Then performing transformations, neither source nore destination parameters may be equal to *nil*. In this case, function will generate an exception. They may be empty however, in this case no transformation will be berformed. Transforming the data inside the same bitmap, when *Src* = *Dst*, has some limitations, generally in this case source and destination regions must not intersect (this is discussed in details when discussing particular functions).

BlockTransfer

BlockTransfer is similar to the *BitBlt* function from Windows GDI.

```
procedure BlockTransfer(
  Dst: TBitmap32;
  DstX: Integer;
  DstY: Integer;
  Src: TBitmap32;
  SrcRect: TRect;
  CombineOp: TDrawMode);
```

```
type TDrawMode=(dmOpaque, dmBlend);
```

It performs copying of the bitmap fragment specified by *SrcRect* into location (*DstX*, *DstY*). If *CombineOp=dmBlend*, the fragment is blended to destination using its alpha channel and *MasterAlpha* property, otherwise the destination pixels are replaced.

It is not required for *DstRect* and *SrcRect* to lie entirely inside the corresponding bitmap, since the function provides necessary clipping.

The result is not specified when transferring data inside the same bitmap (*Src=Dst*) and if in the same time *SrcRect* intersects with *DstRect*. In this case it is recommended to use a temporary bitmap buffer.

StretchTransfer

StretchTransfer is similar to *StretchBlt* or *StretchDIBits* functions from WindowsGDI.

```
procedure StretchTransfer(
  Dst: TBitmap32;
  DstRect: TRect;
  Src: TBitmap32;
  SrcRect: TRect;
  StretchFilter: TStretchFilter;
  CombineOp: TDrawMode);
```

```
type TStretchFilter = (sfNearest, sfLinear, sfSpline);
```

```
type TDrawMode=(dmOpaque, dmBlend);
```

This procedure performs copying and, if necessary, stretching of the bitmap fragment specified by *SrcRect* into location specified by *DstRect*.

StretchFilter defines a color interpolation method for image stretching. The fastest filter is *sfNearest*, although the quality of the stretched image is fair; *sfLinear* is several times slower, but it produces more or less decent results in most cases, *sfSpline* – is an approximation of spline interpolation, for some applications its result may be too smooth and blurry, but when using with large magnification factors, it usually yields better image compared to *sfLinear*.

Of cource, there are tons of other filters which might yield better results, however they are very specific to particular application needs (probably I will include some of them in future versions).

Unlike in *BlockTransfer* function, *SrcRect* **must** lie inside the *Src* bitmap boundaries, otherwise function will generate an exception.

The result is not specified when transferring data inside the same bitmap (*Src=Dst*) and if in the same time *SrcRect* intersects with *DstRect*. In this case it is recommended to use a temporary bitmap buffer.

SEE ALSO: *BlockTransfer*.

Transform

The *Transform* function is responsible for arbitrary geometrical transformations of bitmaps or their fragments. In current version it only supports linear transformations.

```
procedure Transform(
  Dst, Src: TBitmap32;
  SrcRect: TRect;
  Transformation: TTransformation);
```

The *Transformation* parameter is a reference to a descendant of *TTransformation* class defined in Transform32.pas

```
  TTransformation = class(TObject)
  public
    function  GetTransformedBounds(const Src: TRect): TRect; virtual; abstract;
    procedure PrepareTransform; virtual; abstract;
    procedure Transform(DstX, DstY: Integer; out SrcX, SrcY: Integer); virtual; abstract;
    procedure Transform256(DstX, DstY: Integer; out SrcX256, SrcY256: Integer); virtual; abstract;
  end;
```

Only linear transformation is currently implemented.

When *Src.StretchFilter* is not *sfNearest*, *Transform* uses linear (bilinear actually) interpolation for magnification (along any axis) as for minification, it is not as accurate as *StretchTransfer* function. If you need better quality when minimizing the bitmaps, transform them into the temporary buffer so that there is no minification invlolved, then *StretchTransfer* to a final bitmap.

Even if *Src.StretchFilter* is *sfSpline, Transform* operates as if it were *sfLinear.*

There is an issue with antialiasing and edges. How to make them antialiased and still keep the performance? The solution implemented in Graphics32 is similar to the one used in OpenGL (I'm not quite sure about Direct3D since I don't know it, but I think it uses the same method).

You just have to provide the source bitmap (or its region) with transparent edges. If the original image, you'll have to force the alpha channel on its edges to zeroes. Remember, that color is interpolated as well, it means that for nice fadeout the color on the border should match the color of pixels lying next to the border.

In case the bitmap is transformed in *dmOpaque* mode, it might be better to keep the color on the edge equal to the color of the background.

TLinearTransformation

TLinearTransformation defines a transformation that may be used as parameter in the *Transform* function.

The linear transformation is defined by 3x3 homoheneous matrix of double precision floats:

```
type TMatrix3d = array[0..2, 0..2] of Extended;
```

(The '3d' postfix does not have anything to do with 3D transformations, it just means that matrix is 3x3 and each value is of a double precision).

The coordinates are transformed as

$$\begin{bmatrix} x_{\text{dst}} \\ y_{\text{dst}} \\ \text{not used} \end{bmatrix} = \begin{bmatrix} M_{[0,0]} & M_{[1,0]} & M_{[2,0]} \\ M_{[0,1]} & M_{[1,1]} & M_{[2,1]} \\ M_{[0,2]} & M_{[1,2]} & M_{[2,2]} \end{bmatrix} \cdot \begin{bmatrix} x_{\text{src}} \\ y_{\text{src}} \\ 1 \end{bmatrix};$$

Only two top rows are used in a final stage when transforming coordinates.

Properties

Matrix

property Matrix: TMatrix3d;

Holds the transformation matrix;

Methods

GetTransformedBounds

function GetTransformedBounds(**const** Src: TRect): TRect; **override**;

Returns the bounding rectangle of the region that will be obtained from the *Src* rectangle after its transformation.

PrepareTransform

procedure PrepareTransform; **override**;

PrepareTransform must be called before using *Transform* or *Transform256* methods after any changes were made in transformation matrix.

Transform

procedure Transform(DstX, DstY: Integer; **out** SrcX, SrcY: Integer); **override**;
procedure Transform256(DstX, DstY: Integer; **out** SrcX256, SrcY256: Integer); **override**;

These functions provide reverse transformation of the point (*DstX*, *DstY*) back into its position at the source bitmap. *Transform256* provides the same result, only the coordinates are multiplyed by 256.

Clear

procedure Clear;

Resets all the transformations (loads identity matrix).

Rotate

procedure Rotate(Cx, Cy, Alpha: Extended); *// in degrees*

First the origin is translated to (Cx, Cy) point, then the image is rotated around the origin by *Alpha* degrees

$$M = \begin{bmatrix} \cos\alpha & \sin\alpha & 0 \\ -\sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix} M;$$

and finally the origin is shifted back.

Skew

procedure Skew(Fx, Fy: Extended);

Adds the skew to the transformation:

$$M = \begin{bmatrix} 1 & Fx & 0 \\ Fy & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} M \;;$$

Scale

procedure Scale(Sx, Sy: Extended);

Adds the scale transformation:

$$M = \begin{bmatrix} Sx & 0 & 0 \\ 0 & Sy & 0 \\ 0 & 0 & 1 \end{bmatrix} M;$$

Translate **procedure** Translate(Dx, Dy: Extended);

Translates the image:

$$M = \begin{bmatrix} 1 & 0 & Dx \\ 0 & 1 & Dy \\ 0 & 0 & 1 \end{bmatrix} M;$$

Filters

This unit currently implements only a basic operations, more to come in future versions.

Only a few simple functions is currently available. They all support in-place operations, that is *Src* may be the same as *Dst*.

AlphaToGrayScale **procedure** AlphaToGrayscale(Dst, Src: TBitmap32);

IntensityToAlpha **procedure** IntensityToAlpha(Dst, Src: TBitmap32);

Invert **procedure** Invert(Dst, Src: TBitmap32); // inverts all channels including alpha

InvertRGB **procedure** InvertRGB(Dst, Src: TBitmap32); // inverts all channels excluding alpha

ColorToGrayscale **procedure** ColorToGrayscale(Dst, Src: TBitmap32);

ApplyLUT **procedure** ApplyLUT(Dst, Src: TBitmap32; const LUT: TLUT8);
type TLUT8 = **array** [Byte] **of** Byte;

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