The Graphics32 Library
Classes (Alphabetical)

TAdaptiveSuperSampler
TAffineTransformation
TArrowBar
TBitmap32
TBitmap32Collection
TBitmap32Item
TBitmap32List
TBitmap32Resampler
TBitmapLayer
TBitmapPolygonFiller
TBlackmanKernel
TBloatTransformation
TBooleanMap
TBoxKernel
TByteMap
TClassList
TContourRasterizer
TContracter
TConvolver
TCosineKernel
TCubicKernel
TCustomGaugeBar
TCustomImage32
TCustomImgView32
TCustomKernel
TCustomLayer
TCustomMap
TCustomPaintBox32
TCustomPolygonFiller
TCustomRangeBar
TCustomResampler
TCustomSampler
TDilater
TDisturbanceTransformation
TDraftResampler
TERoder
TExpander
TFishEyeTransformation
TGaugeBar
TGradient
TGaussianKernel
THammingKernel
THannKernel
THermiteKernel
TImage32
TImgView32
TIntegerMap
TIVScrollProperties
TKernelResampler
TKernelSampler
TLanczosKernel
TLayerCollection
TLinearKernel
TLinearResampler
TMitchellKernel
TMorphologicalSampler
TNearestResampler
TNearestTransformer
TNestedSampler
TNotifiablePersistent
TPaintBox32
TPaintStages
TPatternSampler
TPointerMap
TPointerMapIterator
TPolygon32
TPositionedLayer
TProgressiveRasterizer
TProjectiveTransformation
TRangeBar
TRasterizer
TRectList
TRegularRasterizer
TRemapTransformation
TRenderThread
TRubberbandLayer
TSelectiveConvolver
TSinshKernel
TSplineKernel
TSuperSampler
TSwizzlingRasterizer
TSyntheticImage32
TTesseralRasterizer
TThreadPersistent
TTransformation
TTransformer
TTwirlTransformation
TVectorMap
TWindowedSincKernel
TWordMap
### Routines

<table>
<thead>
<tr>
<th>AlphaComponent</th>
<th>ColorDiv</th>
<th>Expand</th>
<th>Invert</th>
<th>Rasterize</th>
</tr>
</thead>
<tbody>
<tr>
<td>AlphaToGrayscale</td>
<td>ColorMax</td>
<td>FillLongword</td>
<td>InvertRGB</td>
<td>RedColor</td>
</tr>
<tr>
<td>ApplyBitmask</td>
<td>ColorMin</td>
<td>FillWord</td>
<td>IsRectEmpty</td>
<td>RGBColor</td>
</tr>
<tr>
<td>ApplyLUT</td>
<td>ColorModulate</td>
<td>FixedCeil</td>
<td>Lighten</td>
<td>SetAlpha</td>
</tr>
<tr>
<td>Blend</td>
<td>ColorSub</td>
<td>FixedDivision</td>
<td>Merge</td>
<td>SetBorder</td>
</tr>
<tr>
<td>BlendEx</td>
<td>Colorswap</td>
<td>FixedFloor</td>
<td>MergeEx</td>
<td>SetGamma</td>
</tr>
<tr>
<td>BlendLine</td>
<td>ColorToGrayScale</td>
<td>FixedMultiply</td>
<td>Mirror</td>
<td>ShiftA</td>
</tr>
<tr>
<td>BlendLineEx</td>
<td>Combine</td>
<td>FixedPointMath</td>
<td>MoveLongword</td>
<td>SinCos</td>
</tr>
<tr>
<td>BlendTransfer</td>
<td>Constrain</td>
<td>FixedRound</td>
<td>MoveWord</td>
<td>Stretch</td>
</tr>
<tr>
<td>BlockTransfer</td>
<td>Contract</td>
<td>FixedSquare</td>
<td>OffsetRect</td>
<td>Swap</td>
</tr>
<tr>
<td>BlockTransferX</td>
<td>CopyComponents</td>
<td>FixedSquare</td>
<td>Polygon</td>
<td>SwapColor</td>
</tr>
<tr>
<td>BlueComponent</td>
<td>CreateBitmask</td>
<td>Gray32</td>
<td>Polyline</td>
<td>TestClip</td>
</tr>
<tr>
<td>ChromaKey</td>
<td>CreatingPoints</td>
<td>GreenComponent</td>
<td>PolyPolygon</td>
<td>TestStyle</td>
</tr>
<tr>
<td>Clamp</td>
<td>CreatingRectangles</td>
<td>HSLtoRGB</td>
<td>PolyPolygonBounds</td>
<td>Transform</td>
</tr>
<tr>
<td>Color32</td>
<td>Dilate</td>
<td>InflateRect</td>
<td>PolyPolyline</td>
<td>Transform</td>
</tr>
<tr>
<td>Color32Components</td>
<td>EMMS</td>
<td>Intensity</td>
<td>PtInPolygon</td>
<td>WinColor</td>
</tr>
<tr>
<td>Color32ToRGB</td>
<td>EqualRect</td>
<td>IntensityToAlpha</td>
<td>PtInRect</td>
<td>WinPaint</td>
</tr>
<tr>
<td>ColorAdd</td>
<td>Erode</td>
<td>IntersectRect</td>
<td>Rasterize</td>
<td>Wrap</td>
</tr>
</tbody>
</table>

### Types

<table>
<thead>
<tr>
<th>Arrays</th>
<th>TBlendMemEx</th>
<th>TFillLineEvent</th>
<th>TPixelCombineEvent</th>
<th>TScreen</th>
</tr>
</thead>
<tbody>
<tr>
<td>Color Types</td>
<td>TBlendReg</td>
<td>TFixed</td>
<td>TPolyFillMode</td>
<td>TSize</td>
</tr>
<tr>
<td>Point Types</td>
<td>TBlendRegEx</td>
<td>TFloatMatrix</td>
<td>TRBBackgnd</td>
<td>TSize</td>
</tr>
<tr>
<td>Rectangle Types</td>
<td>TCombineInfo</td>
<td>TKernelMode</td>
<td>TRBHandles</td>
<td>TStretch</td>
</tr>
<tr>
<td>TAntialiasMode</td>
<td>TCombineMem</td>
<td>TLogicalOperator</td>
<td>TRBStyle</td>
<td>TVec</td>
</tr>
<tr>
<td>TAreaChangedEvent</td>
<td>TCombineMode</td>
<td>TLUT8</td>
<td>TRectStyle</td>
<td>TVec</td>
</tr>
<tr>
<td>TBlendLine</td>
<td>TCombineReg</td>
<td>TPaintBoxOptions</td>
<td>TRenderMode</td>
<td>TWidth</td>
</tr>
<tr>
<td>TBlendLineEx</td>
<td>TConversionType</td>
<td>TPaintStage</td>
<td>TRepaintMode</td>
<td>Vec</td>
</tr>
<tr>
<td>TBlendMem</td>
<td>TDrawMode</td>
<td>TPixelAccessMode</td>
<td>TSemMode</td>
<td>Wrap</td>
</tr>
</tbody>
</table>

### Variables
### Constants

<table>
<thead>
<tr>
<th>Area Information Flags</th>
<th>DefaultAAMode</th>
<th>IdentityMatrix</th>
<th>Paint Stage Constants</th>
</tr>
</thead>
<tbody>
<tr>
<td>Color Constants</td>
<td>G32Version</td>
<td>Layer Options Bits</td>
<td></td>
</tr>
</tbody>
</table>

### Units

<table>
<thead>
<tr>
<th>GR32</th>
<th>GR32_Filters</th>
<th>GR32_Math</th>
<th>GR32_Rasterizers</th>
</tr>
</thead>
<tbody>
<tr>
<td>GR32_Blend</td>
<td>GR32_Image</td>
<td>GR32_OrdinalMaps</td>
<td>GR32_Resamplers</td>
</tr>
<tr>
<td>GR32_Collectors</td>
<td>GR32_Layers</td>
<td>GR32_Polygons</td>
<td>GR32_Transforms</td>
</tr>
<tr>
<td>GR32_ExtImage</td>
<td>GR32_Lowlevel</td>
<td>GR32_RangeBars</td>
<td>GR32_VectorMaps</td>
</tr>
</tbody>
</table>

Copyright ©2000-2007 Alex Denisov and Contributors - Graphics32 v1.8.3 - Build on 4-March-2007
Features

Many features in Graphics32 are similar to those in standard *TImage, TBitmap* and *TCanvas* classes, however they were rewritten to accelerate and optimize drawing on 32-bit device-independent bitmaps (DIBs). It also includes a few new options.

Some of its features include:

- Fast per-pixel access up to 100 times faster compared to standard *TCanvas/TBitmap*;
- High-performance Bitmap alpha blending (including per-pixel alpha blending);
- Pixel, line and polygon antialiasing with sub-pixel accuracy (combined with alpha blending);
- Arbitrary polygon transformations and custom fillings;
- Bitmap resampling with high quality reconstruction filters (e.g. Lanczos, Cubic, Mitchell);
- A unique state-of-the-art rasterization system;
- Affine transformations of bitmaps: rotations, scaling, etc with sub-pixel accuracy;
- Arbitrary projective transformations of bitmaps;
- Arbitrary remapping transformations of bitmaps (e.g. for Warping, Morphing);
- Flexible supersampling implementation for maximum sampling quality;
- Flicker-free image displaying components with optimized double buffering via advanced [MicroTiles based repaint optimizer](#);
- Multiple customizible easy-to-use overlay layers;
- Locking of bitmaps for safe multithreading;
- A property editor for RGB and alpha channel loading;
- Design-time loading of image formats supported by standard *TPicture*;

Except for extended features, Graphics32 has some important differences from the standard components. It does not heavily rely on Windows GDI, most of the functions are reimplemented and optimized specifically for 32-bit pixel format.
License

As of Version 1.5.1b Graphics32 is licensed under the terms of the Mozilla Public License. You may not use the Graphics32 library except in compliance with the License. You may obtain a copy of the License at http://www.mozilla.org/MPL
Donate

Donation

Graphics32 is free under the terms of the Mozilla Public License. However, if you wish to express your appreciation for the time and energy spend on developing, documenting, supporting and maintaining Graphics32, the team accepts with gratitude any donation made.

Donating should just be thought of as a way of showing appreciation - it will not grant any extra support or focus on specific feature requests. In general the team strives to accommodate and respond as much as possible to user responses and requests.

Two types of donations can be made. Donations for the team in general, which mainly is used to cover general expense related to Graphics32. And donations to individuals involved in the development of Graphics32 (see the list of features, the list of contributors, the list of changes, source codes or examples to aid your evaluation of the involvement of the individuals).

If you are unsure what to choose, the developer team donation is recommended.

Not Found

The requested URL /donation/table.htm was not found on this server.
Michael, Andre and Mattias

Thank you for your support.
See Also

Changes, Contacts, Contributors, Examples, Features, Naming Conventions

Copyright ©2000-2007 Alex Denisov and Contributors - Graphics32 v1.8.3 - Build on 4-March-2007
Contacts

Graphics32 SourceForge home: http://sourceforge.net/projects/graphics32
Graphics32 new home: http://graphics32.org
Graphics32 support forum: http://graphics32.org/forum
Graphics32 support newsgroup:
news://news.g32.org/g32org.public.graphics32
E-mail: team@graphics32.org

Please, use the newsgroup or forum for general support questions.

Graphics32 old home: http://g32.org
old E-mail: alex@g32.org
See Also
alex@g32, Changes, Donate, Features, License, Naming Conventions, team@graphics32
Naming Conventions

Graphics32 introduces several routines (as well as some class properties and methods) which have similar action but may have different arguments or other realization details.

They follow a simple naming convention:

<table>
<thead>
<tr>
<th>Postfix</th>
<th>Details</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>Property or method does not perform any range checking of its arguments. All the coordinates should be valid.</td>
<td>Line</td>
</tr>
<tr>
<td>S</td>
<td>'Safe' version. Validates coordinates. If necessary, clipping is performed.</td>
<td>LineS</td>
</tr>
<tr>
<td>T</td>
<td>'Transparent' version of the method. Uses the alpha channel of the provided color to blend the drawn primitive with the background pixels.</td>
<td>LineT</td>
</tr>
<tr>
<td>A</td>
<td>Methods with 'A' postfix provide antialiasing of the drawn primitive.</td>
<td>LineA</td>
</tr>
<tr>
<td>X</td>
<td>These functions operate with coordinates in TFixed format. They automatically provide antialiasing.</td>
<td>LineX</td>
</tr>
<tr>
<td>F</td>
<td>Methods with 'F' postfix take coordinates as floating point arguments and perform antialiasing of the drawn primitive.</td>
<td>LineF</td>
</tr>
<tr>
<td>TS, AS, XS, FS</td>
<td>Valid combinations of postfixes described above.</td>
<td>LineFS</td>
</tr>
<tr>
<td>P</td>
<td>'Pattern' version. Usually combined with TS or FS postfixes, it allows for implementation of various effects, like gradient or dashed lines (<a href="#">See Line Patterns</a>).</td>
<td>LineFSP</td>
</tr>
<tr>
<td>W</td>
<td>Wrapmode version. Checks if coordinates are outside boundaries or provided ranges, and performs wrapping according to a wrapmode</td>
<td></td>
</tr>
</tbody>
</table>
See Also

Changes, Contacts, Donate, Features, License, Line Patterns, TWrapMode
Changes

v1.8.3 (5 March 2007)
General changes
Additions:

- Added new ASM/(E)MMX optimized ApplyBitmask routine for logical (bitwise) Xor/Or/And operations using a bitmask.
- Added new TColor32Component basetypes for specifying a component or component sets (related updates to TColor32Entry).
- Added new CopyComponents routine for copying selected ARGB components between bitmaps.
Optimizations:
- Optimized TBitmap32.ResetAlpha, Invert and InvertRGB.
- Optimizations in pamTransparentEdge related routines.
- TPatternSampler now supports nonrectangular sampling patterns.
Bugfixes:

- Bugfix in MMX version of BlockAverage that caused access violations on resampling.
- Fixes for the following bugs reported on the SourceForge tracker: 1560912, 1556318
- Fixed misbehavior when setting TCustomImage32.BufferOversize property. Changes now directly force resizing of the buffer.
- Fixed Kylix compilation issue introduced with new multithreading support.
- Several minor cleanups and fixes.
v1.8.2 (5 February 2007)
General changes
Additions:

- Added new TMultiThreadedRasterizer class for performing regular rasterization using multiple threads. By defining the symbol USE_MULTITHREADING this rasterizer becomes default on multi-core systems.
- Added TPolarTransformation and TPathTransformation classes.
- Added TPixelAccessMode pamTransparentEdge, useful for getting smooth edges with transformations.
- Added new optimized low-level routines to GR32_Math.pas.
Bugfixes:

- Fixes for various bugs reported on the SourceForge tracker: 1445701, 1448030, 1486180, 1456950, 1460821, 1493697, 1610287, 1610292, 1611173
- Fixed a problem in TRenderThread that cause an AV when OnAreaChanged of the destination bitmap was unassigned.
- Fixed overflow on calculation in LineS / LineTS with big line coordinates.
- Fixed buffer overrun error in polygon rasterization.
- Fix: M_ColorMax, M_ColorMin, M_ColorAverage raised critical exception on old Celeron CPU.
- Fixed problem in TKernelResampler that caused artifacts at edges for pamSafe pixel access mode.
- Fixed installation problems in D2005.
- Minor tweaks and optimizations.
General changes
Additions:
- Reintroduced property editor features for all Delphi versions after D5.
- BDS2k6 C++ personality supported.
Bugfixes:

- Fixes for various bugs reported on the SourceForge tracker: 1369894, 1398880, 1403949, 1404989, 1409711, 1412953
- Fixed a compilation problem when CHANGED_IN_PIXELS define was enabled.
- Removed mbLeft MouseDown filter in TCustomLayer.
- TCustomLayer.Changed obeys UpdateCount.
- Fixed a bug in the MicroTiles repaint optimizer where the buffer wasn't completely invalidated.
- CLX and D5 compilation issues fixed (GetPropList).
Detailed changes
v1.8 (1 January 2006)
General changes
Additions:
- Global usage of new TFloat type alias instead of Single.
- Area update support for TBitmapLayer added.
- TTransformation changed from TPersistent to TNotifyablePersistent so as to implement a Changed method that sets TransformValid to False automatically.
- Incorporated the Interlocked* functions in TBitmap32.Lock and TBitmap32.Unlock according to Jouni Airaksinen's suggestions.
- Redundant code clean up + better maintainability using central protected CopyMapTo and CopyPropertiesTo methods.
- Added BlockTransferX routine for bitmap transfers with fixed-point coordinates (using linear interpolation).
- CLX versions of all example projects added (no guarantee that all projects are 100% CLX-compatible).
- Delphi 4 support officially dropped as of this release.
Bugfixes:

- Bug in TLayerCollection.ViewportToLocal method fixed.
- Misbehaviour of TCustomLayer.Visible property fixed.
- TCustomLayer.Changed and TCustomLayer.Update methods fixed (NO_UPDATE problem)
- TBitmap32Item.AssignTo implemented to fix problem "Cannot assign a TBitmap32Item to a TBitmap32Item" (SF bug 1329566).
- Prevent AV if user sets resampler directly (e.g. Bitmap.Resampler := TLinearResampler.Create(Bitmap);).
- Fixed a problem in TCustomImage32 that caused performance overhead for rmDirect repaint mode.
- TSyntheticImage was not automatically rasterized upon creation.
- Update regions in TTesseralRaserizer were not pixel-perfect.
- Fix for memory leaks in AssignFromBitmap sub-method due to thread <-> GDI interaction.
- Fixed a problem in integer version of HSLtoRGB.
- Fixed TBitmap32.GetStippleColor (the ordinary interpolation routine must be used).
- Fixed TBitmap32.SET_T256 and TBitmap32.SET_TS256 for color merging. cmMerge is now handled as a separate case.
- Fixed TBitmap32.FillRectT for cmMerge.
- Fixed bug in _CombineMem routine. The alpha channel was not interpolated, which is contradictory to documentation.
- Revised Merge routines. Merge is now based on precise lookup tables. Moreover the new version is faster than the old buggy MMX version.
- D5 compilation fixes.
Detailed changes
v1.8 beta (9 August 2005)
General changes
Major additions:
- Resampler Framework / Nested Sampling
- Rasterizer Framework
- Buffered transformations / Warping / Remap Transformation
- Repaint optimizations (MicroTiles Repaint optimizer)
Other additions:

- New transformations (Twirl, Bloat, Disturbance, FishEye)
- Fast transformations using TRemapTransformation
- Kernel classes (implementing reconstruction filters) for TKernelResampler
- Antialiasing by using regular and irregular super sampling methods
- Spatial convolution and morphological operations
- Vector maps, supporting .msh fileformat (useful for rasterizing/buffering transformations)
- New ordinal map classes for Boolean-, Word- and Integer-sized data elements
- New WrapMode property and wrapping Pixel properties
- Scrollbar visibility control and new scale modes
- Independent scales in TCustomImgView32 / TImgView32
- TRubberbandLayer enhancements
- New BlendTransfer routine for blending two bitmaps using an arbitrary blend callback routine
- New polygonal antialiasing modes for 2x and 32x antialiasing
- Improved performance in TPolygon32 rasterization routines
- New draw mode, dmTransparent, for color key transparency
- FastMove support
- New class registration mechanism using TClassList
- Help documents updated with new additional topics
- Restructured examples:

  new examples: TextureBlend_Ex, NestedSampling_Ex, Resamplers_Ex, ImgWarping_Ex, Visualization_Ex

  changed examples:
  GradLines_Ex, Polygons_Ex, ImgView_Layers_Ex, Image32_Ex, Sprites_Ex
Bugfixes:
- Fixed AV with Graphics32 usage in DLLs on WinXP
- Fixed jagged lines problem in LineS, LineTS and LineAS
Detailed changes
v1.7.1 (25 February 2005)
Bugfixes

- fixes for AVs in new line drawing methods (LineS, LineTS, LineAS)
- replaced incorrect call of VertLineS and HorzLineS in LineTS and LineAS with VertLineTS and HorzLineTS
- fix for buffer reallocation inefficiency in TCustomPaintBox32 (non-critical, long standing)
  
  This should improve performance on resizing for all TCustomPaintBox32 derived controls.
- added missing call to inherited constructor in TPolygon32 constructor (caused big trouble on Win9x/WinME systems)
- RenderText now ignores Windows-based font smoothing in VCL
  
  (VCL-only change because the QT version that's used in CLX doesn't support disabling font antialiasing, neither in Windows nor in Linux/X11)
- CLX compilation fixes
- C++ Builder compilation fixes
- ColorMul removed (duplicate of ColorModulate)
- minor cleanups
v1.7 (16 January 2005)
New Features

- clipping rect support (all safe drawing operations are clipped to a rect)
- support for custom polygon fillings (via callback or polygon filler objects)
- support for polygon transformation (permanent or while drawing)
- update to transformation classes to make them more suited for general usage
- polygon performance enhancements
- new CombineMode property in TBitmap32
- TRubberbandLayer enhancements
- additional color algebra functions
- more versatile TBitmap32.ResetAlpha method (overloaded)
- Delphi 2005 (Win32) support (optional inlining where applicable)
- small speed enhancements
Bugfixes

- fix for RenderText problems with letters that spread below baseline
- fix for bug in TBitmap32.Rotate180
- fix for problem in TBitmap32.Assign that caused transparent regions in metafiles and icons to appear in blue when they really should be white.
- fix for problem where the buffer content on repaint in TPaintBox32 wasn't validated
- fix for problem where C++ Builder threw linker errors (HDC methods and "hoisted overload" or E2113) in GR32.pas and GR32_Image.pas
- fix for linear/draft bug with bitmaps of size 100x1 et al.
- fix for bug in Transform that caused the outline of the source bitmap to appear in OuterColor
- fix for the 3DNow detection (caused "external exception" on some machines when sfDraft was used)
- fix for dysfunctional OnMouseEnter and OnMouseLeave in TCustomImage32
- fix for CheckParams problem in ApplyLUT
- fix to preserve alpha channel in ApplyLUT and ColorToGrayscale
Changes

- Graphics32 now fully supports CLX cross-platform development;
- Example projects have been successfully converted and tested in Linux;
- Fixed issues in GR32_Transforms that could potentially cause access violation;
- Added sfDraft resampling filter with MMX optimization;
- Added sfCosine resampling filter;
- Added PixelF/X properties that read and write pixels with antialiasing;
- SetPixelF/X methods have been deprecated;
- Added new MMX optimized color algebra routines to GR32_Blend (Add, Subtract, Multiply etc.);
- Added AntiAliasMode property to T_polygon32 for dynamically changing antialias level at runtime;
- Added support for 4x supersampling antialias of polygons (significantly faster but with less quality);
- Unicode support for TBitmap32.Text* and TBitmap32.Rendertext methods;
- Fixed blending bug (the "not fully opaque" issue);
- Performance optimizations (MMX) in blending routines and Transforms;
- Several minor additions and optimizations.
Earlier releases (v0.9 - v1.5.1b)
Summary
See Also

Contacts, Donate, Features, License, Naming Conventions

Copyright ©2000-2007 Alex Denisov and Contributors - Graphics32 v1.8.3 - Build on 4-March-2007
Installation

Graphics32 supports Delphi versions 5, 6, 7, 2005 (Win32), 2006 (Delphi Win32 and C++), and C++ Builder 5 and 6.

As of version 1.6 there is also CLX support in Graphics32. That means you can use it either with CLX on Windows using Delphi or on x86 Linux using the Kylix Delphi environment.

Note: If you want to use the Windows CLX version in Delphi, you'll have to uncomment the \{\$DEFINE CLX\} line in GR32.inc.

Note: For Delphi 2005 and up function inlining is automatically used to improve performance in certain situations. If you want to avoid this, please comment out the \{\$DEFINE USEINLINING\} line in GR32.inc.

Note: Because some properties of range bars and gage bars have been changed, you can obtain some error messages while opening your previous Graphics32 projects. Just ignore these messages and Delphi will fix DFM files automatically.

- Unzip the files;
- Add location of main installation directory to Tools | Environment Options | Library | Library Path.
- Select File | Open... on the menu bar. Set Files of type to Delphi package source, locate and open GR32_DSGN_D6.dpk (GR32_DSGN_K for Kylix, GR32_DSGN_BDS2006 for Borland Developer Studio 2006, GR32_DSGN_D2005 for Delphi 2005, GR32_DSGN_D7 for Delphi7, GR32_DSGN_D6 for Delphi6, GR32_DSGN_D5 for Delphi5 or GR32_DSGN_B5 for C++ Builder);
- A package editor window will appear. Click Compile, then click Install;
- If compiler asks whether you want to save changes, usually it is a good idea to choose No.
Examples

There are a few examples included with the library, which you should be able to find in the respective subdirectories VCL or CLX in the Examples subdirectory.

Here is the list of examples with short descriptions:
**ByteMaps Example**

Directory: General\ByteMaps_Ex *(VCL only due to the 3rd party components used)*

A basic demonstration of mapping a 2D array of values to `TByteMap` and displaying it using `TPalette32`.
GradLines Example

Directory: Drawing\GradLines_Ex

An example on using line patterns to draw gradient lines.
**Image Warping Example**

Directory: Transformation\ImgWarping_Ex

This comprehensive example demonstrates extensive use of TRemapTransformation and TVectorMap. It also implements a useful generic brush weight system, which can be applied to other aspects of image related editing (painting, uniform feathering etc.).
Image32 Example

Directory: General\Image32_Ex

This example demonstrates the properties of TImage32 that control the scale and alignment of the bitmap image.
**ImgView and Layers Example**

Directory: Layers\ImgView_Layers_Ex

A demonstration of using `TImgView32`, `TPositionedLayer`, `TBitmapLayer` and `TRubberbandLayer`. It also shows how it is possible to load RGB and Alpha channels from different image files into the `TBitmap32` object at run-time and demonstrates an application of affine transformations. It also shows how it is possible to 'flatten' layers.
LineStippling Example

Directory: Drawing\LineStippling_Ex

An example on using line patterns to draw dashed lines.
**Nested Sampling Example**

Directory: Resampling\NestedSampling_Ex (*VCL only due to the use of VCL exclusive TSyntheticImage*)

An extensive example that features a visual editor for editing and ordering nested samplers. It demonstrates how different samplers can be combined in order to create dynamically linked sampling chains.

Additionally it shows how various rasterizers affects rendering process.
**PixelF Example**

Directory: Resampling\PixelF_Ex

A demonstration of interpolated **PixelF** property. It lets you compare the result with normal pixel access.
**PixelCombine Example**

Directory: Blending\PixelCombine_Ex

A short demonstration of using the [OnPixelCombine](#) method in TBitmap32.
**ProgressBar Example**

Directory: General\ProgressBar_Ex

A simple visual component, that demonstrates how is it possible to create custom components based on TCustomPaintBox32. Note, that this component is mostly designed for demonstration purposes, do not expect too much from it.
**Polygons Example**

Directory: Drawing\Polygons_Ex

This example shows the application of `TPolygon32` to draw thick lines and other polygons with thick outlines.
RenderText Example

Directory: Drawing\RenderText_Ex

A basic demonstration of using the TBitmap32.RenderText method.
**Resamplers Example**

Directory: Resampling\Resamplers_Ex

An informative demonstration of the different resamplers and kernels available in Graphics32.
**Rotate Example**

Directory: General\Rotate_Ex

A simple example on using [AffineTransformation](#). It shows how to rotate and scale the bitmap at the same time so that it stays within the specified rectangle.
**RotLayer Example**

Directory: Layers\RotLayer_Ex

Features custom layer class creation and demonstrates [TAffineTransformation](#).
Sprites Example

Directory: Layers\Sprites_Ex

A demonstration on adding, removing and animating multiple TBitmapLayer objects to TImage32.
Texture Blend Example
Directory: Blending\TextureBlend_Ex
A short demonstration of using the BlendTransfer routine and how to use custom color algebra routines.
**Transform Example**

Directory: Transformation\Transform_Ex

An example, which demonstrates using the Transformation routine together with TAffineTransformation and TProjectiveTransformation.
Visualization Example

Directory: Transformation\Visualization_Ex

Implements a simple set of movement (displacement transformations) renderings in line with visualizations found in various audioplayers. The formula style roughly follows conventions of Winamp Visualization Studio, and is buffered using TVectorMap. Different types of pixelrenderings (spots, particles and more) is used to visualize the movements.
See Also

### Line Patterns

Graphics32 defines several functions to support non-uniform lines. This includes gradient lines, dashed lines etc.

The idea is pretty simple: `TBitmap32` object holds dynamic array of colors, and a counter, which 'crawls' along the array and reads colors from its position. The line drawing algorithm queries color value from the current counter position at each point, then the counter is automatically incremented to get ready to supply the next value to line rasterisation routine.

The counter, accessed through `StippleCounter` property, wraps itself automatically at the edges of color array. It can move in both directions depending on stipple step, which in turn can be positive or negative. Its value may even be fractional in this case resulting color is interpolated. The step is accessed with `StippleStep` property.

`GetStippleColor` returns color from the current counter position and automatically increments counter position by the counter step, so that next `GetStippleColor` call will return color value from the next position.

Drawing functions that support line patterns have 'P' in their postfix (as in `LineFSP`).

Warning: the counter is not thread-aware, it is shared by all threads accessing the bitmap. Additional care should be taken when multiple threads draw stippled lines in the same bitmap.
See Also

TBitmap32.GetStippleColor, GradLines Example, LineStippling Example, Naming Conventions, TBitmap32.StippleCounter, TBitmap32.StippleStep, TBitmap32

Copyright ©2000-2007 Alex Denisov and Contributors  -  Graphics32 v1.8.3  -  Build on 4-March-2007
Repaint Optimization

1 Introduction

Two basic classes for on screen display exist in Graphics32: TCustomPaintBox32 and TCustomImage32. These classes provide the functionality all other graphical components in GR32 base on.

TCustomPaintBox32 implements a component similar to the TPaintBox component known from Borlands Visual Component Library (VCL). It differs from the latter in the way it handles the content: While TPaintBox directly draws to the displays context, TCustomPaintBox32 uses an in-memory backbuffer. This technique generally called doublebuffering has its up and downsides: While it provides a convenient and simple way to avoid flickering by reducing many on-screen paint operations to just one synchronized buffer transfer (blit) from memory to screen, it does also require a significant amount of memory and bus bandwidth which in turn effectively limits the number of possible updates per second depending on the hardware used. The main problem with the implementation prior to version 1.8 of Graphics32 is that the whole buffer is transferred to screen even if just a small fraction of its area has changed, thus there is a lot of potential to improve on. One implementation that reduced this bandwidth was developed by Mattias Andersson as a patch called Clipping extension for Graphics32. The implementation in version 1.8 partly relies and extends on the techniques used in this patchset.

TCustomImage32 extends TCustomPaintBox32 by replacing the direct painting to the buffer with so called stacked Paintstages. Upon repaint these paintstages are executed in a succesive fashion from bottom to top - each stage drawing to the buffer.

![Figure 1: Paintstages](image)
at runtime

Once a change happens in this stack at any given stage a deferred invalidation of the whole buffer content is triggered, which leads to a complete repaint of all stages once this invalidation request is handled by the application message queue in Windows. This is where the main problem resides: Even with the smallest change (e.g. an updating layer) the whole buffer area needs to be repainted. This approach - though simple - is naive and results in unnecessary CPU and bus and memory bandwidth utilitzation.

To sum up, we have two main problems to overcome, i.e. to optimize:

- forced full scene repaint of paintstages to memory buffer in TCustomImage32.
- forced full scene repaint from memory buffer to screen in TCustomPaintBox32.
2 New Structure and Repaint Optimizer

In order to achieve this, we need to restructure the repaint process in both, TCustomPaintBox32 and TCustomImage32. An external repaint optimizer should take care of the aspect of managing and optimizing changed areas. This abstraction allows better flexibility over an in-place implementation, because the repaint optimizer can be exchanged freely.

TCustomPaintBox32 implements a new property RepaintMode that allows selection of the repaint mode to be used. rmFull is equal to the old full scene repaint whereas rmOptimizer uses the repaint manager to handle only updated areas.
Figure 2: Example comparison of full scene and optimized repaint

Figure 2 shows an example comparison of the old full scene repaint (\textit{rmFull}) and the new optimized repaint (\textit{rmOptimizer}) for simple layer operations like moving or resizing. One can see that the new method breaks the full scene repaint down to just a fractional repaint namely those parts that were changed. Both modes are used in TCustomPaintBox32 for repaint to screen and in TCustomImage32 for
repaint to buffer. Additionally there is one mode called rmDirect which is only available for TCustomPaintBox32 derived controls and does provide a direct repainting to screen. In this mode the deferred repaint technique is replaced by an immediate repaint. This technique is especially useful for the new TSyntheticImage class, which provides incremental painting of the result while still rendering.
3 Measuring Mode

Layers in Graphics32 are a special case that needs to be taken care of separately: Since layers are not forced to stay within their determined bounds (for TPositionedLayer for example), they can basically paint everywhere on the buffer. Thus we need to find some other way of determining which areas the layer is drawing to. For this to work we have extended all safe drawing operations in Graphics32 to support a method called measuring. This method can basically be thought of as a simulation mode or dry-run where nothing is actually drawn to the buffer. However, the Changed event is still triggered. So, this way the repaint optimizer can get information of which areas the operation is drawing to. As a matter of fact the repaint optimizer just needs to iterate through all marked layers (compare Figure 2), calling the Paint method of each layer with the measuring mode enabled. The information gathered in this process are used for the repaint manager's internal optimization work, i.e. unifying overlapping areas and minimizing the number of rectangles to be updated.

Profiling has shown, that the measuring process adds only neglectable overhead to the repaint process. However, the developer needs to take care of certain facts in his custom code to actually take advantage of the performance benefits the repaint optimizer offers.

<table>
<thead>
<tr>
<th>Code 1</th>
<th>Code 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>begin MyDrawingOperation(Buffer); Buffer.Changed; end;</td>
<td>begin if not Buffer.Measuringmode then MyDrawingOperation(Buffer); Buffer.Changed(RectOfAreaThatWasChanged); end;</td>
</tr>
</tbody>
</table>

Code 1 compared to Code 2 illustrates the required changes in pseudo-Pascal-code. As seen in the Code 2, a simple check for active measuring mode is introduced. In this case the actual drawing operation is omitted.

If the developer's code includes calls to the Changed method, those calls need to be changed to only represent the changed area instead of the whole buffer area. Keeping the Changed method unmodified will force a complete buffer invalidation, thus the effect of the repaint optimizer and the partial repaint therefore is reversed. Also, the custom code needs to be fully safe, meaning it has to offer full clipping support.

If the developer's custom code solely relies on the safe drawing operations provided by Graphics32, there is no need to change this code. However, doing
so will likely result in better performance especially if the custom code is calling many safe drawing operations. In this case introducing the changes of *Code 2* could simplify the measuring process a lot by overriding all subordinated checks by one superordinated check for measuring mode.

So, to sum up, there are two possible pitfalls in custom code that can occur with the new optimized repaint approach:

- **Changed calls need to be taken care of** (or else the whole buffer area is repainted).
- **Custom code has to be clippable**, i.e. needs to obey the buffer's ClipRect property (or else visual artifacts and failures appear).
4 Internals

As already mentioned above, the repaint optimizer is responsible for managing and optimizing changed areas, which are described by rectangles. Because there can be quite a lot changes happening between repaints, the area information has to be saved in a space-saving and performance-optimal structure.

The naive approach of saving all rectangles into a list and combining them once the repaint optimizer's method PerformOptimization is called is not suitable. With each TRect instance being 16 byte in size, the memory usage is unacceptable for large sets. The overhead of reallocating such structures is also noticeable. Moreover one has to make sure not to add several overlapping rectangles to the list. Using an algorithm for this matter adds complexity to the process. Thus, we need to find a better and more flexible way of managing possibly overlapping rectangles.

A totally different approach is to subdivide the buffer's dimension into a matrix. Each tile of this matrix would be responsible for a 32 x 32 pixel area in the buffer. New rectangles would simply be rendered to this matrix. The memory usage stays constant because the matrix size is in fixed relation to the size of the buffer size. Also, the problem of handling overlapping reactangles is also easily solved by rendering to the matrix. Additionally unifying tiles to bigger rectangles is obviously less complex than the approach needed for determining and unifying reactangles from a list structure. However, because each tile of our matrix only holds a binary value (filled or empty), the granularity (compare Figure 3) of this approach is quite high and thus too much information is lost.

Figure 3: Granularity comparison of MicroTiles and Tiles
The method used in Graphics32 1.8 is based on the tile method but improves the granularity problem by expanding each tile in the matrix from a binary representation to an integer representation.

![Diagram of 32 x 32 Pixel MicroTiles](image)

**Figure 4: Rectangle rendered to 32 x 32 Pixel MicroTiles**

So, instead of only having to restrict to full or empty as possible values, the tile contains exactly one rectangle that can further define the content. The two 16-bit values in the 32-bit integer of each tile represent the upper left and lower right corner of the inscribed rectangle relative to the upper left position of the tile (Figure 4). This allows a finer granularity and in the worst case (tile completely filled with one inscribed rectangle) the solution equals the tile based approach. However, most times the result is better, thus more information about the original shape is kept.

Because each coordinate is 8-bit wide, the tile size can scale up to 256 x 256 in size.

This method was first implemented by the developer of libart by the name MicroTile Arrays. Graphics32 implements an optimized version of its own and mixes that with some specialities: The MicroTiles Repaint Optimizer implements a simple adaptive algorithm that chooses between full scene, tile and MicroTile based operation mode depending on the current update situation.

For instance with many small rectangles (500+) the MicroTiles based optimization becomes less effective and can pose a performance overhead. In this case the adaptive algorithm will automatically downgrade to the next lower mode, which in that situation is the tiles based mode. Because the granularity is bigger in this mode, the optimization process is also less complex. Once the situation normalizes, it switches back to MicroTiles based operation mode. Thus, a good performance should be guaranteed in almost all cases.
5 Benchmark

The `Sprites_Ext` project was the most important test case of all because it is exceptional in the way that it shows both, the strength and weaknesses of the MicroTiles based approach. For our tests we've extended the project slightly to be able to measure the effective frames (or updates therefore) per second.

![Graph](a)
Figure 5: Benchmark results with **Sprites Ex** on PIII 1.13 GHz, WinXP, GeForce 2 mx

Figure 5 shows two results of benchmarking with different canvas resolutions. Each bitmap layer has a size of either 32 x 32 or 64 x 64 pixel picked randomly. The random seed used in the benchmark is reproducible for each test machine, thus a valid comparision is possible. As seen in the first graph (a) the MicroTiles based optimization works considerable better than the Tiles based approach. However, on our test machine it becomes less effective starting with 70 changed layers and finally the Tiles based approach outpaces it starting from 130 changed layers, because the higher granularity helps while combining the tiles to uniform rectangles. Using the MicroTiles based approach results in too many rectangles, which in this situation are less effective than fewer combined rectangles. This trend continues up to 600 rectangles. The Tiles based approach finally converges against the full scene repaint with MicroTiles being slightly worse due to the overhead involved. With so many layers the canvas area is almost completely covered with updates. For Graph (b) the results are slightly shifted due to the bigger canvas size. Both graphs show that the adaptive approach works well enough to be a feasible solution, however, the overhead of the balancing and scheduling used is noticeable in the switching regions of the graphs.
See Also

Examples, Paint Stages, TCustomPaintBox32.RepaintMode,Sprites Example, TCustomImage32, TCustomPaintBox32, TRepaintMode
Sampling and Rasterization

Sampling

Sampling is a very important concept within digital image processing and image analysis. Sampling is a process where color samples are acquired given their logical coordinates in the \((x, y)\) coordinate space. Graphics32 provides a special class called TCustomSampler, that provides the necessary mechanism for implementing different sampling techniques. A sampler can be conceived as a scalar function \(f(x, y)\) that returns a color sample given a logical coordinate \((x, y)\). A sample may be created synthetically (this is a common technique within ray-tracing, fractal rendering and pattern generation). It may also be acquired from some input hardware device. Another very common method for acquiring samples is resampling.
Resampling

Resampling is the process of reconstructing samples from a discrete input signal. The idea can also be extended from the 1D case to 2D. In the 2D case we can think of the bitmap as our signal. We have a number of pixels, aligned on a rectangular square grid. Hence we only know the actual color values at a number of discrete coordinates. In order to determine the color value of a sample at an arbitrary coordinate in a continuous image space, we need to perform interpolation for reconstructing this sample.

Descendants of TCustomResampler implement various algorithms for performing resampling and sample acquisition. A general algorithm reconstructing samples is to perform convolution in a local neighborhood of the actual sample coordinate. This method is used in TKernelResampler, where a convolution filter is specified by the TKernelSampler.Kernel property.

Graphics32 includes a class called TCustomKernel which is used as an ancestor class for various convolution kernels. For high quality resampling, one should consider using a kernel that approximates the ideal low-pass filter. The ideal low-pass filter is often referred to as a sinc filter. It can be described by the formula

\[ \text{sinc}(x) = \frac{\sin(\pi x)}{\pi x} \]

Since this function has infinite extent, it is not practical for using as a convolution kernel (because of the computational overhead). TWindowedSincKernel is a base class for kernels that use the sinc function together with a window function (also known as tapering function or apodization function). This way the kernel can be constrained to a certain width and reduce the amount of computations.

For further details about resampling, see Resamplers_Ex example project.
**Rasterization**

By *rasterizing* an image, we collect samples for each pixel of an output bitmap. The *rasterizer* is responsible for the order in which output pixels are sampled and how the destination bitmap is updated. A rasterizer class is derived from `TRasterizer`, by overriding the protected `DoRasterize` method.

Instances of `TRasterizer` needs to be associated with a sampler and an output destination bitmap. Some rasterization schemes, such as *swizzling*, may improve cache-performance for certain applications, since samples are collected in a local neighborhood rather than row by row. Rasterizers can also provide various transition effects for creating transitions between bitmaps.

Graphics32 includes the following rasterizers:

- `TRegularRasterizer` — rasterizes the bitmap row by row;
- `TProgressiveRasterizer` — rasterizes in a progressive manner by successively increasing the resolution of the image;
- `TTesseralRasterizer` — rasterization by sub-division;
- `TContourRasterizer` — the rasterization path is determined from the intensity of the collected samples.
**Nested sampling**

If the input of one sampler is the output from another, then we have a *nested sampler*. Nested samplers are derived from the class `TNestedSampler`.

By nesting samplers, it is possible to create a chain of nested samplers between the sampler that generates the actual sample and the rasterizer. This mechanism is illustrated in the below image.

![Nested Sampling Diagram](image)

There are many different useful applications for nested samplers. A sampler may be associated with a transformation. This will transform the input coordinate that is passed to the sampler at the next level.

It is possible to collect more than one sample in a local neighborhood of the pixel coordinate of the output pixel. This permits the use of techniques such as *super sampling*, where several samples are collected in order to estimate the color of the area covered by a pixel in the destination bitmap. If super sampling is not performed, it may cause jagginess and aliasing artifacts in the output image. However, this also depends on what kind of reconstruction method is used if samples are resampled.

Another important class of nested samplers is *kernel samplers*. Kernel samplers compute an output sample from several subsamples in a local region of the input coordinate. Each subsample is combined with a kernel value (contained within a `TIntegerMap` object). A class-specific kernel operation is used to update a buffer for each collected sample. This permits a very simplistic implementation of convolution and morphological operations.

The following is a list of the different nested samplers that are included in Graphics32.

- **Transformers**
  - `TTransformer` — transforms coordinates using an associated `TTransformation` object;
  - `TNearsetTransformer` — the same as above, but for nearest neighbor resampling.

- **Super samplers**
- **TSuperSampler** — performs regular super sampling;
- **TAdaptiveSuperSampler** — performs adaptive super sampling;
- **TPatternSampler** — performs sampling according to a predefined pattern.

**Kernel samplers**

- **TConvolver** — performs convolution;
- **TSelectiveConvolver** — performs selective convolution;
- **TDilater** — performs morphological dilation;
- **TEroder** — performs morphological erosion;
- **TExpander** — special expansion operation;
- **TContracter** — special contraction operation.

For further details about nested sampling, see the NestedSampling_Ex example project.
See Also

**Bitmap Image**

A bitmap image is represented with a `TBitmap32` object and is stored in the `Bitmap` property. Its scale and location within the control is determined by the following properties:

**BitmapAlign** — specifies if the bitmap image is positioned at the top-left corner of the control (baTopLeft), centered (baCenter), tiled (baTile) or its exact location is determined by `OffsetHorz` and `OffsetVert` properties;

**ScaleMode** — indicates if the bitmap image is displayed with its original size (smNormal), stretched to fit the control's boundaries (smStretch), proportionally resized to fit the control's boundaries (smResize) or proportionally scaled using its `Scale` property (smScale).

The bitmap image is combined with the back-buffer according to its `DrawMode` property. And the quality of its resampling is determined by the `StretchFilter` property. If its `DrawMode` is `dmCustom`, the bitmap will fire a series of `OnPixelCombine` events.
See Also

Paint Stages

An order in which TImage32 blends layers and bitmap image to its back-buffer is specified by paint stages.

For example, the default sequence of operations includes:

- Clearing the visible area of the background, that is the parts of the buffered area which are not covered by the bitmap image, or the whole buffer, if the bitmap image is not in dmOpaque draw mode;
- Drawing the dotted frame around the control boundaries (design-time only);
- Drawing the scaled bitmap image;
- Framing the area of the scaled bitmap image with the dotted frame (design-time only);
- Drawing layers;

It is possible to change the order in which stages execute at run-time, add new stages, delete old ones, etc., using the PaintStages property of TImage32, which is basically a dynamic indexed list of stages.
TPaintStage Record

Each paint stage is defined with a TPaintStage record:

```pascal
type TPaintStage = record
  DsgnTime: Boolean;
  RunTime: Boolean;
  Stage: Cardinal; // a PST_* constant
  Parameter: Cardinal; // an optional parameter
end;
```

where the Stage member holds one of the Paint Stage Constants and defines the action associated with the stage.

All stages include additional parameter, which may be ignored or may be used to store additional stage options. For example, PST_DRAW_LAYERS stage uses its parameter as a 32-bit mask to filter out invisible layers.

By default, TImage32 contains the following stages:

<table>
<thead>
<tr>
<th>#</th>
<th>DsgnTime</th>
<th>RunTime</th>
<th>Stage</th>
<th>Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>True</td>
<td>True</td>
<td>PST_CLEAR_BACKGND</td>
<td>not used</td>
</tr>
<tr>
<td>1</td>
<td>False</td>
<td>True</td>
<td>PST_CONTROL_FRAME</td>
<td>not used</td>
</tr>
<tr>
<td>2</td>
<td>True</td>
<td>True</td>
<td>PST_DRAW_BITMAP</td>
<td>not used</td>
</tr>
<tr>
<td>3</td>
<td>False</td>
<td>True</td>
<td>PST_BITMAP_FRAME</td>
<td>not used</td>
</tr>
<tr>
<td>4</td>
<td>True</td>
<td>True</td>
<td>PST_DRAW_LAYERS</td>
<td>$80000000</td>
</tr>
</tbody>
</table>

See 'Using Layers' for explanation on parameter value in PST_DRAW_LAYERS stage.
Customizing TImage32 at Run-Time

A PST_CUSTOM stage deserves a little bit deeper explanation. It causes the control to issue an OnPaintStage event, thus allowing to change TImage32 behavior at run-time.

The OnPaintStage event is of a TPaintStageEvent type:

```
type TPaintStageEvent = procedure(Sender: TObject; Dest: TBitmap32; StageNum: Cardinal) of object;
```

In the event handler, application can perform some custom operations over the back-buffer of the control.

Note, that by default, TImage32 does not generate OnPaintStage. In order to make it to do so, you have to insert a new stage in the PaintStages list, and set its Stage to PST_CUSTOM, or change one of the existent stages, for example:

```
type TForm1 = class(TForm)
            Image32: TImage32;
      procedure Image32InitStages(Sender: TObject); // OnInitStages
      procedure Image32PaintStage(Sender: TObject; Dest: TBitmap32; StageNum: Cardinal); // OnPaintStage
    private
     { Private declarations }
    public
     { Public declarations }
end;
```

```
var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.Image32InitStages(Sender: TObject);
begin
  // change default PST_CLEAR_BACKGND (0-th stage) to a custom handler
  with Image32.PaintStages[0] do
  ```
begin
  Stage := PST_CUSTOM;
  Parameter := 1; // use parameter to tag the stage
end;

// insert another custom stage after the bitmap image
// was drawn, but before the control starts painting layers
with Image32.PaintStages.Insert(4) do
begin
  // Note that for new stages RunTime = True by default
  Stage := PST_CUSTOM;
  Parameter := 2; // use parameter to tag the stage
end;
end;

procedure TForm1.Image32PaintStage(Sender: TObject; Dest: TBitmap32; StageNum: Cardinal);
begin
  // OnPaintStage Handler
  case Image32.PaintStages[StageNum].Parameter of
    1: // do something with the background
    2: // call another handler
  end;
end;
**GDI Overlays**

[ Note: Most likely GDI Overlays will not be used in future versions. Do not use it ]

A final step in TImage32 repainting is the drawing of GDI overlays. This operation is performed after the bitmap image and layers have been combined in a back buffer and copied to the screen canvas.

At this stage, TImage32 fires the `OnGDIOverlay` event, where you can perform drawing using the standard `Canvas` of the TImage32.

The main reason for introducing this stage is that painting of GDI overlays does not affect the contents of the buffer, that is changes in overlay image will not cause the buffer invalidation, however GDI overlays have to be repainted each time the control repaints itself, and they are not flicker-free.
See Also


Copyright ©2000-2007 Alex Denisov and Contributors - Graphics32 v1.8.3 - Build on 4-March-2007
Using Layers

TImage32 holds an indexed collection of layers, referenced by the Layers property. The number of layers is limited only by the amount of free memory. Each layer is an entity, which basically 'knows' how to paint itself to the back-buffer of the control and how to interact with the mouse. Layers are indexed and their indexes are similar to the Z-order of standard controls. Layers with smaller indexes are considered to lie deeper. This common behavior is implemented in the TCustomLayer class.
Common Properties and Methods

The basic layer's behavior is controlled by its `LayerOptions` property, which is a 32-bit unsigned integer value composed of `Layer Options Bits`. This property allows for fast and relatively simple referencing of layers and groups of layers. Consider, for example, the LOB_VISIBLE bit (31-st bit in layer options). When TImage32 repaints its layers at PST_DRAW_LAYERS stage, it uses the stage parameter from corresponding `TPaintStage record` as a bit-mask (default value is LOB_VISIBLE=$80000000). Being compared (logical AND operation) with `LayerOptions` of each layer, this mask determines whether the layer should be painted or not.

Similar situation is with reaction to mouse messages (see below).

Using `LayerOptions` you can easily customize the appearance and modify the order in which TImage32 repaints layers. For example, you may assign several categories to layers, using lower 24-bits of `LayerOptions` and then just change the mask in corresponding `TPaintStage record` to specify exactly which category you want to be displayed.

More of that, remember, that you can insert several PST_DRAW_LAYERS paint stages (even before the PST_DRAW_BITMAP stage), each with its own mask, or, alternatively, call `ExecDrawLayers` while handling PST_CUSTOM stages. Basically it means, that you can realize almost any complex repainting scheme, like overlay layers, underlay layers, layers, which are always on top... or always in the background, etc.
Tracking the Mouse

Each layer is capable of responding to mouse down/move/up messages, which are routed to `OnMouseDown`, `OnMouseMove` and `OnMouseUp` events of the container (TImage32 control). Only one layer can receive a mouse message at the time.

Searching for the layer that receives a mouse message starts from the top-most layer and ends when the first layer satisfying both of the following conditions is found:

- Layer has LOB_MOUSE_EVENTS bit activated in its `LayerOptions`;
- Layer passes a hit test (see below);

There is also a possibility to disable passing of all mouse messages to layers. Just set the `MouseEvents` property of the layer collection to `False`, and TImage32 will generate mouse events right away, without checking for possible receiving layer.

As stated earlier, layer should pass a hit test in order to receive mouse message. The hit test is, basically a boolean response to a couple of coordinates, accomplished by the `HitTest` method:

```
function HitTest(X, Y: Integer): Boolean; virtual;
```

X and Y parameters here are the coordinates of the point specified relative to the top-left corner of TImage32 (in pixels). The function returns a value that indicates that the layer is 'there'. This function is overridden in descendants, for example, in `TBitmapLayer`, this hit test may take into consideration values stored in the alpha channel of the contained bitmap. You can also write a handler for `OnHitTest` event to customize hit tests at run-time.

Similar to standard controls, layers may also capture mouse messages. By default, all mouse messages are captured automatically once the mouse is pressed on top of the layer and until the mouse is released. The layer that has captured the mouse is pointed by the `MouseListener` property of the layer collection. (There is a few things to work on here... will do it later).
Painting Layers

Layers are painted in bottom-to-top order, starting from lower indexes. By default, only visible layers are painted, that is the ones with LOB_VISIBLE bit set, but as it was shown above, this order may be changed.
Positioned Layers

Positioned layers are the layers, you will probably use most of the time. The base class for positioned layers is `TPositionedLayer`. In addition to basic layer behavior, it introduces `Location` property (of the `TFloatRect` type), which specifies layer's position and size. Having location specified as a floating point rectangle helps to avoid rounding errors when layers are resized.

The location can be specified in pixels, relative to `TImage32` top-left corner, or in scaled pixels, relative to the top-left corner of the `bitmap image`. In the second case, the scale of the layer coincides with the scale of the bitmap image and the actual location of the layer relative to the top-left corner of `TImage32` may be obtained with its `GetAdjustedLocation` method.
See Also

Contributors

(in alphabetical order):

**v1.8.x**

**Mattias Andersson** <mattias@centaurix.com>:

**GR32.pas:**
- Added fast integer versions of HSLtoRGB and RGBtoHSL
- Added TCustomSampler and TCustomResampler
- Added dmTransparent draw mode
- Added TBitmap32.WrapMode property
- Added TBitmap32.PixelW property

**GR32_LowLevel.pas:**
- Added clamp, wrap and mirror routines

**GR32_OrdinalMaps.pas:**
- Initial developer
- Added TBooleanMap, TWordMap and TIntegerMap

**GR32_Resamplers.pas:**
- Initial developer
- Added TCustomKernel and descendant classes
- Added TBitmap32Resampler and descendant classes
- Added TNestedSampler
- Added TTransformer
- Added TSuperSampler, TAdaptiveSuperSampler and TPatternSampler
- Added TKernelSampler and descendant classes
- Added auxiliary routines for convolution and morphological operations
- Optimized TKernelResampler.GetSample (with Michael)
- Added BlendTransfer routine
- Added CreateJitteredPattern routine

**GR32_Rasterizers.pas:**
- Initial developer
- Added TRasterizer and descendant classes

**GR32_Transforms.pas:**
- Updated Transform procedure

**GR32_ExtImage.pas:**
- Initial developer
- Added TSyntheticImage and TRenderThread
- Added Rasterize auxiliary routine

**GR32_Containers.pas:**
- Added TClassList

**Examples:**
- Mandelbrot Example
- Nested Sampling Example
- Resampler Example

**Andre Beckedorf <andre@metaexception.de>:**

**GR32.pas:**
- Added MeasuringMode and changed area notification required for repaint optimizer (BeginMeasuringMode, EndMeasuringMode, OnAreaChanged)

**GR32_Containers.pas:**
- Added TPointerMap and TPointerMapIterator
- Added TRectList

**GR32_Image.pas:**
- Added support for repaint optimization to display components (TCustomPaintBox32, TCustomImage32, TCustomImgView32)
- Added property to control scrollbar visibility (TCustomImgView32.Scrollbars.Visibility)
- Added independent X and Y scale properties in TCustomImage32 along with new OnScaleChange event
- Added additional scale modes smOptimal and smOptimalScaled
- Added RepaintMode property to control whether to use repaint optimizer, direct or full scene repaint

**GR32_Layers.pas:**
- Replaced CoordXForm property with more flexible GetViewportShift and GetViewportScale methods in TLayerCollection (e.g. for nested layers)
- Added support for repaint optimization to TCustomLayer descendants
- Added new Update methods to TCustomLayer

**GR32_LowLevel.pas:**
- Initial support for FastCode routines

**GR32_MicroTiles.pas:**
- MicroTiles and Tiles auxiliary routines
- MicroTiles Repaint optimizer
- Adaptive balancing
- MMX optimization for MicroTileUnion

**GR32_Polygon.pas:**
- Added additional 2x and 32x supersampling mode
- Performance optimization

**GR32_RepaintOpt.pas:**
- Abstract Repaint optimizer class

**GR32_System.pas:**
- Added support for EMMX detection (used in repaint optimizer)
- Added TPerfTimer class along with global performance timer
- Added GetTickCount derivative for Linux
- Fixed AV when using Graphics32 in DLLs on WinXP

**Examples:**
- Modified Sprites_Ex, ImgView32_Ex, GradLines_Ex, Polygons_Ex and Image32_Ex Example

**Michael Hansen** <dyster_tid@hotmail.com>:

**GR32 pasa:**
- Fixed Math related additions
- Auxiliary rectangle routines
- Misc. array types
- Added TColor32Component, TColor32Components and related routines (TColorEntry updated)
- Optimized TBitmap32.ResetAlpha

**GR32_Layers.pas:**
- Enhancements to TRubberbandLayer

**GR32_Lowlevel.pas:**
- MMX optimized FillLongword

**GR32_Filters.pas:**
- ASM/MMX/EMMX optimized Invert
- Added fast CopyComponents routine for copying selected ARGB components btwn. bitmaps
- Added ASM/MMX/EMMX ApplyBitmask routine for bitwise Xor/Or/And operations using a bitmask
**GR32_OrdinalMaps.pas (new unit):**
- Minor tweaks for TByteMap

**GR32_Resamplers.pas (new unit):**
- Added kmTableLinear and kmTableNearest kernel modes
- Optimized TLinearResampler
- Fixed TDraftResampler
- GetSampleFloat optimizations
- New overloaded BlendTransfer routine, supporting TBlendRegEx and MasterAlpha
- New PixelAccessMode pamTransparentEdge as a dynamic and fast alternative to SetBorderTransparent
- Minor tweaks in TPatternSampler
- Minor optimizations in pamTransparentEdge related routines

**GR32_Transforms.pas:**
- Added following transformation types:
  - TTwirlTransformation
  - TBloatTransformation
  - TDisturbanceTransformation
  - TFishEyeTransformation
  - TRemapTransformation
- Additional performance tuning

**GR32_VectorMaps.pas (new unit):**
- Initial developer
- Added TVectorMap, supporting loading and saving in Photoshop .msh format

**Examples:**
- Image Warping Example
- Visualization Example
- Resample Example
- Texture blend Example
- Additional updates to various examples (see sources)

**v1.7 and v1.6**

Mattias Andersson <mattias@centaurix.com>:

**GR32.pas:**
- Tweaks
- PixelF/X property
- Some minor CLX/Kylix tweaks
- Added CombineMode

**GR32_Image.pas:**
- Ported to CLX/Kylix

**GR32_Polygons.pas:**
- Added TPolygon32.ContainsPoint method

**GR32_Transforms.pas:**
- Added sfCosine resampling filter

**GR32_Blend.pas**
- Pascal and MMX version of Merge algorithm
- Added MMX versions of almost all PixelCombine functions

**Andre Beckedorf** <andre@metaexception.de>:

**GR32.pas:**
- Added Unicode text methods
- Ported to CLX/Kylix
- Added CombineMode
- Added clipping rect support
- pixel-perfect line drawing methods (LineS, LineTS, LineAS)

**GR32_Image.pas:**
- Some part of the CLX/Kylix related changes in this unit
- Added new overrideable Mouse* methods with Layer information
- Added OverSize property in TCustomImgView32

**GR32_Layers.pas:**
- enhancements to TRubberbandLayer

**GR32_Polygon.pas:**
- Added option for dynamic antialias level adjustment at runtime along with 4x supersampling mode
- Added support for custom polygon fillings (TCustomPolygonFiller, TBitmapPolygonFiller) and transformations
- Added support for clipping to rect
- Added PolylineXSP and PolyPolylineXSP
- Additional performance tuning

**GR32_RangeBars.pas:**
- Ported to CLX/Kylix
GR32_System.pas:
- Added support for SSE, SSE2, 3DNow and 3DNow Ext detection

GR32_Transforms.pas:
- Update to transformation classes to make them suited for general usage

Michael Hansen <dyster_tid@hotmail.com>:
GR32.pas:
- PixelF/X property
- MMX optimized GetPixelF/X FS/XS functions

GR32_Blend.pas:
- Added Difference and Exclusion PixelCombine functions (Pascal)
- Added PixelCombine function setup, to support non MMX cpus
- Worked out bugfix for "not fully opaque" problem in some blendroutines

GR32_Transforms.pas:
- Added sfDraft resampling routines
- MMX optimized sfLinear and sfDraft
- Updated Transform to use PixelX and PixelXS

Contributors:
Arioch /BDV/:
GR32.pas
- Added WinPalette function. (merged by Mattias with some modifications)

Thomas Bauer:
GR32.pas:
- Some bugfixes (merged by Andre)
GR32_Transforms.pas:
- FullEdge related bugfixes (merged by Andre)

Soumitra Bhattacharjee:
GR32_Polygons.pas:
- Added PtInPolygon function. (merged by Mattias with some modifications)

Greg Chapman:
GR32_Filters.pas:
- Report and bugfix for the CheckParams problem in ApplyLUT. (merged
by Andre)

**Cleber:**
*GR32.pas:*
- Added Color32ToRGB(A). (merged by Andre)

**Dieter Köhler:**
*GR32_Layers.pas:*
- TLayerCollection.LocalToViewport
- TLayerCollection.ViewportToLocal
- TPositionedLayer.GetAdjustedRect (all merged by Andre)

**Marc Lafon:**
*GR32_Image.pas:*
- Custom color properties for the bars (merged by Andre)

**Peter Larson:**
*GR32_Polygons.pas:*
- Initial work on polygon transformation and bitmap pattern filling.
- Some optimizations in polygon setup.
*GR32_Transforms.pas:*
- TransformPoints routine.

**Gerd Platl:**
*Documentation:*
- Illustration for TPolyFillMode

**Andrew P. Rybin:**
*GR32_Image.pas:*
- MouseEnter/MouseLeave (merged by Andre with some changes to match structure)

**J. Tulach:**
*GR32.pas:*
- Additional handling of chroma transparency in TBitmap32.Assign
  (merged by Andre with some modifications and comments).
*GR32_Transforms.pas:*
- C++ Builder workaround (merged by Andre)

**Bob Voigt:**
*GR32_Blend.pas:*
- ColorMul and ColorDiv (merged by Andre)

**Jens Weiermann:**

*GR32_Filters.pas:*

- enhancements to ApplyLUT and ColorToGrayscale that preserve alpha channel (merged by Andre)

**Norbert Witternigg:**

- updated the example project files to Borland C++ Builder 6.
See Also

Repaint Optimization
property BitmapHandle: HBITMAP; // read-only;
**Description**

Returns current *bitmap handle*. See *HBITMAP* in Windows SDK documentation for more information.

Note, this property is not the same as *Handle*, which returns *device handle* (HDC).

*This property is only available in the VCL version. For CLX use the property *Pixmap* instead.*
See Also

Handle, PixelPtr, Pixmap, ScanLine
TBitmap32.BitmapInfo

```pascal
property BitmapInfo: TBitmapInfo; // read-only;
```
Description

Returns a `BITMAPINFO` record corresponding to a current DIB data. See `BITMAPINFO` in Windows SDK documentation for more information.

*This property is only available in the VCL version.*
See Also

PixelPtr, ScanLine
TBitmap32.Bits

**property** Bits: **PColor32Array**; // *Read-only*

**type** TColor32Array = array [0..0] of **TColor32**;

**type** PColor32Array = ^TColor32Array;
Description
The bits property contains the address of the first (top-left) pixel in a bitmap. If the bitmap is not allocated (width or height is zero), the returned address is nil.

Note, that numbering of rows in Graphics32 starts from the top-most one.

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 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

TCustomMap.Height, PixelPtr, ScanLine, TColor32, TColor32Array, TCustomMap.Width
property Canvas: TCanvas; // read-only;
Description

Returns current *Canvas instance*. You can use that to draw onto the bitmap. However the methods provided by *TCanvas* are mostly slower than using their native Bitmap32 counterparts. So *try to avoid the usage of Canvas where possible.*

**Special note for CLX users:** Due to the way QT is handling image and bitmap data, please try to avoid using only single Canvas operations. Every time you access the Canvas, the internal *QImage* is unavoidably copied to a *QPixmap* representative and copied back once the Bits property or any other internal method is accessed. So, if you access the Bits and Canvas properties in short interval after each other, that poses a heavy performance hit due to copying the *QImage* to *QPixmap* back and forth.
TBitmap32.Clipping

property Clipping: Boolean;
Description
Determines if a clipping rect was set.
TBitmap32.ClipRect

property ClipRect: TRect;
**Description**
Determines the clipping rectangle that all [safe drawing operations](#) are clipped to.

If no clipping rectangle is set, ClipRect is equal to the current dimension of the bitmap.

*Note:* Validation is performed upon setting of a clipping rect, so the clip rect is always sane.
### TBitmap32.CombineMode

<table>
<thead>
<tr>
<th>property</th>
<th>CombineMode: TCombineMode;</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>TCombineMode = (cmBlend, cmMerge);</td>
</tr>
</tbody>
</table>
Description

TCombineMode defines how drawing operations on the bitmap combine the foreground color with the background. It also determines how the bitmap is combined with the background of another bitmap when DrawMode is set to dmBlend.

- **cmBlend** - Fast blending of foreground color with the background color using the supplied alpha. This method is not suited for working and preserving alpha-channels. Use this if you want to blend directly to the display buffer. Internally uses Blend and BlendEx routines.
- **cmMerge** - Uses a completely different formula that also merges the alpha-channels. This mode is slower than blending but is suited for working with alpha-channels (eg. when composing in temporary buffers). Internally uses Merge and MergeEx routines.
See Also

Blend, BlendEx, Draw, DrawMode, DrawTo, Merge, MergeEx, TCombineMode
TBitmap32.DrawMode

property DrawMode: TDrawMode;

type TDrawMode = (dmOpaque, dmBlend, dmCustom);
Description
Specifies how the bitmap should be combined with a background during pixel transfer and similar operations.

In **dmOpaque** mode, new pixels simply replace the background pixels. In **dmBlend** mode, they are combined using the alpha blending operation defined by [CombineMode](#). This property is used while copying one bitmap into another, scaling, performing linear transformations etc. The **dmCustom** mode allows implementation of custom blending functions using the [OnPixelCombine](#) call-back event.
See Also

CombineMode, Draw, DrawTo, OnPixelCombine, TDrawMode
TBitmap32.Font

    property Font: TFont;
**Description**

Specifies a current font used by text output functions.
See Also

RenderText, TextOut, UpdateFont
This following only applies to the VCL version of Graphics32:

```
property Handle: HDC; // read-only
```
Description

Provides 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:

```pascal
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.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 can change after resizing.

This example is just a simple illustration of using the Handle property. For more information on using the TCanvas object with TBitmap32, see the Canvas property.

This following only applies to the CLX version of Graphics32:

```pascal
property Handle: QPainterH; // read-only
```
Description

This provides a pointer to the current instance of the QPainterH in TBitmap32.

You can use this QPainter instance in external functions. However please notice, that the QPainter is not associated with the QPixmap of TBitmap32. You'll have to call QPainter_begin and QPainter_end manually to do so. See the QT docs for more information on this topic.
See Also

Canvas, TBitmap32

Copyright ©2000-2007 Alex Denisov and Contributors - Graphics32 v1.8.3 - Build on 4-March-2007
TBitmap32.MasterAlpha

property MasterAlpha: TColor32;
**Description**

When blending a bitmap to the screen or to another bitmap, **MasterAlpha** normally 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 used only for bitmap blending, it does not affect pixel/line drawing and other similar routines.

When blending bitmaps in dmCustom draw mode, this property may be used for other purposes. Custom combining routine may use all 32 bits of **MasterAlpha**.

If the bitmap is in dmBlend draw mode and **MasterAlpha** is not in the $[0…255]$ range, the blending result is not specified, it is your responsibility to keep **MasterAlpha** consistent with current draw mode.
See Also

DrawMode, TColor32
TBitmap32.MeasuringMode

property MeasuringMode: Boolean; // readonly
Description

Determines if the bitmap is in measuring mode. In this mode drawing to the bitmap is only simulated. That is, drawing is actually omitted, but the OnAreaChanged event is still triggered.

For detailed description please refer to the additional topic dealing with repaint optimization.
See Also

MeasuringMode, OnAreaChanged, Repaint Optimization
TBitmap32.OuterColor

property OuterColor: TColor32;
**Description**

This property specifies the color returned by [Pixels](#) property when reading the pixel with coordinates that lie outside the bitmap. The default value is $00000000$, which corresponds to a fully transparent black. It is also used when performing linear transformations of a bitmap.
See Also

Pixel, TColor32
TBitmap32.PenColor

property PenColor: TColor32;
Description
Simulates $TCanvas.Pen.Color$ property. **PenColor** is used exclusively in **MoveTo/LineTo** functions.
See Also

LineTo, MoveTo, TColor32
TBitmap32.Pixel

```plaintext
property Pixel[X, Y: Integer]: TColor32; default;
property PixelS[X, Y: Integer]: TColor32;
property PixelW[X, Y: Integer]: TColor32;
property PixelX[X, Y: TFixed]: TColor32;
property PixelXS[X, Y: TFixed]: TColor32;
property PixelXW[X, Y: TFixed]: TColor32;
property PixelF[X, Y: Single]: TColor32;
property PixelFS[X, Y: Single]: TColor32;
property PixelFW[X, Y: Single]: TColor32;
```
Description

**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 the bitmap boundary, the value specified by **OuterColor** is returned. Writing with invalid coordinates will have no effect.

**PixelW** is the wrapping version of the **Pixel** property. When reading pixels from outside the bitmap boundary, the value returned is determined by the currently selected **WrapMode**. Thus always safe.

**PixelX** provides a method for accessing the pixels in fixed-point coordinates. The returned color is computed by performing linear interpolation on four adjacent pixels. Similarly, when pixels are set, they are antialiased.

**PixelXS** is a 'safe' version of the above property.

**PixelXW** is a wrapping version of the property **PixelX**.

**PixelF** is similar to **PixelX** but works with floating point coordinates instead of fixed point.

**PixelFS** is a 'safe' version of the above property.

**PixelFW** is a wrapping version of the above property.
See Also

Naming Conventions, OuterColor, SetPixel, TColor32, TFixed, WrapMode
TBitmap32.PixelPtr

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

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.

Pixel with (0, 0) coordinates has the same address as specified in `Bits` property.
See Also

Bits, Pixel, ScanLine, TBitmap32, TColor32
**TBitmap32.Pixmap**

```plaintext
property Pixmap: QPixmapH; // read-only;
```
**Description**

Returns a pointer to the current Pixmap instance. You can use that if you're working directly with QT related methods.

**Important Note:** Due to the nature of QT handling images and bitmap information, there is a need to notify TBitmap32 about changes you made to the QPixmap. You can use the PixmapChanged property to do so. This will copy the Pixmap content as soon as the Bits property is accessed or some internal function is accessing the image data directly. This will make sure that the content is copied from the QPixmap to the internal QImage in TBitmap32.

*This property is only available in the CLX version. For VCL use the property **BitmapHandle** instead.*
See Also

BitmapHandle
TBitmap32.PixmapChanged

```
property PixmapChanged: Boolean;
```
**Description**

Returns whether the current QPixmap instance has changed or not. You can use that if you're working directly with QT related methods.

**Important Note:** Due to the nature of QT handling images and bitmap information, there is a need to notify *TBitmap32* about changes you made to the *QPixmap*. You can use this property to do so. This will copy the *QPixmap* content as soon as the Bits property is accessed or some internal function is accessing the image data directly. This will make sure that the content is copied from the *QPixmap* to the internal *QImage* in *TBitmap32*.

*This property is only available in the CLX version.*
See Also

Pixmap
TBitmap32.Resampler

property Resampler: TCustomResampler;
Description

Resampler specifies the resampling method used for image stretching.
See Also

Draw, Sampling and Rasterization, TCustomResampler
TBitmap32.ResamplerClassName

```pascal
property ResamplerClassName: string;
```
Description

ResamplerClassName determines which class to instantiate for Resampler.
See Also

Draw, Resampler, TCustomResampler
TBitmap32.ScanLine

**property** ScanLine[Y: Integer]: PColor32Array; // read-only
Description

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

Bits, PixelPtr, TColor32Array
TBitmap32.StippleCounter

property StippleCounter: Single;
Description
Use StippleCounter property to access current value of the stipple counter.
See Also

GetStippleColor, Line Patterns, SetStipple, StippleStep

Copyright ©2000-2007 Alex Denisov and Contributors - Graphics32 v1.8.3 - Build on 4-March-2007
property StippleStep: Single;
Description
Provides access to stipple step. Default value is 1.0.
It is possible to set step to fractional or negative value.
See Also

GetStippleColor, Line Patterns, SetStipple, StippleCounter
property StretchFilter: TStretchFilter;

type TStretchFilter = (sfNearest, sfDraft, sfLinear, sfCosine, sfSpline, sfLanczos, sfMitchell);
Description
This property is *deprecated* as of version 1.8 of Graphics32. Please use Resampler and ResamplerClassName instead.

**StretchFilter** specifies color interpolation method for image stretching as well as for some other operations, like linear transformations. Some functions (transformations, for example) can interpolate only using the **sfNearest** and **sfLinear** filters, other values will be implicitly processed as **sfLinear**.
See Also

Draw, TStretchFilter
TBitmap32.WrapMode

property WrapMode: TWrapMode;

type TWrapMode = (wmClamp, wmRepeat, wmMirror);
**Description**

Specifies how the wrapping *Pixel* property should behave.
See Also

Pixel, TWrapMode
TBitmap32.BeginMeasuring

```plaintext
procedure BeginMeasuring(const Callback: TAreaChangedEvent);
```
Description
Attaches a callback method to the OnAreaChanged event and switches the bitmap into measuring mode. The event handler previously attached to the OnAreaChanged event is preserved and will be reset once measuring is stopped by calling EndMeasuring.
See Also

`EndMeasuring`, `MeasuringMode`, `OnAreaChanged`, `TAreaChangedEvent`
TBitmap32.CanvasAllocated

function CanvasAllocated: Boolean;
**Description**
Indicates whether the internal [Canvas](#) has been allocated.
See Also

Canvas, RenderText, UpdateFont
TBitmap32.Clear

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

Fills the entire bitmap with **FillColor**. If there is no argument specified, method uses clBlack32 ($FF000000).
See Also

TCustomMap.Delete, TColor32
TBitmap32.Create

constructor Create; override;
**Description**

Creates and initializes an instance of `TBitmap32`.

After calling the inherited constructor, `Create` initializes the following properties:

- `BitmapInfo` is filled with data corresponding to 32-bit DIBs;
- `DrawMode` to `dmOpaque`;
- `MasterAlpha` to 255 (`$FF`);
- `OuterColor` to transparent black: `$00000000`;
- `PenColor` to `clWhite32`;
- `StippleCounter` to 0;
- `StippleStep` to 1.0;
- `StretchFilter` to `sfNearest`;
See Also

BitmapInfo, DrawMode, MasterAlpha, OuterColor, PenColor, StippleCounter, StippleStep, TBitmap32, TStretchFilter
TBitmap32.DeleteCanvas

procedure DeleteCanvas;
**Description**

Destroys the internal [Canvas](#) object.
See Also

Canvas, CanvasAllocated, RenderText

Copyright ©2000-2007 Alex Denisov and Contributors - Graphics32 v1.8.3 - Build on 4-March-2007
TBitmap32.Destroy

```cpp
    destructor Destroy; override;
```
Description
Destroys the bitmap object and frees all associated memory.
Do not call Destroy directly, use Free method instead.
TBitmap32.Draw

procedure Draw(DstX, DstY: Integer; Src: TBitmap32); overload;

procedure Draw(DstX, DstY: Integer; const SrcRect: TRect; Src: TBitmap32); overload;

procedure Draw(const DstRect, SrcRect: TRect; Src: TBitmap32); overload;

procedure Draw(const DstRect, SrcRect: TRect; hSrc: HDC); overload; // VCL version

procedure Draw(const DstRect, SrcRect: TRect; SrcPixmap: QPixmapH); overload; // CLX version
**Description**

Renders the image specified by `Src/hSrc` parameter (or part of it specified by `SrcRect` 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` to determine how it should be blended with the background, and if stretching, `Src.StretchFilter` specifies how the image should be stretched.

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 version that uses the `SrcPixmap` is only available in the CLX version of Graphics32. You may use it to transfer data directly from QPixmap instances such as `TBitmap`.

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

BlockTransfer, DrawMode, DrawTo, StretchFilter, StretchTransfer, TBitmap32, TRect
TBitmap32.DrawTo

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Signature</th>
</tr>
</thead>
<tbody>
<tr>
<td>overload</td>
<td>procedure DrawTo(Dst: TBitmap32); overload;</td>
</tr>
<tr>
<td>overload</td>
<td>procedure DrawTo(Dst: TBitmap32; DstX, DstY: Integer); overload;</td>
</tr>
<tr>
<td>overload</td>
<td>procedure DrawTo(Dst: TBitmap32; DstX, DstY: Integer; const SrcRect: TRect); overload;</td>
</tr>
<tr>
<td>overload</td>
<td>procedure DrawTo(Dst: TBitmap32; const DstRect: TRect); overload;</td>
</tr>
<tr>
<td>overload</td>
<td>procedure DrawTo(Dst: TBitmap32; const DstRect, SrcRect: TRect); overload;</td>
</tr>
<tr>
<td>overload</td>
<td>procedure DrawTo(hDst: HDC; DstX, DstY: Integer); overload;</td>
</tr>
<tr>
<td>overload</td>
<td>procedure DrawTo(hDst: HDC; const DstRect, SrcRect: TRect); overload;</td>
</tr>
</tbody>
</table>
Description

The **DrawTo** method renders the bitmap (or part of it specified by **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 renders itself to destination object. See the **Draw** description for details.
See Also

Draw, TBitmap32, TRect
TBitmap32.Empty

function Empty: Boolean; override;
**Description**

Returns *True* if the bitmap is empty, that is *Width* or *Height* is zero. If the bitmap is empty, it has no device context (*Handle* property) allocated.
See Also

Handle, TCustomMap.Height, TCustomMap.Width
TBitmap32.EndMeasuring

procedure EndMeasuring;
**Description**

Stops the measuring mode and switches the bitmap back into normal operation mode. If there was an event handler previously attached to the OnAreaChanged event it is restored.
See Also

BeginMeasuring, MeasuringMode, OnAreaChanged
TBitmap32.FillRect

```plaintext
procedure FillRect(X1, Y1, X2, Y2: Integer; Value: TColor32);
procedure FillRectS(X1, Y1, X2, Y2: Integer; Value: TColor32);
    overload;
procedure FillRectT(X1, Y1, X2, Y2: Integer; Value: TColor32);
procedure FillRectTS(X1, Y1, X2, Y2: Integer; Value: TColor32);
    overload;
procedure FillRectS(const ARect: TRect; Value: TColor32);
    overload;
procedure FillRectS(const ARect: TRect; Value: TColor32);
    overload;
```
Description

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 \( X_2 > X_1; Y_2 > Y_1 \).

The function fills the rectangle up to, but not including the right column and bottom row. When \( X_2 \leq X_1 \) or \( Y_2 \leq Y_1 \), nothing is drawn.
See Also

FrameRect, Naming Conventions, TColor32
**TBitmap32.FlipHorz**

`procedure FlipHorz(Dst: TBitmap32);`
Description
FlipHorz flips the image horizontally. When Dst parameter is specified, the transformed version of the image is copied into Dst, otherwise the procedure performs the in-place flipping.
See Also

FlipVert, TBitmap32
TBitmap32.FlipVert

```
procedure FlipVert(Dst: TBitmap32);
```
**Description**

FlipVert flips the image vertically. When Dst parameter is specified, the transformed version of the image is copied into Dst, otherwise the procedure performs the in-place flipping.
See Also

FlipHorz, TBitmap32
TBitmap32.FrameRect

```plaintext
procedure FrameRectS(X1, Y1, X2, Y2: Integer; Value: TColor32; overload;
procedure FrameRectTS(X1, Y1, X2, Y2: Integer; Value: TColor32; overload;
procedure FrameRectS(const ARect: TRect; Value: TColor32; overload;
procedure FrameRectTS(const ARect: TRect; Value: TColor32; overload;
procedure FrameRectTSP(X1, Y1, X2, Y2: Integer);
```
**Description**

Draws a rectangle frame. Row with X2 coordinate and column with Y2 coordinate are excluded.

If X2 ≤ X1 or Y2 ≤ Y1, the function does not draw the rectangle.

`FrameRectTSP` version supports line patterns.
See Also

FillRect, LineTo, Naming Conventions, TColor32
TBitmap32.GetStippleColor

function GetStippleColor: TColor32;
**Description**

Returns color corresponding to the current counter position in stipple pattern. If the counter step is fractional, returned color will be interpolated. Calling **GetStippleColor** automatically changes the **StippleCounter** value by **StippleStep**.
See Also

Line Patterns, SetStipple, StippleCounter, StippleStep, TColor32
TBitmap32.HandleChanged

procedure TBitmap32.HandleChanged; virtual; // protected
**Description**

This method is called every time the device handle of the bitmap is changed. It calls the [OnHandleChanged](#) event.
See Also

Handle, Naming Conventions, OnHandleChanged
TBitmap32.HorzLine

```plaintext
procedure HorzLine(X1, Y, X2: Integer; Value: TColor32);
procedure HorzLineS(X1, Y, X2: Integer; Value: TColor32);
procedure HorzLineT(X1, Y, X2: Integer; Value: TColor32);
procedure HorzLineTS(X1, Y, X2: Integer; Value: TColor32);
procedure HorzLineTSP(X1, Y, X2: Integer);
```
**Description**

Draws a horizontal line from \((X_1, Y)\) to \((X_2, Y)\). The last point is always included.

These functions works faster compared to Line (I think **HorzLine** 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 \(X_1\) value should be less than or equal to \(X_2\).

**HorzLineTSP** uses a stipple pattern to vary the color along the line.
See Also

Line, Line Patterns, Naming Conventions, TColor32, VertLine
procedure Line(X1, Y1, X2, Y2: Integer; Value: TColor32; L: Boolean = False);

procedure LineS(X1, Y1, X2, Y2: Integer; Value: TColor32; L: Boolean = False);

procedure LineT(X1, Y1, X2, Y2: Integer; Value: TColor32; L: Boolean = False);

procedure LineTS(X1, Y1, X2, Y2: Integer; Value: TColor32; L: Boolean = False);

procedure LineA(X1, Y1, X2, Y2: Integer; Value: TColor32; L: Boolean = False);

procedure LineAS(X1, Y1, X2, Y2: Integer; Value: TColor32; L: Boolean = False);

procedure LineX(X1, Y1, X2, Y2: TFixed; Value: TColor32; L: Boolean = False);

procedure LineXS(X1, Y1, X2, Y2: TFixed; Value: TColor32; L: Boolean = False);

procedure LineF(X1, Y1, X2, Y2: Single; Value: TColor32; L: Boolean = False);

procedure LineFS(X1, Y1, X2, Y2: Single; Value: TColor32; L: Boolean = False);

procedure LineXP(X1, Y1, X2, Y2: Single; L: Boolean = False);

procedure LineXSP(X1, Y1, X2, Y2: Single; L: Boolean = False);

procedure LineFP(X1, Y1, X2, Y2: Single; L: Boolean = False);

procedure LineFSP(X1, Y1, X2, Y2: Single; L: Boolean = False);
Description
Draws a line from \((X_1,Y_1)\) to \((X_2,Y_2)\). Methods with 'S' postfix perform necessary pixel-perfect clipping to a bitmap boundary or clipping rectangle.

**LineA** and **LineAS** use modified versions of Bresenham’s algorithm (also known as Wu’s antialiasing), modified to support line transparency and pixel shape/gamma correction.

**LineX/F/XS/FS** functions use my own algorithm for antialiasing. Line ends have fixed- or floating-point coordinates. These methods work approximately 2–2.5 times slower than **LineA** and **LineAS**.

The last optional parameter \((L)\) determines if the last point \((X_2,Y_2)\) 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, Line methods don’t render the last point.

**LineXP/FP/XSP/FXP** draw antialiased lines with support for color patterns.
See Also

HorzLine, LineTo, MoveTo, Naming Conventions, TColor32, VertLine
**TBitmap32.LineTo**

| procedure LineToS(X, Y: Integer); |
| procedure LineToTS(X, Y: Integer); |
| procedure LineToAS(X, Y: Integer); |
| procedure LineToXS(X, Y: TFixed); |
| procedure LineToFS(X, Y: Single); |
| procedure LineToXSP(X, Y: TFixed); |
| procedure LineToFSP(X, Y: Single); |
**Description**

*LineTo* methods are similar to *TCanvas.LineTo*. The line is drawn from the current raster position, to the position specified by *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' and 'XS' versions use and update independent fixed-point raster position.

*LineToXSP* and *LineToFSP* do not use *PenColor*, instead, they draw a line using current stipple pattern.

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

Line, Line Patterns, MoveTo, Naming Conventions, PenColor, TFixed
**TBitmap32.LoadFromFile**

```delphi
procedure LoadFromFile(const FileName: string);
```
Description

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

ImgView and Layers Example, LoadFromStream, SaveToFile
TBitmap32.LoadFromResourceID

procedure LoadFromResourceID(Instance: THandle; ResID: Integer);
**Description**

Loads an image from a resource. Instance is the handle of the module that contains the resource. ResID is the resource ID for the bitmap.
See Also

LoadFromFile, LoadFromResourceName, LoadFromStream
procedure LoadFromResourceName(Instance: THandle; const ResName: string);
Description
Loads an image from a resource. Instance is the handle of the module that contains the resource. ResName is the name of the resource to load.
See Also

LoadFromFile, LoadFromResourceID, LoadFromStream
TBitmap32.LoadFromStream

procedure LoadFromStream(Stream: TStream);
Description
Loads an image from a stream. This method uses a temporal \textit{TPicture} object to load data and will succeed with any format supported by \textit{TPicture}. 
See Also

LoadFromFile, SaveToStream

Copyright ©2000-2007 Alex Denisov and Contributors - Graphics32 v1.8.3 - Build on 4-March-2007
TBitmap32.MoveTo

<table>
<thead>
<tr>
<th>procedure</th>
<th>MoveTo(X, Y: Integer);</th>
</tr>
</thead>
<tbody>
<tr>
<td>procedure</td>
<td>MoveToX(X, Y: TFixed);</td>
</tr>
<tr>
<td>procedure</td>
<td>MoveToF(X, Y: Single);</td>
</tr>
</tbody>
</table>
Description
Shifts current raster to specified position, analogous to `MoveTo` method of standard `TCanvas`.

Each bitmap maintains separate raster positions for integer and fixed point coordinates. `MoveTo` affects the integer raster position, while `MoveToX` and `MoveToF` affect the fixed-point raster position.

Raster position is applicable only to `MoveTo`, `LineTo` methods and some polygon drawing routines, which use it internally, it is not changed neither by `Line` methods nor by any other method.
See Also

Line, LineTo, Naming Conventions, TFixed
TBitmap32.RaiseRectTS

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

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

FrameRect, Naming Conventions
TBitmap32.RenderText

```pascal
procedure RenderText( X, Y: Integer; const Text: String; AALevel: Integer; Color: TColor32);

procedure RenderTextW( X, Y: Integer; const Text: Widestring; AALevel: Integer; Color: TColor32);
```
**Description**

The **RenderText** or **RenderTextW** (the Unicode-Version) 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.

Note, that quality of the text is also influenced by the system font antialiasing.

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

*Please note that the CLX versions of the non-Unicode methods do also default to Widestring because that is the native QT string format.*
See Also

Font, RenderText Example, TColor32, TextExtent, TextOut
TBitmap32.ResetAlpha

procedure ResetAlpha; overload;
procedure ResetAlpha(const AlphaValue: Byte); overload;
**Description**

Resets the alpha channel of the entire bitmap either to $FF$ or the value defined by **AlphaValue**.
TBitmap32.ResetClipRect

procedure ResetClipRect;
Description

Resets the clipping rect to the current dimensions of the bitmap.
procedure Roll(Dx, Dy: Integer; FillBack: Boolean; FillColor: TColor32);
Description
Use Roll to scroll the whole bitmap Dx pixels right and Dy pixels down. Dx and Dy may be negative.
The content of the new area may optionally be cleared with FillColor, if FillBack is true.
See Also

TColor32

Copyright ©2000-2007 Alex Denisov and Contributors - Graphics32 v1.8.3 - Build on 4-March-2007
## TBitmap32.Rotate

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Description</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rotate90</td>
<td>Rotate 90 degrees</td>
<td>Dst: TBitmap32</td>
</tr>
<tr>
<td>Rotate180</td>
<td>Rotate 180 degrees</td>
<td>Dst: TBitmap32</td>
</tr>
<tr>
<td>Rotate270</td>
<td>Rotate 270 degrees</td>
<td>Dst: TBitmap32</td>
</tr>
</tbody>
</table>
Description
A set of optimized routines for rotating the image 90, 180, and 270 degrees. Angles correspond to the clockwise rotation.
See Also

TBitmap32
TBitmap32.SaveToFile

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

Writes a bitmap image to disk. The image is saved as 32-bit BMP file.
If you need another pixel format or storage format, create intermediate
TBitmap object or other TGraphic descendant, assign it from TBitmap32 and
save it to a file.

See source of TBitmap.SaveToFile for more details.
See Also

LoadFromFile, SaveToStream
TBitmap32.SaveToStream

```
procedure SaveToStream(Stream: TStream);
```
Description
Stores a bitmap image to a stream. The data in the stream is stored in a form compatible with *TBitmap*, where each pixel allocates 32 bits.
[See also]: LoadFromStream, SaveToFile
See Also

LoadFromStream, SaveToFile
procedure SetPixelT(X, Y: Integer; Value: TColor32); overload;
procedure SetPixelT(var Ptr: PColor32; Value: TColor32); overload;
procedure SetPixelTS(X, Y: Integer; Value: TColor32);
procedure SetPixelX(X, Y: TFixed; Value: TColor32); deprecated;
procedure SetPixelXS(X, Y: TFixed; Value: TColor32); deprecated;
procedure SetPixelF(X, Y: Single; Value: TColor32); deprecated;
procedure SetPixelFS(X, Y: Single; Value: TColor32); deprecated;
**Description**

`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 with each call to `SetPixelT`, for example:

```pascal
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 verification. If pixel coordinates lie outside the bitmap area, `SetPixelTS` does nothing.

`SetPixelX/F/XS/FS` methods are deprecated. Please use the `PixelX/F/XS/FS` property instead.
See Also

Naming Conventions, Pixel, TColor32, TFixed
TBitmap32.SetSize

**function** SetSize(NewWidth, NewHeight: Integer): Boolean; 
**override**;

**procedure** SetSize(Source: TPersistent); // implemented in **TCustomMap**
**Description**

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 source dimensions.

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

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

After the **SetSize** call the image the bitmap should be completely redrawn.
See Also

Handle, TCustomMap.Height, TCustomMap, TCustomMap.Width
TBitmap32.SetStipple

procedure SetStipple(NewStipple: TArrayOfColor32); overload;
procedure SetStipple(NewStipple: array of TColor32);
  overload;
Description
Sets a new stipple pattern.
See Also

GetStippleColor, Line Patterns, TArrayOfColor32, TColor32
TBitmap32.TextExtent

function TextExtent(const Text: String): TSize;
function TextExtentW(const Text: Widestring): TSize;
Description
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.

Please note that the CLX versions of the non-Unicode methods do also default to Widestring because that is the native QT string format.
See Also
Font, RenderText, TextHeight, TextOut, TextWidth, TSize
### TBitmap32.TextHeight

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Signature</th>
</tr>
</thead>
<tbody>
<tr>
<td>TextHeight</td>
<td>Returns the height of the text</td>
<td><code>function TextHeight(const Text: String): Integer;</code></td>
</tr>
<tr>
<td>TextHeightW</td>
<td>Returns the height of the text, considering wide characters</td>
<td><code>function TextHeightW(const Text: Widestring): Integer;</code></td>
</tr>
</tbody>
</table>
Description

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

Please note that the CLX versions of the non-Unicode methods do also default to Widestring because that is the native QT string format.
See Also
Font, RenderText, TextExtent, TextOut, TextWidth
### TBitmap32.TextEdit

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Description</th>
<th>Overload</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>procedure TextOut(X, Y: Integer; const Text: String); overload;</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>procedure TextOutW(X, Y: Integer; const Text: Widestring); overload;</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>procedure TextOut(X, Y: Integer; const ClipRect: TRect; const Text: String); overload;</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>procedure TextOutW(X, Y: Integer; const ClipRect: TRect; const Text: Widestring); overload;</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>procedure TextOut(DstRect: TRect; const Flags: Cardinal; const Text: String); overload;</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>procedure TextOutW(DstRect: TRect; const Flags: Cardinal; const Text: Widestring); overload;</code></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Description

Use **TextOut** or **TextOutW** (the Unicode version) to write a string onto the bitmap. The string will be written using current **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 version provides the most flexible text formatting. See description of **DrawText** function in 'Win32 Developer Reference' help file for information on **Flags** and their function.

**Special Note for CLX users:**

Since CLX is based on the QT library rather than the native Win API the flags used in the last two functions are somewhat different from the Win32 flags. You can get more information on the QT specific flags [here](#) and [here](#).

However, to make it easier for you, we have provided some basic mappings that will work in QT just as their Win API counterparts:

| DT_LEFT, DT_RIGHT, DT_TOP, DT_BOTTOM, DT_CENTER,  
| DT_VCENTER, DT_EXPANDtabs, DT_NOCLIP,  
| DT_WORDBREAK, DT_SINGLELINE |

However, the following flags are missing because there is no equivalent value in QT at this time:

| DT_CALCRECT, DT_EDITCONTOL, DT_END_ELLIPSIS and  
| DT_PATH_ELLIPSIS, DT_EXTERNALLEADING,  
| DT_MODIFYSTRING, DT_NOPREFIX, DT_RTLREADING,  
| DT_TABSTOP |

*Please note that the CLX versions of the non-Unicode methods do also default to Widestring because that is the native QT string format.*
See Also

Font, RenderText, TextExtent, TextHeight, TextWidth, TRect
### TBitmap32.TextWidth

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>function TextWidth(const Text: String): Integer;</code></td>
<td></td>
</tr>
<tr>
<td><code>function TextWidthW(const Text: Widestring): Integer;</code></td>
<td></td>
</tr>
</tbody>
</table>
**Description**

Returns the width, in pixels, of a string rendered in the current [Font](#).

*Please note that the CLX versions of the non-Unicode methods do also default to Widestring because that is the native QT string format.*
See Also

Font, RenderText, TextExtent, TextHeight, TextOut
TBitmap32.UpdateFont

procedure UpdateFont;
**Description**

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, TBitmap32
TBitmap32.VertLine

**procedure** VertLine(X, Y1, Y2: Integer; Value: TColor32);

**procedure** VertLineS(X, Y1, Y2: Integer; Value: TColor32);

**procedure** VertLineT(X, Y1, Y2: Integer; Value: TColor32);

**procedure** VertLineTS(X, Y1, Y2: Integer; Value: TColor32);

**procedure** VertLineTSP(X, Y1, Y2: Integer);
**Description**

Draws a vertical line from \((X, Y_1)\) to \((X, Y_2)\). The last point is included. These functions work faster compared to Line. In versions with 'S' postfix necessary clipping to a bitmap coordinate range is provided. In versions without 'S', the \(Y_2\) value should be greater or equal to \(Y_1\).

**VertLineTSP** supports line patterns.
See Also

HorzLine, Line, Line Patterns, Naming Conventions, TColor32
TBitmap32.OnAreaChanged

property OnAreaChanged: TAreaChangedEvent;
**Description**

This event is triggered when a drawing operation changes an area on the bitmap. If the bitmap is in *measuring mode*, the actual drawing operation is omitted, but this event is still triggered.
See Also

MeasuringMode, TAreaChangedEvent
property OnHandleChanged: TNotifyEvent;
Description
This event is called when the `HDC/QPainterH` of property `Handle` changed. Use this event to customize handling of such events.
See Also

Handle
TBitmap32.OnPixelCombine

property OnPixelCombine: TPixelCombineEvent;

type TPixelCombineEvent = procedure(F: TColor32; var B: TColor32; M: TColor32) of object;
**Description**
This event is called when the bitmap is drawn in dmCustom draw mode. Use this event to customize handling of colors. Note, however, that this event is called for every pixel, so keeping the event handler small and fast is in your own interest.
See Also

**DrawMode**, **PixelCombine Example**, **TColor32**, **TPixelCombineEvent**

Copyright ©2000-2007 Alex Denisov and Contributors - Graphics32 v1.8.3 - Build on 4-March-2007
TCustomMap.Height

property Height: Integer;
**Description**

Specifies the height of the contained data array.

Use [SetSize](#) method to change both width and height simultaneously.
See Also

`SetSize`, `Width`
TCustomMap.Width

property Width: Integer;
**Description**
Defines the width of the contained data array.
Writing into the **Width** property will resize the data array.
See Also

Height, SetSize

Copyright ©2000-2007 Alex Denisov and Contributors - Graphics32 v1.8.3 - Build on 4-March-2007
TCustomMap.Delete

procedure Delete; virtual;
Description

By default, **Delete** calls **SetSize**\((0, 0)\). In descendants this is accompanied with deletion of stored data and freeing of the occupied memory.
See Also

SetSize

Copyright ©2000-2007 Alex Denisov and Contributors - Graphics32 v1.8.3 - Build on 4-March-2007
TCustomMap.Empty

function Empty: Boolean; virtual;
Description

Returns *True* if the data set is empty. Normally, it means that either its *Width* or *Height* is 0.
procedure Resized; virtual;

Calls the `OnResize` event.

Descendants of `TCustomMap` call `Resized` internally after changing the size of the contained data array.
See Also

OnResize, TCustomMap
TCustomMap.SetSize

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

Simultaneously changes both **Width** and **Height** of contained data.

The SetSizeFrom method 'knows' how to get the size from the following objects or their descendants: **TCustomMap**, **TGraphic**, **TControl** and **nil**. When other parameter is specified, **TCustomMap** will generate an exception.

The SetSize method is overridden in descendants (**TBitmap32**, **TByteMap**) so that bitmap bits are reallocated and byte map data array is resized.
See Also

Height, TBitmap32, TByteMap, TCustomMap, Width
TCustomMap.OnResize

property OnResize: TNotifyEvent;
Description

OnResize is called each time dimensions of the associated data array change. This event is normally preceeded with the OnChange call.

Similar to OnChange, generation of OnResize may be temporarily suppressed with BeginUpdate/EndUpdate.
See Also

BeginUpdate, EndUpdate, OnChange
property Width: Single;
Description
Returns the width of the associated kernel.
TCustomResampler.Resample

procedure Resample(Dst: TBitmap32; DstRect: TRect; DstClip: TRect; Src: TBitmap32; SrcRect: TRect; CombineOp: TDrawMode; CombineCallBack: TPixelCombineEvent); virtual; abstract;
Description
Descendants must override this method in order to perform resampling of bitmaps.
See Also

Rectangle Types, TBitmap32, TBitmap32Resampler, TDrawMode, TPixelCombineEvent
TCustomSampler.FinalizeSampling

procedure FinalizeSampling; virtual;
Description
Descendants should override this method in order to perform special processing when sampling ends.
Note: **FinalizeSampling** is automatically called by TRasterizer when rasterization ends.
See Also

PrepareSampling, TRasterizer
TCustomSampler.GetSample

function GetSampleInt(X, Y: Integer): TColor32;
function GetSampleFixed(X, Y: TFixed): TColor32;
function GetSampleFloat(X, Y: Single): TColor32;
**Description**

*GetSample* provides a method for acquiring a color sample from the coordinate \((X, Y)\). The sample may be generated by the sampler itself, or it may be acquired from a *nested sampler*. Descendants of *TCustomSampler* must always override at least one of *GetSampleFixed* or *GetSampleFloat*. The different postfixes determines the precision of the input parameters (integer, fixed-point or floating-point).

Prior to calling any GetSample method, *PrepareSampling* should be called, and finally *FinalizeSampling* should be called. Note that any number of calls to the GetSample methods can be made inside this Prepare/FinalizeSampling block.
See Also

Color Types, FinalizeSampling, PrepareSampling, TCustomSampler, TFixed
procedure GetSampleBounds: TRect; virtual; abstract;
Description
Descendants must override this method to define the area changed by the sampler, which is used for clipping purposes. If the sampler do not support returning a bounds rectangle, HasBounds should be set to false.
See Also

HasBounds, Rectangle Types
TCustomSampler.HasBounds

function HasBounds: Boolean; virtual;
Description
Descendants must set this to \textit{true}, if sample bounds will be returned by \texttt{GetSampleBounds}. 
See Also

GetSampleBounds
TCustomSampler.PrepareSampling

procedure PrepareSampling; virtual;
**Description**

Descendants should override this method in order to perform special preparations before the `GetSample` method is called.

Note: **PrepareSampling** is automatically called by `TRasterizer` before rasterization begins.
See Also

FinalizeSampling, GetSample, TRasterizer
TNotifiablePersistent.UpdateCount

property UpdateCount: Integer; // Read-only; protected;
**Description**

The current nesting level of the update lock.

**UpdateCount** is increased each time you call the [BeginUpdate](#) method and is decreased with [EndUpdate](#) calls. The object does not generate [OnChange](#) as long as its **UpdateCount** is greater than 0.
See Also

BeginUpdate, EndUpdate, OnChange
TNotifyablePersistent.BeginUpdate

procedure BeginUpdate;
**Description**

Call this method in order to temporarily disable the update notification. **BeginUpdate** increases the **UpdateCount** property, which disables generation of **OnChange** events. Calls to **BeginUpdate** must be paired with **EndUpdate** and they may be safely nested.
See Also
EndUpdate, OnChange, UpdateCount
TNotifiablePersistent.Changed

procedure Changed;
Description

Calls the OnChange event, provided the change notification is not masked with BeginUpdate.

Descendants of TNotifiablePersistent call Changed after making changes to their data or properties.

You will need to call Changed explicitly in your application after leaving the BeginUpdate... EndUpdate block since EndUpdate itself does not internally call Changed.
See Also

BeginUpdate, Copy of TThreadPersistent, EndUpdate, OnChange
TNotifiablePersistent.Create

constructor Create; virtual;
Description
Creates and initializes an instance of TNotifiablePersistent.
See Also

TNotifiablePersistent
TNotifiablePersistent.Destroy

```plaintext
destructor Destroy; override;
```
Description

Destroys the **TNotifyablePersistent** object.

Do not call **Destroy** directly, use **Free** method instead.
See Also
TNotifiablePersistent
TNotificationPersistent.EndUpdate

procedure EndUpdate;
Description

Re-enables change notification previously disabled with BeginUpdate.

EndUpdate decreases the UpdateCount property and re-enables generation of OnChange events.

EndUpdate itself does not generate the OnChange event. You will need to call Changed explicitly, after calling EndUpdate in case the change notification is required.
See Also

BeginUpdate, Changed, OnChange, UpdateCount
TNotifiablePersistent.OnChange

property OnChange: TNotifyEvent;
**Description**

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

Generation of *OnChange* events can be prevented by calling *BeginUpdate*. 
See Also

BeginUpdate, Changed, TBitmap32, TNotifiablePersistent
TThreadPersistent.LockCount

**property** LockCount: Integer; // *Read-only; protected;*
Description

**LockCount** represents the current nesting level of the threading lock. This is a counter shared by all running threads in the current process.

The object is unlocked only when **LockCount** is 0 and only one thread can lock the object at the time. Call **Lock** to increase **LockCount** value and **Unlock** to decrease it.
See Also

Lock, Unlock
TThreadPersistent.Lock

procedure Lock;
**Description**

Blocks other execution threads from locking the object until the **Unlock** method is called. If another thread tries to call the **Lock** method of an object which is already locked, its execution will be 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. **LockCount** is increased each time **Lock** is called.
See Also

LockCount, Unlock
TThreadPersistent.Unlock

procedure Unlock;
**Description**

Removes the multithreading lock from the object.

Each call to **Unlock** decreases the **LockCount** property, allowing other threads to access the object when **LockCount** reaches 0. Each thread must call **Unlock** once for each time that it locked the object.
See Also
Lock, LockCount

Copyright ©2000-2007 Alex Denisov and Contributors - Graphics32 v1.8.3 - Build on 4-March-2007
AlphaComponent

function AlphaComponent(Color32: TColor32): Integer;
Description
Extracts the alpha component from the parameter.
This function is provided for convenience and backward compatibility only. In performance-critical parts of your code use \texttt{TC\textcolor{red}{olor32}Entry} typecasting or direct conversion:

\begin{verbatim}
Alpha := Color32 shr 24;
\end{verbatim}
See Also

BlueComponent, GreenComponent, RedComponent, TColor32, TColor32Entry

Copyright ©2000-2007 Alex Denisov and Contributors - Graphics32 v1.8.3 - Build on 4-March-2007
BlueComponent

function BlueComponent(Color32: TColor32): Integer;
Description
Extracts the blue component from the parameter.
This function is provided for convenience and backward compatibility only. In performance-critical parts of your code use TColor32Entry typecasting or direct conversion:

\[
\text{Blue := Color32 and } \$0000FF;
\]
See Also

AlphaComponent, GreenComponent, RedComponent, TColor32, TColor32Entry
Color32

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

This function combines its parameters into a 4-byte TColor32.

```pascal
function Color32(WinColor: TColor): TColor32; overload;
```

This function provides conversion of TColor into TColor32.

The pixel format of 32-bit DIBs (ARGB) is different from that used in the standard TColor type (ABGR). Some standard windows colors are coded using special constants which are encoded into RGB form.

The alpha of the resulting color is set to $FF;

```pascal
function Color32(Index: Byte; Palette: PPalette32): TColor32; overload;
```

This function simply picks the color value from the palette, same as

Result := Palette[Index];
See Also

Gray32, TColor32, TPalette32, WinColor

Copyright ©2000-2007 Alex Denisov and Contributors - Graphics32 v1.8.3 - Build on 4-March-2007
Color32Components

function Color32Components(R, G, B, A: Boolean):
TColor32Components;

Returns a set (TColor32Components) of TColor32Component from the provided parameters.
If all parameters are false, an empty set will be returned.
See Also

TColor32Component, TColor32Components
**Color32ToRGB**

```plaintext
procedure Color32ToRGB(Color32: TColor32; var R, G, B: Byte);
procedure Color32ToRGBA(Color32: TColor32; var R, G, B, A: Byte);
```

Both procedures split the `Color32` parameter up into their respective components.

These procedures are provided for convenience only. Use direct conversion in performance-critical parts of your code:

```plaintext
Alpha := Color32 shr 24;
Red := (Color32 and $00FF0000) shr 16;
Green := (Color32 and $0000FF00) shr 8;
Blue := Color32 and $000000FF;
```
Creating Points

Point

function Point(X, Y: Integer): TPoint; overload;
function Point(const FP: TFloatPoint): TPoint; overload;
function Point(const FXP: TFixedPoint): TPoint; overload;

Constructs a point with integer coordinates, or converts it from the one with fixed/floating point coordinates by rounding them to nearest integers.
Provided construction and conversion of points with floating-point coordinates.
Function FixedPoint(X, Y: Integer): TFixedPoint; overload;
function FixedPoint(X, Y: Single): TFixedPoint; overload;
function FixedPoint(const P: TPoint): TFixedPoint; overload;
function FixedPoint(const FP: TFloatPoint): TFixedPoint; overload;

Provides construction and conversion of points with fixed-point coordinates
See Also

TFixedPoint, TFloatPoint, TPoint
Creating Rectangles

Rect

```haskell
function MakeRect(L, T, R, B: Integer): TRect; overload;
function MakeRect(const FR: TFloatRect; Rounding: TRectRounding = rrClosest): TRect; overload;
function MakeRect(const FXR: TFixedRect; Rounding: TRectRounding = rrClosest): TRect; overload;
```

Provides construction and conversion of rectangles with integer coordinates.
**FloatRect**

```pascal
function FloatRect(L, T, R, B: Single): TFloatRect; overload;
function FloatRect(const Rect: TRect): TFloatRect; overload;
function FloatRect(const FXR: TFixedRect): TFloatRect; overload;
```

Provides construction and conversion of rectangles with floating-point coordinates.
FixedRect

function FixedRect(L, T, R, B: TFixed): TFixedRect; overload;
function FixedRect(const Rect: TRect): TFixedRect; overload;
function FixedRect(const FR: TFloatRect): TFixedRect; overload;

Provides construction and conversion of rectangles with fixed-point coordinates.
See Also

TFixedRect, TFloatRect, TRect, TRectRounding
EqualRect

function EqualRect(const R1, R2: TRect): Boolean;
Description
Returns True if the rectangles are identical.
See Also

TRect
Fixed Point Math

Fixed

function Fixed(S: Single): TFixed; overload;
function Fixed(I: Integer): TFixed; overload;

A couple of TFixed constructors.
In case you don't want your program to loose time on extra function call, you can use another method of converting Integer to TFixed:

var
  FX: TFixed;
begin
  FX := IntVal shl 16;
  ...
end;
See Also
GR32_Math, TFixed
Gray32

```cpp
function Gray32(Intensity: Byte; Alpha: Byte): TColor32;
```
**Description**

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

AlphaComponent, BlueComponent, Color32, GreenComponent, RedComponent, TColor32
GreenComponent

function GreenComponent(Color32: TColor32): Integer;
**Description**

Extracts the green component from the parameter.

This function is provided for convenience and backward compatibility only. In performance-critical parts of your code use `TColor32Entry` typecasting or direct conversion:

```
Green := (Color32 and $0000FF00) shr 8;
```
See Also

AlphaComponent, BlueComponent, RedComponent, TColor32, TColor32Entry
HSLtoRGB

function HSLtoRGB(H, S, L: Single): TColor32; overload;
function HSLtoRGB(H, S, L: Integer): TColor32; overload;
**Description**

Conversion from HSL to RGB color space. Each argument should normally be in [0…1] range for the float version and in the [0…255] range for the integer version, although the H value is automatically wrapped.
See Also
Gray32, RGBtoHSL, TColor32
InflateRect

procedure InflateRect(var R: TRect; Dx, Dy: Integer); overload;
procedure InflateRect(var FR: TFloatRect; Dx, Dy: Single); overload;
**Description**  
Increases (or decreases if \textbf{Dx, Dy} negative) the width and height of the specified rectangle:

- \( \text{R.Left} := \text{R.Left} - \text{Dx}; \)
- \( \text{R.Right} := \text{R.Right} + \text{Dx}; \)
- \( \text{R.Top} := \text{R.Top} - \text{Dy}; \)
- \( \text{R.Bottom} := \text{R.Bottom} + \text{Dy}; \)
See Also

OffsetRect, TFloatRect, TRect
Intensity

function Intensity(Color32: TColor32): Integer;
**Description**

Returns weighted intensity of the color, which is calculated as

\[ I = R \times 0.21 + G \times 0.71 + B \times 0.08; \]
See Also

Color32, Gray32, TColor32
### IntersectRect

**Function** `IntersectRect(out Dst: TRect; const R1, R2: TRect): Boolean; overload;`

**Function** `IntersectRect(out Dst: TFloatRect; const FR1, FR2: TFloatRect): Boolean; overload;`
Description
Calculates intersection of the rectangles and writes result into Dst. Functions return True if rectangles intersect, otherwise it returns False and writes an empty rectangle into Dst, that is rectangle with all coordinates equal to 0.
See Also

Creating Rectangles, TFloatRect, TRect
IsRectEmpty

function IsRectEmpty(const R: TRect): Boolean; overload;
function IsRectEmpty(const FR: TRect): Boolean; overload;
Description
Returns True if the rectangle is empty, that is its width or height is not positive.
See Also

TRect
OffsetRect

procedure OffsetRect(var R: TRect; Dx, Dy: Integer); overload;
procedure OffsetRec(var FR: TFloatRect; Dx, Dy: Single); overload;
**Description**

Use these functions to shift rectangles. Dx and Dy parameters specify the distance.
See Also

InflateRect, TFloatRect, TRect
**PtInRect**

```delphi
function PtInRect(const R: TRect; const P: TPoint): Boolean;
```
Description

Returns True if the point lies inside the specified rectangle.
See Also

TPoint, TRect
RedComponent

function RedComponent(Color32: TColor32): Integer;
**Description**

Extracts the red component from the parameter. This function is provided for convenience and backward compatibility only. In performance-critical parts of your code use `TColor32Entry` typecasting or direct conversion:

```
Red := (Color32 and $00FF0000) shr 16;
```
See Also

AlphaComponent, BlueComponent, GreenComponent, TColor32, TColor32Entry
RGBtoHSL

```pascal
procedure RGBtoHSL(RGB: TColor32; var H, S, L : Single);
overload;

procedure RGBtoHSL(RGB: TColor32; var H, S, L : Byte);
overload;
```
Description
Conversion from RGB to HSL color space. The H, S and L components are returned in corresponding var parameters ranging from 0 to 1 for the float version and 0 to 255 for the byte version.
See Also

Gray32, HSLtoRGB, TColor32
SetAlpha

function SetAlpha(Color32: TColor32; NewAlpha: Integer): TColor32;
**Description**
Returns a color with the new alpha channel. This function automatically performs clipping of the *NewAlpha* parameter to [0…255] range.
See Also

TColor32
SetGamma

\textbf{procedure} SetGamma(Gamma: Single = 0.7);
Description

Updates gamma/shape correction table.

Pixel and line antialiasing produces much better results with correction of opacities for partially covered pixels. This accounts both for monitor gamma and for pixel shape correction.

Use this function to update the internal gamma-correction table which is used in pixel and line rasterization.

The default value of 0.7 works fine in most cases.
WinColor

function WinColor(Color32: TColor32): TColor;
Description
Provides conversion of the TColor32 value back into TColor. The highest-order byte (Alpha channel) of resulting color is assigned the $FF$ value.
See Also

Color32, TColor32
WinPalette

function WinPalette(P: TPalette32): HPALETTE;
**Description**

The provided TPalette32 fixed-size buffer of color values is converted into the Win32 API equivalent HPALETTE handle definition of a palette with 256 32-bit color entries.
See Also

Color32, TPalette32

Copyright ©2000-2007 Alex Denisov and Contributors - Graphics32 v1.8.3 - Build on 4-March-2007
Arrays

Dynamic Array Types

TArrayOfByte

\[
\begin{align*}
\textbf{type} & \quad \text{TArrayOfByte} = \text{array of} \; \text{Byte}; \\
\textbf{type} & \quad \text{PArrayOfByte} = ^\text{TArrayOfByte};
\end{align*}
\]

A dynamic array of bytes.
TArrayOfWord

```pascal
type TArrayOfInteger = array of Word;
```

A dynamic array of words.
TArrayOfInteger

\[
\text{type} \ T\text{ArrayOfInteger} = \text{array} \ of \ \text{Integer};
\]

A dynamic array of integers.
TArrayOfSingle
type TArrayOfSingle = array of Single;
A dynamic array of single-precision floating-point values.
TArrayOfArrayOfInteger

type TArrayOfArrayOfInteger = array of TArrayOfInteger;

A dynamic array of dynamic arrays of integers.
Array Reference Types

Most likely you will never use the reference types directly, however the pointer is the one of some importance. It provides indexed access to values stored in memory, and is used as the result type of the ordinal maps bits properties.
TByteArray

\[
\text{type } \text{TByteArray} = \text{array} [0..0] \text{ of Byte;}
\]
\[
\text{type } \text{PByteArray} = ^\text{TByteArray};
\]

A reference to array of byte-typed values.
TWordArray

| type | TWordArray = array [0..0] of Word; |
| type | PWordArray = ^TWordArray; |

A reference to array of word-typed values.
TIntegerArray

```plaintext
  type TIntegerArray = array [0..0] of Integer;
  type PIntegerArray = ^TIntegerArray;
```

A reference to array of integer-typed values.
See Also

TByteMap, TIntegerMap, TWordMap
**Color Types**

**TColor32**

```delphi
type TColor32 = type Longword;
type PColor32 = ^TColor32;
```

A simple ARGB color quad, compatible with 32-bit DIBs.

**TColor32** holds color information in ARGB format:

<table>
<thead>
<tr>
<th>Bits 32...24</th>
<th>Bits 23...16</th>
<th>Bits 15...8</th>
<th>Bits 7...0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alpha</td>
<td>Red</td>
<td>Green</td>
<td>Blue</td>
</tr>
</tbody>
</table>

This order is different from ABGR pixel format used by most Windows API functions and implemented in Delphi as **TColor** type. A couple of functions are provided to convert colors between different standards: **Color32**, **WinColor**.

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

Graphics32 specifies several pre-defined **Color Constants**, which are similar to standard ones except that they have properly filled alpha value and swapped red and blue components.
TColor32Entry

TColor32Entry = packed record
case Integer of
  0: (A: Byte;
      B: Byte;
      G: Byte;
      R: Byte);
  1: (ARGB: TColor32);
  2: (Planes: array [0..3] of Byte);
  3: (Components: array [TColor32Component] of Byte);
end;
type PColor32Entry = ^TColor32Entry;

A useful type for gaining fast direct read/write access to the components of TColor32 by typecasting. The following code segment shows how this typecasting can be used in different manners - we recommend using these approaches instead of auxiliary functions like RedComponent etc.: 

var
  AColor32: TColor32;
  I: Integer;
begin
  // Sets values to individual components
  TColor32Entry(AColor32).A := 255;
  TColor32Entry(AColor32).R := 127;
  TColor32Entry(AColor32).G := 63;
  TColor32Entry(AColor32).B := 31;
  // Sets all planes or components ("channels") to a different random value
  with TColor32Entry(AColor32) do
    for I := Low(Planes) to High(Planes) do
      Planes[I] := Random(255);
  // Write value of Alpha component to Red, Green and Blue components
  with TColor32Entry(AColor32) do
begin
  Components[ccRed] := Components[ccAlpha];
  Components[ccGreen] := Components[ccAlpha];
  Components[ccBlue] := Components[ccAlpha];
end;
end;
**TColor32Component**

```haskell
type TColor32Component = (ccBlue, ccGreen, ccRed, ccAlpha);
```

An enumerated type that specifies a given **TColor32** component or plane.
**TColor32Components**

```plaintext
type TColor32Components = set of TColor32Component;
```

A set of `TColor32Component` which can be used to specify zero or more components. The order of defined components has no importance. To construct a set of components from booleans indicating component presence, use function `Color32Components`. If an empty set is passed to a routine that uses this type, this will (and should) in most cases lead to no processing.
**TColor32Array**

```pascal
  type TColor32Array = array [0..0] of TColor32;
  type PColor32Array = ^TColor32Array;
```

A reference to array of **TColor32**-typed values.

Most likely you will never use the **TColor32Array** type itself, however **PColor32Array** is the one of some importance. It provides indexed access to color values stored in memory. For example, in **TBitmap32**, it is used to access pixel data.
**TArrayOfColor32**

```pascal
type TArrayOfColor32 = array of TColor32;
```

A dynamic array of `TColor32`-typed values.

You may use the standard *SetLength* function for array allocation and dynamic size changes.

Do not confuse `PColor32Array` and `TArrayOfColor32` types. While the first one holds the pointer to a memory location, the second one is a fully functional dynamic array.
TPalette32

```delphi
type TPalette32 = array [Byte] of TColor32;
type PPalette32 = ^TPalette32;
```

A fixed-size array of 256 TColor32-typed values.

TPalette32 type is mostly used to simulate palette-based operations.
See Also

Color Constants, Color32, Color32Components, RedComponent, TBitmap32, WinColor
Point Types

Graphics32 uses coordinate system with an origin at the top-left corner. Due to performance considerations, other coordinate systems are not supported and not planned in future versions.

Point coordinates are defined with following structures:
TPoint

type TPoint = Windows.TPoint;
type PPoint = ^TPoint;

A point with integer coordinates. **TPoint** type, is redeclared from Windows.pas unit. It is compatible with all Delphi/Windows API calls.
TFloatPoint

```pascal
    type TFloatPoint = record
        X, Y: Single;
    end;

    type PFFloatPoint = ^TFloatPoint;
```

Defines a point with single-precision floating-point coordinates.
**TFixedPoint**

```pascal
type TFixedPoint = record
  X, Y: TFixed;
end;

type PFixedPoint = ^TFixedPoint;
```

**TFixedPoint** is bitwise-compatible with *TPointFx* type from Windows.pas, which is used by some API calls. **TFixedPoint** is not assignment-compatible with *TPointFX*, however, appropriate data in memory may be safely referenced by both types and it is still possible to assign between these types using, for example, *Move* procedure from System.pas.
TArrayOfPoint

```pascal
type TArrayOfPoint = array of TPoint;
```

A dynamic array of TPoint.
**TArrayOfFloatPoint**

```pascal
type TArrayOfFloatPoint = array of TFloatPoint;
```

A dynamic array of **TFloatPoint**.
**TArrayOfFixedPoint**

```plaintext
type TArrayOfFixedPoint = array of TFixedPoint;
```

A dynamic array of **TFixedPoint**.
TArrayOfArrayOfPoint

```delphi
type TArrayOfArrayOfPoint = array of TArrayOfPoint;
```
TArrayOfArrayOfFloatPoint

\[
\text{type TArrayOfArrayOfFloatPoint = array of TArrayOfFloatPoint;}
\]
TArrayOfArrayOfFixedPoint

define

type TArrayOfArrayOfFixedPoint = array of TArrayOfFixedPoint;

Maybe last three types look a little cumbersome, but they are what it says: dynamic arrays of dynamic arrays of points. These are the types used in PolyPolyline and PolyPolygon functions to define shapes constructed of several outlines.
See Also

Creating Points, PolyPolygon, PolyPolyline, TFixed
Rectangle Types

Several data types are designated to handle rectangles:
**TRect**

```delphi
type TRect = Windows.TRect;
```

*TRect* is redeclared from *Windows.h* and is compatible with Delphi/Windows API calls.
**TFloatRect**

```plaintext
type TFloatRect = record
  case Integer of
    0: (Left, Top, Right, Bottom: Single);
    1: (TopLeft, BottomRight: TFloatPoint);
  end;
```

A rectangle with floating-point coordinates.
TFixedRect

type TFixedRect = record
  case Integer of
    0: (Left, Top, Right, Bottom: TFixed);
    1: (TopLeft, BottomRight: TFixedPoint);
  end;

A rectangle with fixed-point coordinates.
See Also

Creating Rectangles, TFixed, TFixedPoint, TFloatPoint
TAreaChangedEvent

```delphi
type TAreaChangedEvent = procedure(Sender: TObject; const Area: TRect; const Info: Cardinal) of object;
```
**Description**

TAreaChangedEvent is the type of the OnAreaChanged event in TBitmap32. It indicates which area has changed (Area). A simple information (Info) about the area can be set.

**Info** is normally AREAINFO_RECT, but other flags can be set to indicate what shape the area actually has. This is especially useful for the repaint optimizer. Lines for example are specially treated with the AREAINFO_LINE + LineWidth encoded into the Info parameter.
See Also

Area Information Flags, TBitmap32.DrawMode, TBitmap32.OnAreaChanged, TRect
TCombineMode

```haskell
type TCombineMode = (cmBlend, cmMerge);
```
**Description**

*TCombineMode* defines how *drawing operations* combine the foreground color with the background. It also determines how a bitmap is combined with the background when *DrawMode* is set to *dmBlend*.

- **cmBlend** - Fast blending of foreground color with the background color using the supplied alpha. This method is not suited for working and preserving alpha-channels. Use this more if you want to blend directly to the display buffer.

- **cmMerge** - Uses a completely different formula that also merges the alpha-channels. This mode is slower than blending but is suited for working with alpha-channels (eg. when composing in temporary buffers).
See Also

TBitmap32.DrawMode
TDrawMode

type TDrawMode = (dmOpaque, dmBlend, dmCustom, dmTransparent);
**Description**

**TDrawMode** defines how the bitmap is combined with the background. This is the type for **DrawMode** property of **TBitmap32** class, and it is also passed as parameter in **BlockTransfer** and **StretchTransfer** functions.

- **dmOpaque** – a color information from the foreground image completely replaces the background data;
- **dmBlend** – foreground and background colors are blended together using per-pixel opacity of the foreground bitmap (stored in its alpha channel), multiplied by its **MasterAlpha**;
- **dmCustom** – colors are mixed using **OnPixelCombine** event, that is essentially a call-back function allowing for any custom blending routine.
- **dmTransparent** – color key transparency: any pixel that matches **TBitmap32 OUTERCOLOR** is treated as transparent.
See Also

**TFixed**

```pascal
  type TFixed = type Integer;
  type PFixed = ^TFixed;
```
**Description**

In order to accelerate some calculations, Graphics32 extensively uses fixed point arithmetics, which is mostly based on `TFixed` type:

`TFixed` is a 32-bit fixed point value in 16.16 format, that is 16 bits represent signed integer part and 16 bits represent unsigned fractional part. This format is bitwise-compatible with `TFixed` type from the Windows.pas unit, except it does not use a record to store the value, which allows for faster function parameter passing and faster value copying and assignment.

Conversion from floating to fixed point is very simple:

```pascal
Fixed := Round(Single * 65536);
```
See Also

Fixed Point Math
TPixelCombineEvent

\texttt{type} TPixelCombineEvent = \texttt{procedure}(F: \texttt{TColor32}; \texttt{var} B: \texttt{TColor32}; M: \texttt{TColor32}) \texttt{of object};
**Description**

TPixelCombineEvent is a type for OnPixelCombine callback function. It specifies a function which takes foreground color (F) and blends it with the background (B). This function may optionally use master alpha value (M). Moreover, in dmCustom mode master alpha value does not necessarily have to be in [0…255] range anymore, you are free to use all its 32-bits.
See Also

TBitmap32.DrawMode, TBitmap32.OnPixelCombine, TColor32

Copyright ©2000-2007 Alex Denisov and Contributors - Graphics32 v1.8.3 - Build on 4-March-2007
TRectRounding

    type TRectRounding = (rrClosest, rrOutside, rrInside);
Description

TRectRounding type is involved in conversion from TFloatRect/TFixedRect to TRect. It specifies how the coordinates are rounded:

- rrClosest – a simple `Round()` function is used for each coordinate;
- rrOutside – `Floor()` is used for Left and Top; `Ceil()` – for Right and Bottom;
- rrInside – `Ceil()` is used for Left and Top; `Floor()` – for Right and Bottom;
See Also

Creating Rectangles, TFixedRect, TFloatRect, TRect
TSize

define TSize as record
    Cx, Cy: Integer;
end;
Description
A generic type which holds dimensions of a rectangle, if not stated otherwise, dimensions are specified in pixels.
TStretchFilter

```cpp
type TStretchFilter = (sfNearest, sfDraft, sfLinear, sfCosine, sfSpline, sfLanczos, sfMitchell);
```
Description

When the bitmap is scaled, these filters control how the colors are interpolated:

- **sfNearest** is the fastest filter, although the quality of the stretched image is fair.
- **sfDraft** is a complex resampler, meaning that it uses several different methods. For downampling it uses row-column zoneshifting to preserve more details than sfNearest, however the quality is not as good as sfLinear. For upsampling it uses a linear interpolation method that may compromise details - this happens if both downampling and upsampling is needed. sfDraft is ideal for preview fields and thumbnails. Please note, that the algorithm is not pixel-perfect, that why using a custom cliprect might result in a so called jitter-effect.
- **sfLinear** is several times slower, but it produces more decent results in most cases.
- **sfCosine** is almost equally fast as sfLinear, but it uses cosine interpolation, which produces a sharper image.
- **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.
- **sfLanczos** generally produces the sharpest image (especially when downsampling), however sometimes can produce excessive "ringing" effect - this is the slowest filter.
- **sfMitchell** is often considered as one of the best filters for enlarging images; generally not as sharp as Lanczos.

In some functions filters above **sfLinear** may not be applicable (for example in Transform), in this case the filter value will be substituted with **sfLinear**.
See Also

TBitmap32.StretchFilter, StretchTransfer
TWrapMode

type TWrapMode = (wmClamp, wmRepeat, wmMirror);
Description

TWrapMode is the type for the WrapMode property in TBitmap32, which specifies how to wrapping pixel getters and setters should behave.

wmClamp will repeat the edge pixel if a pixel is read or written outside the bitmap's boundary.

wmRepeat will repeat the relative bitmap colors if it's outside of the bitmap's boundary (also referred to as 'seamless' and 'texture' mode).

wmMirror will mirror the pixel relative to the bitmap if it's outside the bitmap's boundary. This mode is useful for better perceptual results with spatial processing (like blurring).
See Also

TBitmap32, TBitmap32.WrapMode
Area Information Flags

The Area Information Flags are used in the TAreaChangedEvent to further describe the shape of the changed area. Currently Graphics32 defines these types:

```c
const
  // common cases
AREAINFO_RECT = $80000000;
AREAINFO_LINE = $40000000; // 24 bits for line width in pixels...
AREAINFO_ELLIPSE = $20000000;
AREAINFO_ABSOLUTE = $10000000;

AREAINFO_MASK = $FF000000;
```
See Also

TAreaChangedEvent

Copyright ©2000-2007 Alex Denisov and Contributors - Graphics32 v1.8.3 - Build on 4-March-2007
Color Constants

Color constants specify some predefined values of TColor32 type. Their names are similar to standard ones, they just have a '32' postfix:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>clBlack32</td>
<td>$FF000000</td>
</tr>
<tr>
<td>clDimGray32</td>
<td>$FF3F3F3F</td>
</tr>
<tr>
<td>clGray32</td>
<td>$FF7F7F7F</td>
</tr>
<tr>
<td>clLightGray32</td>
<td>$FFBFBFBF</td>
</tr>
<tr>
<td>clWhite32</td>
<td>$FFFFFFFF</td>
</tr>
<tr>
<td>clMaroon32</td>
<td>$FF7F0000</td>
</tr>
<tr>
<td>clGreen32</td>
<td>$FF007F00</td>
</tr>
<tr>
<td>clOlive32</td>
<td>$FF7F7F00</td>
</tr>
<tr>
<td>clNavy32</td>
<td>$FF00007F</td>
</tr>
<tr>
<td>clPurple32</td>
<td>$FF7F007F</td>
</tr>
<tr>
<td>clTeal32</td>
<td>$FF007F7F</td>
</tr>
<tr>
<td>clRed32</td>
<td>$FFFF0000</td>
</tr>
<tr>
<td>clLime32</td>
<td>$FF00FF00</td>
</tr>
<tr>
<td>clYellow32</td>
<td>$FFFFFF00</td>
</tr>
<tr>
<td>clBlue32</td>
<td>$FF0000FF</td>
</tr>
<tr>
<td>clFuchsia32</td>
<td>$FFFF00FF</td>
</tr>
<tr>
<td>clAqua32</td>
<td>$FF00FFFF</td>
</tr>
</tbody>
</table>

In addition, Graphics32 defines a few colors with 50% transparency:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>clTrWhite32</td>
<td>$7FFFFFFF</td>
</tr>
<tr>
<td>clTrBlack32</td>
<td>$7F000000</td>
</tr>
<tr>
<td>clTrRed32</td>
<td>$7FFF0000</td>
</tr>
<tr>
<td>clTrGreen32</td>
<td>$7F00FF00</td>
</tr>
<tr>
<td>clTrBlue32</td>
<td>$7F0000FF</td>
</tr>
</tbody>
</table>
See Also

TColor32
G32Version

The **Graphics32Version** constant is a string that holds version number of the library.

The version number is a simple string which may optionally include sub-version literal, e.g. '0.98a', '0.99', '1.0' etc.
**Blend**

*function* BlendReg(F, B: TColor32): TColor32;

*procedure* BlendMem(F: TColor32; var B: TColor32);
Description
Mixes a foreground (F) color with the background color (B) using the alpha component of the foreground color.

\[ S_{RGB} = F_A \times F_{RGB} + (1 - F_A) \times B_{RGB}; \]

\textbf{BlendReg} takes parameters and produces the result operating on CPU registers.

\textbf{BlendMem} operates with the background color referenced by a memory address.

Using \textbf{BlendMem} is more efficient when blending/combining data to a bitmap, since it excludes writing operation for transparent pixels and reading operation for opaque ones.

Note, that after using the \textbf{Blend} function, you have to call \texttt{EMMS}. Otherwise CPU will be unable to handle floating point instructions.
See Also

BlendEx, Combine, EMMS, Merge, MergeEx, TColor32
BlendEx

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

Mixes a foreground color with the background color using alpha of the foreground color scaled by the master alpha value \( M \).

\[
S_{RGB} = (M \cdot F_A) \cdot F_{RGB} + (1 - (M \cdot F_A)) \cdot B_{RGB};
\]

\( M \) is defined as \text{TC}	ext{olor}32 to avoid unnecessary type conversions, it must store only values in [0..255] range, the function does not perform range checking and the result in case \( M > 255 \) is not specified.

\textbf{BlendRegEx} takes parameters and produces the result operating on CPU registers.

\textbf{BlendMemEx} operates with the background color referenced by a memory address.

Using \textbf{BlendMemEx} is more efficient when blending/combining data to a bitmap since it excludes writing operation for transparent pixels and reading operation for opaque ones.

Note, that after using \textbf{BlendEx} functions, you have to call \texttt{EMMS}. Otherwise CPU will be unable to handle floating point instructions.
See Also

Blend, Combine, EMMS, Merge, MergeEx, TColor32
BlendLine

**procedure** BlendLine(Src, Dst: PColor32; Count: Integer);
Description
An optimized row processing version of Blend.
See Also

Blend, BlendLineEx, Color Types
BlendLineEx

procedure (Src, Dst: PColor32; Count: Integer; M: TColor32);
Description
An optimized row processing version of BlendEx.
See Also

BlendEx, BlendLine, Color Types
ColorAdd

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

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.
See Also

ColorSub, TColor32
ColorDiv

function ColorDiv(C1, C2: TColor32): TColor32;
Description
The resulting color components are the result of the division of corresponding components from C2 and C1.
See Also

TColor32
ColorMax

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

Resulting color components are maximums of corresponding components in C1 and C2.
See Also

ColorMin, TColor32
ColorMin

function ColorMin(C1, C2: TColor32): TColor32;
Description
Resultin color components are minimums of corresponding components in C1 and C2.
See Also

ColorMax, TColor32

Copyright ©2000-2007 Alex Denisov and Contributors - Graphics32 v1.8.3 - Build on 4-March-2007
ColorModulate

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

The resulting color components are calculated as products of corresponding components in \textbf{C1} and \textbf{C2} divided by $FF$: 
See Also

TColor32
ColorSub

function ColorSub(C1, C2: TColor32): TColor32;
Description
Subtracts $\mathbf{C_2}$ from $\mathbf{C_1}$. The resulting color components are clamped to $[0\ldots 255]$ range. The alpha channel is subtracted as well.
See Also
ColorAdd, TColor32
function CombineReg(X, Y, W: TColor32): TColor32;
procedure CombineMem(F: TColor32; var B: TColor32; W: TColor32);
Description

Returns the color with components calculated as linear interpolation between X and Y colors. The W parameter, which should be in [0..255] range, specifies the weight of the first color (X). The alpha channel is interpolated as well.

\[ S_{RGBA} = W \cdot X_{RGBA} + (1 - W) \cdot Y_{RGBA}; \]

**CombineReg** takes parameters and produces the result operating on CPU registers.

**CombineMem** operates with the background color referenced by a memory address.

Using **CombineMem** is more efficient when blending/combining data to a bitmap since it excludes writing operation for transparent pixels and reading operation for opaque ones.

Note, that after using **Combine** functions, you have to call **EMMS**. Otherwise CPU will be unable to handle floating point instructions.
See Also

Blend, BlendEx, EMMS, TColor32
EMMS

procedure EMMS;
Description

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

This function must be called after **Blend**, **BlendEx** or **Combine** calls, otherwise CPU will not be able to execute floating point instructions.

If you are using color algebra (such as **ColorAdd**, **ColorMax** etc.) that use MMX internally you also have to make sure that **EMMS** is called.

A typical example of using blending functions:

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

Blend, BlendEx, ColorAdd, ColorMax, Combine
Lighten

function Lighten(C: TColor32; Amount: Integer): TColor32;
**Description**

The resulting color is formed from the color components of the parameter C by adding *Amount* to each of them and clipping the sum to [0…255] range. *Amount* may be negative.
See Also

TColor32
function MergeReg(F, B: TColor32): TColor32;
procedure MergeMem(F: TColor32; var B: TColor32);
Description
Merges a foreground (F) color with the background color (B) using the alpha component of the foreground color. It does merge the alpha-channels.

\[ S_A = 1 - (1 - F_A) (1 - B_A) \]
\[ S_{RGB} = (F_A * F_{RGB} + B_A * (1 - F_A) * B_{RGB}) / S_A \]

**MergeReg** takes parameters and produces the result operating on CPU registers.

**MergeMem** operates with the background color referenced by a memory address.

Note, that after using the **Merge** function, you have to call **EMMS**. Otherwise CPU will be unable to handle floating point instructions.
See Also

Blend, BlendEx, Combine, EMMS, MergeEx, TColor32
MergeEx

function MergeRegEx(F, B, M: TColor32): TColor32;

procedure MergeMemEx(F: TColor32; var B: TColor32; M: TColor32);
Description
Merges the foreground color with the background color using alpha of the foreground color scaled by the master alpha value $M$. Alpha-Channels are also merged.

\[
S_A = 1 - (1 - (M \times F_A)) \times (1 - B_A)
\]
\[
S_{RGB} = ((M \times F_A) \times F_{RGB} + B_A \times (1 - F_A) \times B_{RGB}) / S_A
\]

$M$ is defined as `TColor32` to avoid unnecessary type conversions, it must store only values in [0..255] range, the function does not perform range checking and the result in case $M > 255$ is not specified.

**MergeRegEx** takes parameters and produces the result operating on CPU registers.

**MergeMemEx** operates with the background color referenced by a memory address.

Note, that after using **MergeEx** functions, you have to call **EMMS**. Otherwise CPU will be unable to handle floating point instructions.
See Also

Blend, BlendEx, Combine, EMMS, Merge, TColor32
TBlendLine

```plaintext
type TBlendLine = procedure (Src, Dst: PColor32; Count: Integer);
```
Description

TBlendline is a type for line blending procedures. It is mainly used internally for handling specific CPU feature optimized versions (MMX etc.). See source code for details.
See Also

BlendLine, Color Types
TBlendLineEx

```delphi
 type TBlendLineEx = procedure (Src, Dst: PColor32; Count: Integer; M: TColor32);
```
**Description**

*TBlendline* is a type for line blending procedures with additional master alpha parameter. It is mainly used internally for handling specific CPU feature optimized versions (MMX etc.). See source code for details.
See Also

BlendLineEx
TBlendMem

```pascal
  type TBlendMem = procedure (F: TColor32; var B: TColor32);
```
**Description**

*TBlendMem* is a type for blending procedures. It is mainly used internally for handling specific CPU feature optimized versions (MMX etc.). See source code for details.
See Also

Blend, Color Types
TBlendMemEx

\[
\text{type } \text{TBlendMemEx} = \text{procedure (F: TColor32; var B: TColor32; M: TColor32);}
\]
**Description**

**TBlendMemEx** is a type for blending procedures. It is mainly used internally for handling specific CPU feature optimized versions (MMX etc.). See source code for details.
See Also

BlendEx, Color Types
type TBlendReg = function (F, B: TColor32): TColor32;
**Description**

**TBlendReg** is a type for blending procedures. It is mainly used internally for handling specific CPU feature optimized versions (MMX etc.). See source code for details.
See Also

Blend, Color Types
TBlendRegEx

\[
\text{type TBlendRegEx = function (F, B, M: TColor32): TColor32};
\]
Description

TBlendRegEx is a type for blending procedures. It is mainly used internally for handling specific CPU feature optimized versions (MMX etc.). See source code for details.
See Also

BlendEx, Color Types

Copyright ©2000-2007 Alex Denisov and Contributors - Graphics32 v1.8.3 - Build on 4-March-2007
TCombineMem

```delphi
type TCombineMem = procedure (F: TColor32; var B: TColor32; W: TColor32);
```
Description

TCombineMem is a type for combining procedures. It is mainly used internally for handling specific CPU feature optimized versions (MMX etc.). See source code for details.
See Also

Combine
TCombineReg

```plaintext
type TCombineReg = function (X, Y, W: TColor32): TColor32;
```
Description

TCombineReg is a type for color combining procedures. It is mainly used internally for handling specific CPU feature optimized versions (MMX etc.). See source code for details.
See Also

Color Types, Combine
MMX_ACTIVE

var MMX_ACTIVE: Boolean;
**Description**

This variable indicates if the CPU supports MMX instructions. **MMX_ACTIVE** is initialized automatically upon the library initialization.
TClassList.Find

function Find(AClassName: string): TClass;
**Description**

Searches the list for a specific class name. If no match is found, this function method will return *nil*. 
TClassList.GetClassNames

procedure GetClassNames(Strings: TStrings);
Description

Adds names of all classes currently in the list to the string list provided by the parameter Strings.
TPointerMap.Count

property Count: Integer;
**Description**

Returns the amount of entries in the pointer map.
TPointerMap.Data

*property* Data[Item: PItem]: PData; default;
Description

Returns or sets the associated data for an entry **Item**. If the entry does not exist in the map, an exception of type **EListError** will be raised.
The data lookup is fast because it uses a hash-like structure internally.
TPointerMap.Add

```haskell
function Add(NewItem: PItem): PPData; overload;
function Add(NewItem: PItem; out IsNew: Boolean): PPData; overload;
function Add(NewItem: PItem; NewData: PData): PPData; overload;
function Add(NewItem: PItem; NewData: PData; out IsNew: Boolean): PPData; overload;
```
**Description**

Adds a new item to the pointer map. The function method will return a pointer to the data **PPData**.

**NewItem** specifies the pointer of the new item;

**NewData** specified the pointer of the data associated with the new item;

**IsNew** returns whether the item was added to the list or if an existing copy was modified.
TPointerMap.Clear

procedure Clear;
Description
Clears all entries from the pointer map.
**TPointerMap.Contains**

```pascal
function Contains(Item: PItem): Boolean;
```
Description
Returns whether an entry exists in the pointer map.
TPointerMap.Find

function Find(Item: PItem; out Data: PPData): Boolean;
Description

Returns whether an entry **Item** exists in the pointer map. If it exists, the Data pointer is returned by **Data**.
TPointerMap.Remove

function Remove(Item: PItem): PData;
Description
Remove the entry specified by Item from the pointer map and returns its Data.
### TPointerMapIterator.Data

| property | Data: PData; |
**Description**

Returns the associated **Data** entry of current entry **Item** of the iterator.
See Also

Item
TPointerMapIterator.Item

```pascal
property Item: PItem;
```
Description
Returns the current entry **Item** of the iterator.
TPointerMapIterator.Next

function Next: Boolean;
**Description**

Retrieves the next entry in the pointer map and saves a copy to the Data and Item property. Returns True if the operation succeeded, False if there is no more entry to fetch.
See Also

Data, Item
constructor Create(Rasterizer: TRasterizer; Bitmap: TBitmap32; DstRect: TRect; Suspended: Boolean);
**Description**

Creates an instance of `TRenderThread`. If **Suspended** then the rasterization process will not be started, otherwise the threaded rasterization will start immediatly.
See Also

Rectangle Types, TBitmap32, TRasterizer, TRenderThread
TSyntheticImage32.AutoRasterize

property AutoRasterize: Boolean;
Description

If AutoRasterize then the synthetic image will rasterize when the associated rasterizer has changed, otherwise Rasterize has to be called in order to start rasterization. AutoRasterize is set true upon creation.
See Also

Rasterize, Rasterizer
property BitmapAlign: TBitmapAlign;
type TBitmapAlign = (baTopLeft, baCenter, baTile, baCustom);
**Description**

Specifies how the buffer bitmap is located within the control's boundaries.
TSyntheticImage32.Buffer

```
property Buffer: TBitmap32; // Read only;
```
**Description**

Provides direct access to the back buffer.
See Also

TBitmap32

Copyright ©2000-2007 Alex Denisov and Contributors - Graphics32 v1.8.3 - Build on 4-March-2007
TSyntheticImage32.DstRect

| property  | DstRect: TRect; |
Description

**DstRect** specifies the rasterization boundaries when [render mode](#) is `rnmConstrained`. 
See Also

Rectangle Types, RenderMode
property Rasterizer: TRasterizer;
**Description**

This property specifies which rasterizer is used to render to the synthetic image control.
See Also

TRasterizer
property RenderMode: TRenderMode;
**Description**

Specifies whether the complete control is updated or only the area specified by the `DstRect` property.
See Also

DstRect, TRenderMode
TSyntheticImage32.Rasterize

procedure Rasterize;
Description

Rasterize starts the rasterization process.
procedure Rasterize(Rasterizer: TRasterizer; Bitmap: TBitmap32; DstRect: TRect);
Description

**Rasterize** is a simple [TRenderThread](#) based rasterize auxiliary routine. Calling Rasterize will start up a separate rasterization thread and rasterize through that with the provided **Rasterizer**.
See Also

Rectangle Types, TBitmap32, TRasterizer, TRenderThread
**TRenderMode**

```plaintext
type TRenderMode = (rnmFull, rnmConstrained);
```
Description

TRenderMode defines how the synthetic image will render or rasterize.

- **rnmFull** – a color information from the foreground image completely replaces the background data;
- **rnmConstrained** – foreground and background colors are blended together using per-pixel opacity of the foreground bitmap (stored in its alpha channel), multiplied by its MasterAlpha;
See Also

TBitmap32.MasterAlpha
procedure AlphaToGrayscale(Dst, Src: TBitmap32);
Description
This function transforms an alpha channel from Src bitmap into grayscale (R=A, G=A, B=A) color in Dst bitmap. If necessary, Dst bitmap is resized to fit Src dimensions.

It writes only RGB components into Dst. The alpha channel (A) remains intact, provided that dimensions of Dst and Src match before AlphaToGrayScale call.

This function is useful when it is necessary to visualize the alpha channel, or when you want to store it as standard bitmap into a file.
See Also

IntensityToAlpha, TBitmap32
ApplyBitmask

procedure ApplyBitmask(
    Dst: TBitmap32;
    DstX: Integer;
    DstY: Integer;
    Src: TBitmap32;
    SrcRect: TRect;
    Bitmask: TColor32;
    LogicalOperator: TLogicalOperator;
);

procedure ApplyBitmask(
    ABitmap: TBitmap32;
    ARect: TRect;
    Bitmask: TColor32;
    LogicalOperator: TLogicalOperator;
);
**Description**

`ApplyBitmask` performs a logical (bitwise) operation on a bitmap fragment specified by `SrcRect` into location `(DstX, DstY)` or directly in-place on `ABitmap`. The operation is carried out in the following manner: "Source Pixel [Logical Operation] Bitmask" regardless of present destination or in-place operation. The logical operation is defined by the provided `Logical Operator`.

Each byte in the bitmask will be used as component corresponding operand. The following combinations illustrates different applications:

<table>
<thead>
<tr>
<th>Description</th>
<th>Parameter Values</th>
<th>Source</th>
</tr>
</thead>
</table>
| Zero out Red and Blue components, leave the rest untouched | LogicalOperator = IoAND
Bitmask = $FF00FF00 | ![Image](image1.jpg) |
| Inversion of Blue component, leave the rest untouched | LogicalOperator = IoXOR
Bitmask = $000000FF | ![Image](image2.jpg) |
| Full power to Red component, leave the rest untouched | LogicalOperator = IoOR
Bitmask = $00FF0000 | ![Image](image3.jpg) |
| Weird filter                                    | LogicalOperator = IoOR
Bitmask = $BABEC0DE | ![Image](image4.jpg) |

As seen above, the bitmask does not necessarily have to contain 0 or $FF in each component segment - intermediate values can also be used however with limited realistic applications (read: weird filters).

You may use auxiliary function `CreateBitmask` to create bitmasks.
See Also

Color Types, CopyComponents, CreateBitmask, Invert, TBitmap32,
TLogicalOperator, TRect
procedure ApplyLUT(Dst, Src: TBitmap32; const LUT: TLUT8;
PreserveAlpha: Boolean = False);
Description

Converts color from \texttt{Src} to \texttt{Dst} using a look-up table (\texttt{LUT} parameter) to map the color components. The alpha channel of the result is set to $\$FF$ by default. However, you can override this behaviour by setting \texttt{PreserveAlpha} to \texttt{True}.

\[
\begin{align*}
R_{\text{DST}} &= \text{LUT}[R_{\text{SRC}}]; \\
G_{\text{DST}} &= \text{LUT}[G_{\text{SRC}}]; \\
B_{\text{DST}} &= \text{LUT}[B_{\text{SRC}}];
\end{align*}
\]

If \texttt{PreserveAlpha} then $A_{\text{DST}} = A_{\text{SRC}}$ else $A_{\text{DST}} = \$FF$;

This function supports in-place operation, that is \texttt{Dst} may be the same as \texttt{Src}.
See Also

TBitmap32, TLUT8
ChromaKey

procedure ChromaKey(ABitmap: TBitmap32; TrColor: TColor32);
**Description**

Preprocesses a Bitmap for standard color key transparency, i.e. areas that are of color *TrColor* are made transparent.

It is possible to use this function several times for different colors.
See Also

Color Types, TBitmap32
procedure ColorToGrayscale(Dst, Src: TBitmap32;
  PreserveAlpha: Boolean = False);
Description
Calculates color intensity from Src and writes it as a grayscale image into Dst. The alpha channel in Dst will be set to $FF$ by default. However, you can override this behaviour by setting PreserveAlpha to True.
This function supports in-place operation, that is Dst may be the same as Src.
See Also

AlphaToGrayscale, TBitmap32
procedure CopyComponents(
    Dst: TBitmap32;
    Src: TBitmap32;
    Components: TColor32Components); overload;

procedure CopyComponents(
    Dst: TBitmap32;
    DstX: Integer;
    DstY: Integer;
    Src: TBitmap32;
    SrcRect: TRect;
    Components: TColor32Components); overload;
Description

CopyComponents copies the provided TColor32Components from Source to Destination (read: no blending etc. is involved). The first version doesn't take specific source and destination settings and will therefore resize (and implicitly clear) destination so it is same size as the source. The second version will not modify size of the destination and therefore leaves the components not overwritten by the operation untouched.
See Also

ApplyBitmask, TBitmap32, TColor32Components, TRect
CreateBitmask

function CreateBitmask(Components: TColor32Components): TColor32; overload;

This function takes a set of TColor32Component and creates a TColor32. If a given component is present in the set, the result component will get the value 255; if not present the result component will be zero. The context of application is to prepare a Bitmask parameter for the routine ApplyBitmask. To construct bitmasks containing any number in range [0..255] per component, use Color32.
See Also

ApplyBitmask, Color32, TColor32, TColor32Component, TColor32Components
IntensityToAlpha

procedure IntensityToAlpha(Dst, Src: TBitmap32);
Description
This function transforms the color intensity, stored in RGB channels of \texttt{Src} bitmap into the alpha channel in \texttt{Dst}. If necessary, the \texttt{Dst} bitmap is resized to fit \texttt{Src} dimensions.

The intensity is calculated using the \texttt{Intensity} function.

The function writes only the alpha component into \texttt{Dst}. Other channels remain intact (provided dimensions of \texttt{Dst} and \texttt{Src} match before \texttt{IntensityToAlpha} call).

\texttt{IntensityToAlpha} is useful when it is necessary to load an alpha channel stored in as an image.

The in-place operation is fully supported, that is \texttt{Dst} may be the same as \texttt{Src}.
See Also

AlphaToGrayscale, Intensity, TBitmap32
Invert

**procedure** Invert(Dst, Src: **TBitmap32**; Components : **TColor32Components** = [ccAlpha, ccRed, ccGreen, ccBlue]);
**Description**

Inverts the given set of components. If no component set is provided all color components, including the alpha component will be inverted.

This function supports in-place operation, that is \textbf{Dst} may be the same as \textbf{Src}. 

See Also

InvertRGB, TBitmap32, TColor32Components
InvertRGB

procedure InvertRGB(Dst, Src: TBitmap32);
**Description**
Inverts RGB color channels, and copies the alpha channel without inversion. If necessary, the Dst bitmap is resized to fit Src dimensions. This function is equal to calling Invert with Components = [ccRed, ccGreen, ccBlue]. This function supports in-place operation, that is Dst may be the same as Src.
See Also

Invert, TBitmap32
TLogicalOperator

define TLogicalOperator = (loXOR, loAND, loOR);
**Description**

An enumerated type that specifies logical (bitwise) operators for use in [ApplyBitmask](#).
See Also

ApplyBitmask

Copyright ©2000-2007 Alex Denisov and Contributors - Graphics32 v1.8.3 - Build on 4-March-2007
type TLUT8 = array [Byte] of Byte;
Description
A look-up table of 256 values, each ranging from 0 to 255.
TLUT8 is used to transform color data with ApplyLUT procedure.
See Also

ApplyLUT
TBitmap32Collection.Items

| property | Items[Index: Integer]: TBitmap32Item; |
**Description**

Provides indexed access to stored bitmap items.

Bitmap items are instances of `TBitmap32Item` object. Each bitmap item, in turn, contains a `TBitmap32` object, accessible through its `Bitmap` property.

The number of items may be obtained by reading inherited `Count` property.
See Also

TBitmap32Item.Bitmap, TBitmap32, TBitmap32Item
TBitmap32Collection.Add

function Add: TBitmap32Item;
**Description**

Creates a new instance of the bitmap item with the empty bitmap, adds it to the collection and returns reference to the new bitmap item.
Alternatively, you may use `TBitmap32Item.Create` method to add new items to the collection.
See Also

TBitmap32Item.Create, TBitmap32Item
TBitmap32Collection.Create

**constructor** Create(AOwner: TPersistent; ItemClass: TBitmap32ItemClass);

**type** TBitmap32ItemClass = class of TBitmap32Item;
**Description**

Creates an instance of [TBitmap32Collection](#).

ItemClass specifies must specify [TBitmap32Item](#) or its descendant. When calling collection's [Add](#) method it will create instances of the class specified in [ItemClass](#) parameter.
See Also

Add, TBitmap32Collection, TBitmap32Item
property Bitmap: TBitmap32;
**Description**
Provides access to the contained bitmap.
See Also

TBitmap32
TBitmap32Item.Create

constructor Create(Collection: TCollection); override;

**Description**

Creates an instance of `TBitmap32Item`.

When creating bitmap items at run time, pass the corresponding `TBitmap32Collection` object as the parameter.

Alternatively, you may use `TBitmap32Collection.Add` method to create new bitmap items:

```pascal
var BitmapCollection: TBitmap32Collection;
    B1, B2: TBitmap32Item;
begin
    BitmapCollection := TBitmap32Collection.Create;
    try
        B1 := TBitmap32Item.Create(BitmapCollection);
        B2 := BitmapCollection.Add;
        // some operations with B1, B2...
    finally
        BitmapCollection.Free;
    end;
end;
```

Note, that it is not necessary to call the `Free` method for bitmap items, since they are owned by the collection.
See Also

TBitmap32Collection.Add, TBitmap32Collection, TBitmap32Item
TBitmap32Item.Destroy

**destructor** Destroy; **override**;
Description
Removes a bitmap item from the collection, destroys it and frees allocated memory.
Do not call **Destroy** in your application, use standard **Free** method instead.
Since bitmap items are owned by the collection, they all are destroyed automatically when the collection is destroyed. There is no need to call **Free** method for bitmap items unless you want them to be explicitly destroyed *before* the collection is destroyed.
See Also

TBitmap32Collection.Items
TBitmap32List.Bitmap

property Bitmap[Index: Integer]: TBitmap32; default;
**Description**

Provides an indexed access to stored bitmaps.

Using `Bitmap[Index]` is the same as using `Bitmaps[Index].Bitmap`. 
See Also

Bitmaps, TBitmap32
TBitmap32List.Bitmaps

property Bitmaps: TBitmap32Collection;
Description
Provides indexed access to contained TBitmap32Collection object. The total number of items may be obtained through collection's Count property.
See Also

TBitmap32Collection
TBitmap32List.Create

constructor Create(AOwner: TComponent); override;
Description
Creates an instance of TBitmap32List.
See Also

TBitmap32List
TBitmap32List.Destroy

**destructor** Destroy; **override**;
**Description**

Destroys the contained bitmap collection, including all the bitmaps in it, then calls the inherited destructor.

Do not call **Destroy** in your application, use standard **Free** method instead.
TCustomImage32.Bitmap

| property Bitmap: TBitmap32; |
Description
Provides access to the bitmap image.
See Also

Bitmap Image, TBitmap32
property BitmapAlign: TBitmapAlign;
type TBitmapAlign = (baTopLeft, baCenter, baTile, baCustom);
**Description**

Specifies how the bitmap image is located within the control's boundaries. Note, that in TCustomImgView32 this property should always remain baCustom.
See Also

Bitmap Image, TCustomImgView32
TCustomImage32.Layers

property Layers: TLayerCollection;
Description
Provides access to the layer collection.
See Also
TLayerCollection, Using Layers
TCustomImage32.OffsetHorz

property OffsetHorz: Single;
**Description**

Specifies the horizontal offset of the bitmap image in baCustom BitmapAlign mode.
See Also

Bitmap Image, BitmapAlign
**TCustomImage32.OffsetVert**

| **property** OffsetVert: Single; |
**Description**

Specifies the vertical offset of the bitmap image in baCustom **BitmapAlign** mode.
See Also

Bitmap Image, BitmapAlign

Copyright ©2000-2007 Alex Denisov and Contributors - Graphics32 v1.8.3 - Build on 4-March-2007
property PaintStages: TPaintStages;
**Description**

Provides access to paint stages of the control, allowing to customize the order in which it repaints itself.
See Also

Paint Stages, TPaintStages
TCustomImage32.Scale

property Scale: Single;
Description
Specifies the X and Y scale of the bitmap image when ScaleMode is smScale or smOptimalScaled.
Warning: do not set too small scales in baTile BitmapAlign mode, as this can have severe impact on performance.
See Also

Bitmap Image, BitmapAlign, Scale, ScaleMode, ScaleX

Copyright ©2000-2007 Alex Denisov and Contributors - Graphics32 v1.8.3 - Build on 4-March-2007
TCustomImage32.ScaleMode

<table>
<thead>
<tr>
<th>property</th>
<th>ScaleMode: <strong>TScaleMode</strong>;</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>TScaleMode = (smNormal, smStretch, smScale, smResize, smOptimal, smOptimalScaled);</td>
</tr>
</tbody>
</table>
Description

Specifies the current scale mode for the bitmap image.

**smNormal** will display the bitmap image with its original size;

**smStretch** will stretch the bitmap image to fit the control's boundaries;

**smResize** will proportionally resize the bitmap image to fit the control's boundaries;

**smScale** will proportionally scale the bitmap image using the factor provided by the [Scale](#) property;

**smOptimal** will proportionally fit the bitmap image into the control's boundaries if the viewport is smaller than the image size. Otherwise it will be displayed with its original size;

**smOptimalScaled** will proportionally fit the bitmap image into the control's boundaries if the viewport is smaller than the image size multiplied by the [Scale](#) property. Otherwise it will be displayed with its original size.

Note, that [TCustomImgView32](#) descendants only support smScale, smOptimal and smOptimalScaled mode.
See Also

Bitmap Image, Scale, TCustomImgView32, TScaleMode
TCustomImage32.ScaleX

property ScaleX: Single;
**Description**

Specifies the X scale of the bitmap image when `ScaleMode` is `smScale` or `smOptimalScaled`.

Warning: do not set too small scales in baTile `BitmapAlign` mode, as this can have severe impact on performance.
See Also

Bitmap Image, BitmapAlign, Scale, ScaleMode, ScaleY
TCustomImage32.ScaleY

property ScaleY: Single;
**Description**

Specifies the Y scale of the bitmap image when `ScaleMode` is smScale or smOptimalScaled.

Warning: do not set too small scales in baTile `BitmapAlign` mode, as this can have severe impact on performance.
See Also

Bitmap Image, BitmapAlign, Scale, ScaleMode, ScaleX
TCustomImage32.BeginUpdate

procedure BeginUpdate; virtual;
**Description**
Disables change notifications until the paired `EndUpdate` method is called.

`BeginUpdate...EndUpdate` blocks may be nested.
See Also

EndUpdate, OnChange
function BitmapToControl(const APoint: TPoint): TPoint;
Description
Transforms point coordinates from the reference frame of bitmap image to control's reference frame.
See Also

Bitmap Image, ControlToBitmap, TPoint
TCustomImage32.Changed

    procedure Changed; virtual;
**Description**

In case the update notification is not masked with `BeginUpdate`, `Changed` invalidates the control and then calls the `OnChange` event.
See Also

BeginUpdate, EndUpdate, TCustomPaintBox32.Invalidate, OnChange
function ControlToBitmap(const APoint: TPoint): TPoint;
Description

Transforms point coordinates from control's reference frame to the reference frame of bitmap image.
See Also
Bitmap Image, BitmapToControl, TPoint
TCustomImage32.Create

    constructor Create(AOwner: TComponent); override;
**Description**

Creates an instance of [TCustomImage32](#). After calling the inherited constructor, **Create** initializes the following properties:

- **BitmapAlign** to baTopLeft;
- **OffsetHorz** to 0;
- **OffsetVert** to 0;
- **Scale** to 1.0;
- **ScaleMode** to smNormal;
See Also

BitmapAlign, OffsetHorz, OffsetVert, Scale, ScaleMode, TCustomPaintBox32
TCustomImage32.Destroy

destructor Destroy; override;
**Description**

Destroys an instance of [TCustomImage32](#), freeing the memory allocated to bitmap image and layers.

Do not call the **Destroy** directly in your application. Call the standard **Free** method instead.
See Also

TCustomImage32
TCustomImage32.EndUpdate

property EndUpdate: TNotifyEvent;
Description
Re-enables generation of the OnChange event, masked with BeginUpdate method.

BeginUpdate...EndUpdate pairs may be safely nested.
See Also

BeginUpdate, OnChange

Copyright ©2000-2007 Alex Denisov and Contributors - Graphics32 v1.8.3 - Build on 4-March-2007
procedure ExecBitmapFrame(Dest: TBitmap32; Stage: Integer);
virtual;
**Description**

Draw a dotted frame around the scaled bitmap image in design time. This method is called by the control when it executes PST_BITMAP_FRAME paint stage.
See Also

Paint Stages, TBitmap32
TCustomImage32.ExecClearBackgnd

procedure ExecClearBuffer(Dest: TBitmap32; Stage: Integer);
virtual;
**Description**

Clears the visible area of the buffer with the color specified in the `Color` property of the control.

This method is called by the control when it executes `PST_CLEAR_BACKGND` paint stage.
See Also

Paint Stages, TBitmap32

Copyright ©2000-2007 Alex Denisov and Contributors - Graphics32 v1.8.3 - Build on 4-March-2007
procedure ExecClearBuffer(Dest: TBitmap32; Stage: Integer);
virtual;
Description
Clears the whole buffer with the color specified in the *Color* property of the control.

This method is called by the control when it executes PST_CLEAR_BUFFER paint stage. Note, however, that by default, *TCustomImage32* does not execute PST_CLEARBUFFER stage, instead of it it uses PST_CLEAR_BACKGND, which works faster in some cases.
See Also

Paint Stages, TBitmap32, TCustomImage32
TCustomImage32.ExecControlFrame

procedure ExecControlFrame(Dest: TBitmap32; Stage: Integer);
virtual;
Description

Draw a dotted frame around control boundaries in design-ime. This method is called by the control when it executes PST_CONTROL_FRAME paint stage.
See Also

Paint Stages, TBitmap32
procedure ExecCustom(Dest: TBitmap32; Stage: Integer);
   virtual;
Description

Calls the OnPaintStage event.

This method is called by the control when it executes PST_CUSTOM stage.
See Also
OnPaintStage, Paint Stages, TBitmap32
TCustomImage32.ExecDrawBitmap

procedure ExecDrawBitmap(Dest: TBitmap32; Stage: Integer); virtual;
**Description**

Paints the bitmap image.

This method is called by the control when it executes PST_DRAW_BITMAP stage.
See Also

Paint Stages, TBitmap32
TCustomImage32.ExecDrawLayers

procedure ExecDrawLayers(Dest: TBitmap32; Stage: Integer);
virtual;
**Description**

Paints layers using the layer mask of the corresponding stage. This method is called by the control when it executes PST_DRAW_LAYERS stage.
See Also

Paint Stages, TBitmap32

Copyright ©2000-2007 Alex Denisov and Contributors - Graphics32 v1.8.3 - Build on 4-March-2007
TCustomImage32.GetPictureRect

function GetPictureRect: TRect;
**Description**

Returns the location of the bitmap image after an appropriate `ScaleMode` and `BitmapAlign` is applied. In bmTile `ScaleMode` it returns the location of a top-left tile.
### TCustomImage32.GetPictureSize

| function | GetPictureSize: TSize; |
Description

Returns the size of the bitmap image after an appropriate ScaleMode is applied. In bmTile ScaleMode it returns the size of a tile.
See Also

**Bitmap Image**, **GetPictureRect**, **ScaleMode**, **TSize**
TCustomImage32.Resize

procedure Resize; override;
procedure SetupBitmap(
    DoClear: Boolean = False;
    clearColor: TColor32 = $FF000000);
Description
Sets the size of the bitmap image to the size of the buffered area. This method does not account for current ScaleMode.

If DoClear = True, the resized image will be filled with a color specified in the ClearColor parameter.
See Also

Bitmap Image, TCustomPaintBox32.GetViewportRect, ScaleMode, TColor32
property OnBitmapResize: TNotifyEvent;
Description

This event is generated after the bitmap image is resized, that is its **Width** or **Height** property changes. **OnBitmapResize** is *not* generated when **Scale** or **ScaleMode** is changed.
See Also

TCustomMap.Height, Scale, ScaleMode, TCustomMap.Width
TCustomImage32OnChange

propertyOnChange: TNotifyEvent;
**Description**

**OnChange** is an abstract change notification event, which is called by some of the descendants of **TCustomPaintBox32** immediately after changes have been made to their contents.

In **TCustomImage32**, for example, this includes redirection of change notification events from the contained bitmap and from layers.

This event, however, is not called by **TCustomPaintBox32** control itself, unless you call the **Changed** method explicitly.

Change notification may be disabled with **BeginUpdate** call and re-enabled with **EndUpdate** call.
property OnGDIOverlay: TNotifyEvent;
Description

[ Note: Most likely this event will be removed in future versions. Do not use it ]

This event is called after the image has been combined with layers in the back-buffer and copied to the screen. In the event handler you can provide additional drawing using the standard *Canvas* of the control.

Drawing at this stage will not affect the contents of the back-buffer.
See Also

OnPaintStage, Paint Stages

Copyright ©2000-2007 Alex Denisov and Contributors - Graphics32 v1.8.3 - Build on 4-March-2007
TCustomImage32.OnInitStages

property OnInitStages: TNotifyEvent;
**Description**

OnInitStages is called by TCustomImage32 in the process of initialization right after it finishes building default list of paint stages. You may also alter the stages in the OnCreate event handler of the owner form or anywhere else in your program.
See Also

Paint Stages, TCustomImage32
property OnMouseDown: TImgMouseEvent;

type TImgMouseEvent = procedure(Sender: TObject; Button: TMouseButton; Shift: TShiftState; X, Y: Integer; Layer: TCustomLayer) of object;
**Description**

This event occurs when user pushes the mouse button in the image area. The **Layer** parameter indicates the layer under the mouse, or nil in case the mouse is over the background area.

Whether the layer is under the mouse or not, is determined by the **HitTest** function and by the LOB_MOUSEEVENTS bit in **LayerOptions**. The top-most layer which passes the test is passed into the **Layer** parameter of the event.

Note, that once mouse was pressed, the layer (or the background) captures the mouse messages until the mouse button is released.
See Also

`TCustomLayer.HitTest`, `TCustomLayer.LayerOptions`, `OnMouseMove`, `OnMouseUp`, `TCustomLayer`
**TCustomImage32.OnMouseMove**

```plaintext
property OnMouseMove: TImgMouseMoveEvent;

type TImgMouseMoveEvent = procedure(Sender: TObject;
                                          Shift: TShiftState; X, Y: Integer; Layer: TCustomLayer) of object;
```
**Description**

This event is fired while the mouse is dragged over the image. **Layer** indicates the layer under the mouse, nil – when there is no layer under the mouse, or the layer which has captured mouse events.
See Also

OnMouseDown, OnMouseUp, TCustomLayer
**TCustomImage32.OnMouseUp**

```plaintext
property OnMouseUp: TImgMouseEvent;

type TImgMouseEvent = procedure(Sender: TObject; Button: TMouseButton; Shift: TShiftState; X, Y: Integer; Layer: TCustomLayer) of object;
```
Description

This event occurs when the mouse button is released. **Layer** indicates the layer under the mouse, nil – when there is no layer under the mouse, or the layer which has captured mouse events.
See Also

Changed, TCustomLayer
property OnPaintStage: TPaintStageEvent;

type TPaintStageEvent = procedure(Sender: TObject; Dest: TBitmap32; StageNum: Cardinal) of object;
Description
An event, called by the control while it repaints itself. By default, TCustomImage32 does not generate OnPaintStage events, but this can be changed by adding PST_CUSTOM paint stages.
In the event handler, do not paint to Buffer directly, use the passed Dest parameter instead.
See Also

`TCustomPaintBox32.Buffer`, `Paint Stages`, `TBitmap32`, `TCustomImage32`
TCustomImage32.OnPixelCombine

property OnPixelCombine: TPixelCombineEvent;

type TPixelCombineEvent = procedure(F: TColor32; var B: TColor32; M: TColor32) of object;
Description
This event is redirected from the bitmap image, in order to make it accessible in the Delphi's Object Manager. There is no difference between using `TCustomImage32.OnPixelCombine` and `TBitmap32.OnPixelCombine`. 
See Also

TBitmap32.OnPixelCombine, TColor32, TCustomImage32, TPixelCombineEvent
TCustomImage32.OnScaleChange

property OnScaleChange: TNotifyEvent;
Description

OnScaleChange is change notification event, which is called whenever the Scale property changes.
See Also

Changed, Scale, ScaleX, TCustomImage32
TCustomImgView32.Centered

property Centered: Boolean;
Description
Indicates the alignment of the container bitmap image. When Centered is True (default) the image is automatically positioned at the center of the control, otherwise it is located at the top-left corner.
TCustomImgView32.OverSize

property OverSize: Integer;
**Description**

This property introduces a frame of **OverSize** pixels in each direction around the bitmap. This will keep the bitmap scrollable even if it is scaled to fit the viewport. **OverSize** is set to 0 by default.
property ScrollBars: TIVScrollProperties;
**Description**
Controls appearance of scroll bars.
The scroll bars in `TCustomImgView32` are of the `TCustomRangeBar` type. Use `ScrollBars` to control properties of the vertical and horizontal scroll bar simultaneously.
See Also

TCustomImgView32, TCustomRangeBar, TIVScrollProperties
property SizeGrip: TSizeGripStyle;
type TSizeGripStyle = (sgAuto, sgNone, sgAlways);
Description

Modify the SizeGrip property to control the appearance of the resizing grip at the lower-right corner of the control:

In `shAuto` mode, visibility of the grip is determined similar to tat in other standard controls (e.g. TScrollBox), it becomes visible when control is client-aligned (if it has parents, they should be client-aligned as well).

Use `snNone` and `sgAlways` styles to specify its appearance explicitly.
See Also

TSizeGripStyle
TCustomImgView32.Create

constructor Create(AOwner: TComponent); override;
**Description**

Creates an instance of `TCustomImgView32`.

After calling the inherited constructor, `Create` initializes the `Centered` property to `True`.
See Also

Centered, TCustomImgView32
TCustomImgView32.Destroy

destructor Destroy; override;
Description

Destroys the instance of TCustomImgView32 class and frees all associated memory.
Do not call Destroy directly, use Free method instead.
See Also

TCustomImgView32
TCustomImgView32.GetViewportRect

function GetViewportRect: TRect; override;
**Description**

This property is overridden to return the rectangle, corresponding to the control dimensions minus the area covered by the scroll bars.
See Also

TIVScrollProperties.Size, TRect
TCustomImgView32.Loaded

procedure Loaded; override;
Description

**Loaded** is overridden to adjust positions of scroll bars after the control loads it properties from the DFM file.

Normally, you should not use it in your application.
TCustomImgView32.Resize

procedure Resize; override;
Description

**Resize** is overridden to adjust positions of scroll bars after the control resizes. Normally, you should not use it in your application.
procedure Scroll(Dx, Dy: Integer);
Description

Scrolls the viewport by the offsets given by the Dx and Dy parameters.
TCustomImgView32.ScrollToCenter

procedure ScrollToCenter(X, Y: Integer);
Description
Relatively scrolls the center of the viewport to the X and Y coordinates.
TCustomImgView32.OnScroll

property OnScroll: TNotifyEvent;
Description

This event is generated when user scrolls the image using scroll bars.
TCustomPaintBox32.Buffer

```object
property Buffer: TBitmap32; // Read only;
```
**Description**

Provides direct access to the back buffer.

Note that the actual size of the buffered may be up to BufferOversize pixels larger than the buffered area.
See Also

BufferOversize, BufferValid, TBitmap32
property BufferOversize: Integer; // protected
**Description**

**BufferOversize** specifies how the back buffer is reallocated when the control is resized.

When **BufferOversize** equals 0, the back buffer is reallocated each time the control is resized to suite the new control dimensions.

When set to some positive value (40 by default), the back buffer is allowed to be up to **BufferOversize** pixels larger than the control is in each direction. The buffer resizing algorithm tracks size changes and reallocates the buffer only when its size becomes smaller than size of the buffered area or when the difference between the sizes exceeds the specified amount of pixels.
See Also

Buffer, GetViewportRect
property BufferValid: Boolean; // protected
**Description**

BufferValid specifies if the current content of the Buffer bitmap is up-to-date and whether the buffer needs to be repainted.

When TCustomPaintBox32 repaints itself, it checks whether the buffer is valid. If BufferValid is True, the control simply flushes its contents to the screen, otherwise, it executes DoPaintBuffer method, which updates the buffer and sets BufferValid to True, then it copies content of the buffer to the screen.

Normally, this property should not be accessed in your application, since TCustomPaintBox32 tracks the buffer state and updates it automatically.
See Also

Buffer, DoPaintBuffer, TCustomPaintBox32
property Options: TPaintBoxOptions;

type TPaintBoxOptions = set of (pboWantArrowKeys, pboAutoFocus);
Description
This property controls additional options of the paint box and its descendants:

**pboWantArrowKeys** - indicates whether the keyboard arrow keys appear in OnKeyDown event;

**pboAutoFocus** - indicates that the paint box automatically captures keyboard focus when clicked with the mouse. The focus is acquired before the OnMouseDown event is fired.
See Also

TPaintBoxOptions
property RepaintMode: TRepaintMode;
type TRepaintMode = (rmFull, rmDirect, rmOptimizer);
**Description**

This property controls the way repaints of the contents are handled by TCustomPaintBox32 and its descendants:

- **rmFull** - indicates whether always to repaint everything;
- **rmDirect** - indicates to use a direct repaint rather than a deferred invalidation. Please note, that this mode is not supported for layer supporting control like TCustomImage32 and its descendants;
- **rmOptimizer** - indicates to use the repaint optimizer to just repaint changed areas. Especially layer operations benefit from this mode.
See Also

Repaint Optimization, TCustomImage32, TPaintBoxOptions
TCustomPaintBox32.Destroy

```cpp
destructor Destroy; override;
```
**Description**
Destroys an instance of TCustomPaintBox32 object and frees all associated memory.
Do not call **Destroy** directly, use **Free** method instead.
See Also
TCustomPaintBox32
TCustomPaintBox32.DoPaintBuffer

procedure DoPaintBuffer; virtual;
Description

DoPaintBuffer is an abstract virtual method, called by the controll during its repainting.
**TCustomPaintBox32.Flush**

```plaintext
procedure Flush; overload;
procedure Flush(const SrcRect: TRect); overload;
```
**Description**

Immediately copies the contents of the buffer, or its part passed in `SrcRect` to the screen. Use this function when you want to bypass normal Windows invalidating/repainting procedure, for example when the buffer is painted by the other thread.

Both functions are thread-safe.
See Also

TRect
TCustomPaintBox32.ForceFullInvalidate

procedure ForceFullInvalidate;
Description

**ForceFullInvalidate** is similar to **Invalidate**. However, it will override the **repaint optimizer** for the following deferred repaint.
See Also

Invalidate, Repaint Optimization, RepaintMode
TCustomPaintBox32.GetViewportRect

function GetViewportRect: TRect; virtual; // protected
Description

Returns coordinates of the buffered area relative to control's client area. By default, it coincides with the control's client rectangle, that is, the function returns (0, 0, ClientWidth, ClientHeight) rectangle.

Descendants may override this method and return a different rectangle if they have unbuffered areas, like scroll bars.
See Also

TRect
TCustomPaintBox32.Invalidate

procedure Invalidate; override;
Description

Invalidate is overridden from TWinControl. In addition to standard behavior, it also invalidates the buffer content, so that the buffer gets actually updated during the following paint procedure.

You do not have to call this method explicitly in your application.
TCustomPaintBox32.Loaded

procedure Loaded; override;
Description

**Loaded** is overridden from *TWinControl* to keep the buffer dimensions consistent with control's dimensions. You do not have to call this method explicitly in your application.
TCustomPaintBox32.Resize

procedure Resize; override;
Description

Resize is overridden from TWinControl. It keeps the size of the buffer consistent with control's dimensions after the control resizes. Normally, you will not call this method in your applications.

Whether the buffer is actually resized or not is influenced by the BufferOversize property.

Resizing of the control will also invalidate its buffer.

Resize generates the standard OnResize event as well.
See Also

BufferOversize
procedure SetBounds(ALeft, ATop, AWidth, AHeight: Integer); override;
Description

SetBounds is overridden from TWinControl to keep buffer dimensions consistent with the control size.

You do not have to call this method explicitly in your application.
property OnMouseEnter: TNotifyEvent;
Description

OnMouseEnter is called by TPaintBox32 when the Mouse cursor has entered the control.
See Also

TCustomPaintBox32
property OnMouseLeave: TNotifyEvent;
Description

OnMouseLeave is called by TPaintBox32 when the Mouse cursor has left the control.
See Also

TCustomPaintBox32
**TIVScrollProperties.Backgnd**

<table>
<thead>
<tr>
<th>property</th>
<th>Backgnd: TRBBackgnd;</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>TRBBackgnd = (bgPattern, bgSolid);</td>
</tr>
</tbody>
</table>
Description
Provides access to the Backgd property in scroll bars.
See Also

TArrowBar.Backgnd, TRBBackgnd
TIVScrollProperties.BorderStyle

property BorderStyle: TBorderStyle;

type TBorderStyle = (bsNone, bsSingle); // is declared in Forms.pas
**Description**

Provides access to the **BorderStyle** property in scroll bars.
See Also
TArrowBar.BorderStyle
TIVScrollProperties.ButtonSize

property ButtonSize: Integer;
Description
Provides access to the ButtonSize property in scroll bars.
See Also

TArrowBar.ButtonSize
TIVScrollProperties(HandleColor

```delphi
property HandleColor: TColor;
```
Description
Provides access to the HandleColor property in scroll bars.
See Also

TArrowBar.HandleColor

Copyright ©2000-2007 Alex Denisov and Contributors - Graphics32 v1.8.3 - Build on 4-March-2007
TIVScrollProperties.Increment

    property Increment: Integer;
Description
Provides access to the Increment property in scroll bars.
See Also

TCustomRangeBar.Increment
TIVScrollProperties.ShowArrows

**property** ShowArrows: Boolean;
Description
Provides access to the ShowArrows property in scroll bars.
See Also

ShowArrows
TIVScrollProperties.ShowHandleGrip

```plaintext
property ShowHandleGrip: Boolean;
```
**Description**

Provides access to the `ShowHandleGrip` property in scroll bars.
See Also

TArrowBar.ShowHandleGrip
TIVScrollProperties.Size

property Size: Integer;
**Description**

Simultaneously controls the *Height* of horizontal and *Width* of vertical scroll bars.

If **Size** is 0 (default), the corresponding size is set to the default one, specified in current desktop settings.
TIVScrollProperties.Style

property Style: TRBStyle;
Description
Provides access to the Style property in scroll bars.
See Also
TArrowBar.Style, TRBStyle
property Visibility: TScrollbarVisibility;
type TScrollBarVisibility = (svAlways, svHidden, svAuto);
Description

Controls the visibility of the scrollbars.

svAlways will always display the scrollbars;

svHidden will always hide the scrollbars;

svAuto will only show the scrollbars if the (scaled) workspace does not completely fit into the current viewport.
TPaintBox32.Create

constructor Create(AOwner: TComponent);
Description
Creates an instance of TPaintBox32.
See Also

TPaintBox32
TPaintBox32.DoPaintBuffer

procedure DoPaintBuffer; override; // protected
Description
This method is overridden from TCustomPaintBox32, it calls the OnPaintBuffer event.
See Also

TCustomPaintBox32.DoPaintBuffer, OnPaintBuffer, TCustomPaintBox32
TPaintBox32.Flush

procedure Flush; overload;
procedure Flush(const SrcRect: TRect); overload;
**Description**

Immediately copies the contents of the buffer, or its part passed in `SrcRect` to the screen. Use this function when you want to bypass normal Windows invalidating/repainting procedure, for example when the buffer is painted by the other thread.

Both functions are thread-safe.
TPaintBox32.OnPaintBuffer

property OnPaintBuffer: TNotifyEvent;
Description

OnChange is called by TPaintBox32 when it needs to repaint its back buffer itself.
See Also

TCustomPaintBox32.Buffer, DoPaintBuffer, TCustomPaintBox32
TPaintStages.Items

property Items[Index: Integer]: PPaintStage; default; // Read only;
**Description**

The **Items** property can be used to obtain a pointer to the specified stage. The index of the first paint stage is 0.
See Also

TPaintStage
TPaintStages.Add

function Add: PPaintStage;
**Description**

Adds a new stage to the list and returns pointer to stage record. By default, new stages have their RunTime member set to *True*. 
See Also
Delete, Insert, TPaintStage
TPaintStages.Clear

procedure Clear;
Description
Removes all stages from the list.
TPaintStages.Count

    function Count: Integer;
Description
Returns the number of stored stages.
TPaintStages.Delete

procedure Delete(Index: Integer);
Description
Removes the stage at the position specified by the **Index** parameter. The index of subsequent stages automatically decreases.
See Also

Add, Insert
TPaintStages.Destroy

**destructor** Destroy; **override**;
**Description**

Destroys an instance of `TPaintStages` object and frees the memory allocated to store the list of stages.

Do not call **Destroy** directly, call the standard **Free** method instead.
See Also

TPaintStages
TPaintStages.Insert

function Insert(Index: Integer): PPaintStage;
Description

Insert a new stage into the list at specified position and returns a pointer to it. **Index** indicates the position, where the new stage is inserted. The item that occupied that position before insertion, is shifted up together with all subsequent items.

If Index is greater than the number of stages, the new stage will be added at the end of the list. Similarly, if Index is less or equal than zero, all previous elements are shifted up and the new stage is inserted in the beginning of the list.

By default, new stages have their RunTime member set to *True*. 
See Also
Add, Delete, TPaintStage

Copyright ©2000-2007 Alex Denisov and Contributors - Graphics32 v1.8.3 - Build on 4-March-2007
TPaintBoxOptions

```go
type TPaintBoxOptions = (pboWantArrowKeys, pboAutoFocus);
```
Description

TPaintBoxOptions is the type for the Options property in TCustomPaintBox32 and its descendants.
See Also

TCustomPaintBox32.Options, TCustomPaintBox32
TPaintStage

```pascal
type TPaintStage = record
  DsgnTime: Boolean;
  RunTime: Boolean;
  Stage: Cardinal;  // a PST_* constant
  Parameter: Cardinal;  // an optional parameter
end;

type PPaintStage = ^TPaintStage;
```
**Description**

Defines a drawing stage in TCustomImage32.

**DsgnTime** — indicates that the stage must be executed at design-time;

**RunTime** — indicates that the stage must be executed at run-time;

**Stage** — is one of the Paint Stage Constants, which specifies the stage action;

**Parameter** — is an optional parameter used in some paint stages.
See Also

Paint Stage Constants, Paint Stages, TCustomImage32.PaintStages, Repaint Optimization, TCustomImage32
TRepaintMode

    type TRepaintMode = (rmFull, rmDirect, rmOptimizer);
Description

TRepaintMode is the type for the RepaintMode property in TCustomPaintBox32 and its descendants.
See Also

TCustomPaintBox32.RepaintMode, TCustomPaintBox32
**TScaleMode**

```cpp
type TScaleMode = (smNormal, smStretch, smScale, smResize,
                  smOptimal, smOptimalScaled);
```
Description

TScaleMode is a type for the ScaleMode property in TCustomImage32.
See Also

TCustomImage32.ScaleMode, TCustomImage32
TScrollBarVisibility

```
type TScrollBarVisibility = (svAlways, svHidden, svAuto);
```
**Description**

TScrollBarVisibility is the type for the Visibility property in TIVScrollProperties.
See Also
TIVScrollProperties, TIVScrollProperties.Visibility
TSizeGripStyle

```go
type TSizeGripStyle = (sgAuto, sgNone, sgAlways);
```
Description

TSizeGripStyle is a type for the SizeGrip property in TCustomImgView, which determines appearance of a sizing grip at the bottom-right corner.
See Also
GR32_Image, TCustomImgView32.SizeGrip

Copyright ©2000-2007 Alex Denisov and Contributors - Graphics32 v1.8.3 - Build on 4-March-2007
Paint Stage Constants

Paint stage constants specify the type of operation associated with a paint stage in `TCustomImage32` and its descendants. There is seven types of paint stages, which can be used by `TCustomImage32` to paint its contents. Each paint stage has 'PST_' prefix:

<table>
<thead>
<tr>
<th>Stage Name</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PST_CUSTOM</td>
<td>1</td>
<td>Calls the OnPaintStage event</td>
</tr>
<tr>
<td>PST_CLEAR_BUFFER</td>
<td>2</td>
<td>Clears the buffer</td>
</tr>
<tr>
<td>PST_CLEAR_BACKGND</td>
<td>3</td>
<td>Clears visible buffer area</td>
</tr>
<tr>
<td>PST_DRAW_BITMAP</td>
<td>4</td>
<td>Draws a scaled bitmap image</td>
</tr>
<tr>
<td>PST_DRAW LAYERS</td>
<td>5</td>
<td>Draws layers</td>
</tr>
<tr>
<td>PST_CONTROL_FRAME</td>
<td>6</td>
<td>Draws a dotted frame around the control</td>
</tr>
<tr>
<td>PST_BITMAP_FRAME</td>
<td>7</td>
<td>Draws a dotted frame around the scaled bitmap image</td>
</tr>
</tbody>
</table>
See Also

Paint Stages, TCustomImage32
TBitmapLayer.AlphaHit

    property AlphaHit: Boolean;
Description

AlphaHit determines how the layer performs its HitTest.

If AlphaHit=False, layer only passes points inside the rectangle returned by GetAdjustedLocation.

If AlphaHit=True, layer checks the opacity of the pixel lying under the specified point and passes the hit test only when the alpha channel of the pixel is greater when zero.
See Also
TPositionedLayer.GetAdjustedLocation, TCustomLayer.HitTest

Copyright ©2000-2007 Alex Denisov and Contributors - Graphics32 v1.8.3 - Build on 4-March-2007
<table>
<thead>
<tr>
<th>property</th>
<th>Bitmap: TBitmap32;</th>
</tr>
</thead>
</table>

**TBitmapLayer.Bitmap**
Description
Specifies the contained image.
See Also

TBitmap32
TBitmapLayer.Cropped

property Cropped: Boolean;
**Description**

When **Cropped** is *True*, layer will paint only the portion of its bitmap, that intersects with the bitmap image of the container:
See Also

**Bitmap Image**
TBitmapLayer.Create

constructor Create(ALayerCollection: TLayerCollection);
override;
**Description**

Creates an instance of *TBitmapLayer*.

After calling the inherited constructor, **Create** initializes the internal bitmap and returns an object with both **AlphaHit** and **Cropped** properties set to *False*. 
See Also

AlphaHit, TCustomLayer.Create, Cropped, TBitmapLayer, TLayerCollection
TBitmapLayer.Destroy

**destructor** Destroy; **override**;
Description

Removes the bitmap layer from the collection and destroys it. The memory allocated for the bitmap is freed as well.

Do not call Destroy in your application directly, use the standard Free method instead.
TCustomLayer.Cursor

```delphi
property Cursor: TCursor;
```
Description

Specifies the appearance of the mouse pointer, when it passes over the region covered by the layer. Whether this property used or not is influenced by LOB_MOUSE_EVENTS bit in LayerOptions, as well as by the hit test.
See Also
LayerOptions, Using Layers
TCustomLayer.Index

property Index: Integer;
Description

Indicates the position of the layer in the LayerCollection to which it belongs. The background layer has an Index value of 0, the layer directly on top of the background layer has an Index value of 1, and so forth. Change Index to reorder the layers in a LayerCollection.
See Also
LayerCollection, Using Layers
TCustomLayer.LayerCollection

**property** LayerCollection: TLayerCollection;
**Description**

Points to the LayerCollection, to which the layer belongs.
See Also

TLayerCollection, Using Layers
TCustomLayer.LayerOptions

property LayerOptions: Cardinal;
Description

A 32-bit unsigned integer value composed of Layer Options Bits, which controls the layer's behavior.

Note that the lower 24-bits of LayerOptions have no predefined meaning; they are available for customization purposes, like layer categories, etc.
See Also

Layer Options Bits, Using Layers
property LayerStates: TLayerStates; // Read only

type TLayerState = (IsMouseLeft, IsMouseRight, IsMouseMiddle);

type TLayerStates = set of TLayerState;
**Description**

Indicates the current state of the layer. At the moment, this property contains only information on pressed mouse buttons, but it will be expanded in future versions.
See Also

Using Layers
TCustomLayer.MouseEvents

property MouseEvents: Boolean;
**Description**

Returns True if the LOB_MOUSE_EVENTS bit in LayerOptions is set; otherwise False is returned. Setting MouseEvents changes the LOB_MOUSE_EVENTS bit in LayerOptions accordingly.
See Also

Layer Options Bits, LayerOptions, Using Layers
TCustomLayer.Tag

property Tag: Integer;
Description

Tag has no predefined meaning, it can be used to store any 32-bit value as part of the layer.
See Also

Using Layers
TCustomLayer.Visible

*property* Visible: Boolean;
Description

Returns True if the LOB_VISIBLE bit in LayerOptions is set; otherwise False is returned. Setting Visible changes the LOB_VISIBLE bit in LayerOptions accordingly.
See Also

Layer Options Bits, LayerOptions, Using Layers
TCustomLayer.BringToFront

procedure BringToFront;
**Description**

Makes the layer the top-most layer of the layer collection it belongs to. After calling **BringToFront**, the layer's Index value is the highest among the layers in the same layer collection.
See Also

Index, LayerCollection, Using Layers
TCustomLayer.Changed

procedure Changed;
procedure Changed(const Rect: TRect);
**Description**

Will mark the layer for update and calls the `OnChange` event, provided the change notification is not masked with `BeginUpdate`. If a rectangle of the changed area (`Rect`) is set, only this area will be updated - provided the `RepaintMode` of the owning `TCustomImage32` instance is set to `rmOptimizer`. 
See Also

BeginUpdate, OnChange, TLayerCollection.Owner, TCustomPaintBox32.RepaintMode, TCustomImage32, TRect, TRepaintMode, Update, Using Layers
TCustomLayer.Create

constructor Create(ALayerCollection: TLayerCollection); virtual;
**Description**

Creates a new instance of `TCustomLayer` and adds it to the specified layer collection.

By default, new layers are created with only the LOB_VISIBLE bit set in their `LayerOptions`. 
See Also

LayerOptions, TCustomLayer, TLayerCollection, Using Layers
TCustomLayer.Destroy

destructor Destroy; override;
**Description**

Removes the layer from its layer collection, destroys the layer instance, and frees all allocated memory.

Do not call **Destroy** directly. Instead, use the **Free** method which verifies that the layer reference is assigned and only than calls **Destroy**.
See Also

LayerCollection, Using Layers
TCustomLayer.HitTest

function HitTest(X, Y: Integer): Boolean;
**Description**

Performs a hit test of the layer at specified point and returns *True* if the point have passed the test.

After performing default hit test, specific for particular type of a layer, this method also calls *OnHitTest* event, in case you need some run-time customization.

By default:

- *TCustomLayer* always passes the hit test;
- *TPositionedLayer* passes only the points lying inside the rectangle, returned by *GetAdjustedLocation*;
- Depending on the *AlphaHit* property, *TBitmapLayer* is capable of passing only coordinates corresponding to non-transparent pixels.
See Also

TBitmapLayer.AlphaHit, TPositionedLayer.GetAdjustedLocation, OnHitTest, TBitmapLayer, TCustomLayer, TPositionedLayer, Using Layers
procedure SendToBack;
**Description**

Makes the layer the background layer of the layer collection it belongs to. Calling **SendToBack** has the same effect as setting the layer's **Index** value to 0.
See Also

Index, LayerCollection, Using Layers
TCustomLayer.Update

procedure Update;
procedure Update(const Rect: TRect);
Description

Update will mark the layer for a repaint. If a rectangle of the changed area (Rect) is set, only this area will be repainted - provided the RepaintMode of the owning TCustomImage32 instance is set to rmOptimizer.
See Also

Changed, TLayerCollection.Owner, TCustomPaintBox32.RepaintMode, TCustomImage32, TRect, TRepaintMode, Using Layers
**TCustomLayer.OnHitTest**

```
property OnHitTest: THitTestEvent;

type THitTestEvent = procedure(Sender: TObject; X, Y: Integer;
var Passed: Boolean) of object;
```
**Description**

Provide a handler for this event to customize the mouse hit testing. The X and Y parameters indicate the mouse position relative to `TCustomImage32` (or its descendant) top-left corner.
See Also

TCustomImage32, Using Layers
property OnMouseDown: TMouseEvent;

type TMouseEvent = procedure (Sender: TObject; Button: TMouseButton; TMouseButtons; Shift: TShiftState; X, Y: Integer) of object;
Description

This event is obsolete and remains here only for backward compatibility. Use the `OnMouseDown` event of `TCustomImage32` instead.

[See also]: Using Layers, OnMouseMove, OnMouseUp
See Also

property OnMouseMove: TMouseMoveEvent;
type TMouseMoveEvent = procedure(Sender: TObject; Shift: TShiftState; X, Y: Integer) of object;
**Description**

This event is obsolete and remains here only for backward compatibility. Use the [OnMouseMove](#) event of [TCustomImage32](#) instead.
See Also

TCustomLayer.OnMouseUp

property OnMouseUp: TMouseEvent;

{property}

{type TMouseEvent = procedure (Sender: TObject; Button:
TMouseButton ; Shift: TShiftState; X, Y: Integer) of object;
Description

This event is obsolete and remains here only for backward compatibility. Use the OnMouseUp event of TCustomImage32 instead.
See Also

OnMouseMove, TCustomImage32.OnMouseUp, TCustomImage32, Using Layers
property OnPaint: TPaintLayerEvent;

type TPaintLayerEvent = procedure(Sender: TObject; Buffer: TBitmap32) of object;
Description
Provide a handler for this event to paint the layer. Use the \texttt{TBitmap32} instance indicated by the \texttt{Buffer} parameter to paint the layer.
See Also

TBitmap32, Using Layers

Copyright ©2000-2007 Alex Denisov and Contributors - Graphics32 v1.8.3 - Build on 4-March-2007
TLayerCollection.CoordXForm

<table>
<thead>
<tr>
<th>property CoordXForm: PCoordXForm;</th>
</tr>
</thead>
</table>
Description

This property is *deprecated* as of version 1.8 of Graphics32. Please use GetViewportScale and GetViewportShift instead.
See Also

GetViewportScale, GetViewportShift, LocalToViewport, Using Layers, ViewportToLocal
property Count: Integer; // Read only
**Description**

Indicates the amount of layers stored in the collection.
See Also

Items, Using Layers
Procedure: GetViewportScale

```delphi
procedure GetViewportScale(var ScaleX, ScaleY: Single);
```
Description

Returns the X and Y scale of the viewport. In the default implementation this equals the ScaleX and ScaleY property of the connected TCustomImage32 descendant.

Note: If you previously used the deprecated CoordXForm property, use GetViewportScale instead to retrieve the viewport scale.
See Also

CoordXForm, TCustomImage32.ScaleX, TCustomImage32.ScaleY,
TCustomImage32, Using Layers
TLayerCollection.GetViewportShift

procedure GetViewportShift(var ShiftX, ShiftY: Single);
Description
Returns the X and Y shift of the viewport.
Note: If you previously used the deprecated CoordXForm property, use GetViewportShift instead to retrieve the viewport shift.
See Also

CoordXForm, Using Layers

Copyright ©2000-2007 Alex Denisov and Contributors - Graphics32 v1.8.3 - Build on 4-March-2007
**TLayerCollection.Items**

**property** Items[Index: Integer]: TCustomLayer;
**Description**

Provides indexed access to contained layers. Index of layers is zero-based.
See Also

Count, TCustomLayer, Using Layers
function LocalToViewport(const APoint: TFloatPoint; AScaled: Boolean): TFloatPoint;
Description
Transforms a given point (APoint) from the local coordinate system to the viewport's coordinate system and returns the result. If AScaled is False the output is equal to the input point.
See Also

Using Layers
property MouseEvents: Boolean;
**Description**
Determines if mouse messages from the owner (TCustomImage32) do propagate to layers.
See Also

TCustomImage32, Using Layers
property MouseListener: TCustomLayer;
Description

Indicates the layer which has captured mouse messages. By default, the layer captures mouse if the left mouse button is pressed and releases it when the mouse button is released.

Note, that since layers have nothing to do with windowed controls, this property is just a simulation of standard mouse capturing. Windows still 'thinks' that the mouse is captured to the owner of the layer collection (TCustomImage32), which in turn redirects mouse messages to an appropriate layer.
See Also

**TCustomImage32, TCustomLayer, Using Layers**
TLayerCollection.Owner

property Owner: TPersistent;
Description
Indicates the owner of the layer collection. Usually the owner is the displaying control like TCustomImage32 or TCustomImgView32.
See Also

TCustomImage32, TCustomImgView32, Using Layers
function ViewportToLocal(const APoint: TFloatPoint; AScaled: Boolean): TFloatPoint;
**Description**

Transforms a given point (**APoint**) from the viewport's coordinate system to the local coordinate system and returns the result. If **AScaled** is *False* the output is equal to the input point.
**TLayerCollection.Add**

```delphi
function Add(ItemClass: TLayerClass): TCustomLayer;

type TLayerClass = class of TCustomLayer;
```
**Description**

Adds a new layer to the collection. The layer is added to the end of the list and it's `Index` property will become equal to `Count` - 1.

The class of the layer is specified in the `ItemClass` parameter and you will need to typecast the result to `ItemClass` if you need to access its specific properties, for example:

```pascal
var
  BL: TBitmapLayer;
beg
  BL := TBitmapLayer(MyImage32.Layers.Add(TBitmapLayer));
  ...
end;
```

Alternatively, you can use another approach to add layers to the collection:

```pascal
var
  BL: TBitmapLayer;
beg
  BL := TBitmapLayer.Create(MyImage32.Layers);
end;
```
See Also

Count, TCustomLayer.Index, TCustomLayer, Using Layers
TLayerCollection.Clear

procedure Clear;
Description
Removes and destroys all the layers from the collection.
See Also

Using Layers
TLayerCollection.Create

**constructor** Create(AOwner: TPersistent);
Description

Creates a new instance of `TLayerCollection`. This constructor is used internally in constructor of `TCustomImage32` to initialize its `Layers` property.
See Also

TCustomImage32.Layers, TCustomImage32, TLayerCollection, Using Layers

Copyright ©2000-2007 Alex Denisov and Contributors - Graphics32 v1.8.3 - Build on 4-March-2007
procedure Delete(Index: Integer);
**Description**

Removes and destroy the layer with specified index from the collection.
See Also

Using Layers
TLayerCollection.Destroy

destructor Destroy; override;
Description

Destroys an instance of TLayerCollection together with all contained layers. This destructor is used in the destructor of TCustomImage32 to property deallocate its Layers property.
See Also

TCustomImage32.Layers, TCustomImage32, TLayerCollection, Using Layers

Copyright ©2000-2007 Alex Denisov and Contributors - Graphics32 v1.8.3 - Build on 4-March-2007
function Insert(Index: Integer; ItemClass: TLayerClass): TCustomLayer;

type TLayerClass = class of TCustomLayer;
**Description**
Inserts a new layer at the specified position. Insert works similar to the Add method.
See Also
Add, TCustomLayer, Using Layers
$\text{TPositionedLayer.Location}$

| property Location: $\text{FloatRect}$; |  |
**Description**

Determines layer's position and size.

Note, that the layer is not constrained to lie entirely within the rectangle specified by *Location*. It may perform drawing outside the *Location* and may catch mouse messages from other areas.

When the layer is *Scaled*, its actual location its position after scaling may be obtained through *GetAdjustedLocation* method.
See Also

GetAdjustedLocation, Scaled, TFloatRect, Using Layers
TPositionedLayer.Scaled

property Scaled: Boolean;
Description

Specifies if the layer is scaled and determines the result returned by `GetAdjustedLocation`. When set to `True`, scaling is influenced by the layout of bitmap image in the container (`TCustomImage32` object or its descendant).
See Also

Bitmap Image, GetAdjustedLocation, TCustomImage32, Using Layers
TPositionedLayer.Create

**constructor** Create(ALayerCollection: TLayerCollection);
override;
Description

Creates the instance of TPositionedLayer.

After calling the inherited constructor, Create initializes the following properties:

- LayerOptions to LOB_VISIBLE or LOB_MOUSE_EVENTS;
- Scaled to False;
- Location to FloatRect(0, 0, 64, 64);
See Also

TCustomLayer.Create, TCustomLayer.LayerOptions, Location, Scaled, TLayerCollection, TPositionedLayer, Using Layers
TPositionedLayer.GetAdjustedLocation

function GetAdjustedLocation: TFloatRect;
Description
When the Scaled property is True, GetAdjustedLocation returns scaled Location, otherwise it returns Location as it is, without scaling.

It is up to you do decide if you want to use the result as a floating-point rectangle, to round the coordinates before painting:

```pascal
procedure TForm1.OnLayerPaintHandler(Sender: TObject; Buffer: TBitmap32);
var
  R: TRect;
begin
  if Sender is TPositionedLayer then
    with TPositionedLayer(Sender) do
    begin
      R := MakeRect(GetAdjustedLocation);
      Buffer.FrameRectS(R.Left, R.Top, R.Right, R.Bottom, clRed32);
    end;
end;
```
See Also

GetAdjustedRect, Location, Scaled, TFloatRect, Using Layers
TPositionedLayer.GetAdjustedRect

```pascal
function GetAdjustedRect(const R: TFloatRect): TFloatRect;
```
**Description**

When the `Scaled` property is `True`, `GetAdjustedRect` returns the scaled version of the input rect (`R`), otherwise it returns `R` as it is, without scaling.
See Also

GetAdjustedLocation, Scaled, TFloatRect, Using Layers
TRubberbandLayer.ChildLayer

property ChildLayer: TPositionedLayer;
**Description**

Points to the position layer, controlled by the rubber band. TRubberbandLayer will keep Location of its child layer the same as its own Location.
See Also

TPositionedLayer.Location, TPositionedLayer, TRubberbandLayer
TRubberbandLayer.FrameStippleCounter

property FrameStippleCounter: Single;
**Description**

Specifies the current position of the stipple pattern of the surrounding frame.

Hint: By using a timer and modifying the FrameStippleCounter, it is possible to animate the frame of the rubberband.
See Also

SetFrameStipple
property FrameStippleStep: Single;
**Description**

Specifies the stepping for the stipple pattern of the surrounding frame.
See Also

SetFrameStipple
TRubberbandLayer.HandleFill

property HandleFill: TColor32;
Description

Specifies the color of the drag handle.
TRubberbandLayer.HandleFrame

property HandleFrame: TColor32;
Description
Specifies the frame color of the drag handle.
**TRubberbandLayer.Handles**

**property** Handles: TRBHandles;

**type** TRBHandles = set of (rhCenter, rhSides, rhCorners, rhFrame, rhNotLeftSide, rhNotRightSide, rhNotTopSide, rhNotBottomSide, rhNotTLCorner, rhNotTRCorner, rhNotBLCorner, rhNotBRCorner);
Description
Determines which handles are visible and draggable by the mouse.

- rhCenter – corresponds to the internal area of the control, that is the area inside the rectangle specified by Location;
- rhSides – corresponds to the handles on the edges of the layer;
- rhCorners – corresponds to the handles at the corners of the layer;
- rhFrame - corresponds to the frame surrounding the layer;
- rhNotLeftSide, rhNotRightSide, rhNotTopSide, rhNotBottomSide - set to disable one or more sides (needs rhSides);
- rhNotTLCorner, rhNotTRCorner, rhNotBLCorner, rhNotBRCorner - set to disable one or more corners (needs rhCorners);
See Also

TPositionedLayer.Location, TRBHandles
TRubberbandLayer.HandleSize

property HandleSize: Integer;
**Description**

Determines the size of the handles. The actual size in pixels may be calculated as

\[ \text{HandleSize} \times 2 + 1 \]
TRubberbandLayer.MaxHeight

**property** MaxHeight: Single;
**Description**

Determines the maximum height of the rubber band while it resizes. When **MaxHeight** is less than or equal to **MinHeight**, this property is disregarded and the maximum height becomes unlimited.
See Also

MaxWidth, MinHeight
TRubberbandLayer.MaxWidth

property MaxWidth: Single;
Description
Determines the maximum width of the rubber band while it resizes. When MaxWidth is less than or equal to MinWidth, this property is disregarded and the maximum width becomes unlimited.
See Also

MinWidth
TRubberbandLayer.MinHeight

**property** MinHeight: Single;
Description
Determines the minimum height of the rubber band while it resizes.
See Also

MinWidth
TRubberbandLayer.MinWidth

property MinWidth: Single;


**Description**

Determines the minimum width of the rubber band while it resizes.
See Also

MinHeight
TRubberbandLayer.Create

**constructor** Create(ALayerCollection: TLayerCollection);
**override**;
Description
Creates an instance of TRubberbandLayer and adds it into the specified collection.
See Also

TCustomLayer.Create, TLayerCollection, TRubberbandLayer

Copyright ©2000-2007 Alex Denisov and Contributors - Graphics32 v1.8.3 - Build on 4-March-2007
TRubberbandLayer.Destroy

defstructor Destroy; override;
**Description**
Removes the rubber band layer from the collection and destroys it.
Do not call **Destroy** directly in your application, use the standard **Free** method instead.
TRubberbandLayer.SetFrameStipple

procedure SetFrameStipple(const Value: Array of TColor32);
Description
Sets the stipple pattern for the frame in the rubberband.
See Also

Color Types, FrameStippleStep
property OnResizing: TRBResizingEvent;

type TRBResizingEvent = procedure(
  Sender: TObject;  const OldLocation: TFloatRect;
  var NewLocation: TFloatRect;
  DragState: TDragState;
  Shift: TShiftState) of object;

type TDragState = (dsNone, dsMove, dsSizeL, dsSizeT,
  dsSizeR, dsSizeB, dsSizeTL, dsSizeTR, dsSizeBL, dsSizeBR);
**Description**

**OnResizing** allows for runtime customization of the rubber band which is being moved or resized using the mouse. This event is called only when the user clicks and drags the layer with the mouse, it is not generated when you set the **Location** property explicitly.

**OldLocation** specifies location of the layer before the user started dragging it.

**NewLocation** determines the new location of the rubber band. You may change the coordinates in order to provide required response to dragging or resizing operation.

The **DragState** parameter indicates how the layer is being dragged/resized. **Shift** indicates the state of the Alt, Ctrl, and Shift keys.
See Also

TPositionedLayer.Location
TRubberbandLayer.OnUserChange

```
property OnUserChange: TNotifyEvent;
```
Description

The event is called automatically when the layer is moved or resized with the mouse. **OnUserChange** is not called when you change **Location** explicitly.
See Also

TPositionedLayer.Location
TRBHandles

\[
\text{type TRBHandles} = \text{set of} \ (\text{rhCenter, rhSides, rhCorners, rhFrame, rhNotLeftSide, rhNotRightSide, rhNotTopSide, rhNotBottomSide, rhNotTLCorner, rhNotTRCorner, rhNotBLCorner, rhNotBRCorner});
\]
Description

A type for Handles property in TRubberbandLayer.
See Also

TRubberbandLayer::Handles, TRubberbandLayer

Copyright ©2000-2007 Alex Denisov and Contributors - Graphics32 v1.8.3 - Build on 4-March-2007
**Layer Options Bits**

The basic behavior of the layer is controlled by its `LayerOptions` property, an unsigned 32-bit integer field of bits. The upper 8 bits in this field are reserved for default options:

<table>
<thead>
<tr>
<th>Bit #</th>
<th>Constant Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>LOB_VISIBLE</td>
<td>Controls the layer visibility</td>
</tr>
<tr>
<td>30</td>
<td>LOB_GDI_OVERLAY</td>
<td>Indicates that the layer performs drawing when its owner (TCustomImage32 control or its descendant) draws its GDI Overlays. [ Note: Most likely, this constant will not be used in future versions. Do not use it ]</td>
</tr>
<tr>
<td>29</td>
<td>LOB_MOUSE_EVENTS</td>
<td>Specifies whether the layer responds to mouse messages.</td>
</tr>
<tr>
<td>28</td>
<td>LOB_NO_UPDATE</td>
<td>Disables automatic repainting when the layer changes its location or other properties.</td>
</tr>
<tr>
<td>27</td>
<td>LOB_NO_CAPTURE</td>
<td>Allows to override automatic capturing of mouse messages when the left mouse is pressed on top of the layer. This bit has no effect if LOB_MOUSE_EVENTS is not set.</td>
</tr>
<tr>
<td>26</td>
<td>LOB_RESERVED_26</td>
<td>Reserved for future use</td>
</tr>
<tr>
<td>25</td>
<td>LOB_RESERVED_25</td>
<td>Reserved for future use</td>
</tr>
<tr>
<td>24</td>
<td>LOB_RESERVED_24</td>
<td>Reserved for future use</td>
</tr>
</tbody>
</table>

Other 24 bits can be used for any other purpose, as customized layer classification etc.
See Also

TCustomLayer.LayerOptions
Clamp

| function  | Clamp(const Value: Integer): TColor32; overload; |
| function  | Clamp(Value: Integer; Max: Integer): Integer; overload; |
| function  | Clamp(Value: Integer; Min: Integer; Max: Integer): Integer; overload; |
**Description**

First version restricts `Value` to [0..255] range.
Second version restricts `Value` to [0..Max] range.
Third version restricts `Value` to [Min..Max] range.
function ColorSwap(WinColor: TColor): TColor32;
Description

ColorSwap exchanges ARGB <-> ABGR and fills A with $FF$
Constrain

function Constrain(const Value: Integer; const Lo: Integer; const Hi: Integer): Integer;
**Description**

Returns Value constrained to [Lo..Hi] range.
See Also

Clamp
procedure FillLongword(var X; Count: Integer; Value: Longword);
**Description**

An optimized analogue of FillChar for 32 bit values.
See Also

FillWord
procedure FillWord(var X; Count: Integer; Value: Longword);
Description
An optimized analogue of FillChar for 16 bit values. Note that only the lower word of the Value parameter is used.
See Also

FillLongword
function Mirror(Value: Integer; Max: Integer): Integer; overload;
function Mirror(Value: Integer; Min: Integer; Max: Integer): Integer; overload;
**Description**

First version mirrors integer *Value* in \([0..Max]\) range. E.g. if *Value* is 10 and *Max* is 9, result will be 8.

Second version mirrors integer *Value* in \([Min..Max]\) range.
See Also

Clamp, Wrap
MoveLongword

```procedure MoveLongword(const Source; var Dest; Count: Integer);```
Description
An analogue of Move for 32 bit values.
See Also

MoveWord
procedure MoveWord(const Source; var Dest; Count: Integer);
**Description**

An analogue of *Move* for 16 bit values.
See Also

MoveLongword

Copyright ©2000-2007 Alex Denisov and Contributors - Graphics32 v1.8.3 - Build on 4-March-2007
Shift Arithmetic Right

function SAR_4(Value: Integer): Integer; // Value div 16
function SAR_8(Value: Integer): Integer; // Value div 256
function SAR_9(Value: Integer): Integer; // Value div 512
function SAR_11(Value: Integer): Integer;
function SAR_12(Value: Integer): Integer;
function SAR_13(Value: Integer): Integer;
function SAR_14(Value: Integer): Integer;
function SAR_15(Value: Integer): Integer;
function SAR_16(Value: Integer): Integer;
**Description**

Auxiliary arithmetic shift right routines (SAR). These can be used for faster division of signed integers. SAR differs from the `shr` instruction by preserving the sign of the `Value`. For example, calling SAR_4 equals `Value div 16 (2^4)` and calling SAR_9 equals `Value div 512 (2^9)`.
procedure Swap(var A: Integer; var B: Integer);
Description
Exchange two 32-bit values.
See Also

FillWord
SwapConstrain

function SwapConstrain(const Value: Integer; Constrain1: Integer; Constrain2: Integer): Integer;
**Description**

Returns *Value* constrained to [Min(Constrain1, Constrain2)..Max(Constrain1, Constrain2)] range.
See Also

Clamp, Constrain, Swap
TestClip

function TestClip(var A: Integer; var B: Integer; const Size: Integer): Boolean; overload;

function TestClip(var A: Integer; var B: Integer; const Start: Integer; const Stop: Integer): Boolean; overload;
**Description**

First version: Exchange $A <-> B$ only if $B < A$, then restrict both to $[0..Size-1]$ range. Returns true if resulting range has common points with $[0..Size-1]$ range.

Second version: Exchange $A <-> B$ only if $B < A$, then restrict both to $[Start..Stop-1]$ range. Returns true if resulting range has common points with $[Start..Stop-1]$ range.
See Also

Clamp, Constrain
procedure TestSwap(var A: Integer; var B: Integer);
Description
Exchange A <-> B only if B < A.
function Wrap(Value: Integer; Max: Integer): Integer; overload;
function Wrap(Value: Integer; Min: Integer; Max: Integer): Integer; overload;
Description

First version wraps integer Value in [0..Max] range. E.g. if Value is 10 and Max is 9, result will be 0.

Second version wraps integer Value in [Min..Max] range.
See Also

Clamp, Wrap
Wrap Procedure Types

type TWrapProc = function (Value, Max: Integer): Integer;
TWrapProcEx = function (Value, Min, Max: Integer): Integer;
Description

These function types are used in relation to wrapmodes. While \texttt{TWrapProc} implicitly uses zero as minimum value, \texttt{TWrapProcEx} lets the caller define the complete minimum to maximum range.
Fixed Ceil

**FixedCeil**

```cpp
function FixedCeil(A: TFixed): Integer;
```

An implementation of the *Ceil* function for `TFixed` type.
See Also

Fixed Floor, Fixed Point Math, Fixed Round, TFixed
Fixed Division

**FixedDiv**

```cpp
function FixedDiv(A, B, TFixed): TFixed;
```

This is the same as using `MulDiv(A, $10000, B)`.
See Also

Fixed Multiply, Fixed Point Math, TFixed
Fixed Floor

function FixedFloor(A: TFixed): Integer;

An analogue to *Floor* function.
See Also

Fixed Ceil, Fixed Point Math, Fixed Round, TFixed
Fixed Multiply

FixedMul

function FixedMul(A, B, TFixed): TFixed;

This is the same as using MulDiv(A, B, $10000).
See Also

Fixed Division, Fixed Point Math, TFixed
Fixed Round

**Function** FixedRound

```
function FixedRound(A, B, TFixed): TFixed;
```

Provides correct rounding of fixed-point values.
See Also

Fixed Point Math, TFixed
Fixed Square

**FixedSqr**

```pascal
function FixedSqr(Value: TFixed): TFixed;
```

This is the same as using *FixedMul*(Value, Value).
See Also

Fixed Point Math, Fixed Square Root, TFixed
Fixed Square Root

FixedSqrtLP

function FixedSqrtLP(Value: TFixed): TFixed;

Low precision (8 bit) fixed math squareroot.
FixedSqrtHP

function FixedSqrtHP(Value: TFixed): TFixed;

High precision (16 bit) fixed math square root. This routine is slower than the low precision version.
See Also

Fixed Point Math, Fixed Square, TFixed
SinCos

SinCos

procedure SinCos(const Theta: Single; var Sin: Single; var Cos: Single); overload;

procedure SinCos(const Theta: Single; const Radius: Single; var Sin: Single; var Cos: Single); overload;

Single precision versions of *SinCos*.
property Bits: PByteArray;
**Description**

Returns the pointer to the internal array of *byte packed* bits. Data is stored in row-major order, top-left "pixel" comes first. Note that since the boolean map stores its values packed in bytes (8 bits or booleans per byte), the last element of the array may contain invalid values. It is recommended only to process width * height - 1 elements, and use the value property to change the last packed byte element of the array.
See Also

TByteArray, Value
TBooleanMap.Value

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

Provides coordinate-based access to stored booleans. This function does not perform any range checking of its arguments. Be sure, that the boolean map is not empty and that both X and Y are in a valid range.
TByteMap.Bits

property Bits: PWordArray;
**Description**

Returns the pointer to the internal array of words. Data is stored in row-major order, top-left "pixel" comes first.
See Also

TWordArray, ValPtr, Value
TByteMap.ValPtr

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

Returns a pointer to the specific byte in the array.

Copyright ©2000-2007 Alex Denisov and Contributors - Graphics32 v1.8.3 - Build on 4-March-2007
TByteMap.Value

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

Provides coordinate-based access to stored bytes. This function does not perform any range checking of its arguments. Be sure, that the byte map is not empty and that both $X$ and $Y$ are in a valid range.
procedure TByteMap.Assign(Source: TPersistent); override;

**Description**

Copies data from another object, specified by the **Source** parameter.

This method supports following classes:

- *nil* – assigns an empty data array;
- **TByteMap** – copies data bytes from another byte map;
- **TBitmap32** – transforms bitmap data using the **ReadFrom** method with `ctWeightedRGB` conversion type.

**TByteMap** also overrides the protected **AssignTo** method, making it possible to assign bitmaps from byte maps, so that both following lines are correct:

```pascal
ByteMap.Assign(Bitmap32); // an analog to BM := Intensity(B32);
Bitmap32.Assign(ByteMap); // an analog to B32 := Gray32(BM);
```
See Also

ReadFrom, TByteMap
TByteMap.Clear

procedure Clear(FillValue: Byte);
Description

Fills the entire byte map with the specified value.
TByteMap.Destroy

```
  destructor Destroy; override;
```
**Description**

Destroys the bytemap object and frees all the associated memory. Do not call **Destroy** directly, use the **Free** method instead.
TByteMap.Empty

function Empty: Boolean; override;
Description

Returns True if the byte map contains no data, that is Width or Height is equal to 0.
See Also

TCustomMap.Height, TCustomMap.Width
TByteMap.ReadFrom

procedure ReadFrom(Source: TBitmap32; Conversion: TConversionType);
**Description**

**ReadFrom** allows reading and converting data from **TBitmap32** objects into the byte map.

First, the byte map is resized to fit the **Source** bitmap dimensions, then the 32-bit RGBA color is transformed to 8-bits depending on the conversion type:

<table>
<thead>
<tr>
<th>Conversion</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>ctRed</td>
<td>Copies red channel</td>
</tr>
<tr>
<td>ctGreen</td>
<td>Copies green channel</td>
</tr>
<tr>
<td>ctBlue</td>
<td>Copies blue channel</td>
</tr>
<tr>
<td>ctAlpha</td>
<td>Copies alpha channel</td>
</tr>
<tr>
<td>ctUniformRGB</td>
<td>Copies averaged value: ((R + G + B) / 3)</td>
</tr>
<tr>
<td>ctWeightedRGB</td>
<td>Copies intensity: (R \times 0.21 + G \times 0.71 + B \times 0.08)</td>
</tr>
</tbody>
</table>
See Also

TBitmap32, TConversionType
TByteMap.SetSize

function SetSize(NewWidth, NewHeight: Integer): Boolean;
override;

procedure SetSize(Source: TPersistent); // implemented in TCustomMap
Description

Call **SetSize** to set new dimensions of the bytemap. If one of the arguments is zero, the bytemap is considered empty and its **Bytes** property is set to **nil**.

Calling **SetSize** works faster than changing both **Width** and **Height** properties.

If you use another bitmap, byte map or control as an argument, the byte map will be sized to corresponding dimensions.
See Also

Bytes, TCustomMap.Height, TCustomMap, TCustomMap.Width
TByteMap.WriteTo

procedure WriteTo(Dest: TBitmap32; Conversion: TConversionType); overload;
procedure WriteTo(Dest: TBitmap32; const Palette: TPalette32); overload;
**Description**

WriteTo fills the Dest bitmap using the values stored in the byte map. If it is necessary, the destination bitmap is resized to fit the byte map dimensions.

The following table shows how 8-bit data is transformed into 32-bit RGBA color depending on Conversion parameter.

<table>
<thead>
<tr>
<th>Conversion</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>ctRed</td>
<td>Copies bytes into red channel</td>
</tr>
<tr>
<td>ctGreen</td>
<td>Copies bytes into green channel</td>
</tr>
<tr>
<td>ctBlue</td>
<td>Copies bytes into blue channel</td>
</tr>
<tr>
<td>ctAlpha</td>
<td>Copies bytes into alpha channel</td>
</tr>
<tr>
<td>ctUniformRGB</td>
<td>Copies the same byte value into red, green and blue channels.</td>
</tr>
<tr>
<td>ctWeightedRGB</td>
<td></td>
</tr>
</tbody>
</table>

WriteTo method does not distinguish between ctUniformRGB and ctWeightedRGB conversion types.

Note, that this method alters only the specified color channels, other channels remain intact.

The second overloaded version with Palette parameter uses TPalette32 array to map byte values into RGBA colors.
See Also

TBitmap32, TConversionType, TPalette32
TIntegerMap.Bits

property Bits: PIntegerArray;
**Description**

Returns the pointer to the internal array of integers. Data is stored in row-major order, top-left "pixel" comes first.
See Also

TIntegerArray, ValPtr, Value
TIntegerMap.ValPtr

property ValPtr[X, Y: Integer]: PInteger;
**Description**

Returns a pointer to the specific integer in the array.
TIntegerMap.Value

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

Provides coordinate-based access to stored integers. This function does not perform any range checking of its arguments. Be sure, that the map is not empty and that both \( X \) and \( Y \) are in a valid range.
TWordMap.Bits

property Bits: PByteArray;
**Description**

Returns the pointer to the internal array of bytes. Data is stored in row-major order, top-left "pixel" comes first.
See Also

TByteArray, ValPtr, Value
property ValPtr[X, Y: Integer]: PWord;
Description

Returns a pointer to the specific word in the array.

Copyright ©2000-2007 Alex Denisov and Contributors - Graphics32 v1.8.3 - Build on 4-March-2007
TWordMap.Value

property Value[X, Y: Integer]: Word; default;
Description

Provides coordinate-based access to stored words. This function does not perform any range checking of its arguments. Be sure, that the map is not empty and that both X and Y are in a valid range.
TConversionType

```delphi
type TConversionType = (
  ctRed,
  ctGreen,
  ctBlue,
  ctAlpha,
  ctUniformRGB,
  ctWeightedRGB);
```
Description
Specifies how data is converted from 32-bit ARGB bitmap to 8-bit bytemap and back.
See Also

TByteMap.ReadFrom, TByteMap.WriteTo
property OffsetX: Integer;
Description
Determines how much in pixels the tiling should be shifted in horizontal direction.
This is especially useful if your polygon is moved and you want to move the pattern tiling relatively to it.
TBitmapPolygonFiller.OffseyJ

property OffsetY: Integer;
**Description**

Determines how much in pixels the tiling should be shifted in vertical direction.

This is especially useful if your polygon is moved and you want to move the pattern tiling relatively to it.
TBitmapPolygonFiller.Pattern

| property Pattern: TBitmap32; |
**Description**

This property determines the pattern that is used to fill a polygon. The pattern will be tiled across the polygons.

Please note that the draw transformation won't have any effect on the polygon. For that purpose please preprocess your pattern accordingly.
See Also

TBitmap32
function GetFillLine: TFillLineEvent; abstract;
Description
Descendants should override this abstract method. If overridden it should return a TFillLineEvent callback that is used to fill lines in a polygon.
See Also

TFillLineEvent
TPolygon32.Antialiased

**property** Antialiased: Boolean;
Description

Determines the quality of the shape when it is drawn with Draw, DrawFill or DrawEdge methods.

When Antialiased = False, the object uses PolyPolylineTS and PolyPolygonTS functions, otherwise, it uses PolyPolylineXS and PolyPolygonXS functions.
See Also

Draw, DrawEdge, DrawFill, PolyPolyline
TPolygon32.AntialiasMode

| property  | AntialiasMode: | TAntialiasMode; |
**Description**

Determines how the shape is antialiased. See [TAntialiasMode](#) for details.
See Also

TAntialiasMode
TPolygon32.Closed

property Closed: Boolean;
**Description**

Specifies whether each contour defined by [Points](#) is a closed shape, or if it has to be considered as polyline.

This property is ignored by [Draw](#) and [DrawFill](#) methods, but it is important when using [Outline](#) method.
See Also

Draw, DrawFill, Outline, Points
TPolygon32.FillMode

property FillMode: TPolyFillMode;
Description
Determines the shape's fill mode.
See Also

TPolyFillMode
TPolygon32.Normals

**property** Normals: TArrayOfArrayOfFixedPoint;
Description
This array stores unit vectors showing the direction normal to each line in each contour. **Normals** are used in **Grow**.
See Also

Grow, Points, TArrayOfArrayOfFixedPoint
TPolygon32.Points

property Points: TArrayOfArrayOfFixedPoint;
**Description**

This property defines an array of contours. Each contour is, in turn, an array of vertexes.
See Also

Normals, TArrayOfArrayOfFixedPoint

Copyright ©2000-2007 Alex Denisov and Contributors - Graphics32 v1.8.3 - Build on 4-March-2007
TPolygon32.Add

procedure Add(const P: TFixedPoint);
**Description**

Adds a new vertex to the current contour in the polygon. In order to start a new contour, use the `NewLine` method.
See Also

NewLine, TFixedPoint
TPolygon32.AddPoints

procedure AddPoints(const First: TFixedPoint; Count: Integer);
Description
Copies Count number of points beginning at point First to the current outline.
See Also

TFixedPoint
TPolygon32.Assign

procedure Assign(Source: TPersistent); virtual;
Description

If **Source** is also of type **TPolygon32**, Assign copies all properties, points and normals over to the current instance. The old values are replaced in this process.
See Also

TPolygon32
TPolygon32.Clear

procedure Clear;
**Description**

Clears the information on all contours, associated with the polygon.
TPolygon32.ContainsPoint

function ContainsPoint\(\text{const } P: \text{TFixedPoint}\): \text{Boolean};
**Description**

This method returns true if a point \( \mathbf{P} \) is contained within any of the contours in the current TPolygon32 instance and false otherwise.
See Also

Point Types, PtInPolygon, TFixedPoint
TPolygon32.Create

constructor Create; override;
Description
Creates and initializes an instance of TPolygon32.
After calling the inherited constructor, Create initializes the following properties:

- Closed to True;
- Antialiased to False;
- FillMode to pfAlternate;
See Also

Antialiased, Closed, FillMode, TPolygon32
TPolygon32.Destoy

**destructor** Destroy; **override**;
**Description**

Destroys the polygon object and frees all associated memory. Do not call **Destroy** directly, use **Free** method instead.
TPolygon32.Draw

```pascal
procedure Draw(Bitmap: TBitmap32; OutlineColor, FillColor: TColor32; Transformation: TTransformation = nil); overload;

procedure Draw(Bitmap: TBitmap32; OutlineColor: TColor32;
FillLineCallback: TFillLineEvent; Transformation: TTransformation = nil); overload;

procedure Draw(Bitmap: TBitmap32; OutlineColor: TColor32;
Filler: TCustomPolygonFiller; Transformation: TTransformation = nil); overload;
```
Description
Fills the polygon and draws the outline transformed by Transformation. This method is a simple combination of DrawFill and DrawEdge methods.
See Also

DrawEdge, DrawFill, TBitmap32, TColor32, TCustomPolygonFiller, TFillLineEvent, TTransformation
procedure DrawEdge(Bitmap: TBitmap32; Color: TColor32; Transformation: TTransformation = nil);
Description
Frames the edge of the polygon with specified color and uses the Transformation while drawing.
The quality of the line is determined by Antialiased property.
When Closed property is True, each polygon is automatically closed.
See Also

Antialiased, Closed, Draw, DrawFill, TBitmap32, TColor32, TTransformation
procedure DrawFill(Bitmap: TBitmap32; Color: TColor32; Transformation: TTransformation = nil); overload;

procedure DrawFill(Bitmap: TBitmap32; FillLineCallback: TFillLineEvent; Transformation: TTransformation = nil); overload;

procedure DrawFill(Bitmap: TBitmap32; Filler: TCustomPolygonFiller; Transformation: TTransformation = nil); overload;
Description
Fills the polygon shape using one of the following options using the current FillMode:

- simple color
- custom fillline callback
- custom filler object

Optionally a Transformation is applied while drawing.
The Closed property is ignored and all contours are automatically considered closed.
The edge quality of the filling is determined by the Antialiased property.
See Also

Antialiased, Closed, Draw, DrawEdge, FillMode, TBitmap32, TColor32, TCustomPolygonFiller, TFillLineEvent, TTransformation
TPolygon32.GetBoundingRect

function GetBoundingRect: TFixedRect;
**Description**

Returns the common bounding rectangle for all outlines in the current polygon.
See Also

Rectangle Types
TPolygon32.Grow

```pascal
function Grow(const Delta: TFixed; EdgeSharpness: Single = 0): TPolygon32;
```
**Description**

Creates a new instance of TPolygon32 object and changes its shape similar to dilation (or erosion when Delta is negative) morphological filters.

The direction of growth is determined by the order of the points in the contour. Positive direction is to the left of each line (see image below).

Using **Grow** method with Delta = 0 will simply create a new instance of the polygon with the same data as in original polygon.

The **EdgeSharpness** parameter controls generation of new vertices. Its value range is [0…0.99].

![Diagram of polygon changes with EdgeSharpness](image)
See Also

TFixed, TPolygon32
TPolygon32.NewLine

procedure NewLine;
Description

Starts a new contour. New points, introduced with Add function, will be inserted into the new contour.
See Also

Add

Copyright ©2000-2007 Alex Denisov and Contributors  -  Graphics32 v1.8.3  -  Build on 4-March-2007
TPolygon32.Offset

procedure Offset(const Dx, Dy: TFixed);
**Description**

Shifts all the points in all contours \( D_x \) pixels right and \( D_y \) pixels down. Values of \( D_x \) and \( D_y \) can be negative.
See Also

TFixed
TPolygon32.Outline

function Outline: TPolygon32;
Description
This functions takes each contour (polyline) and constructs a new TPolygon32 object by outlining each contour.
The operation is similar to using morphological edge detection filter for bitmaps, but it operates over polygon vertices.
When outlining a closed polygon, this function actually returns two contours for each initial closed contour. In case of polylines (Closed property is False), it returns a single contour (see image below).
Note, this technique produces good results only for antialiased polygons.
This function is essential for thick line drawing, all you have to do, is to take an original polygon, build its outline and grow it to get the desired thickness:

```
var P, Outline: TPolygon32;
begin
  Outline := P.Outline; // create an outline
  Outline.Grow(Fixed(1), 1); // make it 2-pixel wide outline (it grows in both directions)
  Outline.DrawFill(DstBitmap, clBlack32);
end;
```

![Diagram showing the process of outlining and growing a polygon](image.png)
See Also

Closed, TPolygon32
TPolygon32.Transform

procedure Transform(Transformation: TTransformation);
Description
Transforms all points in the current instance using the supplied Transformation instance. This is different from the transformation while drawing because it is a permanent operation.
See Also

Transformation
<table>
<thead>
<tr>
<th>Procedure</th>
<th>Bitmap Type</th>
<th>Points Type</th>
<th>Color Type</th>
<th>Mode Type</th>
<th>Transformation Type</th>
<th>Overload</th>
</tr>
</thead>
<tbody>
<tr>
<td>PolygonTS</td>
<td>TBitmap32</td>
<td>TArrayOfFixedPoint</td>
<td>TColor32</td>
<td>TPolyFillMode = pfAlternate</td>
<td>TTransformation = nil</td>
<td>overload</td>
</tr>
<tr>
<td>PolygonTS</td>
<td>TBitmap32</td>
<td>TArrayOfFixedPoint</td>
<td>TFillLineEvent</td>
<td>TPolyFillMode = pfAlternate</td>
<td>TTransformation = nil</td>
<td>overload</td>
</tr>
<tr>
<td>PolygonTS</td>
<td>TBitmap32</td>
<td>TArrayOfFixedPoint</td>
<td>TCustomPolygonFiller</td>
<td>TPolyFillMode = pfAlternate</td>
<td>TTransformation = nil</td>
<td>overload</td>
</tr>
<tr>
<td>PolygonXS</td>
<td>TBitmap32</td>
<td>TArrayOfFixedPoint</td>
<td>TColor32</td>
<td>TPolyFillMode = pfAlternate</td>
<td>TAntialiasMode = DefaultAAMode</td>
<td>overload</td>
</tr>
<tr>
<td>PolygonXS</td>
<td>TBitmap32</td>
<td>TArrayOfFixedPoint</td>
<td>TFillLineEvent</td>
<td>TPolyFillMode = pfAlternate</td>
<td>TAntialiasMode = DefaultAAMode</td>
<td>overload</td>
</tr>
<tr>
<td>PolygonXS</td>
<td>TBitmap32</td>
<td>TArrayOfFixedPoint</td>
<td>TCustomPolygonFiller</td>
<td>TPolyFillMode = pfAlternate</td>
<td>TAntialiasMode = DefaultAAMode</td>
<td>overload</td>
</tr>
</tbody>
</table>
**Description**

Fills the shape defined by the **Points** parameter and (optionally) transformed by the **Transformation** parameter using one of the following options:

- **simple color**
- **custom fillline callback**
- **custom filler object**

Unlike GDI polygons, this function allows for transparency and optional antialiasing.

**PolygonTS** draws a non-antialiased polygon with optional transparency. When drawing, fixed-point coordinates of vertices are rounded.

**PolygonXS** provides full antialiasing and does not perform rounding of vertex coordinates. You can also specify the antialias mode **AAMode** that should be used.

All functions perform clipping.
See Also

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultAA

DefaultA
Polyline

```pascal
procedure PolyLineTS(Bitmap: TBitmap32; const Points: TArrayOfFixedPoint; Color: TColor32; Closed: Boolean=True; Transformation: TTransformation = nil);

procedure PolyLineAS(Bitmap: TBitmap32; const Points: TArrayOfFixedPoint; Color: TColor32; Closed: Boolean=True; Transformation: TTransformation = nil);

procedure PolyLineXS(Bitmap: TBitmap32; const Points: TArrayOfFixedPoint; Color: TColor32; Closed: Boolean=True; Transformation: TTransformation = nil);

procedure PolyLineXSP(Bitmap: TBitmap32; const Points: TArrayOfFixedPoint; Closed: Boolean=True; Transformation: TTransformation = nil);
```
**Description**

Draws a series of lines, connecting points passed in. The points are transformed by **Transformation**. If **Closed** is True, the polygon is closed automatically by drawing a line from the last vertex to the first.

**PolyLineTS** draws a non-antialiased line with optional transparency. When drawing, fixed-point coordinates of vertexes are rounded.

**PolyLineAS** draws an antialiased line, fixed-point coordinates of vertexes are rounded.

**PolyLineXS** provides full antialiasing and does not perform rounding of vertexes’ coordinates. **PolyLineXSP** uses the stipple pattern defined in the **Bitmap** instead of a color.

All functions perform clipping.
See Also

Naming Conventions, TArrayOfFixedPoint, TBitmap32, TColor32, TTransformation
PolyPolygon

procedure PolyPolygonTS(Bitmap: TBitmap32; const Points: TArrayOfArrayOfFixedPoint; Color: TColor32; Mode: TPolyFillMode = pfAlternate);

procedure PolyPolygonXS(Bitmap: TBitmap32; const Points: TArrayOfArrayOfFixedPoint; Color: TColor32; Mode: TPolyFillMode = pfAlternate; AAMode: TAntialiasMode = DefaultAA_Mode);
Description

Fills the shape similar to that done by Polygon function. The shape is constructed from several contours, which are specified in Points parameter. The filling is performed according to Mode parameter.

For PolyPolygonXS you can also specify the antialias mode that should be used.
See Also

DefaultAAMode, Naming Conventions, Polygon, PolyPolyline, TAntialiasMode, TArrayArrayOfFixedPoint, TBitmap32, TColor32, TPolyFillMode
## PolyPolygonBounds

| function | PolyPolygonBounds(const Points: TArrayArrayOfFixedPoint): TFixedRect; |
**Description**

Returns the boundaries of the input array.
See Also

Point Types, Rectangle Types
PolyPolyline

procedure PolyPolylineTS(Bitmap: TBitmap32; const Points: TArrayOfArrayOfFixedPoint; Color: TColor32; Closed: Boolean=True; Transformation: TTransformation=nil);

procedure PolyPolylineAS(Bitmap: TBitmap32; const Points: TArrayOfArrayOfFixedPoint; Color: TColor32; Closed: Boolean=True; Transformation: TTransformation=nil);

procedure PolyPolylineXS(Bitmap: TBitmap32; const Points: TArrayOfArrayOfFixedPoint; Color: TColor32; Closed: Boolean=True; Transformation: TTransformation=nil);

procedure PolyPolylineXSP(Bitmap: TBitmap32; const Points: TArrayOfArrayOfFixedPoint; Closed: Boolean=True; Transformation: TTransformation=nil);
Description
PolyPolyline is the same as Polyline, but takes array of polylines as parameter. The Closed parameter influences all polylines in the array.
See Also

Naming Conventions, Polyline, PolyPolygon, TArrayOfArrayOfFixedPoint, TBitmap32, TColor32
function PtInPolygon(const Pt: TFixedPoint; const Points: TArrayOfFixedPoint): Boolean;
Description
Returns true if \textbf{Pt} is a coordinate within the polygon region specified by an array of fixed-point coordinates.
See Also

TPolygon32.ContainsPoint, Point Types, TArrayOfFixedPoint, TFixedPoint
TAntialiasMode

```plaintext
type TAntialiasMode = (am32times, am16times, am8times, am4times, am2times);
```
**Description**

Specifies how polygons are antialiased.

Here is a small illustration that demonstrates the different modes and their output quality:

32x supersampling generally generates the best result. However, it is also the slowest mode.

16x supersampling generates a very smooth result. Generally, there is no noticeable difference in quality between this and the 32x mode.

8x supersampling is a good compromise between speed and quality.

4x supersampling offers an acceptable tradeoff between speed and quality.

2x supersampling offers the worst quality, but still has a higher precision than rendering without any antialiasing.

If you need fast but still antialiased polygons (for example in an editor) we recommend using the 4x or 8x antialias mode.
See Also

Polygon
TFillLineEvent

```plaintext
type TFillLineEvent = procedure(Dst: PColor32; DstX, DstY, Length: Integer; AlphaValues: PColor32) of object;
```
Description
Use this definition to create custom fill line callbacks.

**Dst** specifies the first position in the destination buffer that needs to be drawn to. You can use the coordinates **DstX** and **DstY** for calculations.

**Length** determines the length of the line to draw.

**AlphaValues** is a pointer to the first alpha value of the edge for each pixel on that line. This is used by the PolygonXS and PolyPolygonXS procedures only. For PolygonTS and PolyPolygonTS this parameter is **nil**, so please do appropriate checks.

Callbacks based on this specification can be used with the [DrawFill] and [Draw] methods of [TPolygon32] or with the [Polygon] and [PolyPolygon] routines.
See Also

TPolygon32.Draw, TPolygon32.DrawFill, Polygon, PolyPolygon, TPolygon32
TPolyFillMode

```plaintext
type TPolyFillMode = (pfAlternate, pfWinding);
```
Description
Specifies how polygons are filled.

- pfAlternate
- pfWinding
See Also

Polygon
DefaultAAMode

The DefaultAAMode constant of type TAntialiasMode sets the default antialias mode to am8times.
See Also

TAntialiasMode

Copyright ©2000-2007 Alex Denisov and Contributors - Graphics32 v1.8.3 - Build on 4-March-2007
TArrowBar.Backgnd

**property** Backgnd: `TRBBackgnd`;

**type** `TRBBackgnd = (bgPattern, bgSolid);`
Description

Determines appearance of the background:

- `bgPattern`
- `bgSolid`

Note, that this property is available only when `Style=rbsMac`. 
See Also
Style, TRBBackgnd
TArrowBar.BorderStyle

property BorderStyle: TBorderStyle;

type TBorderStyle = (bsNone, bsSingle); // declared in Forms.pas
**Description**

When **BorderStyle** is bsSingle, control is painted with a recessed border.
TArrowBar.ButtonSize

```plaintext
property ButtonSize: Integer;
```
**Description**

Specifies the size of the scroll button in pixels.

When **ButtonSize** is set to 0 (default), this size is calculated and updated automatically to keep the button square.

If you want to hide buttons set **ShowArrows** property to False.
See Also

ShowArrows
**TArrowBar.HandleColor**

```diff

- property HandleColor: TColor;

```
**Description**

Specifies a color of the handle:

![Color Swatches]

Although `TArrowBar` itself does not have any handles, this property is used in descendants `TCustomRangeBar` and `TCustomGaugeBar`.

Note, that this property is available only when `Style=rbsMac`. 
See Also

Style, TArrowBar, TCustomGaugeBar, TCustomRangeBar
TArrowBar.Kind

property Kind: TScrollBarKind;

type TScrollBarKind = (sbHorizontal, sbVertical); // defined in Forms.pas
Description
Indicates orientation of the control.
| **property** | ShowArrows: Boolean; |
Description
Determines visibility of scrolling buttons.
TArrowBar.ShowHandleGrip

```delphi
property ShowHandleGrip: Boolean;
```
Description

Determines visibility of a handle grip:

Although `TArrowBar` itself does not have any handles, this property is used in descendants `TCustomRangeBar` and `TCustomGaugeBar`.

Note, that this property is available only when `Style=rbsMac`. 
See Also

Style, TArrowBar, TCustomGaugeBar, TCustomRangeBar
TArrowBar.Style

```pascal
property Style: TRBStyle;
type TRBStyle = (rbsDefault, rbsMac);
```
**Description**

Determines appearance of the control. Note that some properties are accessible only when Style is set to rbsMac.

The default value is rbsDefault;
TArrowBar.Create

constructor Create(AOwner: TComponent); override;
Description
Creates and initializes an instance of TArrowBar.
After calling the inherited constructor, **Create** initializes the following properties:

- **BackColor** to bgPattern;
- **ButtonSize** to 0;
- **Color** to clBtnShadow;
- **Style** to rbsDefault;
- **HandleColor** to clBtnShadow;
- **Kind** to sbHorizontal;
- **ShowArrows** to True;
- **ShowHandleGrip** to True;
See Also

`Backgnd`, `ButtonSize`, `HandleColor`, `Kind`, `ShowArrows`, `ShowHandleGrip`, `Style`, `TArrowBar`
TArrowBar.Destroy

  destructor Destroy; override;
Description
Destroys the arrowbar object and frees all associated memory.
Do not call **Destroy** directly, use **Free** method instead.
TArrowBar.Paint

procedure Paint; override;
Description
Paint is overridden from \textit{TCustomControl}. It draws the bar and scroll buttons.
TArrowBarOnChange

property OnChange: TNotifyEvent;
Description

OnChange is called each time control changes.
This event is never called by TArrowBar itself, but it is shared by descendants TCustomRangeBar and TCustomGaugeBar.
See Also

OnUserChange, TArrowBar, TCustomGaugeBar, TCustomRangeBar
**TArrowBar.OnUserChange**

```plaintext
property OnUserChange: TNotifyEvent;
```
**Description**

**OnUserChange** is called when control is changed by user, as reaction to mouse events. It is not called then the control is changed due to some internal program activity. This event is called after **OnChange**.

This event is never called by **TArrowBar** itself, but it is shared by descendants **TCustomRangeBar** and **TCustomGaugeBar**.
See Also

OnChange, TArrowBar, TCustomGaugeBar, TCustomRangeBar
TCustomGaugeBar.HandleSize

property HandleSize: Integer;
Description

Specifies the size of the handle:

If **HandleSize** is 0 (default), the size is calculated and updated automatically to keep the handle square.
See Also

TArrowBar.ButtonSize
TCustomGaugeBar.LargeChange

property LargeChange: Integer;


**Description**
Determine how much Position changes when the user clicks the scroll bar on either side of the handle.

![Scroll Bar](image)
See Also

Position, SmallChange

Copyright ©2000-2007 Alex Denisov and Contributors - Graphics32 v1.8.3 - Build on 4-March-2007
TCustomGaugeBar.Max

property Max: Integer;
Description

The **Max** and **Min** properties define the available range for **Position**.
See Also

Min, Position
TCustomGaugeBar.Min

**property** Min: Integer;
Description
The **Max** and **Min** properties define the available range for **Position**.
See Also

Max, Position

Copyright ©2000-2007 Alex Denisov and Contributors - Graphics32 v1.8.3 - Build on 4-March-2007
TCustomGaugeBar.Position

property Position: Integer;
**Description**

Specifies the current position of the scrollbar.

Read **Position** to determine the current position of the thumb tab.

Set **Position** to programmatically move the handle. The available range of positions is determined by the difference between the **Max** property and the **Min** property.

When **Position** is changed programmatically, gauge bar will generate **OnChange** event, when it is changed by user (by clicking scrolling buttons, dragging the handle or clicking on the background area), the control will additionally generate **OnUserChange** event.
See Also

Max, Min, TArrowBarOnChange, TArrowBarOnUserChange
`property` SmallChange: Integer;
Description
Determines how much Position changes when the user clicks the scrolling buttons.
See Also

LargeChange, Position
TCustomGaugeBar.Create

constructor Create(AOwner: TComponent); override;
**Description**

Creates and initializes an instance of `TCustomGaugeBar`.

After calling the inherited constructor, `Create` initializes the following properties:

- **HandleSize** to 0;
- **LargeChange** to 1;
- **Max** to 100;
- **Min** to 0;
- **Position** to 0;
See Also

HANDLESIZE, LARGECHANGE, MAX, MIN, POSITION, TCUSTOMGAUGEBAR

Copyright ©2000-2007 Alex Denisov and Contributors - Graphics32 v1.8.3 - Build on 4-March-2007
procedure Paint; override;
**Description**

Paint is overriden from `TArrowBar.Paint`. In addition to bar and scrolling buttons, it paints the handle.
See Also

TArrowBar::Paint

Copyright ©2000-2007 Alex Denisov and Contributors - Graphics32 v1.8.3 - Build on 4-March-2007
property Centered: Boolean;
**Description**

Determines how the handle is repositioned when `Range` changes.

If **Centered** is False (default), the control behaves similar to standard `TControlScrollBar`. That is it tries to preserve the `Position` property.

If **Centered** is True, the control changes `Position`, trying to keep the centre at the same location.

This property also influences how the handle is adjusted when range bar changes its `Window` size.
See Also

Position, Range, Window
TCustomRangeBar.Increment

property Increment: Single;
**Description**

Determines the change in Position when user clicks scrolling arrows. Default value is 8.
property Position: Single;
**Description**

Determines the current position of the handle.

Note, that **Position** is a single precision floating-point value. Which allows for more predictable behavior then **Centered** property is set to *True*, since the handle won't drift due to rounding errors.
See Also

Centered, Range, Window
property Range: Single;
**Description**

Determines the scrollable range.

When range is less than the value stored in [Window](#) property (or the size of the control, in case Window = 0), the control draws itself disabled.
See Also

Position, Window
TCustomRangeBar.Window

property Window: Single;
**Description**

The size of the associated visible area:

In case the value of the property is set to zero, range bar assumes that it is equal to its own size, as shown in the picture above.

Using this property is required when the size of the scrollable window is different from the size of the range bar.
See Also

Position, Range
TCustomRangeBar.Create

constructor Create(AOwner: TComponent); override;
**Description**

Creates and initializes an instance of `TCustomRangeBar`. After calling the inherited constructor, `Create` initializes the following properties:

- `Centered` to False;
- `Increment` to 8;
- `Position` to 0;
- `Range` to 0;
See Also

Centered, Increment, Position, Range, TCustomRangeBar
procedure Paint; override;
Description

Paint is overridden from TArrowBar.Paint. In addition to the bar itself and scrolling buttons, it paints the handle.
See Also

TArrowBar.Paint
TRBBackgnd

```haskell
  type TRBBackgnd = (bgPattern, bgSolid);
```
**Description**

Specifies a type of background filling of the arrow bar control and its descendants. See [TArrowBar.Backgnd](#) for more information.
See Also

TArrowBar.Backgnd
TRBStyle

```plaintext
type TRBStyle = (bgDefault, bgMac);
```
**Description**

Specifies the arrow bar style. See [TArrowBar.Style](#) for more information.
See Also

TArrowBar.Style
TProgressiveRasterizer.Steps

property Steps: Integer;
Description

This property controls how many progressive steps the progressive rasterization process will use, i.e. the intermediate resolution progressions.
TProgressiveRasterizer.UpdateRows

property UpdateRows: Boolean;
**Description**

This property defines how the rasterizer will trigger repaints. If true, each row of the destination bitmap will be repainted as the rasterizer progress to next row. If false, only step progression will trigger a repaint.
TRasterizer.Sampler

property Sampler: TCustomSampler;
Description

Associates a sampler class, which is used in the rasterization process to retrieve color samples.
See Also

Rasterize, TCustomSampler

Copyright ©2000-2007 Alex Denisov and Contributors - Graphics32 v1.8.3 - Build on 4-March-2007
procedure Rasterize(Dst: TBitmap32); overload;
procedure Rasterize(Dst: TBitmap32; const DstRect: TRect); overload;
procedure Rasterize(Dst: TBitmap32; const DstRect: TRect; const CombineInfo: TCombineInfo); overload;
procedure Rasterize(Dst: TBitmap32; const DstRect: TRect; Src: TBitmap32); overload;
Description
Calling rasterize will start the rasterization process.
See Also

Rectangle Types, TBitmap32, TCombineInfo
TRegularRasterizer.UpdateRowCount

property UpdateRowCount: Integer;
Description
This property controls how many rows should be repainted as the rasterization is performed.
**TSwizzlingRasterizer.BlockSize**

```plaintext
property BlockSize: Integer;
```
Description
This property controls how large blocks should be repainted as the rasterization is performed. The size of the block is $2^{\text{BlockSize}}$. 
TCombineInfo

PCombineInfo = ^TCombineInfo;
TCombineInfo = record
  SrcAlpha: Integer;
  DrawMode: TDrawMode;
  CombineMode: TCombineMode;
  CombineCallback: TPixelCombineEvent;
 TransparentColor: TColor32;
end;
Description

TCombineInfo contains blending and combine related properties, similar to what is contained in TBitmap32. This structure is used in routines which does not have a single source bitmap, which would normally provide this information (for example a special overloaded version of Rasterize).
See Also

Color Types, TRasterizer.Rasterize, TBitmap32, TCombineMode, TDrawMode, TPixelCombineEvent
TAdaptiveSuperSampler.Level

property Level: Integer;
Description
This property will set the maximum possible recursion depth.
TAdaptiveSuperSampler.Tolerance

property Tolerance: Integer;
**Description**

Adaptive supersampling uses a tolerance parameter to limit the recursion depth. If the difference between two samples is less than the tolerance, then the recursion will stop. By default the difference between two colors is computed as the sum of the componentwise differences between the R, G and B components. It is possible to change this evaluation by overriding the protected CompareColors method.
procedure Create(Sampler: TCustomSampler); override;
**Description**

Call Create to instantiate an adaptive super sampler at runtime.
Sampler is an instance of TCustomSampler used to retrieve samples by the adaptive super sampler.
TAdaptiveSuperSampler.GetSample

function GetSampleInt(X, Y: Integer): TColor32;
function GetSampleFixed(X, Y: TFixed): TColor32;
function GetSampleFloat(X, Y: Single): TColor32;
**Description**

**GetSample** provides a method for acquiring a color sample from the coordinate \((X, Y)\). The sample may be generated by the sampler itself, or it may be acquired from a *nested sampler*. Descendants of **TCustomSampler** must always override at least one of **GetSampleFixed** or **GetSampleFloat**. The different postfixes determine the precision of the input parameters (integer, fixed-point or floating-point).
See Also

Color Types, TCustomSampler, TFixed
TBitmap32Resampler.Bitmap

property Bitmap: TBitmap32;
Description

Any instance of a TBitmap32Resampler descendant is associated with a bitmap. The Resample method is used whenever TBitmap32.Draw or TBitmap32.DrawTo is invoked. The GetSample method allows the resampler to reconstruct a single color sample from the bitmap.
See Also

body, TBitmap32.Draw, TBitmap32.DrawTo, GetSampleFloat, TBitmap32, TCUSTOMRESAMPLER

Copyright ©2000-2007 Alex Denisov and Contributors - Graphics32 v1.8.3 - Build on 4-March-2007
TBitmap32Resampler.PIXELACCESSMODE

property PixelAccessMode: TPixelAccessMode;
**Description**

Determines how pixels are accessed. See [TPixelAccessMode](#) for details.
See Also

TPixelAccessMode
TBitmap32Resampler.TransformerClass

property TransformerClass: TTransformerClass;
Description

Each resampler is associated with a default transformer class.
TBitmap32Resampler.Changed

procedure Changed;
Description
Notifies the associated TBitmap32 instance that a property has changed and calls the inherited Changed method.
TBitmap32Resampler.Create

constructor Create(ABitmap: TBitmap32); reintroduce; virtual;
Description

Creates and initializes an instance of TBitmap32Resampler.

After calling the inherited constructor, Create initializes the following properties:

- **Bitmap** to Bitmap;
- **PixelAccessMode** to pamSafe;
- **TransformerClass** to TTransformer;
See Also

Bitmap, PixelAccessMode, TBmap32Resampler, TransformerClass, TTransformer
TBitmap32Resampler.GetSampleBounds

    function GetSampleBounds: TRect; override;
Description

The **GetSampleBounds** method returns the ClipRect property of the associated bitmap instance.
function HasBounds: Boolean; override;
Description
For TBitmap32Resampler descendants HasBounds always returns true, indicating that the sampling is constrained to the ClipRect property of the associated bitmap.
See Also

TBitmap32Resampler
TBitmap32Resampler.PrepareSampling

procedure PrepareSampling; override;
Description

The **PrepareSampling** method should be called before calling GetSample. See [TCustomSampler.PrepareSampling](#) for further details.
See Also

TCustomSampler.PrepareSampling
TBoxKernel.Filter

function Filter(Value: Single): Single; override;
Description
Returns the function value of this kernel given its input parameter.
TBoxKernel.GetWidth

functionGetWidth:Single;override;
Description

Returns the width of this kernel.
TContracter.GetSample

function GetSampleInt(X, Y: Integer): TColor32;
function GetSampleFixed(X, Y: TFixed): TColor32;
function GetSampleFloat(X, Y: Single): TColor32;
**Description**

GetSample provides a method for acquiring a color sample from the coordinate \((X, Y)\). The sample may be generated by the sampler itself, or it may be acquired from a nested sampler. Descendants of TCustomSampler must always override at least one of GetSampleFixed or GetSampleFloat. The different postfixes determines the precision of the input parameters (integer, fixed-point or floating-point).
See Also

Color Types, TCustomSampler, TFixed
TContracter.PrepareSampling

```plaintext
procedure PrepareSampling; override;
```
Description

The **PrepareSampling** method should be called before calling GetSample. See [TCustomSampler.PrepareSampling](#) for further details.
TCosineKernel.Filter

procedure Filter: Single; override;
**Description**

Returns the function value of this kernel given its input parameter.
TCosineKernel.GetWidth

functionGetWidth: Single; override;
Description
Returns the width of this kernel.
TCubicKernel.Coeff

property Coeff: Single;
Description
This is a coefficient of the cubic polynomial described by this kernel. See TCubicKernel description for details.
See Also

TCubicKernel
TCubicKernel.Create

constructor Create; override;
**Description**

Creates and instantiates a `TCubicKernel` object. The `Coeff` property is initialized to -0.5.
See Also

Coeff, TCubicKernel
TCubicKernel.Filter

function Filter(Value: Single): Single; override;
**Description**

Returns the function value of this kernel given its input parameter.
TCubicKernel.GetWidth

functionGetWidth: Single; override;
**Description**

Returns the width of this kernel.
TCubicKernel.RangeCheck

function RangeCheck: Boolean; override;
Description
Indicates whether or not this kernel needs range-checking.
TCustomKernel.Observer

property Observer: TNotifiablePersistent;
**Description**

The **Observer** property is a class that will be notified of any changes to the properties of the kernel (by a change notification).
See Also

TNotifiablePersistent
TCustomKernel.Changed

procedure Changed;
**Description**

Changed is called when a property is changed; this will also cause the *Observer* property to be notified.
See Also

Owner
TCustomKernel.Create

procedure Create; virtual;
Description
Create will instantiate a TCustomKernel object.
See Also

TCustomKernel
TCustomKernel.Filter

function Filter: Single; virtual; abstract;
**Description**

Returns the function value of this kernel given its input parameter. Descendants should override this function.
function GetWidth: Single; override;
Description
Returns the width of this kernel.
TCustomKernel.RangeCheck

function RangeCheck: Single; override;
Description

Descendants should override this procedure if the Kernels require range-checking of the final convolved result. Typically this is needed for kernels with negative lobes.
TEroder.Create

**constructor** Create(ASampler: TCustomSampler); **override**;
Description
Creates and instantiates a TEroder object.
T GaussianKernel.Sigma

property Sigma: Single;
**Description**

Sigma is a parameter of the gaussian window.
T\text{GaussianKernel}.Create

\text{constructor Create; override;}
Description
A brief description of Create.
THERmiteKernel.Bias

property Bias: Single;
Description

The **Bias** property will affect the left-right alignment of the kernel function.
The image contains a single line of text that appears to be a property definition. The text is as follows:

```
property Tension: Single;
```
Description
The tension parameter will influence the smoothness of the curve.
THermiteKernel.Create

constructor Create; override;
Description
Call create to instantiate a THermiteKernel object. Upon creation the properties initialized by setting
- Bias to 0;
- Tension to 0.
See Also
Bias, Tension, THermiteKernel
function Filter(Value: Single): Single; override;
Description
Returns the function value of this kernel given its input parameter.
THERMITEKERNEL.GETWIDTH

function getWidth: Single; override;
Description
Returns the width of this kernel.
THERMITEKERNEL.RANGECHECK

function RangeCheck: Boolean; override;
**Description**

Indicates whether or not this kernel needs range-checking.
TKernelResampler.Kernel

property Kernel: TCustomKernel;
Description
This property specifies which kernel is used for performing resampling. Samples are reconstructed through convolution with this kernel.
See Also

TCustomKernel

Copyright ©2000-2007 Alex Denisov and Contributors - Graphics32 v1.8.3 - Build on 4-March-2007
TKernelResampler.KernelClassName

```csharp
property KernelClassName: string;
```
Description

**KernelClassName** specifies the class name of the current associated kernel. It should never be changed at run-time. It is only used by the object inspector at design-time.
TKernelResampler.KernelMode

property KernelMode: TKernelMode;
Description
Describes the current kernel mode. See TKernelMode for details.
See Also

TKernelMode
TKernelResampler.TableSize

**property** TableSize: Integer;
Description

**TableSize** determines the size of the table used in GetSample for kmTableNearest and kmTableLinear kernel modes.

For kmTableNearest the weightcurve of the kernels is quantized in proportional to the tablesize. Perceptually the side effect of the table nearest approach, becomes visible as the tablesize decreases. With a table size of 2, the actual result becomes similar to using the box kernel.

For kmTableLinear the weightcurve of the kernels is approximized via linear interpolation between the quantized values in the table. Perceptually the side effect of the table linear approach, becomes visible as the table size decreases. With a table size of 2, the actual result becomes similar to using the linear kernel.
See Also

GetSample, TBoxKernel, TKernelMode, TLinearKernel
TKernelResampler.Create

**constructor** Create(Bitmap: TBitmap32); **override**;
Description
Call Create to construct a new TKernelResampler instance.
After calling the inherited constructor, the following properties are set:
  - TableSize is set to 32;
  - Kernel is set to TNearestKernel;
See Also

_Body, TBoxKernel_
TKernelResampler::Destroy

defstructor Destroy; override;
Description
Destroys the resampler object and frees all associated memory.
Do not call Destroy directly, use Free instead.
TKernelResampler.FinalizeSampling

procedure FinalizeSampling; override;
Description

The **FinalizeSampling** method should be paired with **PrepareSampling**. It is called when the sampling is finished. See **TCustomSampler.FinalizeSampling** for further details.
See Also

TCustomSampler.FinalizeSampling, TCustomSampler.PrepareSampling
TKernelResampler.GetSample

-function GetSampleInt(X, Y: Integer): TColor32;
-function GetSampleFixed(X, Y: TFixed): TColor32;
-function GetSampleFloat(X, Y: Single): TColor32;
Description

GetSample provides a method for acquiring a color sample from the coordinate \((X, Y)\). The sample may be generated by the sampler itself, or it may be acquired from a nested sampler. Descendants of TCustomSampler must always override at least one of GetSampleFixed or GetSampleFloat. The different postfixes determines the precision of the input parameters (integer, fixed-point or floating-point).
See Also

Color Types, TCustomSampler, TFixed
TKernelResampler.PrepareSampling

procedure PrepareSampling; override;
Description
The **PrepareSampling** method should be called before calling GetSample. See [TCustomSampler.PrepareSampling](#) for further details.
See Also

TCustomSampler.PrepareSampling
TKernelSampler.CenterX

property CenterX: Integer;
Description

CenterX determines the vertical position of the kernel relative to the sample as the UpdateBuffer method is applied.
TKernelSampler.CenterY

property CenterY: Integer;
**Description**

**CenterY** determines the horizontal position of the kernel relative to the sample as the *UpdateBuffer* method is applied.
TKernelSampler.Kernel

property Kernel: TIntegerMap;
Description

The kernel property is used for describing the convolution kernel (for convolution) and the structuring element (for morphological operations).
TKernelSampler.Create

**constructor** Create(ASampler: TCustomSampler); override;
**Description**

Creates and instantiates a [TKernelSampler](#) object.
See Also

TCustomKernel, TKernelSampler
TKernelSampler::Destroy

\textbf{destructor} Destroy; \textbf{override};
Description

Destroys the kernel sampler object and frees all associated memory. Do not call **Destroy** directly, use **Free** method instead.
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Return Type</th>
<th>Override</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetSampleInt</td>
<td>Get sample at integer indices</td>
<td>TColor32</td>
<td>override</td>
</tr>
<tr>
<td>GetSampleFixed</td>
<td>Get sample at fixed indices</td>
<td>TColor32</td>
<td>override</td>
</tr>
</tbody>
</table>
**Description**

Collects samples from the area defined by the kernel and invokes the `UpdateBuffer` for each new sample. When all samples have been processed, the buffer is converted into the `TColor32` format using the protected `ConvertBuffer` method.

See [TCustomSampler.GetSample](#) for further details.
See Also

Color Types, TCustomSampler.GetSample, TFixed
TLinearKernel.Filter

function Filter(Value: Single): Single; override;
Description
A brief description of Window.

Copyright ©2000-2007 Alex Denisov and Contributors - Graphics32 v1.8.3 - Build on 4-March-2007
TLinearKernel.GetWidth

functionGetWidth:Single;override;
Description

Returns the width of this kernel.
TLinearResampler.Create

constructor Create(Bitmap: TBitmap32); override;
Description
A brief description of Window.
See Also
TBitmap32
TLinearResampler.Destroy

defuctor Destroy; override;
Description
Destroys the resampler object and frees all associated memory.
Do not call Destroy directly, use Free method instead.
TLinearResampler.GetSample

function GetSampleInt(X, Y: Integer): TColor32;
function GetSampleFixed(X, Y: TFixed): TColor32;
function GetSampleFloat(X, Y: Single): TColor32;
Description

GetSample provides a method for acquiring a color sample from the coordinate \((X, Y)\). The sample may be generated by the sampler itself, or it may be acquired from a nested sampler. Descendants of TCustomSampler must always override at least one of GetSampleFixed or GetSampleFloat. The different postfixes determines the precision of the input parameters (integer, fixed-point or floating-point).
See Also

Color Types, TCustomSampler, TFixed
TLinearResampler.PrepareSampling

procedure PrepareSampling; override;
Description

The **PrepareSampling** method should be called before calling GetSample. See [TCustomSampler.PrepareSampling](#) for further details.
See Also

TCustomSampler.PrepareSampling
TMitchellKernel.Filter

function Filter(Value: Single): Single; override;
Description
A brief description of Window.
TMitchellKernel.GetWidth

functionGetWidth:Single;override;
**Description**

Returns the width of this kernel.
TMitchellKernel.RangeCheck

function RangeCheck: Boolean; override;
Description
Indicates whether or not this kernel needs range-checking.
TNearestResampler.Create

*constructor* Create(ABitmap: TBitmap32); *override*;
**Description**

Create instantiates a `TNearestResampler` object by first calling the inherited constructor and then initializing the `TransformerClass` property to `TNearestTransformer`. 
See Also

TBitmap32, TNearResampler, TNearTransformer, TBitmap32Resampler.TransformerClass
TNearestResampler.GetSample

function GetSampleInt(X, Y: Integer): TColor32;
function GetSampleFixed(X, Y: TFixed): TColor32;
function GetSampleFloat(X, Y: Single): TColor32;
**Description**

**GetSample** provides a method for acquiring a color sample from the coordinate \((X, Y)\). The sample may be generated by the sampler itself, or it may be acquired from a *nested sampler*. Descendants of **TCustomSampler** must always override at least one of **GetSampleFixed** or **GetSampleFloat**. The different postfixes determines the precision of the input parameters (integer, fixed-point or floating-point).
See Also

Color Types, TCustomSampler, TFixed
TNearestResampler.PrepareSampling

procedure PrepareSampling; override;
Description

The **PrepareSampling** method should be called before calling GetSample. See [TCustomSampler.PrepareSampling](#) for further details.
See Also

TCustomSampler.PrepareSampling
TNearestTransformer.GetSample

function GetSampleInt(X, Y: Integer): TColor32;
function GetSampleFixed(X, Y: TFixed): TColor32;
function GetSampleFloat(X, Y: Single): TColor32;
Description

**GetSample** provides a method for acquiring a color sample from the coordinate (X, Y). The sample may be generated by the sampler itself, or it may be acquired from a nested sampler. Descendants of `TCustomSampler` must always override at least one of **GetSampleFixed** or **GetSampleFloat**. The different postfixes determines the precision of the input parameters (integer, fixed-point or floating-point).
See Also

Color Types, TCustomSampler, TFixed
TNestedSampler.Sampler

**property** Sampler: **TCustomSampler**;
**Description**

When the `TNestedSampler` class is instantiated, it requires a separate sampler to forward samples to its own particular `GetSample` implementation. Thus it is essential that this property is properly set up to point to another sampler (circular references must also be avoided).
See Also

body, TCustomSampler.GetSample, TCustomSampler
TNestedSampler.Create

procedure Create(Sampler: TCustomSampler); override;
Description
Creates and initializes an instance of **TNesetSampler**.
Sampler is an instance of **TCustomSampler** used by descendants to retrieve samples in a nested fashion.
See Also

BODY, TNestedSampler
TPatternSampler.Pattern

    property Pattern: TFixedSamplePattern;

    type TFixedSamplePattern = array of array of TArrayOfFixedPoint;
**Description**

**Pattern** is a two-dimensional array of `TArrayOfFixedPoint`. Pixels are super-sampled by collecting sub-samples for each sample coordinate specified in one of the elements of the Pattern. For irregular sampling it is useful to use tiles of irregularly generated samples, since this will reduce aliasing. Thus, the sample pattern can be varied from pixel to pixel by adjusting the size of the **Pattern** matrix. This will cause the sample pattern to be repeated.
See Also

Point Types
TPatternSampler.Destroy

procedure Destroy; override;
**Description**
Destroys the kernel object and frees all associated memory.
Do not call **Destroy** directly, use **Free** method instead.
TPatternSampler.GetSample

```pascal
function GetSampleInt(X, Y: Integer): TColor32;
function GetSampleFixed(X, Y: TFixed): TColor32;
function GetSampleFloat(X, Y: Single): TColor32;
```
Description

**GetSample** provides a method for acquiring a color sample from the coordinate \((X, Y)\). The sample may be generated by the sampler itself, or it may be acquired from a *nested sampler*. Descendants of **TCustomSampler** must always override at least one of **GetSampleFixed** or **GetSampleFloat**. The different postfixes determines the precision of the input parameters (integer, fixed-point or floating-point).
See Also

Color Types, TCustomSampler, TFixed
TSelectiveConvolver.Delta

property Delta: Single;
Description

Selective convolution requires a Delta parameter for determining which color samples should be included in the convolution. For each sample location one reference sample \( C \) is chosen. Color samples that are not within the threshold range \([C - \text{Delta}, C + \text{Delta}]\), will be excluded from the convolution.
TSelectivConvolver.Create

constructor Create(ASampler: TCustomSampler); override;
Description

Create instantiates an TSelectiveConvolver object and then sets Delta to 30.
See Also

Delta, TSelectiveConvolver
TSelectConvolver.GetSample

function GetSampleInt(X, Y: Integer): TColor32;
function GetSampleFixed(X, Y: TFixed): TColor32;
function GetSampleFloat(X, Y: Single): TColor32;
Description

**GetSample** provides a method for acquiring a color sample from the coordinate \((X, Y)\). The sample may be generated by the sampler itself, or it may be acquired from a *nested sampler*. Descendants of **TCustomSampler** must always override at least one of **GetSampleFixed** or **GetSampleFloat**. The different postfixes determines the precision of the input parameters (integer, fixed-point or floating-point).
See Also

Color Types, TCustomSampler, TFixed
TSinshKernel.Coeff

property Coeff: Single;
**Description**

The **Coeff** property determines the shape of the filter. The coefficient should be strictly positive; when it approaches zero, the curve will resemble the ideal *sinc* filter.
TSinshKernel.Width

    property Width: Single;
Description
Set and get the width of the $\sinh$ kernel.
TSinshKernel.Create

constructor Create; override;
Description

Creates and instantiates a `TSinshKernel` object.

After calling the inherited constructor the following properties are set:

- `Width` is set to 3;
- `Coeff` is set to 0.5.
See Also

Coeff, TSinshKernel, Width

Copyright ©2000-2007 Alex Denisov and Contributors - Graphics32 v1.8.3 - Build on 4-March-2007
TSinshKernel.Filter

function Filter: Single; override;
Description
Returns the function value of this kernel given its input parameter.
TSinshKernel.GetWidth

function GetWidth: Single; override;
**Description**

Returns the width of the kernel.
TSinshKernel.SetWidth

```pascal
procedure SetWidth(Value: Single);
```
**Description**

**TSinshKernel** supports adjustment of the width of the kernel using **SetWidth**.
TSplineKernel.Filter

function Filter: Single; override;
Description

Returns the function value of this kernel given its input parameter.
TSplineKernel.GetWidth

functionGetWidth: Single; override;
**Description**

Returns the width of the kernel.
TSuperSampler.SamplingX

property SamplingX: Integer;
Description
This property determines the number of vertical gridlines.
TSuperSampler.SamplingY

procedure SamplingY;
Description
This property determines the number of horizontal gridlines.
### TSuperSampler.Create

| constructor Create(ASampler: TCustomSampler); override; |
Description
Call **Create** to instantiate a new super sampler.
After the inherited constructor is called the **SamplingX** and **SamplingY** properties are set to 4.
See Also

SamplingX, SamplingY, TCustomSampler
TSuperSampler.GetSample

function GetSampleInt(X, Y: Integer): TColor32;
function GetSampleFixed(X, Y: TFixed): TColor32;
function GetSampleFloat(X, Y: Single): TColor32;
**Description**

**GetSample** provides a method for acquiring a color sample from the coordinate \((X, Y)\). The sample may be generated by the sampler itself, or it may be acquired from a *nested sampler*. Descendants of **TCustomSampler** must always override at least one of **GetSampleFixed** or **GetSampleFloat**. The different postfixes determines the precision of the input parameters (integer, fixed-point or floating-point).
See Also

Color Types, TCustomSampler, TFixed
TTransformer.Transformation

property Transformation: TTransformation;
**Description**

*Transformation* specifies which transformation should be used for transforming sample coordinates.
See Also

Transformation

Copyright ©2000-2007 Alex Denisov and Contributors - Graphics32 v1.8.3 - Build on 4-March-2007
TTransformer.GetSample

function GetSampleInt(X, Y: Integer): TColor32;
function GetSampleFixed(X, Y: TFixed): TColor32;
function GetSampleFloat(X, Y: Single): TColor32;
Description

GetSample provides a method for acquiring a color sample from the coordinate \((X, Y)\). The sample may be generated by the sampler itself, or it may be acquired from a nested sampler. Descendants of TCustomSampler must always override at least one of GetSampleFixed or GetSampleFloat. The different postfixes determines the precision of the input parameters (integer, fixed-point or floating-point).
See Also

Color Types, TCustomSampler, TFixed
TTransformer.GetSampleBounds

function GetSampleBounds: TRect; override;
**Description**

Fetches the sample bounds of the next nested sampler and uses `TTransformation.GetTransformedBounds` to update it to the correct (transformed) output rectangle.
See Also

Transformation.GetTransformedBounds
TTransformer.HasBounds

function HasBounds: Boolean;
**Description**

Returns *true* if the associated transformation supports the [GetTransformedBounds](#) method.
See Also

Transformation.GetTransformedBounds
TTransformer.PrepareSampling

procedure PrepareSampling;
Description

The **PrepareSampling** method should be called before calling GetSample. See [TCustomSampler.PrepareSampling](#) for further details.
See Also
TCustomSampler::PrepareSampling
property Width: Single;
Description

This property allows user-control of the width of the kernel. This will adjust the width of the windows implemented in descendant classes.
TWindowedSincKernel.Create

constructor Create; override;
**Description**

Call `create` to instantiate a `TWindowedSincKernel` object. The `Width` property will be initialized to 3.
See Also

TWindowedSincKernel, Width

Copyright ©2000-2007 Alex Denisov and Contributors - Graphics32 v1.8.3 - Build on 4-March-2007
TWindowedSincKernel.Filter

function Filter(Value: Single): Single;
Description
Returns the function value of this kernel given its input parameter.
TWindowedSincKernel.GetWidth

function GetWidth; override;
Description
Returns the width of this kernel.
**TWindowedSincKernel.SetWidth**

```plaintext
procedure SetWidth(Value: Single);
```
**Description**

Sets the width of this kernel.
procedure BlendTransfer(
    Dst: TBitmap32;
    DstX: Integer;
    DstY: Integer;
    DstClip: TRect;
    SrcF: TBitmap32;
    SrcRectF: TRect;
    SrcB: TBitmap32;
    SrcRectB: TRect;
    BlendCallback: TBlendReg); overload;

procedure BlendTransfer(
    Dst: TBitmap32;
    DstX: Integer;
    DstY: Integer;
    DstClip: TRect;
    SrcF: TBitmap32;
    SrcRectF: TRect;
    SrcB: TBitmap32;
    SrcRectB: TRect;
    BlendCallback: TBlendRegEx;
    MasterAlpha: Integer); overload;
Description

**BlendTransfer** is a three parameter version of **BlockTransfer**. **SrcF** is blended with **SrcB** via the provided **BlendCallback** callback routine, and the resulting color is written to the destination. There are two versions, one supporting blend callbacks of type **TBlendReg**, and one supporting the extended **TBlendRegEx**. For the latter, the parameter **MasterAlpha** is passed when calling the extended blendcallback. Note that all blend and combine settings in the provided bitmap parameters are ignored.
See Also

BlockTransfer, Rectangle Types, StretchTransfer, TBitmap32, TBlendReg, TBlendRegEx, Texture Blend Example
procedure BlockTransfer(
    Dst: TBitmap32;
    DstX: Integer;
    DstY: Integer;
    DstClip: TRect;
    Src: TBitmap32;
    SrcRect: TRect;
    CombineOp: TDrawMode;
    CombineCallBack: TPixelCombineEvent = nil);
Description

BlockTransfer is similar to the BitBlt function from Windows GDI. It performs copying of a bitmap fragment specified by SrcRect into location (DstX, DstY) with optional alpha blending or using user-specified combining function. This operation is constrained to the DstClip rectangle specified in the destination bitmap coordinates.

If CombineOp=dmOpaque, the fragment simply replaces destination pixels. In dmBlend mode it is blended to destination using its alpha channel and MasterAlpha property. In dmCusom mode, the function calls CombineCallBack function for pixel combining.

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.

Neither Src nor Dst bitmaps 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 performed.

This routine used in TBitmap32.Draw and TBitmap32.DrawTo methods.
See Also


Copyright ©2000-2007 Alex Denisov and Contributors - Graphics32 v1.8.3 - Build on 4-March-2007
procedure BlockTransferX(
    Dst: TBitmap32;
    DstX: TFixed;
    DstY: TFixed;
    Src: TBitmap32;
    SrcRect: TRect;
    CombineOp: TDrawMode;
    CombineCallBack: TPixelCombineEvent = nil);
Description

**BlockTransferX** is an extended version of [BlockTransfer](#) that allows a destination coordinate \((\text{DstX}, \text{DstY})\) in fixed-point coordinates. This means that the source bitmap is not copied directly but rather transposed a fractional distance before it is copied. In order to still retain reasonable performance, linear interpolation is used for the fractional displacement.

See [BlockTransfer](#) for a detailed description.
See Also

BlockTransfer, StretchTransfer, TBitmap32, TDrawMode, TPixelCombineEvent, TRect
procedure Contract(Src, Dst: TBitmap32; Kernel: TIntegerMap;
CenterX, CenterY: Integer);
Description

Contract performs rasterization using the TContracter sampler. The Kernel parameter specifies a weight matrix which is centered at the coordinated (X + CenterX, Y + CenterY) for each pixel coordinate (X, Y).

Output pixels are computed from the formula

\[(f \circ b)(s, t) = \max_{x,y} \{b(x, y)\} - (1 - f) \circ b\]

where \(f\) corresponds to the source bitmap and \(b\) corresponds to the kernel (formally described as a structuring element).
See Also

TBitmap32, TContracter, TIntegerMap
Dilate

procedure Dilate(Src, Dst: TBitmap32; Kernel: TIntegerMap;
CenterX, CenterY: Integer);
Description

**Dilate** performs rasterization using the **TDilater** sampler. The **Kernel** parameter specifies a weight matrix which is centered at the coordinated \((X + CenterX, Y + CenterY)\) for each pixel coordinate \((X, Y)\).

Output pixels are computed from the formula

\[
(f \oplus b)(s, t) = \max_{x,y} \{ f(s - x, t - y) + b(x, y) \}
\]

where \(f\) corresponds to the source bitmap and \(b\) corresponds to the kernel (formally described as a *structuring element*).
See Also

TBitmap32, TDilater, TIntegerMap
procedure Erode(Src, Dst: TBitmap32; Kernel: TIntegerMap; CenterX, CenterY: Integer);
**Description**

**Erode** performs rasterization using the T*Eroder* sampler. The **Kernel** parameter specifies a weight matrix which is centered at the coordinated \((X + \text{CenterX}, Y + \text{CenterY})\) for each pixel coordinate \((X, Y)\).

Output pixels are computed from the formula

\[
(f \ominus b)(s, t) = \min_{x,y}(f(s - x, t - y) - b(x, y))
\]

where \(f\) corresponds to the source bitmap and \(b\) corresponds to the kernel (formally described as a *structuring element*).
See Also

TBitmap32, TEroder, TIntegerMap
procedure Expand(Src, Dst: TBitmap32; Kernel: TIntegerMap;
CenterX, CenterY: Integer);
**Description**

**Expand** performs rasterization using the **TExpander** sampler. The **Kernel** parameter specifies a weight matrix which is centered at the coordinated \((X + CenterX, Y + CenterY)\) for each pixel coordinate \((X, Y)\).

Output pixels are computed from the formula

\[
(f \circ b)(s, t) = \max_{x,y}\{f(s - x, t - y)b(x, y)\}
\]

where \(f\) corresponds to the source bitmap and \(b\) corresponds to the kernel (formally described as a *structuring element*).
See Also

TBitmap32, TExpander, TIntegerMap
StretchTransfer

procedure StretchTransfer(
    Dst: TBitmap32;
    DstRect: TRect;
    DstClip: TRect;
    Src: TBitmap32;
    SrcRect: TRect;
    StretchFilter: TStretchFilter;
    CombineOp: TDrawMode;
    CombineCallBack: TPixelCombineEvent = nil);
Description

StretchTransfer is similar to StretchBlt or StretchDIBits functions from WindowsGDI. This procedure performs copying and, if necessary, stretching of the bitmap fragment specified by SrcRect into location in Dst specified by DstRect. This operation is constrained to the DstClip rectangle specified in the destination bitmap coordinates.

For more information on the stretch filters take a look here.

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.

This routine used in TBitmap32.Draw and TBitmap32.DrawTo methods.
See Also

BlockTransfer, TBitmap32.Draw, TBitmap32.DrawTo, TBitmap32, TDrawMode, TPixelCombineEvent, TRect, TStretchFilter

Copyright ©2000-2007 Alex Denisov and Contributors - Graphics32 v1.8.3 - Build on 4-March-2007
TKernelMode

```plaintext
type TKernelMode = (kmDynamic, kmTableNearest, kmTableLinear);
```
Description

TKernelMode defines how a kernel will be used in TKernelResampler, when GetSample methods are called.

- **kmDynamic** - Uses direct calls to the kernels filter method. Being precise, this mode is also quite expensive. The curve display in the resamplers example shows how the different weight curves are configured.

- **kmTableNearest** - Uses precalculated weight tables. Being the fastest, this mode can produce fair results. An option is to increase the tablesize.

- **kmTableLinear** - Uses precalculated weight tables, but in contrast to kmTableNearest this mode will perform linear interpolation between precalculated weights. Being somewhat slower than kmTableNearest, but faster than kmDynamic, this mode produces results good results.
See Also

Examples, TKernelResampler.GetSample, TKernelResampler.TableSize, TCustomKernel, TKernelResampler
TPixelAccessMode

```haskell
  type TPixelAccessMode = (pamUnsafe, pamSafe, pamWrap);
```
Description

TPixelAccessMode determines how TKernelResampler handles pixels outside the bitmap boundary when invoking the GetSample method.

- **pamUnsafe** - No boundary checks are performed. This means that we will read from memory outside the bitmap, if coordinates outside the bitmap area are passed.
- **pamSafe** - If the coordinate is outside the clipping rectangle of the bitmap, then the bitmap's outer color will be returned.
- **pamWrap** - Coordinates will be wrapped using the current WrapMode and will be restricted to the clipping rectangle.
- **pamTransparentEdge** - If the coordinate is outside the clipping rectangle of the bitmap, then a transparent result is returned. Moreover edges are resampled as if there were transparent outer edges. This can be used to achieve antialiased edges when transforming bitmaps with opaque edges; The functionality is similar to SetBorderTransparent.
See Also

TKernelResampler.GetSample, SetBorderTransparent, TKernelResampler, TBitmap32.WrapMode

Copyright ©2000-2007 Alex Denisov and Contributors - Graphics32 v1.8.3 - Build on 4-March-2007
TAffineTransformation.Matrix

Matrix: TFloatMatrix;
Description
Stores the transformation matrix. The matrix is accessed as a field instead of a property in order to make it Borland C++ Builder compatible.
See Also

TFloatMatrix
TAffineTransformation.Clear

procedure Clear;
Description
Resets the transformation (loads IdentityMatrix into the Matrix field).
See Also

IdentityMatrix, Matrix
TAffineTransformation.Create

constructor Create; virtual;
**Description**

Creates and initializes an instance of `TAffineTransformation`. Upon creation, the `Matrix` field is initialized to `IdentityMatrix`. 
See Also

IdentityMatrix, Matrix, TAffineTransformation
function GetTransformedBounds: TRect; override;
**Description**

Returns the bounding rectangle that surrounds the destination area affected by the transformation. The coordinates of the rectangle are specified relative to the destination bitmap origin.
See Also

TRect
TAffineTransformation.Rotate

procedure Rotate(Cx, Cy, Alpha: Single);
Description
At first, the origin is translated to \((Cx, Cy)\) point, then the image is rotated around the origin by \textbf{Alpha} degrees

\[
M = \begin{bmatrix}
\cos \alpha & \sin \alpha & 0 \\
-\sin \alpha & \cos \alpha & 0 \\
0 & 0 & 1
\end{bmatrix}
\]

and finally, the origin is shifted back.
TAffineTransform.Scale

procedure Scale(Sx, Sy: Single);
Description

Adds scale to the transformation:

\[
\mathbf{M} = \begin{bmatrix}
S_x & 0 & 0 \\
0 & S_y & 0 \\
0 & 0 & 1
\end{bmatrix}
\]
TAffineTransformation.Skew

procedure Skew(Fx, Fy: Single);
Description

Adds the skew to the transformation:

\[ M = \begin{bmatrix}
1 & F_x & 0 \\
F_y & 1 & 0 \\
0 & 0 & 1
\end{bmatrix} M; \]

Copyright ©2000-2007 Alex Denisov and Contributors - Graphics32 v1.8.3 - Build on 4-March-2007
procedure Translate(Dx, Dy: Single);
Description

Translates the image:

\[ M = \begin{bmatrix} 1 & 0 & D_x \\ 0 & 1 & D_y \\ 0 & 0 & 1 \end{bmatrix} M' \]
TBlobTransformation.BloatPower

property BloatPower: Single;
Description

BloatPower specifies the scaling of the bloat formula.
TDisturbanceTransformation.Disturbance

property Disturbance: Single;
**Description**

**Disturbance** specifies how extreme the random displacement will be scaled.
TProjectiveTransformation.X0

property X0: Single;
Description
Specifies destination coordinate of the top-left point of the source rectangle.
TProjectiveTransformation.X1

property X1: Single;
Description
Specifies destination coordinate of the top-right point of the source rectangle.
TProjectiveTransformation.X3

property X3: Single;
**Description**

Specifies destination coordinate of the bottom-right point of the source rectangle.
TProjectiveTransformation.Y0

property Y0: Single;
Description
Specifies destination coordinate of the top-left point of the source rectangle.
TProjectiveTransformation.Y1

property Y0: Single;
Description
Specifies destination coordinate of the top-right point of the source rectangle.
TProjectiveTransformation.Y2

**property** Y2: Single;
Description
Specifies destination coordinate of the bottom-left point of the source rectangle.
TProjectiveTransformation.Y3

property Y3: Single;
**Description**

Specifies destination coordinate of the bottom-right point of the source rectangle.
TProjectiveTransformation.GetTransformedBounds

function GetTransformedBounds: TRect; override;
**Description**

Returns the bounding rectangle that surrounds the destination area affected by the transformation. The coordinates of the rectangle are specified relative to the destination bitmap origin.
See Also

TRect
TRemapTransformation.MappingRect

**property** MappingRect: *TFloatRect*;
Description

**MappingRect** specifies the scaling and positioning of the input coordinate space in conjunction with the usage of the subordinate vector map. I.e. changing the mapping rectangle will change the mapping *onto* the transformed space. Consider the following two images:

Here a fisheye transformation was rasterized centered on the subordinate vector map (using `RasterizeTransformation`). In the left image, the mapping rectangle (the red *rubberband layer*) is centered and shrunked somewhat. In the right image, the rectangle was changed to another position and scaled, causing a different mapping. The rasterized transformation has not been changed.

As seen above, the the mapping rectangle is not behaving like a transformed bounds rectangle. Thus for intuitive user interface, one may prefer another type of representation. The above representation is solely chosen for the sake of explanatory value.
See Also

GetTransformedBounds, Offset, RasterizeTransformation, Scale, TFloatRect, TRubberbandLayer
TRemapTransformation.Offset

property Offset: TFloatVector;
**Description**

**Offset** specifies an offset applied to the transformation vectors during transform. Setting offset will not change the subordinate vectormap, it is applied in the actual transform procedure.
See Also

MappingRect, Scale, Vector Types
function GetTransformedBounds: TRect; override;
**Description**

TRemapTransformation does not support transformed bounds retrieval. This can be determined calling `HasTransformedBounds`. However if called anyway, it returns `Rect(-MaxInt, -MaxInt, MaxInt, MaxInt)`. The reason is that the transformed bounds cannot be predicted without transforming each vector in the vector map (an extreme overhead in performance), so the absolute most extreme transformed bounds will be returned.
See Also

HasTransformedBounds, TRect
function HasTransformedBounds: Boolean; override;
Description
Returns false. For details see GetTransformedBounds
See Also

GetTransformedBounds
TRemapTransformation.Scale

procedure Scale(Sx: Single; Sy: Single);
Description

Scale stores internal scaling values, which multiplies the vectors when performing the transformation. If $S_x = 0$ and $S_y = 0$, no transformation will appear. Setting scaling values will not change the subordinate vectormap, it is applied in the actual transform procedure. The image warping example shows what effect scaling has.
See Also

Image Warping Example, MappingRect, Offset
TTransformation.SrcRect

property SrcRect: TFloatRect;
Description

**SrcRect** specifies the boundary of the incoming image fragment. Since it is a **TFloatRect**, the boundary has floating point coordinates.
See Also

TFloatRect
TTransformation.GetTransformedBounds

function GetTransformedBounds: TRect; virtual; abstract;
**Description**

Returns the bounding rectangle that surrounds the destination area affected by the transformation. The coordinates of the rectangle are specified relative to the destination bitmap origin. Not all derived transformation types may be able to specify transformed bounds. In that case, the source rectangle is returned.
See Also

HasTransformedBounds, TRect
TTransformation.HasTransformedBounds

```plaintext
function HasTransformedBounds: Boolean;
```
**Description**

Returns true if a given derived transformation can specify its transformed bounds, otherwise the result is false. If custom descendants cannot specify transformed bounds, this method must be overridden, since the default result is *true*.
See Also

GetTransformedBounds
TTransformation.ReverseTransform

function ReverseTransform(const P: TPoint): TPoint; overload; virtual;
function ReverseTransform(const P: TFixedPoint): TFixedPoint; overload; virtual;
function ReverseTransform(const P: TFloatPoint): TFloatPoint; overload; virtual;
Description
Reverse-transforms the supplied point $P$ and returns the result.
TTransformation.Transform

function Transform(const P: TPoint): TPoint; overload; virtual;

function Transform(const P: TFixedPoint): TFixedPoint; overload; virtual;

function Transform(const P: TFloatPoint): TFloatPoint; overload; virtual;
Description
Forward transforms the supplied point \( P \) and returns the result. Note that not all derived transformation classes supports forward transformation. If the given class do not support forward transformation an ETransformNotImplemented exception will be raised.
property Twirl: Single;
Description

Twirl controls the power of the twirl transformation. The parameter is rather sensitive, a realistic setting could e.g. be as low as 0.001. It is recommended that one experiments finding a useful range within a given enviroment, as no common range for the parameter can be specified.
procedure RasterizeTransformation(Vectormap: TVectormap;
  Transformation: TTransformation; DstRect: TRect;
  CombineMode: TVectorCombineMode = vcmAdd;
  CombineCallback: TVectorCombineEvent = nil);
Description

**RasterizeTransformation** provides a convenient **TTransformation** rendering/buffering system, meaning that other transformation classes can be precalculated into the vector map.

The **Transformation** parameter is a reference to a descendant of an abstract **TTransformation** class. It specifies all necessary transformation parameters.

**CombineMode** is vcmAdd by default, and the transformation will be added to the existing vectors (note that the routine will handle the conversion to relative vector space). When combine mode is vcmReplace, transformation vectors will replace the existing ones in the vectormap. When combine mode is vcmCustom, the last parameter **CombineCallback** will be used. Note that even in vcmCustom mode, vectors will be made relative by RasterizeTransformation.
See Also

Rectangle Types, TRemapTransformation, TTransformation, TVectorCombineEvent, TVectorCombineMode, TVectorMap
SetBorderTransparent

procedure SetBorderTransparent(ABitmap: TBitmap32; ARect: TRect);
**Description**

This is an auxiliary function that sets the alpha channel along the edges of specified rectangle to zero. It may be used on a bitmap before passing it to `Transform` function as a workaround to the edge antialiasing problem. For a dynamic option see `TPixelAccessMode` `pamTransparentEdge`, which can do the same trick without any destructive editing.
See Also

TBitmap32, TPixelAccessMode, Transform, TRect
procedure Transform(Dst, Src: TBitmap32; Transformation: TTransformation);
**Description**

*Transform* is responsible for arbitrary geometrical transformations of bitmaps or their fragments. The current version supports only affine and projective transformations.

The *Transformation* parameter is a reference to a descendant of an abstract *TTransformation* class. It specifies all necessary transformation parameters.

When *Src.StretchFilter* is different from sfNearest, *Transform* uses bilinear 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 involved, then *StretchTransfer* to a final bitmap.

The function does not support spline interpolation, 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. You just have to provide the source bitmap (or its region) with transparent edges. In the original image, you’ll have to force the alpha channel on its edges to zeroes, for example, using the *SetBorderTransparent* method.

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 close to the color of the background.
See Also

SetBorderTransparent, TBitmap32.StretchFilter, StretchTransfer, TAffineTransformation, TBitmap32, TProjectiveTransformation, Transform Example, TTransformation
TransformPoints

**function** TransformPoints(Points: TArrayOfArrayOfFixedPoint; Transformation: TTransformation): TArrayOfArrayOfFixedPoint;
Description
Transforms all Points using the supplied Transformation instance and returns the new set of transformed points.
See Also

Transformation
TFloatMatrix

```pascal
type TFloatMatrix = array [0..2, 0..2] of Single;
```
Description
A 3x3 transformation matrix used in affine transformations.
See Also

TAffineTransformation
const IdentityMatrix: TFloatMatrix =(
    (1, 0, 0),
    (0, 1, 0),
    (0, 0, 1));
Description
An identity matrix, which is loaded in Matrix field of TAffineTransformation class when calling its Clear method.
See Also

### property FixedVector\([X, Y: \text{Integer}]\): \text{TFixedVector}; default;

### property FixedVectorS\([X, Y: \text{Integer}]\): \text{TFixedVector};

### property FixedVectorX\([X, Y: \text{TFixed}]\): \text{TFixedVector};

### property FixedVectorXS\([X, Y: \text{TFixed}]\): \text{TFixedVector};

### property FloatVector\([X, Y: \text{Integer}]\): \text{TFloatVector};

### property FloatVectorS\([X, Y: \text{Integer}]\): \text{TFloatVector};

### property FloatVectorF\([X, Y: \text{Single}]\): \text{TFloatVector};

### property FloatVectorFS\([X, Y: \text{Single}]\): \text{TFloatVector};
Description

**FixedVector** property sets the value of the vector in the vectormap. Reading it, will return the vector value of the vector located at specified coordinates. This property does not validate the specified coordinates, so use it only when completely certain that the ranges provided is within the vectormap boundaries. **FixedVector** is declared as default property, you may use it as shown below:

```latex
Vectormap[10, 20] := Vectormap[20, 10]; // copy a vector from (20,10) to (10,20) position
```

**FixedVectorS** is a 'safe' version of the **FixedVector** property. When reading vectors from the outside the bitmap boundary, a vector of (0, 0) is returned. Writing with invalid coordinates will have no effect.

**FixedVectorX** provides a method for accessing the vectors in TFixed based coordinates. The returned vector is computed by performing linear interpolation on four adjacent vectors. Similarly, when vectors are set, they are distributed weighted over four adjacent vectors.

**FixedVectorXS** is a 'safe' version of the above property.

**FloatVector/S/F/FS** Provides floating point compatible versions of the above properties. Note that the floating point versions are provided for convenience only; coding for optimal performance, consider using fixedpoint versions (which will allow execution inside MMX enabled code segments). TVectormap stores its vectors natively in TFixedVector format, hence no extra precision is gained by using floating point versions.
See Also

Fixed Point Math, Naming Conventions, TFixed, TFixedVector, TFloatVector

Copyright ©2000-2007 Alex Denisov and Contributors - Graphics32 v1.8.3 - Build on 4-March-2007
property VectorCombineMode: TVectorCombineMode;
**Description**

*TVectorCombineMode* determines the vector combine mode used in TVectormap.Merge.
See Also

TCombineMode
TVectorMap.Vectors

property Vectors: PFixedPointArray;
**Description**

The read-only vectors property contains the address of the first (top-left) vector in a vectormap. If the vectormap is not allocated (width or height is zero), the returned address is *nil*.

Note, that numbering of rows in Graphics32 starts from the top-most one.

Data is continuously allocated in memory, row by row. You may safely access `Width * Height` elements, each of them is a 8-byte `TFixedVector` value.
See Also

TBitmap32.Bits, Height, Vector Types, Width
TVectorMap.Clear

procedure Clear;
Description

Fills the entire vectormap with zero vectors.
TVectorMap.Destroy

destructor Destroy; override;
Description

Destroys the vectormap object and frees all the associated memory.
Do not call Destroy directly, use the Free method instead.
function Empty: Boolean; override;
Description

Returns True if the vector map contains no data, that is Width or Height is equal to 0.
See Also

TCustomMap.Height, TCustomMap.Width
TVectorMap.GetTrimmedBounds

```pascal
function GetTrimmedBounds: TRect;
```
**Description**

Returns the smallest rectangle that surrounds non zero vectors in a vectormap. This is useful for optimizing heavy processing like supersampling by passing this rectangle as a clipping rectangle. Note that the procedure of analyzing a vectormap is rather expensive, thus carefully determine if the call is worth the cost.
See Also

Image Warping Example, TRect
`TVectorMap.LoadFromFile`
Description

Loads a vector map from a file. Format supported is Photoshop .msh format, compatible with the Liquify plugin. See source for more details.
See Also

Image Warping Example, SaveToFile
TVectorMap.Merge

procedure Merge(DstLeft: Integer; DstTop: Integer; Src: TVectorMap; SrcRect: TRect);
**Description**

**Merge** provides a method for merging a vector map. **VectorCombineMode** specifies how the source vector map is merged. If **VectorCombineMode** is set to vcmCustom, **OnVectorCombine** will be used. The source parameter must not necessarily be some other vector map, so for example in-place masking or scaling is possible.
See Also

Rectangle Types, TVectorMap, VectorCombineMode
TVectorMap.SaveToFile

```pascal
procedure SaveToFile(const FileName: string);
```
Description

Writes a vector map to disk. Format supported is Photoshop .msh format, compatible with the Liquify plugin. See source for more details.
See Also

Image Warping Example, LoadFromFile
property OnVectorCombine: TVectorCombineEvent;

type TVectorCombineEvent = procedure (F, P: TFixedVector;
  var B: TFixedVector) of object;
**Description**

This event is called when vector map **Merge** is called in vcmCustom **vector combine mode**. Use this event to customize handling of vector combining. Note, however, that this event is called for every Vector in the map, so keeping the event handler small and fast is in your own interest.
See Also

Merge, TVectorCombineEvent, Vector Types, VectorCombineMode
TVectorCombineEvent

type TVectorCombineEvent = procedure (F, P: TFixedVector; var B: TFixedVector) of object;
**Description**

`TVectorCombineEvent` is a type for `OnVectorCombine` callback function, used in `Merge`, when `VectorCombineMode` is `vcmCustom`. It specifies a function which takes foreground vector (F) and mixes it with the background vector (B).

This function may optionally use progresson vector (P). The progression vector will by the merge routine be specified in the range X,Y: [-FIXEDONE..FIXEDONE], regardless of the specified source rect in `Merge`. This enables in-place masking of the vector map to be merged.
See Also


Copyright ©2000-2007 Alex Denisov and Contributors - Graphics32 v1.8.3 - Build on 4-March-2007
**TVectorCombineMode**

```plaintext
type TVectorCombineMode = (vcmAdd, vcmReplace, vcmCustom);
```
Description

TVectorCombineMode defines how vectors will be combined (e.g. in TVectormap.Merge).

- **vcmAdd** - Adds the source vectors.
- **vcmReplace** - Replaces with source vectors.
- **vcmCustom** - Uses a specified OnVectorCombine event.
Vector Types

Graphics32 currently uses a relative vector system, meaning that a vector have to be applied to a given absolute vector. The relative system propose a wider flexibility, but also a small overhead (the two additions needed). Future versions may introduce a relative/absolute distinction in relevant routines. To make a vector absolute, simply apply actual absolute vertices, but keep in mind that some implementations may expect relative vectors (e.g. TRemapTransformation.Vectormap uses relative vectors).

Vector types are defined with following structures:
TFloatVector

```plaintext
  type TFloatVector = TFloatPoint;
  type PFloatVector = ^TFloatVector;
```

Provides convenient naming, and compatibility with TFloatPoint.
**TFixedVector**

```pascal
type TFloatVector = TFixedPoint;
type PFloatVector = ^TFloatVector;
```

Provides convenient naming, and compatibility with TFixedPoint.
TArrayOfFloatPoint

- `type TArrayOfFloatVector = array of TFloatVector;`
- `type PArrayOfFloatVector = ^TArrayOfFloatVector;`

A dynamic array of `TFloatVector`. 
TArrayOfFixedPoint

- **type** TArrayOfFixedVector = array of TFixedVector;
- **type** PArrayOfFixedVector = ^TArrayOfFixedVector;

A dynamic array of TFixedVector.
See Also

Creating Points, TVectorMap

Copyright ©2000-2007 Alex Denisov and Contributors - Graphics32 v1.8.3 - Build on 4-March-2007
TAdaptiveSuperSampler
Hierarchy

TPersistent
  └── TNotifiablePersistent
  └── TCustomSampler
  └── TNestedSampler
**Description**

Adaptive supersampling is different from ordinary supersampling in the sense that samples are chosen adaptively. It is a recursive method that collects more samples at areas with rapid transitions.

The advantage with this method that makes it more attractive than ordinary supersampling is that we can perform supersampling only at areas where it is needed. However, the recursion itself may cause overhead, so there is a trade-off between the cost of the associated nested sampling method and the cost of the adaptive recursion.

**Reference**

<table>
<thead>
<tr>
<th>Properties</th>
<th>Methods</th>
<th>Events</th>
</tr>
</thead>
<tbody>
<tr>
<td>In TAdaptiveSuperSampler:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Level</td>
<td>Create</td>
<td></td>
</tr>
<tr>
<td>Tolerance</td>
<td>GetSample</td>
<td></td>
</tr>
<tr>
<td>In TNestedSampler:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sampler</td>
<td></td>
<td></td>
</tr>
<tr>
<td>In TCustomSampler:</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>FinalizeSampling</td>
<td></td>
</tr>
<tr>
<td></td>
<td>GetSampleBounds</td>
<td></td>
</tr>
<tr>
<td></td>
<td>HasBounds</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PrepareSampling</td>
<td></td>
</tr>
</tbody>
</table>

In TNotifiablePersistent:

<table>
<thead>
<tr>
<th>UpdateCount</th>
<th>BeginUpdate</th>
<th>OnChange</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Changed</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Destroy</td>
<td></td>
</tr>
<tr>
<td></td>
<td>EndUpdate</td>
<td></td>
</tr>
</tbody>
</table>

Copyright ©2000-2007 Alex Denisov and Contributors - Graphics32 v1.8.3 - Build on 4-March-2007
TAffineTransformation
Hierarchy

TObject
  | TTransformation
**Description**

The transformation is defined by 3x3 homogeneous matrix of single-precision floats, \( \text{TFloatMatrix} \):

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

Only the first two rows are used for coordinate transformation at the final stage.

**Reference**

<table>
<thead>
<tr>
<th>Fields</th>
<th>Properties</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>In TAffineTransform:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Matrix</td>
<td></td>
<td>Clear</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Create</td>
</tr>
<tr>
<td></td>
<td></td>
<td>GetTransformedBounds</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Rotate</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Scale</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Skew</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Translate</td>
</tr>
<tr>
<td>In TTransformation:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SrcRect</td>
<td>HasTransformedBounds</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ReverseTransform</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Transform</td>
<td></td>
</tr>
</tbody>
</table>
See Also

ImgView and Layers Example, Rotate Example, RotLayer Example, TFloatMatrix, Transform Example
TArrowBar
Hierarchy
TCustomControl
**Description**

A common ancestor for range bars and gauge bars. Both its descendants are similar to standard windows scroll bars. **TArrowBar** declares common properties which define visual appearance of scroll bars.

**Reference**

<table>
<thead>
<tr>
<th>Properties</th>
<th>Methods</th>
<th>Events</th>
</tr>
</thead>
<tbody>
<tr>
<td>Backgnd</td>
<td>Create</td>
<td>OnChange</td>
</tr>
<tr>
<td>BorderStyle</td>
<td>Destroy</td>
<td>OnUserChange</td>
</tr>
<tr>
<td>ButtonSize</td>
<td>Paint</td>
<td></td>
</tr>
<tr>
<td>HandleColor</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Kind</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ShowArrows</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ShowHandleGrip</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Style</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
See Also

TCustomGaugeBar, TCustomRangeBar
TBitmap32
Hierarchy

TPersistent

| TNotifiablePersistent
| TThreadPersistent
| TCustomMap
Description

`TBitmap32` is the central 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 device context (DC).

`TBitmap32` overrides `Assign` and `AssignTo` methods (inherited from `TPersistent`) to provide compatibility with standard objects: `TBitmap`, `TPicture` and `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).

`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` objects. 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 used in multi-threaded applications.
**Change Notification**

Change notification events (**OnCange**, **OnResize**), inherited from **TThreadPersistent** and **TCustomMap** are generated by most drawing/resizing etc. functions with a few exceptions. Due to performance considerations the following functions are not accompanied with event generation:

- Pixel-based operations (**SetPixelT**, **SetPixelF**...);
- **HorzLine*** and **VertLine*** functions;

If necessary, the **OnCange** event may be generated explicitly by calling the **Changed** methods.

Normally, if the bitmap is a part of some container (**TImage32**, **TBitmapLayer**, etc.), its change notification events are linked to the container, causing repaint operations. When making several simultaneous changes, it may be beneficial to enclose them in **BeginUpdate**…**EndUpdate** block followed by the **Changed** call to reduce the amount of repaintings of the container.

**Reference**

<table>
<thead>
<tr>
<th>Properties</th>
<th>Methods</th>
<th>Events</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>BitmapHandle</strong></td>
<td><strong>BeginMeasuring</strong></td>
<td><strong>OnAreaChanged</strong></td>
</tr>
<tr>
<td><strong>BitmapInfo</strong></td>
<td><strong>CanvasAllocated</strong></td>
<td><strong>OnHandleChanged</strong></td>
</tr>
<tr>
<td><strong>Bits</strong></td>
<td><strong>Clear</strong></td>
<td><strong>OnPixelCombine</strong></td>
</tr>
<tr>
<td><strong>Canvas</strong></td>
<td><strong>Create</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Clipping</strong></td>
<td><strong>DeleteCanvas</strong></td>
<td></td>
</tr>
<tr>
<td><strong>ClipRect</strong></td>
<td><strong>Destroy</strong></td>
<td></td>
</tr>
<tr>
<td><strong>CombineMode</strong></td>
<td><strong>Draw</strong></td>
<td></td>
</tr>
<tr>
<td><strong>DrawMode</strong></td>
<td><strong>DrawTo</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Font</strong></td>
<td><strong>Empty</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Handle</strong></td>
<td><strong>EndMeasuring</strong></td>
<td></td>
</tr>
<tr>
<td><strong>MasterAlpha</strong></td>
<td><strong>FillRect</strong></td>
<td></td>
</tr>
<tr>
<td><strong>MeasuringMode</strong></td>
<td><strong>FlipHorz</strong></td>
<td></td>
</tr>
<tr>
<td><strong>OuterColor</strong></td>
<td><strong>FlipVert</strong></td>
<td></td>
</tr>
<tr>
<td><strong>PenColor</strong></td>
<td><strong>FrameRect</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Pixel</strong></td>
<td><strong>GetStippleColor</strong></td>
<td></td>
</tr>
<tr>
<td><strong>PixelPtr</strong></td>
<td><strong>HandleChanged</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Pixmap</strong></td>
<td><strong>HorzLine</strong></td>
<td></td>
</tr>
<tr>
<td><strong>PixmapChanged</strong></td>
<td><strong>Line</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Resampler</strong></td>
<td><strong>LineTo</strong></td>
<td></td>
</tr>
<tr>
<td><strong>ResamplerClassName</strong></td>
<td><strong>LoadFromFile</strong></td>
<td></td>
</tr>
<tr>
<td><strong>ScanLine</strong></td>
<td><strong>LoadFromResourceID</strong></td>
<td></td>
</tr>
<tr>
<td>StippleCounter</td>
<td>LoadFromResourceName</td>
<td></td>
</tr>
<tr>
<td>---------------</td>
<td>----------------------</td>
<td></td>
</tr>
<tr>
<td>StippleStep</td>
<td>LoadFromStream</td>
<td></td>
</tr>
<tr>
<td>StretchFilter</td>
<td>MoveTo</td>
<td></td>
</tr>
<tr>
<td>WrapMode</td>
<td>RaiseRectTS</td>
<td></td>
</tr>
<tr>
<td></td>
<td>RenderText</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ResetAlpha</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ResetClipRect</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Roll</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Rotate</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SaveToFile</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SaveToStream</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SetPixel</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SetSize</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SetStipple</td>
<td></td>
</tr>
<tr>
<td></td>
<td>TextExtent</td>
<td></td>
</tr>
<tr>
<td></td>
<td>TextHeight</td>
<td></td>
</tr>
<tr>
<td></td>
<td>TextOut</td>
<td></td>
</tr>
<tr>
<td></td>
<td>TextWidth</td>
<td></td>
</tr>
<tr>
<td></td>
<td>UpdateFont</td>
<td></td>
</tr>
<tr>
<td></td>
<td>VertLine</td>
<td></td>
</tr>
</tbody>
</table>

In TCustomMap:

<table>
<thead>
<tr>
<th>Height</th>
<th>Delete</th>
<th>OnResize</th>
</tr>
</thead>
<tbody>
<tr>
<td>Width</td>
<td>Resized</td>
<td></td>
</tr>
</tbody>
</table>

In TThreadPersistent:

<table>
<thead>
<tr>
<th>LockCount</th>
<th>Lock</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Unlock</td>
</tr>
</tbody>
</table>

In TNotifiablePersistent:

<table>
<thead>
<tr>
<th>UpdateCount</th>
<th>BeginUpdate</th>
<th>OnChange</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Changed</td>
<td></td>
</tr>
<tr>
<td></td>
<td>EndUpdate</td>
<td></td>
</tr>
</tbody>
</table>
See Also

BeginUpdate, Changed, EndUpdate, OnChange, TBitmapLayer, TCUSTOMMap, TImage32, TThreadPersistent
TBitmap32Collection
Hierarchy

TCollection
Description

**TBitmap32Collection** is a collection of **TBitmap32Item** objects.

Reference

<table>
<thead>
<tr>
<th>Properties</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>Items</td>
<td>Add</td>
</tr>
<tr>
<td></td>
<td>Create</td>
</tr>
</tbody>
</table>
See Also

TBitmap32Item
TBitmap32Item
Hierarchy

TCollectionItem
Description

TBitmap32Item is a simple descendant of standard TCollectionItem, specialized for storing bitmaps.

Reference

<table>
<thead>
<tr>
<th>Properties</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bitmap</td>
<td>Create</td>
</tr>
<tr>
<td></td>
<td>Destroy</td>
</tr>
</tbody>
</table>

Copyright ©2000-2007 Alex Denisov and Contributors - Graphics32 v1.8.3 - Build on 4-March-2007
TBitmap32List
Hierarchy

TComponent
Description
A VCL wrapper around TBitmap32Collection object.

TBitmap32List provides simple means for storing bitmaps in design time. The TBitmap32List object is quite different from the standard TImageList. While TImageList, in fact, stores all images on a single bitmap, and all of its images have the same dimensions, TBitmap32List stores its items as independent bitmaps, which means, it can contain bitmaps of different sizes.

Reference

<table>
<thead>
<tr>
<th>Properties</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>In TBitmap32List:</td>
<td></td>
</tr>
<tr>
<td>Bitmap</td>
<td>Create</td>
</tr>
<tr>
<td>Bitmaps</td>
<td>Destroy</td>
</tr>
</tbody>
</table>
See Also

TBitmap32Collection
TBitmap32Resampler
Hierarchy

TPersistent
  | TNotifiablePersistent
  | TCustomSampler
  | TCustomResampler
Description

TBitmap32Resampler is an abstract ancestor for bitmap resampling classes. A resampler is used for resizing bitmaps and for reconstructing color samples from a rasterized image.

This class is instantiated with one associated bitmap. This bitmap is always used when reconstructing a sample using the GetSample method.

For further information about resampling, see the Sampling and Rasterization topic.

Reference

<table>
<thead>
<tr>
<th>Properties</th>
<th>Methods</th>
<th>Events</th>
</tr>
</thead>
<tbody>
<tr>
<td>In TBitmap32Resampler:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bitmap</td>
<td>Changed</td>
<td></td>
</tr>
<tr>
<td>PixelAccessMode</td>
<td>Create</td>
<td></td>
</tr>
<tr>
<td>TransformerClass</td>
<td>GetSampleBounds</td>
<td></td>
</tr>
<tr>
<td></td>
<td>HasBounds</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PrepareSampling</td>
<td></td>
</tr>
<tr>
<td>In TCustomResampler:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Width</td>
<td>Resample</td>
<td></td>
</tr>
<tr>
<td>In TCustomSampler:</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>FinalizeSampling</td>
<td></td>
</tr>
<tr>
<td></td>
<td>GetSample</td>
<td></td>
</tr>
<tr>
<td>In TNotifyablePersistent:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>UpdateCount</td>
<td>BeginUpdate</td>
<td>OnChange</td>
</tr>
<tr>
<td></td>
<td>Destroy</td>
<td></td>
</tr>
<tr>
<td></td>
<td>EndUpdate</td>
<td></td>
</tr>
</tbody>
</table>
See Also

Resamplers Example, Sampling and Rasterization
TBitmapLayer
Hierarchy

TPersistent
  | TNotifiablePersistent
  | TCustomLayer
  | TPositionedLayer
**Description**

In addition to behavior inherited from `TPositionedLayer`, `TBitmapLayer`, stores and displays a `TBitmap32` object.

The contained `Bitmap` is displayed and scaled using its `DrawMode`, `MasterAlpha` and `StretchFilter` properties.

**Reference**

<table>
<thead>
<tr>
<th>Properties</th>
<th>Methods</th>
<th>Events</th>
</tr>
</thead>
<tbody>
<tr>
<td>In <code>TBitmapLayer</code>:</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>AlphaHit</code></td>
<td><code>Create</code></td>
<td></td>
</tr>
<tr>
<td><code>Bitmap</code></td>
<td><code>Destroy</code></td>
<td></td>
</tr>
<tr>
<td><code>Cropped</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>In <code>TPositionedLayer</code>:</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>Location</code></td>
<td><code>GetAdjustedLocation</code></td>
<td></td>
</tr>
<tr>
<td><code>Scaled</code></td>
<td><code>GetAdjustedRect</code></td>
<td></td>
</tr>
<tr>
<td>In <code>TCustomLayer</code>:</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>Cursor</code></td>
<td><code>BringToFront</code></td>
<td><code>OnHitTest</code></td>
</tr>
<tr>
<td><code>Index</code></td>
<td><code>Changed</code></td>
<td><code>OnMouseDown</code></td>
</tr>
<tr>
<td><code>LayerCollection</code></td>
<td><code>HitTest</code></td>
<td><code>OnMouseMove</code></td>
</tr>
<tr>
<td><code>LayerOptions</code></td>
<td><code>SendToBack</code></td>
<td><code>OnMouseUp</code></td>
</tr>
<tr>
<td><code>LayerStates</code></td>
<td><code>Update</code></td>
<td><code>OnPaint</code></td>
</tr>
<tr>
<td><code>MouseEvents</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>Tag</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>Visible</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>In <code>TNotifiablePersistent</code>:</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>UpdateCount</code></td>
<td><code>BeginUpdate</code></td>
<td><code>OnChange</code></td>
</tr>
<tr>
<td></td>
<td><code>EndUpdate</code></td>
<td></td>
</tr>
</tbody>
</table>
See Also

TBitmapPolygonFiller
Hierarchy

TObject
| TCustomPolygonFiller
Description
TBitmapPolygonFiller provides an easy way to have a polygon filled with a pattern (TBitmap32).
It fully supports the pattern's DrawMode setting. Custom pixel combiners are also supported.
You can directly use it in the DrawFill and Draw methods of TPolygon32 or in the Polygon and PolyPolygon routines.

Reference

<table>
<thead>
<tr>
<th>Properties</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>In TBitmapPolygonFiller:</td>
<td></td>
</tr>
<tr>
<td>OffsetX</td>
<td></td>
</tr>
<tr>
<td>OffsetY</td>
<td></td>
</tr>
<tr>
<td>Pattern</td>
<td></td>
</tr>
<tr>
<td>In TCustomPolygonFiller:</td>
<td>GetFillLine</td>
</tr>
</tbody>
</table>
See Also

TPolygon32.Draw, TPolygon32.DrawFill, TBitmap32.DrawMode,
TBitmap32.OnPixelCombine, Polygon, PolyPolygon, TBitmap32,
TCustomPolygonFiller, TPolygon32
TBlackmanKernel
Hierarchy

TPersistent
  |
TCustomKernel
  |
TWindowedSincKernel
**Description**

A sinc kernel constrained by a Blackman window function.

**Reference**

<table>
<thead>
<tr>
<th>Properties</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>In TWindowedSincKernel:</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Width</strong></td>
<td><strong>Create</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Filter</strong></td>
</tr>
<tr>
<td></td>
<td><strong>GetWidth</strong></td>
</tr>
<tr>
<td></td>
<td><strong>SetWidth</strong></td>
</tr>
<tr>
<td><strong>In TCustomKernel:</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Observer</strong></td>
<td><strong>Changed</strong></td>
</tr>
<tr>
<td></td>
<td><strong>RangeCheck</strong></td>
</tr>
</tbody>
</table>
See Also

Resamplers Example
TBloatTransformation
Hierarchy

TObject

| TTransformation
Description

The bloat transformation class provides an adjustable type of lens distortion. Setting **bloat power** to a positive value gives a fisheye-like distortion, but with less spheric effect (as seen in the left image below). Setting the bloat power to a negative value will affect the transformation in the opposite way (as seen in the right image below).

Reference

<table>
<thead>
<tr>
<th>Properties</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>In TBlobTransformation:</td>
<td></td>
</tr>
<tr>
<td><strong>BloatPower</strong></td>
<td></td>
</tr>
<tr>
<td>In TTransformation:</td>
<td></td>
</tr>
<tr>
<td><strong>SrcRect</strong></td>
<td><strong>GetTransformedBounds</strong></td>
</tr>
<tr>
<td></td>
<td><strong>HasTransformedBounds</strong></td>
</tr>
<tr>
<td></td>
<td><strong>ReverseTransform</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Transform</strong></td>
</tr>
</tbody>
</table>
See Also
TFishEyeTransformation
TBooleanMap
Hierarchy

TPersistent
  |
  TNotifiablePersistent
  |
  TThreadPersistent
  |
  TCustomMap
Description
The **TBooleanMap** class implements a two dimensional matrix of booleans. It may be useful for storing masks or information about areas to process, and in general data that requires or can be represented as boolean types. The class implements efficient storage of booleans: a boolean value only takes up one bit of memory, and is stored in sets of eight bits (one byte).

Reference

<table>
<thead>
<tr>
<th>Properties</th>
<th>Methods</th>
<th>Events</th>
</tr>
</thead>
<tbody>
<tr>
<td>In TBooleanMap:</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Bits</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Value</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>In TCustomMap:</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Height</strong></td>
<td><strong>Delete</strong></td>
<td><strong>OnResize</strong></td>
</tr>
<tr>
<td><strong>Width</strong></td>
<td><strong>Empty</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Resized</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>SetSize</strong></td>
<td></td>
</tr>
<tr>
<td>In TThreadPersistent:</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>LockCount</strong></td>
<td><strong>Lock</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Unlock</strong></td>
<td></td>
</tr>
<tr>
<td>In TNotifiablePersistent:</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>UpdateCount</strong></td>
<td><strong>BeginUpdate</strong></td>
<td><strong>OnChange</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Changed</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Create</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Destroy</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>EndUpdate</strong></td>
<td></td>
</tr>
</tbody>
</table>
See Also

TByteMap, TIntegerMap, TWordMap
TBoxKernel
Hierarchy

TPersistent

| TCustonKernel
**Description**

**TBoxKernel** is one of the simplest and most fundamental kernels. Convolving with this kernel corresponds to the same approach used within nearest neighbor interpolation. The filter is defined as

\[
h(x) = \begin{cases} 
1 & 0 \leq |x| < .5 \\
0 & .5 \leq |x|
\end{cases}
\]

and it is commonly known as a box filter, top-hat function or a Fourier window.

**Reference**

<table>
<thead>
<tr>
<th>Properties</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>In TBoxKernel:</td>
<td></td>
</tr>
<tr>
<td>Observer</td>
<td>Changed</td>
</tr>
<tr>
<td>Create</td>
<td>RangeCheck</td>
</tr>
<tr>
<td>In TCustomKernel:</td>
<td></td>
</tr>
<tr>
<td>Filter</td>
<td>GetWidth</td>
</tr>
</tbody>
</table>
See Also

Resamplers Example
TByteMap
**Hierarchy**

TPersistent

  | TNotifyablePersistent
  | TThreadPersistent
  | TCustomMap
Description

The **TByteMap** class may be used to simulate palette-based operations or to access separate color layers in **TBitmap32**.

**TByteMap** is an ancestor of **TCustomMap** and is assignment compatible with **TBitmap32** objects in both directions.

Reference

<table>
<thead>
<tr>
<th>Properties</th>
<th>Methods</th>
<th>Events</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>In TByteMap:</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bits</td>
<td>Assign</td>
<td></td>
</tr>
<tr>
<td>ValPtr</td>
<td>Clear</td>
<td></td>
</tr>
<tr>
<td>Value</td>
<td>Destroy</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Empty</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ReadFrom</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SetSize</td>
<td></td>
</tr>
<tr>
<td></td>
<td>WriteTo</td>
<td></td>
</tr>
<tr>
<td><strong>In TCustomMap:</strong></td>
<td></td>
<td>OnResize</td>
</tr>
<tr>
<td>Height</td>
<td>Delete</td>
<td></td>
</tr>
<tr>
<td>Width</td>
<td>Resized</td>
<td></td>
</tr>
<tr>
<td><strong>In TThreadPersistent:</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LockCount</td>
<td>Lock</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Unlock</td>
<td></td>
</tr>
<tr>
<td><strong>In TNotifiablePersistent:</strong></td>
<td></td>
<td>OnChange</td>
</tr>
<tr>
<td>UpdateCount</td>
<td>BeginUpdate</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Changed</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Create</td>
<td></td>
</tr>
<tr>
<td></td>
<td>EndUpdate</td>
<td></td>
</tr>
</tbody>
</table>
See Also

ByteMaps Example, TBitmap32, TBooleanMap, TCustomMap, TIntegerMap, TWordMap
TClassList
Hierarchy
TLList
**Description**

TClassList is a container class for storing class types in list form. Handling is similar to TList, but instead of storing pointers it stores class types. Please refer to the TList documentation for details.

We'll cover the additional methods for this class.

**Reference**

<table>
<thead>
<tr>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>In TClassList:</td>
</tr>
<tr>
<td>Find</td>
</tr>
<tr>
<td>GetClassNames</td>
</tr>
</tbody>
</table>
TContourRasterizer
Hierarchy

TPersistent
  └ TNotificationPersistent
  └ TThreadPersistent
Description

This rasterizer determines its rasterization path from the intensity difference between samples. It always chooses the path with lowest intensity difference. Due to a higher probability of near execution code and memory loads, this may optimize the CPU cache usage; hence increase in overall performance. To see how this rasterizer works, check the nested sampling example and choose this rasterizer.

Reference

<table>
<thead>
<tr>
<th>Properties</th>
<th>Methods</th>
<th>Events</th>
</tr>
</thead>
<tbody>
<tr>
<td>In TThreadPersistent:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LockCount</td>
<td>Lock</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Unlock</td>
<td></td>
</tr>
<tr>
<td>In TNotifiablePersistent:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>UpdateCount</td>
<td>BeginUpdate</td>
<td>OnChange</td>
</tr>
<tr>
<td></td>
<td>Changed</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Create</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Destroy</td>
<td></td>
</tr>
<tr>
<td></td>
<td>EndUpdate</td>
<td></td>
</tr>
</tbody>
</table>
See Also

Nested Sampling Example
TContracter
Hierarchy

TPersistent
  | TNotifiablePersistent
  | TCustomSampler
  | TNestedSampler
  | TKernelSampler
Description

This can be thought of as the inverse to the expand operation. The output for a sample coordinate \((s, t)\) is given by

\[
(f \circ b)(s, t) = \max_{x,y} \{ b(x, y) \} - (1 - f) \circ b
\]

where \(f\) is a sampler and \(b\) is a structuring element. The structuring element is defined by the Kernel property of the class. See topic about Sampling and Rasterization for more info.

Reference

<table>
<thead>
<tr>
<th>Properties</th>
<th>Methods</th>
<th>Events</th>
</tr>
</thead>
<tbody>
<tr>
<td>In TContracter:</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>GetSample</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PrepareSampling</td>
<td></td>
</tr>
<tr>
<td>In TKernelSampler:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CenterX</td>
<td>Create</td>
<td></td>
</tr>
<tr>
<td>CenterY</td>
<td>Destroy</td>
<td></td>
</tr>
<tr>
<td>Kernel</td>
<td></td>
<td></td>
</tr>
<tr>
<td>In TNestedSampler:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sampler</td>
<td></td>
<td></td>
</tr>
<tr>
<td>In TCustomSampler:</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>FinalizeSampling</td>
<td></td>
</tr>
<tr>
<td></td>
<td>GetSampleBounds</td>
<td></td>
</tr>
<tr>
<td></td>
<td>HasBounds</td>
<td></td>
</tr>
<tr>
<td>In TNotifyablePersistent:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>UpdateCount</td>
<td>BeginUpdate</td>
<td>OnChange</td>
</tr>
<tr>
<td></td>
<td>Changed</td>
<td></td>
</tr>
<tr>
<td></td>
<td>EndUpdate</td>
<td></td>
</tr>
</tbody>
</table>
See Also

Sampling and Rasterization, TExpander
TConvolver
Hierarchy

TPersistent
  └── TNotifiablePersistent
  └── TCustomSampler
  └── TNestedSampler
  └── TKernelSampler
**Description**

TConvolver provides functionality for performing discrete convolution within a chain of nested samplers. It will collect samples from its associated nested sampler within the region defined by the **Kernel** property.

**Reference**

<table>
<thead>
<tr>
<th>Properties</th>
<th>Methods</th>
<th>Events</th>
</tr>
</thead>
<tbody>
<tr>
<td>In TKernelSampler:</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>CenterX</strong></td>
<td><strong>Create</strong></td>
<td></td>
</tr>
<tr>
<td><strong>CenterY</strong></td>
<td><strong>Destroy</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Kernel</strong></td>
<td><strong>GetSample</strong></td>
<td></td>
</tr>
<tr>
<td>In TNestedSampler:</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Sampler</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>In TCustomSampler:</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>FinalizeSampling</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>GetSampleBounds</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>HasBounds</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>PrepareSampling</strong></td>
<td></td>
</tr>
<tr>
<td>In TNotifiablePersistent:</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>UpdateCount</strong></td>
<td><strong>BeginUpdate</strong></td>
<td><strong>OnChange</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Changed</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>EndUpdate</strong></td>
<td></td>
</tr>
</tbody>
</table>
TCosineKernel
Hierarchy

TPersistent

| TCustomKernel
**Description**

The TCosineKernel class implements a cosine reconstruction filter.

**Reference**

<table>
<thead>
<tr>
<th>Properties</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>In TCosineKernel:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Filter</td>
</tr>
<tr>
<td></td>
<td>GetWidth</td>
</tr>
<tr>
<td>In TCustomKernel:</td>
<td></td>
</tr>
<tr>
<td>Observer</td>
<td>Changed</td>
</tr>
<tr>
<td></td>
<td>Create</td>
</tr>
<tr>
<td></td>
<td>RangeCheck</td>
</tr>
</tbody>
</table>
See Also

Resamplers Example
TCubicKernel
Hierarchy

TPersistent

| TCustomKernel
**Description**

This class implements a reconstruction filter described by a cubic polynomial. The formula for the filter is

\[
h(x) = \begin{cases} 
(a + 2)|x|^3 - (a + 3)|x|^2 + 1 & 0 \leq |x| < 1 \\
ax^3 - 5ax^2 + 8ax - 4a & 1 \leq |x| < 2 \\
0 & 2 \leq |x|
\end{cases}
\]

Cubic filtering is used commonly for high-quality resampling.

**Reference**

<table>
<thead>
<tr>
<th>Properties</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>In TCubicKernel:</td>
<td></td>
</tr>
<tr>
<td>Coeff</td>
<td>Create</td>
</tr>
<tr>
<td></td>
<td>Filter</td>
</tr>
<tr>
<td></td>
<td>GetWidth</td>
</tr>
<tr>
<td></td>
<td>RangeCheck</td>
</tr>
<tr>
<td>In TCustomKernel:</td>
<td></td>
</tr>
<tr>
<td>Observer</td>
<td>Changed</td>
</tr>
</tbody>
</table>
See Also

Resamplers Example
TCustomGaugeBar
Hierarchy

TCustomControl

| TArrowBar
**Description**
A base class for `TGaugeBar` objects.
Behavior and appearance of `TCustomGaugeBar` is very similar to standard
`TScrollBar`, defined in StdCtrls.pas.

**Reference**

<table>
<thead>
<tr>
<th>Properties</th>
<th>Methods</th>
<th>Events</th>
</tr>
</thead>
</table>
| In `TCustomGaugeBar`:
| `HandleSize`       | `Create`     |              |
| `LargeChange`      | `Paint`      |              |
| `Max`              |              |              |
| `Min`              |              |              |
| `Position`         |              |              |
| `SmallChange`      |              |              |
| In `TArrowBar`:
| `Backgnd`          | `Destroy`    | `OnChange`   |
| `BorderStyle`      | `OnChange`   | `OnUserChange` |
| `ButtonSize`       |              |              |
| `HandleColor`      |              |              |
| `Kind`             |              |              |
| `ShowArrows`       |              |              |
| `ShowHandleGrip`   |              |              |
| `Style`            |              |              |
See Also

TGaugeBar
TCustomImage32
Hierarchy

TCustomControl
  |
  TCustomPaintBox32
Description

A visual control capable of displaying bitmap images.

**TCustomImage32** is a base class for **TImage32**. In addition to **TCustomPaintBox32** behavior, it contains an easy-manageable bitmap and a collection of layers.

It introduces several properties to determine how the bitmap image is displayed within the boundaries of the control.

For more information on how to use **TCustomImage32** and its descendants, see [Using TImage32](#).

Reference

```
<table>
<thead>
<tr>
<th>Properties</th>
<th>Methods</th>
<th>Events</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>In TCustomImage32:</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bitmap</td>
<td>BeginUpdate</td>
<td>OnBitmapResize</td>
</tr>
<tr>
<td>BitmapAlign</td>
<td>BitmapToControl</td>
<td>OnChange</td>
</tr>
<tr>
<td>Layers</td>
<td>Changed</td>
<td>OnGDIOverlay</td>
</tr>
<tr>
<td>OffsetHorz</td>
<td>ControlToBitmap</td>
<td>OnInitStages</td>
</tr>
<tr>
<td>OffsetVert</td>
<td>Create</td>
<td>OnMouseDown</td>
</tr>
<tr>
<td>PaintStages</td>
<td>Destroy</td>
<td>OnMouseMove</td>
</tr>
<tr>
<td>Scale</td>
<td>EndUpdate</td>
<td>OnMouseUp</td>
</tr>
<tr>
<td>ScaleMode</td>
<td>ExecBitmapFrame</td>
<td>OnPaintStage</td>
</tr>
<tr>
<td>ScaleX</td>
<td>ExecClearBackgnd</td>
<td>OnPixelCombine</td>
</tr>
<tr>
<td>ScaleY</td>
<td>ExecClearBuffer</td>
<td>OnScaleChange</td>
</tr>
<tr>
<td></td>
<td>ExecControlFrame</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ExecCustom</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ExecDrawBitmap</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ExecDrawLayers</td>
<td></td>
</tr>
<tr>
<td></td>
<td>GetPictureRect</td>
<td></td>
</tr>
<tr>
<td></td>
<td>GetPictureSize</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Resize</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SetupBitmap</td>
<td></td>
</tr>
<tr>
<td><strong>In TCustomPaintBox32:</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Buffer</td>
<td>DoPaintBuffer</td>
<td>OnMouseEnter</td>
</tr>
<tr>
<td>BufferOversize</td>
<td>Flush</td>
<td>OnMouseLeave</td>
</tr>
<tr>
<td>BufferValid</td>
<td>ForceFullInvalidate</td>
<td></td>
</tr>
<tr>
<td>Options</td>
<td>GetViewportRect</td>
<td></td>
</tr>
<tr>
<td>RepaintMode</td>
<td>Invalidate</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Loaded</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SetBounds</td>
<td></td>
</tr>
</tbody>
</table>
```
See Also

Image32 Example, ImgView and Layers Example, Sprites Example, TCustomPaintBox32, TImage32, Using TImage32
Hierarchy

TCustomControl
  |
  TCustomPaintBox32
  |
  TCustomImage32
**Description**

*TCustomImgView32* is an extension of *TCustomImage32*, which additionally provides a couple of scroll bars to simplify operations with images larger than the control area.

Most of its behavior is inherited from *TCustomImage32*, however some properties are obsolete:

- **BitmapAlign** is no longer used, since the control is always in *baCustom* mode;
- **ScaleMode** is no longer used, the control is always in *smScale* mode;

For more information on how to use *TCustomImgView32* and its descendants, see [*Using TImage32*](#).

**Reference**

<table>
<thead>
<tr>
<th>Properties</th>
<th>Methods</th>
<th>Events</th>
</tr>
</thead>
<tbody>
<tr>
<td>In <em>TCustomImgView32</em>:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Centered</td>
<td>Create</td>
<td>OnScroll</td>
</tr>
<tr>
<td>OverSize</td>
<td>Destroy</td>
<td></td>
</tr>
<tr>
<td>ScrollBars</td>
<td>GetViewportRect</td>
<td></td>
</tr>
<tr>
<td>SizeGrip</td>
<td>Loaded</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>In <em>TCustomImage32</em>:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bitmap</td>
<td>BeginUpdate</td>
<td>OnBitmapResize</td>
</tr>
<tr>
<td>BitmapAlign</td>
<td>BitmapToControl</td>
<td>OnChange</td>
</tr>
<tr>
<td>Layers</td>
<td>Changed</td>
<td>OnGDIOverlay</td>
</tr>
<tr>
<td>OffsetHorz</td>
<td>ControlToBitmap</td>
<td>OnInitStages</td>
</tr>
<tr>
<td>OffsetVert</td>
<td>EndUpdate</td>
<td>OnMouseDown</td>
</tr>
<tr>
<td>PaintStages</td>
<td>ExecBitmapFrame</td>
<td>OnMouseMove</td>
</tr>
<tr>
<td>Scale</td>
<td>ExecClearBackgnd</td>
<td>OnMouseUp</td>
</tr>
<tr>
<td>ScaleMode</td>
<td>ExecClearBuffer</td>
<td>OnPaintStage</td>
</tr>
<tr>
<td>ScaleX</td>
<td>ExecControlFrame</td>
<td>OnPixelCombine</td>
</tr>
<tr>
<td>ScaleY</td>
<td>ExecCustom</td>
<td>OnScaleChange</td>
</tr>
<tr>
<td></td>
<td>ExecDrawBitmap</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ExecDrawLayers</td>
<td></td>
</tr>
<tr>
<td></td>
<td>GetPictureRect</td>
<td></td>
</tr>
<tr>
<td></td>
<td>GetPictureSize</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SetupBitmap</td>
<td></td>
</tr>
</tbody>
</table>

In *TCustomPaintBox32*: |
<table>
<thead>
<tr>
<th>Buffer</th>
<th>DoPaintBuffer</th>
<th>OnMouseEnter</th>
</tr>
</thead>
<tbody>
<tr>
<td>BufferOversize</td>
<td>Flush</td>
<td>OnMouseLeave</td>
</tr>
<tr>
<td>BufferValid</td>
<td>ForceFullInvalidate</td>
<td></td>
</tr>
<tr>
<td>Options</td>
<td>Invalidate</td>
<td></td>
</tr>
<tr>
<td>RepaintMode</td>
<td>SetBounds</td>
<td></td>
</tr>
</tbody>
</table>
See Also

TCustomImage32, Using TImage32
TCustomKernel
Hierarchy

TPersistent
Description

The **TCustomKernel** class is an ancestor class for reconstruction filters. A reconstruction filter is used when resampling discretized data (for example, when resizing an image).

Reference

<table>
<thead>
<tr>
<th>Properties</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Observer</strong></td>
<td><strong>Changed</strong></td>
</tr>
<tr>
<td><strong>Create</strong></td>
<td><strong>Filter</strong></td>
</tr>
<tr>
<td><strong>GetWidth</strong></td>
<td><strong>RangeCheck</strong></td>
</tr>
</tbody>
</table>
See Also

Resamplers Example

Copyright ©2000-2007 Alex Denisov and Contributors - Graphics32 v1.8.3 - Build on 4-March-2007
TCustomLayer
Hierarchy

TPersistent
  |
TNotifiablePersistent
**Description**

*TCustomLayer* is the base class for all layers. It does not perform any drawing itself. Instead it calls an *OnPaint* event. Applications must provide a handler for this event in order to paint the layer.

**Reference**

<table>
<thead>
<tr>
<th>Properties</th>
<th>Methods</th>
<th>Events</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>In TCustomLayer:</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cursor</td>
<td>BringToFront</td>
<td>OnHitTest</td>
</tr>
<tr>
<td>Index</td>
<td>Changed</td>
<td>OnMouseDown</td>
</tr>
<tr>
<td>LayerCollection</td>
<td>Create</td>
<td>OnMouseMove</td>
</tr>
<tr>
<td>LayerOptions</td>
<td>Destroy</td>
<td>OnMouseUp</td>
</tr>
<tr>
<td>LayerStates</td>
<td>HitTest</td>
<td>OnPaint</td>
</tr>
<tr>
<td>MouseEventArgs</td>
<td>SendToBack</td>
<td></td>
</tr>
<tr>
<td>Tag</td>
<td>Update</td>
<td></td>
</tr>
<tr>
<td>Visible</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>In TNotifiablePersistent:</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>UpdateCount</td>
<td>BeginUpdate</td>
<td>OnChange</td>
</tr>
<tr>
<td></td>
<td>EndUpdate</td>
<td></td>
</tr>
</tbody>
</table>
See Also

TCustomLayer, Using Layers
Hierarchy

TPersistent
|-- TNotifiablePersistent
|-- TThreadPersistent
### Description

**TCustomMap** is a common ancestor for objects that hold 2D arrays of data.

### Reference

<table>
<thead>
<tr>
<th>Properties</th>
<th>Methods</th>
<th>Events</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>In TCustomMap:</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Height</td>
<td>Delete</td>
<td>OnResize</td>
</tr>
<tr>
<td>Width</td>
<td>Empty</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Resized</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SetSize</td>
<td></td>
</tr>
<tr>
<td><strong>In TThreadPersistent:</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LockCount</td>
<td>Lock</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Unlock</td>
<td></td>
</tr>
<tr>
<td><strong>In TNotifiablePersistent:</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>UpdateCount</td>
<td>BeginUpdate</td>
<td>OnChange</td>
</tr>
<tr>
<td></td>
<td>Changed</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Create</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Destroy</td>
<td></td>
</tr>
<tr>
<td></td>
<td>EndUpdate</td>
<td></td>
</tr>
</tbody>
</table>

Copyright ©2000-2007 Alex Denisov and Contributors - Graphics32 v1.8.3 - Build on 4-March-2007
Hierarchy

TCustomControl
**Description**

`TCustomPaintBox32` is a common ancestor for `TPaintBox32` and `TCustomImage32` objects.

In addition to properties, inherited from `TCustomControl`, it provides optimized double buffering. The back buffer is a `TBitmap32` object, which stores an image data before flushing it to the screen.

Since `TCustomPaintBox32` is double buffered, there is no need to redraw its contents every time the control receives WM_PAINT message, as with standard `TPaintBox`. The repainting is still required when the control is resized.

**Reference**

<table>
<thead>
<tr>
<th>Properties</th>
<th>Methods</th>
<th>Events</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Buffer</code></td>
<td><code>Destroy</code></td>
<td><code>OnMouseEnter</code></td>
</tr>
<tr>
<td><code>BufferOversize</code></td>
<td><code>DoPaintBuffer</code></td>
<td><code>OnMouseLeave</code></td>
</tr>
<tr>
<td><code>BufferValid</code></td>
<td><code>Flush</code></td>
<td></td>
</tr>
<tr>
<td><code>Options</code></td>
<td><code>ForceFullInvalidate</code></td>
<td></td>
</tr>
<tr>
<td><code>RepaintMode</code></td>
<td><code>GetViewportRect</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>Invalidate</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>Loaded</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>Resize</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>SetBounds</code></td>
<td></td>
</tr>
</tbody>
</table>
See Also

TBitmap32, TCustomImage32, TPaintBox32
TCustomPolygonFiller
Hierarchy

TObject
**Description**

In addition to the standard color fillings for polygons, Graphics32 provides the ability to create new custom polygon fillers based on the TCustomPolygonFiller class or a `TFillLineEvent`-based callback.

If you want to create your own polygon filler, subclass from this class and override the `GetFillLine` method.

All classes based on TCustomPolygonFiller can be used with the `DrawFill` and `Draw` methods of `TPolygon32` or with the `Polygon` and `PolyPolygon` routines.

*Note:* Don't use this class directly as it is partly abstract.

**Reference**

<table>
<thead>
<tr>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>In TCustomPolygonFiller:</td>
</tr>
<tr>
<td><code>GetFillLine</code></td>
</tr>
</tbody>
</table>
See Also

TPolygon32.Draw, TPolygon32.DrawFill, Polygon, PolyPolygon, TBitmapPolygonFiller, TFillLineEvent, TPolygon32
TCustomRangeBar
Hierarchy

TCustomControl

| TArrowBar |
Description
A base class for TRangeBar objects.
Behavior and appearance of TCustomRangeBar is similar to standard TControlScrollBar, defined in Forms.pas, however it does not use the common control library.

Although range bar may operate as a stand-alone control, most of the time you will use it in conjunction with another control, which has a scrollable region:

Range bars are less limited compared to TControlScrollBar. For example, the size of the range bar can be different from the size of the scrollable control and it may be positioned anywhere in the form. Another difference is that range bars do not hide themselves when scrolling range is less than or equal to the size of the scrollable area, they just become disabled.

In addition, range bars have nothing to do with scrollable control's client area. They neither shift client origin, nor determine their range automatically.

Reference

<table>
<thead>
<tr>
<th>Properties</th>
<th>Methods</th>
<th>Events</th>
</tr>
</thead>
<tbody>
<tr>
<td>In TCustomRangeBar:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Centered</td>
<td>Create</td>
<td></td>
</tr>
<tr>
<td>Increment</td>
<td>Paint</td>
<td></td>
</tr>
<tr>
<td>Position</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Range</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Window</td>
<td></td>
<td></td>
</tr>
<tr>
<td>In TArrowBar:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Backgnd</td>
<td>Destroy</td>
<td>OnChange</td>
</tr>
<tr>
<td>BorderStyle</td>
<td></td>
<td>OnUserChange</td>
</tr>
<tr>
<td>ButtonSize</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HandleColor</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Kind</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ShowArrows</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ShowHandleGrip</td>
<td>Style</td>
<td></td>
</tr>
<tr>
<td>----------------</td>
<td>-------</td>
<td></td>
</tr>
</tbody>
</table>


See Also

TRangeBar

Copyright ©2000-2007 Alex Denisov and Contributors - Graphics32 v1.8.3 - Build on 4-March-2007
TCustomResampler
Hierarchy

TPersistent
  |
  TNotifiablePersistent
  |
  TCustomSampler
**Description**

**TCustomResampler** is a common ancestor for objects that resamples 2D arrays of data.

**Reference**

<table>
<thead>
<tr>
<th>Properties</th>
<th>Methods</th>
<th>Events</th>
</tr>
</thead>
<tbody>
<tr>
<td>In TCustomResampler:</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Width</strong></td>
<td><strong>Resample</strong></td>
<td></td>
</tr>
<tr>
<td>In TCustomSampler:</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>FinalizeSampling</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>GetSample</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>GetSampleBounds</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>HasBounds</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>PrepareSampling</strong></td>
<td></td>
</tr>
<tr>
<td>In TNotifyablePersistent:</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>UpdateCount</strong></td>
<td><strong>BeginUpdate</strong></td>
<td><strong>OnChange</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Changed</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Create</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Destroy</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>EndUpdate</strong></td>
<td></td>
</tr>
</tbody>
</table>
See Also

TBitmap32Resampler
TCustomSampler
Hierarchy

TPersistent

| TNotifiablePersistent
**Description**

**TCustomSampler** is a common ancestor for objects that can provide color samples.

**Reference**

<table>
<thead>
<tr>
<th>Properties</th>
<th>Methods</th>
<th>Events</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>FinalizeSampling</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>GetSample</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>GetSampleBounds</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>HasBounds</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>PrepareSampling</strong></td>
<td></td>
</tr>
</tbody>
</table>

In TCustomSampler:

<table>
<thead>
<tr>
<th>Updates</th>
<th>Method</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>BeginUpdate</strong></td>
<td><strong>OnChange</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Changed</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Create</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Destroy</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>EndUpdate</strong></td>
<td></td>
</tr>
</tbody>
</table>
See Also

Sampling and Rasterization
TDilater
Hierarchy

TPersistent
  |- TNotifiablePersistent
  |
  TCustomSampler
  |
  TNestedSampler
  |
  TKernelSampler
  |
  TMorphologicalSampler
**Description**

**TDilater** is a nested sampler for performing morphological dilation. This operation is defined as

\[(f \oplus b)(s, t) = \max_{x,y} \{ f(s - x, t - y) + b(x, y) \}\]

where \(f\) is a sampler and \(b\) is a structuring element. By performing morphological dilation, high intensity image features will become expanded. This is the opposite effect to morphological **dilation**. The structuring element is defined by the **Kernel** property of the class. See topic about **Sampling and Rasterization** for more info.

**Reference**

<table>
<thead>
<tr>
<th>Properties</th>
<th>Methods</th>
<th>Events</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>In TKernelSampler:</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>CenterX</strong></td>
<td><strong>Create</strong></td>
<td></td>
</tr>
<tr>
<td><strong>CenterY</strong></td>
<td><strong>Destroy</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Kernel</strong></td>
<td><strong>GetSample</strong></td>
<td></td>
</tr>
<tr>
<td><strong>In TNestedSampler:</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Sampler</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>In TCustomSampler:</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>FinalizeSampling</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>GetSampleBounds</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>HasBounds</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>PrepareSampling</strong></td>
<td></td>
</tr>
<tr>
<td><strong>In TNotifyablePersistent:</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>UpdateCount</strong></td>
<td><strong>BeginUpdate</strong></td>
<td><strong>OnChange</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Changed</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>EndUpdate</strong></td>
<td></td>
</tr>
</tbody>
</table>
See Also

Sampling and Rasterization, TDilater
TDisturbanceTransformation
Hierarchy

TObject

TTransformation
Description

This transformation class implements a randomizing displacement. The scaling of the randomization can be controlled from the Disturbance property. The effect of the transformation is visualized in the following image:

Reference

<table>
<thead>
<tr>
<th>Properties</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>In TDisturbance</td>
<td></td>
</tr>
<tr>
<td>Disturbance</td>
<td></td>
</tr>
<tr>
<td>In TTransformation:</td>
<td></td>
</tr>
<tr>
<td>SrcRect</td>
<td>GetTransformedBounds</td>
</tr>
<tr>
<td></td>
<td>HasTransformedBounds</td>
</tr>
<tr>
<td></td>
<td>ReverseTransform</td>
</tr>
<tr>
<td></td>
<td>Transform</td>
</tr>
</tbody>
</table>

Copyright ©2000-2007 Alex Denisov and Contributors - Graphics32 v1.8.3 - Build on 4-March-2007
TDraftResampler
Hierarchy

TPersistent
  \|-- TNotifiablePersistent
  \|-- TCustomSampler
  \|-- TCustomResampler
  \|-- TBitmap32Resampler
  \|-- TLinearResampler
**Description**

This class implements an algorithm for fast downsampling. The result is better than nearest neighbor interpolation, but it is not quite as good as linear resampling. Fast downsampling is very useful for generation of thumbnail images.

For upsampling the same method is used as in **TLinearResampler**.

**Reference**

<table>
<thead>
<tr>
<th>Properties</th>
<th>Methods</th>
<th>Events</th>
</tr>
</thead>
<tbody>
<tr>
<td>In TLinearResampler:</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Create</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Destroy</td>
<td></td>
</tr>
<tr>
<td></td>
<td>GetSample</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PrepareSampling</td>
<td></td>
</tr>
<tr>
<td>In TBitmap32Resampler:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bitmap</td>
<td>Changed</td>
<td></td>
</tr>
<tr>
<td>PixelAccessMode</td>
<td>GetSampleBounds</td>
<td></td>
</tr>
<tr>
<td>TransformerClass</td>
<td>HasBounds</td>
<td></td>
</tr>
<tr>
<td>In TCustomResampler:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Width</td>
<td>Resample</td>
<td></td>
</tr>
<tr>
<td>In TCustomSampler:</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>FinalizeSampling</td>
<td></td>
</tr>
<tr>
<td>In TNotifiablePersistent:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>UpdateCount</td>
<td>BeginUpdate</td>
<td>OnChange</td>
</tr>
<tr>
<td></td>
<td>EndUpdate</td>
<td></td>
</tr>
</tbody>
</table>
See Also
Resamplers Example, TLinearResampler
TEroder
Hierarchy

TPersistent
  └ TNotifiablePersistent
  └ TCustomSampler
  └ TNestedSampler
  └ TKernelSampler
  └ TMorphologicalSampler
**Description**

**TEroder** is a nested sampler for performing morphological erosion. This operation is defined as

\[
(f \oplus b)(s, t) = \max_{x,y} \{ f(s - x, t - y) + b(x, y) \}
\]

where \( f \) is a *sampler* and \( b \) is a *structuring element*. Unlike dilation, which expands the image features, erosion will cause the high intensity image features to become thinner. The structuring element is defined by the **Kernel** property of the class. See topic about **Sampling and Rasterization** for more info.

**Reference**

<table>
<thead>
<tr>
<th>Properties</th>
<th>Methods</th>
<th>Events</th>
</tr>
</thead>
<tbody>
<tr>
<td>In <strong>TEroder</strong>:</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Create</td>
<td></td>
</tr>
<tr>
<td>In <strong>TKernelSampler</strong>:</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>CenterX</strong></td>
<td>Destroy</td>
<td></td>
</tr>
<tr>
<td><strong>CenterY</strong></td>
<td>GetSample</td>
<td></td>
</tr>
<tr>
<td><strong>Kernel</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>In <strong>TNestedSampler</strong>:</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Sampler</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>In <strong>TCustomSampler</strong>:</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>FinalizeSampling</td>
<td></td>
</tr>
<tr>
<td></td>
<td>GetSampleBounds</td>
<td></td>
</tr>
<tr>
<td></td>
<td>HasBounds</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PrepareSampling</td>
<td></td>
</tr>
<tr>
<td>In <strong>TNotifiablePersistent</strong>:</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>UpdateCount</strong></td>
<td>BeginUpdate</td>
<td>OnChange</td>
</tr>
<tr>
<td></td>
<td>Changed</td>
<td></td>
</tr>
<tr>
<td></td>
<td>EndUpdate</td>
<td></td>
</tr>
</tbody>
</table>
See Also

Resamplers Example, Sampling and Rasterization, T Dilater
TExpander
Hierarchy

TPersistent
  \- TNotifiablePersistent
  \- TCustomSampler
  \- TNestedSampler
  \- TKernelSampler
**Description**

**TExpander** implements a neighborhood operation similar to morphological dilation. However, unlike ordinary dilation, where the structuring element is added to the color samples, the expand operation will multiply the weights. We define this operator as

\[ (f \otimes b)(s, t) = \max_{x,y} \{ f(s-x, t-y) b(x, y) \} \]

where \( f \) is a sampler and \( b \) is a structuring element. The structuring element is defined by the **Kernel** property of the class. See topic about **Sampling and Rasterization** for more info.

**Reference**

<table>
<thead>
<tr>
<th>Properties</th>
<th>Methods</th>
<th>Events</th>
</tr>
</thead>
<tbody>
<tr>
<td>In TKernelSampler:</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>CenterX</strong></td>
<td><strong>Create</strong></td>
<td></td>
</tr>
<tr>
<td><strong>CenterY</strong></td>
<td><strong>Destroy</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Kernel</strong></td>
<td><strong>GetSample</strong></td>
<td></td>
</tr>
<tr>
<td>In TNestedSampler:</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Sampler</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>In TCustomSampler:</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>FinalizeSampling</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>GetSampleBounds</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>HasBounds</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>PrepareSampling</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>In TNotifyablePersistent:</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>UpdateCount</strong></td>
<td><strong>BeginUpdate</strong></td>
<td><strong>OnChange</strong></td>
</tr>
<tr>
<td><strong>Changed</strong></td>
<td><strong>EndUpdate</strong></td>
<td></td>
</tr>
</tbody>
</table>
See Also

Sampling and Rasterization, TDilatex
TFishEyeTransformation
Hierarchy

 TObject
  |  TTransformation
**Description**

The fisheye transformation implements a transformation similar to wideangle lens distortion ("fisheyes"). The effect of the transformation can be seen in the image below. For illustrative purposes the distortion were amplified by rasterizing the transformation to the vector map of `TRemapTransformation` and adjusting the `scale` factors of the latter.

![Image of fisheye transformation](image)

**Reference**

<table>
<thead>
<tr>
<th>Properties</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>SrcRect</code></td>
<td><code>GetTransformedBounds</code></td>
</tr>
<tr>
<td></td>
<td><code>HasTransformedBounds</code></td>
</tr>
<tr>
<td></td>
<td><code>ReverseTransform</code></td>
</tr>
<tr>
<td></td>
<td><code>Transform</code></td>
</tr>
</tbody>
</table>
See Also

RasterizeTransformation, TRemapTransformation.Scale, TRemapTransformation
TGaugeBar
Hierarchy

TCustomControl
  |
  TArrowBar
  |
  TCustomGaugeBar
**Description**

*TGaugeBar* is a control similar to standard *TScrollBar*.

*TGaugeBar* implements the generic behavior introduced in *TCustomGaugeBar*. It publishes some inherited properties from *TCustomGaugeBar*, but does not introduce any new behavior.

**Reference**

<table>
<thead>
<tr>
<th>Properties</th>
<th>Methods</th>
<th>Events</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>In TCustomGaugeBar:</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HandleSize</td>
<td>Create</td>
<td></td>
</tr>
<tr>
<td>LargeChange</td>
<td>Paint</td>
<td></td>
</tr>
<tr>
<td>Max</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Min</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Position</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SmallChange</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>In TArrowBar:</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Backgnd</td>
<td>Destroy</td>
<td>OnChange</td>
</tr>
<tr>
<td>BorderStyle</td>
<td></td>
<td>OnUserChange</td>
</tr>
<tr>
<td>ButtonSize</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HandleColor</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Kind</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ShowArrows</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ShowHandleGrip</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Style</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
See Also

TCustomGaugeBar
T Gaussian Kernel
Hierarchy

TPersistent
  |
  TCustomKernel
  |
  TWindowedSincKernel
**Description**

This class implements a *Sinc* filter constrained by a *Gaussian* window function.

**Reference**

<table>
<thead>
<tr>
<th>Properties</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>In T GaussianKernel:</td>
<td></td>
</tr>
<tr>
<td>Sigma</td>
<td>Create</td>
</tr>
<tr>
<td>In T WindowedSincKernel:</td>
<td></td>
</tr>
<tr>
<td>Width</td>
<td>Filter</td>
</tr>
<tr>
<td></td>
<td>GetWidth</td>
</tr>
<tr>
<td></td>
<td>SetWidth</td>
</tr>
<tr>
<td>In T CustomKernel:</td>
<td></td>
</tr>
<tr>
<td>Observer</td>
<td>Changed</td>
</tr>
<tr>
<td></td>
<td>RangeCheck</td>
</tr>
</tbody>
</table>
See Also

Resamplers Example
THammingKernel
Hierarchy

TPersistent
  |
  TCustomKernel
**Description**

This class implements a *Sinc* filter constrained by the *Hamming* window function.

**Reference**

<table>
<thead>
<tr>
<th>Properties</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Observer</strong></td>
<td><strong>Changed</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Create</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Filter</strong></td>
</tr>
<tr>
<td></td>
<td><strong>GetWidth</strong></td>
</tr>
<tr>
<td></td>
<td><strong>RangeCheck</strong></td>
</tr>
</tbody>
</table>
See Also

Resamplers Example
THannKernel
Hierarchy

TPersistent

| TCustomKernel
**Description**

This class implements a *Sinc* filter constrained by the *Hann* window function.

**Reference**

<table>
<thead>
<tr>
<th>Properties</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>In TCustomKernel:</td>
<td></td>
</tr>
<tr>
<td><strong>Observer</strong></td>
<td><strong>Changed</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Create</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Filter</strong></td>
</tr>
<tr>
<td></td>
<td><strong>GetWidth</strong></td>
</tr>
<tr>
<td></td>
<td><strong>RangeCheck</strong></td>
</tr>
</tbody>
</table>
See Also

Resamplers Example
Hierarchy

TPersistent

| TCustomKernel
Description
An implementation of the hermite kernel, including Bias and Tension parameters.

Reference

<table>
<thead>
<tr>
<th>Properties</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bias</td>
<td>Create</td>
</tr>
<tr>
<td>Tension</td>
<td>Filter</td>
</tr>
<tr>
<td></td>
<td>GetWidth</td>
</tr>
<tr>
<td></td>
<td>RangeCheck</td>
</tr>
<tr>
<td>Observer</td>
<td>Changed</td>
</tr>
</tbody>
</table>

In THermiteKernel:

In TCustomKernel:
See Also

Resamplers Example
TImage32
Hierarchy

TCustomControl
  └─ TCustomPaintBox32
  └─ TCustomImage32
**Description**

An image displaying control.

**TImage32** implements the generic behavior introduced in **TCustomImage32**. It publishes some inherited properties from **TCustomImage32**, but does not introduce any new behavior.

**TImage32** allows loading of its **Bitmap** property from supported image files in design-time using a property editor, associated with **TBitmap32** objects. The same property editor is activated when you double-click the control in design-time.

For more information on **TImage32**, see [Using TImage32](#).

**Reference**

<table>
<thead>
<tr>
<th>Properties</th>
<th>Methods</th>
<th>Events</th>
</tr>
</thead>
<tbody>
<tr>
<td>In <strong>TCustomImage32</strong>:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bitmap</td>
<td>BeginUpdate</td>
<td>OnBitmapResize</td>
</tr>
<tr>
<td>BitmapAlign</td>
<td>BitmapToControl</td>
<td>OnChange</td>
</tr>
<tr>
<td>Layers</td>
<td>Changed</td>
<td>OnGDIOverlay</td>
</tr>
<tr>
<td>OffsetHorz</td>
<td>ControlToBitmap</td>
<td>OnInitStages</td>
</tr>
<tr>
<td>OffsetVert</td>
<td>Create</td>
<td>OnMouseDown</td>
</tr>
<tr>
<td>PaintStages</td>
<td>Destroy</td>
<td>OnMouseMove</td>
</tr>
<tr>
<td>Scale</td>
<td>EndUpdate</td>
<td>OnMouseUp</td>
</tr>
<tr>
<td>ScaleMode</td>
<td>ExecBitmapFrame</td>
<td>OnPaintStage</td>
</tr>
<tr>
<td>ScaleX</td>
<td>ExecClearBackgnd</td>
<td>OnPixelCombine</td>
</tr>
<tr>
<td>ScaleY</td>
<td>ExecClearBuffer</td>
<td>OnScaleChange</td>
</tr>
<tr>
<td></td>
<td>ExecControlFrame</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ExecCustom</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ExecDrawBitmap</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ExecDrawLayers</td>
<td></td>
</tr>
<tr>
<td></td>
<td>GetPictureRect</td>
<td></td>
</tr>
<tr>
<td></td>
<td>GetPictureSize</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Resize</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SetupBitmap</td>
<td></td>
</tr>
<tr>
<td>In <strong>TCustomPaintBox32</strong>:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Buffer</td>
<td>DoPaintBuffer</td>
<td>OnMouseEnter</td>
</tr>
<tr>
<td>BufferOversize</td>
<td>Flush</td>
<td>OnMouseLeave</td>
</tr>
<tr>
<td>BufferValid</td>
<td>ForceFullInvalidate</td>
<td></td>
</tr>
<tr>
<td>Options</td>
<td>GetViewportRect</td>
<td></td>
</tr>
<tr>
<td>RepaintMode</td>
<td>Invalidate</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Loaded</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SetBounds</td>
<td></td>
</tr>
</tbody>
</table>
See Also

Image32 Example, ImgView and Layers Example, Sprites Example, TBitmap32, TCustomImage32, Using TImage32
Hierarchy

TCustomControl
  \- TCustomPaintBox32
  \- TCustomImage32
  \- TCustomImgView32
Description
An image displaying control with scroll bars.

TImgView32 implements the generic behavior introduced in TCustomImgView32. It publishes some inherited properties from TCustomImgView32, but does not introduce any new behavior.

Reference

<table>
<thead>
<tr>
<th>Properties</th>
<th>Methods</th>
<th>Events</th>
</tr>
</thead>
<tbody>
<tr>
<td>In TCustomImgView32:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Centered</td>
<td>Create</td>
<td>OnScroll</td>
</tr>
<tr>
<td>OverSize</td>
<td>Destroy</td>
<td></td>
</tr>
<tr>
<td>ScrollBars</td>
<td>GetViewportRect</td>
<td></td>
</tr>
<tr>
<td>SizeGrip</td>
<td>Loaded</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Resize</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Scroll</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ScrollToCenter</td>
<td></td>
</tr>
<tr>
<td>In TCustomImage32:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bitmap</td>
<td>BeginUpdate</td>
<td>OnBitmapResize</td>
</tr>
<tr>
<td>BitmapAlign</td>
<td>BitmapToControl</td>
<td>OnChange</td>
</tr>
<tr>
<td>Layers</td>
<td>Changed</td>
<td>OnGDIOverlay</td>
</tr>
<tr>
<td>OffsetHorz</td>
<td>ControlToBitmap</td>
<td>OnInitStages</td>
</tr>
<tr>
<td>OffsetVert</td>
<td>EndUpdate</td>
<td>OnMouseDown</td>
</tr>
<tr>
<td>PaintStages</td>
<td>ExecBitmapFrame</td>
<td>OnMouseMove</td>
</tr>
<tr>
<td>Scale</td>
<td>ExecClearBackgnd</td>
<td>OnMouseUp</td>
</tr>
<tr>
<td>ScaleMode</td>
<td>ExecClearBuffer</td>
<td>OnPaintStage</td>
</tr>
<tr>
<td>ScaleX</td>
<td>ExecControlFrame</td>
<td>OnPixelCombine</td>
</tr>
<tr>
<td>ScaleY</td>
<td>ExecCustom</td>
<td>OnScaleChange</td>
</tr>
<tr>
<td></td>
<td>ExecDrawBitmap</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ExecDrawLayers</td>
<td></td>
</tr>
<tr>
<td></td>
<td>GetPictureRect</td>
<td></td>
</tr>
<tr>
<td></td>
<td>GetPictureSize</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SetupBitmap</td>
<td></td>
</tr>
<tr>
<td>In TCustomPaintBox32:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Buffer</td>
<td>DoPaintBuffer</td>
<td>OnMouseEnter</td>
</tr>
<tr>
<td>BufferOversize</td>
<td>Flush</td>
<td>OnMouseLeave</td>
</tr>
<tr>
<td>BufferValid</td>
<td>ForceFullInvalid</td>
<td></td>
</tr>
<tr>
<td>Options</td>
<td>Invalidate</td>
<td></td>
</tr>
<tr>
<td>RepaintMode</td>
<td>SetBounds</td>
<td></td>
</tr>
</tbody>
</table>
See Also

TCustomImgView32
TIntegerMap
Hierarchy

TPersistent
  └── TNotifiablePersistent
  └── TThreadPersistent
  └── TCustomMap
Description

The **TIntegerMap** class implements a two dimensional matrix of integers. It may be useful for storing height map data, and in general data that requires or can be represented as 32 bit signed integers.

Reference

<table>
<thead>
<tr>
<th>Properties</th>
<th>Methods</th>
<th>Events</th>
</tr>
</thead>
<tbody>
<tr>
<td>In <strong>TIntegerMap</strong>:</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Bits</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>ValPtr</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Value</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>In <strong>TCustomMap</strong>:</td>
<td><strong>Delete</strong></td>
<td><strong>OnResize</strong></td>
</tr>
<tr>
<td><strong>Height</strong></td>
<td><strong>Empty</strong></td>
<td><strong>Resized</strong></td>
</tr>
<tr>
<td><strong>Width</strong></td>
<td></td>
<td><strong>SetSize</strong></td>
</tr>
<tr>
<td>In <strong>TThreadPersistent</strong>:</td>
<td><strong>Lock</strong></td>
<td></td>
</tr>
<tr>
<td><strong>LockCount</strong></td>
<td></td>
<td><strong>Unlock</strong></td>
</tr>
<tr>
<td>In <strong>TNotifyablePersistent</strong>:</td>
<td><strong>BeginUpdate</strong></td>
<td><strong>OnChange</strong></td>
</tr>
<tr>
<td><strong>UpdateCount</strong></td>
<td><strong>Changed</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Create</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Destroy</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>EndUpdate</strong></td>
<td></td>
</tr>
</tbody>
</table>
See Also
GR32_ByteMaps, TBooleanMap, TWordMap
TIVScrollProperties
Hierarchy

TPersistent
Description

**TIVScrollProperties** is an auxiliary class for accessing properties of scroll bars (**TCustomRangeBar** controls) that are hosted by the **TCustomImgView32** control. Basically, it just redirects writing and reading of properties to/from corresponding range bar controls, which allows changing their appearance at both design- and run-time.

Reference

<table>
<thead>
<tr>
<th>Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>In TIVScrollProperties:</td>
</tr>
<tr>
<td>Backgnd</td>
</tr>
<tr>
<td>BorderStyle</td>
</tr>
<tr>
<td>ButtonSize</td>
</tr>
<tr>
<td>HandleColor</td>
</tr>
<tr>
<td>Increment</td>
</tr>
<tr>
<td>ShowArrows</td>
</tr>
<tr>
<td>ShowHandleGrip</td>
</tr>
<tr>
<td>Size</td>
</tr>
<tr>
<td>Style</td>
</tr>
<tr>
<td>Visibility</td>
</tr>
</tbody>
</table>
See Also

TCustomImgView32.ScrollBars, TCustomImgView32, TCustomRangeBar
TKernelResampler
Hierarchy

TPersistent
  \- TNotifiablePersistent
    \- TCustomSampler
      \- TCustomResampler
        \- TBitmap32Resampler
**Description**

**TKernelResampler** is a general class for image resampling using arbitrary convolution filters. It implements a fast method for resizing images, by precomputing kernel weights in *mapping tables* or *coefficient bins.*

For single-sample look-ups, the **GetSample** method supports three different kernel modes:

- **kmDynamic** — kernel weights are computed on-the-fly;
- **kmTableNearest** — kernel weights are stored in a table, nearest-neighbor is used for the look-up;
- **kmTableLinear** — same as above, but uses linear interpolation for the look-up.

For further information, see topic about [Sampling and Rasterization](#).

**Reference**

<table>
<thead>
<tr>
<th>Properties</th>
<th>Methods</th>
<th>Events</th>
</tr>
</thead>
<tbody>
<tr>
<td>In TKernelResampler:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Kernel</td>
<td>Create</td>
<td></td>
</tr>
<tr>
<td>ClassName</td>
<td>Destroy</td>
<td></td>
</tr>
<tr>
<td>KernelMode</td>
<td>FinalizeSampling</td>
<td></td>
</tr>
<tr>
<td>TableSize</td>
<td>GetSample</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PrepareSampling</td>
<td></td>
</tr>
<tr>
<td>In TBitmap32Resampler:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bitmap</td>
<td>Changed</td>
<td></td>
</tr>
<tr>
<td>PixelAccessMode</td>
<td>GetSampleBounds</td>
<td></td>
</tr>
<tr>
<td>TransformerClass</td>
<td>HasBounds</td>
<td></td>
</tr>
<tr>
<td>In TCustomResampler:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Width</td>
<td>Resample</td>
<td></td>
</tr>
<tr>
<td>In TNotifyablePersistent:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>UpdateCount</td>
<td>BeginUpdate</td>
<td>OnChange</td>
</tr>
<tr>
<td></td>
<td>EndUpdate</td>
<td></td>
</tr>
</tbody>
</table>
See Also

Resamplers Example, Sampling and Rasterization
TKernelSampler
Hierarchy

TPersistent
  |
  TNotifyablePersistent
  |
  TCustomSampler
  |
  TNestedSampler
**Description**

TKernelSampler is a base class for samplers that compute an output sample by collecting a number of samples in a local region of the actual sample coordinate. Subsamples are collected from a regular sampling grid by storing the sample.

The size of the grid is equal to the size of the associated kernel. Descendants of this class should override the protected *UpdateBuffer* method.

**Reference**

<table>
<thead>
<tr>
<th>Properties</th>
<th>Methods</th>
<th>Events</th>
</tr>
</thead>
<tbody>
<tr>
<td>In TKernelSampler:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>centerX</td>
<td>Create</td>
<td></td>
</tr>
<tr>
<td>centerY</td>
<td>Destroy</td>
<td></td>
</tr>
<tr>
<td>kernel</td>
<td>GetSample</td>
<td></td>
</tr>
<tr>
<td>In TNestedSampler:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sampler</td>
<td></td>
<td></td>
</tr>
<tr>
<td>In TCustomSampler:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>finalizeSampling</td>
<td></td>
<td></td>
</tr>
<tr>
<td>getSampleBounds</td>
<td></td>
<td></td>
</tr>
<tr>
<td>hasBounds</td>
<td></td>
<td></td>
</tr>
<tr>
<td>prepareSampling</td>
<td></td>
<td></td>
</tr>
<tr>
<td>In TNotifiablePersistent:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>updateCount</td>
<td>beginUpdate</td>
<td>onChange</td>
</tr>
<tr>
<td>changed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>endUpdate</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Copyright ©2000-2007 Alex Denisov and Contributors - Graphics32 v1.8.3 - Build on 4-March-2007
TLanczosKernel
Hierarchy

TPersistent
  ↓
TCustomKernel
  ↓
TWindowedSincKernel
Description
This class implements the *Lanczos* filter function. It uses the a single lobe of the *sinc* filter as a window.

Reference

<table>
<thead>
<tr>
<th>Properties</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Width</strong></td>
<td><strong>Create</strong></td>
</tr>
<tr>
<td><strong>Filter</strong></td>
<td></td>
</tr>
<tr>
<td><strong>GetWidth</strong></td>
<td></td>
</tr>
<tr>
<td><strong>SetWidth</strong></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>In TWindowedSincKernel:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Observer</strong></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
See Also

Resamplers Example
TLayerCollection
Hierarchy
TPersistent
**Description**

**TLayerCollection** is a container for **TCustomLayer** objects and their descendants.

**Reference**

<table>
<thead>
<tr>
<th>Properties</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CoordXForm</strong></td>
<td><strong>Add</strong></td>
</tr>
<tr>
<td><strong>Count</strong></td>
<td><strong>Clear</strong></td>
</tr>
<tr>
<td><strong>GetViewportScale</strong></td>
<td><strong>Create</strong></td>
</tr>
<tr>
<td><strong>GetViewportShift</strong></td>
<td><strong>Delete</strong></td>
</tr>
<tr>
<td><strong>Items</strong></td>
<td><strong>Destroy</strong></td>
</tr>
<tr>
<td><strong>LocalToViewport</strong></td>
<td><strong>Insert</strong></td>
</tr>
<tr>
<td><strong>MouseEvents</strong></td>
<td></td>
</tr>
<tr>
<td><strong>MouseListener</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Owner</strong></td>
<td></td>
</tr>
<tr>
<td><strong>ViewportToLocal</strong></td>
<td></td>
</tr>
</tbody>
</table>
See Also

TCustomLayer, Using Layers

Copyright ©2000-2007 Alex Denisov and Contributors - Graphics32 v1.8.3 - Build on 4-March-2007
TLinearKernel
Hierarchy

TPersistent
  |
TCustomKernel
**Description**

This class implements a linear reconstruction filter. *Linear interpolation* is a first-degree method that passes a straight line through every two consecutive points of the input signal.

The filter is described by the following interpolation kernel:

\[
h(x) = \begin{cases} 
1 - |x| & 0 \leq |x| < 1 \\
0 & 1 \leq |x| 
\end{cases}
\]

In literature \( h(x) \) is often referred to as a *triangle filter, tent filter, roof function, Chateau function* or a *Bartlett window*.

**Reference**

<table>
<thead>
<tr>
<th>Properties</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>In TLinearKernel:</td>
<td>Filter</td>
</tr>
<tr>
<td></td>
<td>GetWidth</td>
</tr>
<tr>
<td>In TCustomKernel:</td>
<td>Observer</td>
</tr>
<tr>
<td></td>
<td>Changed</td>
</tr>
<tr>
<td></td>
<td>Create</td>
</tr>
<tr>
<td></td>
<td>RangeCheck</td>
</tr>
</tbody>
</table>
See Also

Resamplers Example
TLinearResampler
**Hierarchy**

TPersistent

| TNotifiablePersistent
| TCustomSampler
| TCustomResampler
| TBitmap32Resampler
Description

TLinearResampler implements a linear resampling algorithm. This algorithm is used only if the bounds of the destination bitmap is larger than the bounds of the source bitmap. Otherwise resampling is performed using the same method as in TKernelResampler.

Reference

<table>
<thead>
<tr>
<th>Properties</th>
<th>Methods</th>
<th>Events</th>
</tr>
</thead>
<tbody>
<tr>
<td>In TLinearResampler:</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Create</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Destroy</td>
<td></td>
</tr>
<tr>
<td></td>
<td>GetSample</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PrepareSampling</td>
<td></td>
</tr>
<tr>
<td>In TBitmap32Resampler:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bitmap</td>
<td>Changed</td>
<td></td>
</tr>
<tr>
<td>PixelAccessMode</td>
<td>GetSampleBounds</td>
<td></td>
</tr>
<tr>
<td>TransformerClass</td>
<td>HasBounds</td>
<td></td>
</tr>
<tr>
<td>In TCustomResampler:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Width</td>
<td>Resample</td>
<td></td>
</tr>
<tr>
<td>In TCustomSampler:</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>FinalizeSampling</td>
<td></td>
</tr>
<tr>
<td>In TNotifiablePersistent:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>UpdateCount</td>
<td>BeginUpdate</td>
<td>OnChange</td>
</tr>
<tr>
<td></td>
<td>EndUpdate</td>
<td></td>
</tr>
</tbody>
</table>
See Also

Kernel Example, Resampler Example, TKernelResampler
TMitchellKernel
Hierarchy

TPersistent

| TCustomKernel
**Description**
An implementation of a special case of the cubic filter described by Mitchell.

**Reference**

<table>
<thead>
<tr>
<th>Properties</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>In TMitchellKernel:</td>
<td></td>
</tr>
<tr>
<td>Filter</td>
<td></td>
</tr>
<tr>
<td>GetWidth</td>
<td></td>
</tr>
<tr>
<td>RangeCheck</td>
<td></td>
</tr>
<tr>
<td>In TCustomKernel:</td>
<td></td>
</tr>
<tr>
<td>Observer</td>
<td>Changed</td>
</tr>
<tr>
<td>Create</td>
<td></td>
</tr>
</tbody>
</table>
See Also

Kernel Example, Resampler Example
TMorphologicalSampler
Hierarchy

TPersistent
- TNotifiablePersistent
- TCustomSampler
- TNestedSampler
- TKernelSampler
Description
Abstract ancestor class for T Dilater and T Eroder. You should not instantiate this class itself.

Reference

<table>
<thead>
<tr>
<th>Properties</th>
<th>Methods</th>
<th>Events</th>
</tr>
</thead>
<tbody>
<tr>
<td>In TKernelSampler:</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>CenterX</strong></td>
<td><strong>Create</strong></td>
<td></td>
</tr>
<tr>
<td><strong>CenterY</strong></td>
<td><strong>Destroy</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Kernel</strong></td>
<td><strong>GetSample</strong></td>
<td></td>
</tr>
<tr>
<td>In TNestedSampler:</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Sampler</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>In TCustomSampler:</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>FinalizeSampling</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>GetSampleBounds</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>HasBounds</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>PrepareSampling</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>In TNotifyablePersistent:</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>UpdateCount</strong></td>
<td><strong>BeginUpdate</strong></td>
<td><strong>OnChange</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Changed</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>EndUpdate</strong></td>
<td></td>
</tr>
</tbody>
</table>
See Also

TEroder.Create, Kernel Example, Resampler Example, TDilater
TNearestResampler
Hierarchy

TPersistent
  └── TNotifiablePersistent
      └── TCustomSampler
          └── TCustomResampler
              └── TBitmap32Resampler
**Description**
This class implements a fast resampling method based on the nearest neighbor interpolation algorithm.

**Reference**

<table>
<thead>
<tr>
<th>Properties</th>
<th>Methods</th>
<th>Events</th>
</tr>
</thead>
<tbody>
<tr>
<td>In TNearestResampler:</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Create</td>
<td></td>
</tr>
<tr>
<td></td>
<td>GetSample</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PrepareSampling</td>
<td></td>
</tr>
<tr>
<td>In TBitmap32Resampler:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bitmap</td>
<td>Changed</td>
<td></td>
</tr>
<tr>
<td>PixelAccessMode</td>
<td>GetSampleBounds</td>
<td></td>
</tr>
<tr>
<td>TransformerClass</td>
<td>HasBounds</td>
<td></td>
</tr>
<tr>
<td>In TCustomResampler:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Width</td>
<td>Resample</td>
<td></td>
</tr>
<tr>
<td>In TCustomSampler:</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>FinalizeSampling</td>
<td></td>
</tr>
<tr>
<td>In TNotifiablePersistent:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>UpdateCount</td>
<td>BeginUpdate</td>
<td>OnChange</td>
</tr>
<tr>
<td></td>
<td>Destroy</td>
<td></td>
</tr>
<tr>
<td></td>
<td>EndUpdate</td>
<td></td>
</tr>
</tbody>
</table>
See Also

Kernel Example, Resampler Example
TNearestTransformer
Hierarchy

TPersistent
  | TNotifiablePersistent
  | TCustomSampler
  | TNestedSampler
  | TTransformer
**Description**

**Reference**

<table>
<thead>
<tr>
<th>Properties</th>
<th>Methods</th>
<th>Events</th>
</tr>
</thead>
<tbody>
<tr>
<td>In TNearestTransformer:</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>GetSample</td>
<td></td>
</tr>
<tr>
<td>In TTransformer:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Transformation</td>
<td>GetSampleBounds</td>
<td></td>
</tr>
<tr>
<td></td>
<td>HasBounds</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PrepareSampling</td>
<td></td>
</tr>
<tr>
<td>In TNestedSampler:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sampler</td>
<td>Create</td>
<td></td>
</tr>
<tr>
<td>In TCustomSampler:</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>FinalizeSampling</td>
<td></td>
</tr>
<tr>
<td>In TNotifiablePersistent:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>UpdateCount</td>
<td>BeginUpdate</td>
<td>OnChange</td>
</tr>
<tr>
<td></td>
<td>Changed</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Destroy</td>
<td></td>
</tr>
<tr>
<td></td>
<td>EndUpdate</td>
<td></td>
</tr>
</tbody>
</table>
See Also

Kernel Example, Resampler Example
TNestedSampler
Hierarchy

TPersistent
  
  TNotifiablePersistent

  TCustomSampler
**Description**

TNestedSampler is a base class for *chained* or *nested* samplers. Descendants of TNestedSampler should provide sampling methods that rely on the use of another sampler. This nested sampling mechanism can be used beneficially when implementing super samplers or samplers that transform coordinates or perform other intermediate sampling operations.

See the topic about Sampling and Rasterization for additional information.

**Reference**

<table>
<thead>
<tr>
<th>Properties</th>
<th>Methods</th>
<th>Events</th>
</tr>
</thead>
<tbody>
<tr>
<td>In TNestedSampler:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sampler</td>
<td>Create</td>
<td></td>
</tr>
<tr>
<td>In TCustomSampler:</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>FinalizeSampling</td>
<td></td>
</tr>
<tr>
<td></td>
<td>GetSample</td>
<td></td>
</tr>
<tr>
<td></td>
<td>GetSampleBounds</td>
<td></td>
</tr>
<tr>
<td></td>
<td>HasBounds</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PrepareSampling</td>
<td></td>
</tr>
<tr>
<td>In TNotifyablePersistent:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>UpdateCount</td>
<td>BeginUpdate</td>
<td>OnChange</td>
</tr>
<tr>
<td></td>
<td>Changed</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Destroy</td>
<td></td>
</tr>
<tr>
<td></td>
<td>EndUpdate</td>
<td></td>
</tr>
</tbody>
</table>
See Also

body, Sampling and Rasterization
TNotifiablePersistent
Hierarchy
TPersistent
### Description

**TNotifiablePersistent** extends the standard **TPersistent** class with change notification events. That is, it provides methods and events allowing its descendants to issue notification on their changes. For example, **TBitmap32** uses **OnChange** to notify its container (usually **TImage32** or **TBitmapLayer**) that it was modified and its data has to be repainted to the screen. **TNotifiablePersistent**, however, does not use or implement automatic change notification itself. This is done in descendants.

### Reference

<table>
<thead>
<tr>
<th>Properties</th>
<th>Methods</th>
<th>Events</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>UpdateCount</strong></td>
<td><strong>BeginUpdate</strong></td>
<td><strong>OnChange</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Changed</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Create</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Destroy</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>EndUpdate</strong></td>
<td></td>
</tr>
</tbody>
</table>
See Also

TBitmap32, TBitmapLayer, TImage32
TPaintBox32
Hierarchy

TCustomControl

| TCustomPaintBox32
Description

TPaintBox32 implements the generic behavior introduced in TCustomPaintBox32. It publishes some inherited properties from its ancestors and, in addition, it calls the OnPaintBuffer event when the buffer needs to be repainted.

Reference

<table>
<thead>
<tr>
<th>Properties</th>
<th>Methods</th>
<th>Events</th>
</tr>
</thead>
<tbody>
<tr>
<td>In TPaintBox32:</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Create</td>
<td>OnPaintBuffer</td>
</tr>
<tr>
<td></td>
<td>DoPaintBuffer</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Flush</td>
<td></td>
</tr>
<tr>
<td>In TCustomPaintBox32:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Buffer</td>
<td>Destroy</td>
<td>OnMouseEnter</td>
</tr>
<tr>
<td>BufferOversize</td>
<td>ForceFullInvalid</td>
<td>OnMouseLeave</td>
</tr>
<tr>
<td>BufferValid</td>
<td>GetViewportRect</td>
<td></td>
</tr>
<tr>
<td>Options</td>
<td>Invalidate</td>
<td></td>
</tr>
<tr>
<td>RepaintMode</td>
<td>Loaded</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Resize</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SetBounds</td>
<td></td>
</tr>
</tbody>
</table>
See Also

TCustomPaintBox32
TPaintStages
Hierarchy
TObject
**Description**

A dynamic list of paint stages.

This class is similar to standard *TList*, but instead of pointers, it holds an array of *TPaintStage* records.

**Reference**

<table>
<thead>
<tr>
<th>Properties</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Items</strong></td>
<td><strong>Add</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Clear</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Count</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Delete</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Destroy</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Insert</strong></td>
</tr>
</tbody>
</table>
See Also

Paint Stages, TCustomImage32.PaintStages, TPaintStage
TPatternSampler
Hierarchy

TPersistent
  \|-- TNotifiablePersistent
  \|-- TCustomSampler
  \|-- TNestedSampler
**Description**

**TPatternSampler** provides a mechanism for performing sampling according to a preinitialize sample pattern. This pattern is implemented as a matrix of TArrayOfFixedPoint. Each pixel in the rasterized output bitmap will have a designated set of sampling points specified by wrapping this sample pattern across the image.

**Reference**

<table>
<thead>
<tr>
<th>Properties</th>
<th>Methods</th>
<th>Events</th>
</tr>
</thead>
<tbody>
<tr>
<td>In TPatternSampler:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pattern</td>
<td>Destroy</td>
<td></td>
</tr>
<tr>
<td></td>
<td>GetSample</td>
<td></td>
</tr>
<tr>
<td>In TNestedSampler:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sampler</td>
<td>Create</td>
<td></td>
</tr>
<tr>
<td>In TCustomSampler:</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>FinalizeSampling</td>
<td></td>
</tr>
<tr>
<td></td>
<td>GetSampleBounds</td>
<td></td>
</tr>
<tr>
<td></td>
<td>HasBounds</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PrepareSampling</td>
<td></td>
</tr>
<tr>
<td>In TNotifyablePersistent:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>UpdateCount</td>
<td>BeginUpdate</td>
<td>OnChange</td>
</tr>
<tr>
<td></td>
<td>Changed</td>
<td></td>
</tr>
<tr>
<td></td>
<td>EndUpdate</td>
<td></td>
</tr>
</tbody>
</table>
See Also

Kernel Example, Point Types, Resampler Example

Copyright ©2000-2007 Alex Denisov and Contributors - Graphics32 v1.8.3 - Build on 4-March-2007
TPointerMap
Hierarchy
TObject
Description
TPointerMap implements a simple pointer to pointer map. It uses a hash-like structure internally, thus, data lookup is very fast compared to an iterative approach - especially if you have many entries in the map. However, it is also more limited than a list approach since it does not support index based operation. If you need to iterate through a pointer map please refer to TPointerMapIterator.

Reference

<table>
<thead>
<tr>
<th>Properties</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>Count</td>
<td>Add</td>
</tr>
<tr>
<td>Data</td>
<td>Clear</td>
</tr>
<tr>
<td></td>
<td>Contains</td>
</tr>
<tr>
<td></td>
<td>Find</td>
</tr>
<tr>
<td></td>
<td>Remove</td>
</tr>
</tbody>
</table>
See Also

TPointerMapIterator
TPointerMapIterator
Hierarchy

TObject
**Description**

TPointerMapIterator is an auxiliary class for iterating through pointer maps - an operation not directly supported by TPointerMap.

The iterator should be used in the following fashion:

```pascal
with TPointerMapIterator.Create(MyPointerMap) do
try
  while Next do
    begin
      // do something with Item and Data here...
    end;
finally
  Free;
end;
```

**Reference**

<table>
<thead>
<tr>
<th>Properties</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data</td>
<td>Next</td>
</tr>
<tr>
<td>Item</td>
<td></td>
</tr>
</tbody>
</table>

Copyright ©2000-2007 Alex Denisov and Contributors  -  Graphics32 v1.8.3  -  Build on 4-March-2007
TPolygon32
**Hierarchy**

TPersistent
  └ TNotifiablePersistent
    └ TThreadPersistent
**Description**

In addition to common polygon drawing functions, Graphics32 provides an auxiliary class **TPolygon32**, which may be used for storing and drawing polygons and polylines, allows drawing of lines and polylines more than 1 pixel wide and provides some morphological modifications of polygons.

**TPolygon32** is capable of storing more than a single polygon, it can represent a shape, outlines by several polygons, referred here as contours and specified in **Points** property. This allows rendering of complex shapes (with holes, etc.). In future versions we plan to implement direct TTF or Type1 font rendering.

Contour is considered as a closed polygon if the **Closed** property is *True*, otherwise it is treated as polyline. Each contour is, in turn, a dynamic array of **TFixedPoint**.

Polygons may be constructed by modifying other polygons using the **Grow** method or by adding sequentially new points with **Add** or starting new contours with **NewLine**.

**Reference**

<table>
<thead>
<tr>
<th>Properties</th>
<th>Methods</th>
<th>Events</th>
</tr>
</thead>
<tbody>
<tr>
<td>In TPolygon32:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Antialiased</td>
<td>Add</td>
<td></td>
</tr>
<tr>
<td>AntialiasMode</td>
<td>AddPoints</td>
<td></td>
</tr>
<tr>
<td>Closed</td>
<td>Assign</td>
<td></td>
</tr>
<tr>
<td>FillMode</td>
<td>Clear</td>
<td></td>
</tr>
<tr>
<td>Normals</td>
<td>ContainsPoint</td>
<td></td>
</tr>
<tr>
<td>Points</td>
<td>Create</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Destroy</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Draw</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DrawEdge</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DrawFill</td>
<td></td>
</tr>
<tr>
<td></td>
<td>GetBoundingRect</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Grow</td>
<td></td>
</tr>
<tr>
<td></td>
<td>NewLine</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Offset</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Outline</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Transform</td>
<td></td>
</tr>
<tr>
<td>In TThreadPersistent:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LockCount</td>
<td>Lock</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Unlock</td>
<td></td>
</tr>
<tr>
<td>In TNotifiablePersistent:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>UpdateCount</td>
<td>BeginUpdate</td>
<td>OnChange</td>
</tr>
<tr>
<td>-------------</td>
<td>-------------</td>
<td>----------</td>
</tr>
<tr>
<td></td>
<td>Changed</td>
<td></td>
</tr>
<tr>
<td></td>
<td>EndUpdate</td>
<td></td>
</tr>
</tbody>
</table>
See Also

Polygons Example, TFixedPoint

Copyright ©2000-2007 Alex Denisov and Contributors - Graphics32 v1.8.3 - Build on 4-March-2007
TPositionedLayer
Hierarchy

TPersistent
  └── TNotifierPersistent
  └── TCustomLayer
**Description**

**TPositionedLayer** is the base class for layers that are positioned within the boundaries of their container (**TCustomImage32** or its descendant). The position and size is determined by the **Location** property.

When **TPositionedLayer** performs its **HitTest**, those points inside the rectangle returned by **GetAdjustedLocation** pass the test.

The class does not perform any drawing itself, however it is capable of calling the **OnPaint** event, where you can perform custom drawing.

**Reference**

<table>
<thead>
<tr>
<th>Properties</th>
<th>Methods</th>
<th>Events</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>In TPositionedLayer:</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Location</td>
<td>Create</td>
<td></td>
</tr>
<tr>
<td>Scaled</td>
<td>GetAdjustedLocation</td>
<td></td>
</tr>
<tr>
<td></td>
<td>GetAdjustedRect</td>
<td></td>
</tr>
<tr>
<td><strong>In TCustomLayer:</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cursor</td>
<td>BringToFront</td>
<td>OnHitTest</td>
</tr>
<tr>
<td>Index</td>
<td>Changed</td>
<td>OnMouseDown</td>
</tr>
<tr>
<td>LayerCollection</td>
<td>Destroy</td>
<td>OnMouseMove</td>
</tr>
<tr>
<td>LayerOptions</td>
<td>HitTest</td>
<td>OnMouseUp</td>
</tr>
<tr>
<td>LayerStates</td>
<td>SendToBack</td>
<td>OnPaint</td>
</tr>
<tr>
<td>MouseEventArgs</td>
<td>Update</td>
<td></td>
</tr>
<tr>
<td>Tag</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Visible</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>In TNotifiablePersistent:</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>UpdateCount</td>
<td>BeginUpdate</td>
<td>OnChange</td>
</tr>
<tr>
<td></td>
<td>EndUpdate</td>
<td></td>
</tr>
</tbody>
</table>
See Also

TCustomImage32, Using Layers
TProgressiveRasterizer
Hierarchy

TCustomRasterizer

| TRasterizer
**Description**

This rasterizer class provides resolution subdivided rasterization. This is useful for visualizing while rasterizing. The `steps` property determines the number of intermediate steps in the rasterization. To see how this work, take a look at the nested sampling example with the progressive rasterizer enabled.

**Reference**

<table>
<thead>
<tr>
<th>Properties</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>In TProgressiveRasterizer:</td>
<td></td>
</tr>
<tr>
<td><strong>Steps</strong></td>
<td></td>
</tr>
<tr>
<td><strong>UpdateRows</strong></td>
<td></td>
</tr>
<tr>
<td>In TRasterizer:</td>
<td></td>
</tr>
<tr>
<td><strong>Sampler</strong></td>
<td><strong>Rasterize</strong></td>
</tr>
</tbody>
</table>
See Also

Nested Sampling Example
TProjectiveTransformation
Hierarchy

TOBject

| TTransformation
Description

This class specifies an arbitrary projective transformation. It transforms a rectangle from the source image into quadrilateral, defined by 4 points:

Note, that the destination quadrilateral has to be convex, otherwise the result is not specified.

Reference

<table>
<thead>
<tr>
<th>Properties</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>In TProjectiveTransformation:</td>
<td></td>
</tr>
<tr>
<td>X0</td>
<td>GetTransformedBounds</td>
</tr>
<tr>
<td>X1</td>
<td></td>
</tr>
<tr>
<td>X3</td>
<td></td>
</tr>
<tr>
<td>Y0</td>
<td></td>
</tr>
<tr>
<td>Y1</td>
<td></td>
</tr>
<tr>
<td>Y2</td>
<td></td>
</tr>
<tr>
<td>Y3</td>
<td></td>
</tr>
<tr>
<td>In TTransformation:</td>
<td></td>
</tr>
<tr>
<td>SrcRect</td>
<td>HasTransformedBounds</td>
</tr>
<tr>
<td></td>
<td>ReverseTransform</td>
</tr>
<tr>
<td></td>
<td>Transform</td>
</tr>
</tbody>
</table>

![Diagram showing the transformation of a rectangle to a quadrilateral](image)
See Also

Transform Example
TRangeBar
Hierarchy

TCustomControl
  | TArrowBar
  | TCustomRangeBar
Description

**TRangeBar** is a control similar to standard *TControlScrollBar*. Normally it is associated with some other control that has a scrollable region.

**TRangeBar** implements the generic behavior introduced in *TCustomRangeBar*. It publishes some inherited properties from *TCustomRangeBar*, but does not introduce any new behavior.

Reference

<table>
<thead>
<tr>
<th>Properties</th>
<th>Methods</th>
<th>Events</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>In TCustomRangeBar:</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Centered</td>
<td>Create</td>
<td></td>
</tr>
<tr>
<td>Increment</td>
<td>Paint</td>
<td></td>
</tr>
<tr>
<td>Position</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Range</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Window</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>In TArrowBar:</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Backgnd</td>
<td>Destroy</td>
<td>OnChange</td>
</tr>
<tr>
<td>BorderStyle</td>
<td></td>
<td>OnUserChange</td>
</tr>
<tr>
<td>ButtonSize</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HandleColor</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Kind</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ShowArrows</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ShowHandleGrip</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Style</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
See Also

TCustomGaugeBar, TCustomRangeBar
Hierarchy

TCustomRasterizer
Description
The base class for TBitmap32-specific rasterizers. A rasterizer is the final step in a sampler setup, the step responsible for calling the associated samplers and writing the results into the provided destination bitmap. A separated rasterization process allows for different types of on-screen realizations. For example a progressive rasterizer will increase the resolution of the result in progressive steps, which can keep the end user informed about the progression of the overall process. Moreover it provides perceptual information about the rendering, which can be useful for validation purposes.

Reference

<table>
<thead>
<tr>
<th>Properties</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sampler</td>
<td>Rasterize</td>
</tr>
</tbody>
</table>

In TRasterizer:
See Also

Sampling and Rasterization, TBitmap32, TProgressiveRasterizer
TRectList

Hierarchy
   TObject
Description

TRectList is a container class for storing rectangles. Handling is similar to TList, but instead of storing pointers it stores TRect. Please refer to the TList documentation for further details.

Internally the rectangles are saved in an array structure instead of a pointer based approach, so as to have a more memory efficient storage.
TRegularRasterizer
Hierarchy

TCustomRasterizer

| TRasterizer |
**Description**

The regular rasterizer performs a simple iterative rasterization comparable to a straightforward set of scanline loops. It is also the default rasterization class used internally in Graphics32.

**Reference**

<table>
<thead>
<tr>
<th>Properties</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>In TRegularRasterizer:</td>
<td></td>
</tr>
<tr>
<td>UpdateRowCount</td>
<td></td>
</tr>
<tr>
<td>In TRasterizer:</td>
<td></td>
</tr>
<tr>
<td>Sampler</td>
<td>Rasterize</td>
</tr>
</tbody>
</table>

Copyright ©2000-2007 Alex Denisov and Contributors - Graphics32 v1.8.3 - Build on 4-March-2007
TRemapTransformation
Hierarchy

TObject

| TTransformation
Description

This TTransformation wraps TVectormap and implements a set of properties to control how the transformation uses the vectors of TVectormap in its transformation procedures.

The class itself will not provide any significant transformation algorithm: the subordinate vector map needs to be precalculated. The auxiliary routine RasterizeTransformation is useful for this purpose.

Reference

<table>
<thead>
<tr>
<th>Properties</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>In TRemapTransformation:</td>
<td></td>
</tr>
<tr>
<td>MappingRect</td>
<td>GetTransformedBounds</td>
</tr>
<tr>
<td>Offset</td>
<td>HasTransformedBounds</td>
</tr>
<tr>
<td>Scale</td>
<td></td>
</tr>
<tr>
<td>In TTransformation:</td>
<td></td>
</tr>
<tr>
<td>SrcRect</td>
<td>ReverseTransform</td>
</tr>
<tr>
<td></td>
<td>Transform</td>
</tr>
</tbody>
</table>
See Also

Image Warping Example, RasterizeTransformation, TVectorMap
Hierarchy

TPersistent
Description

**TRenderThread** extends the standard **TPersistent** class with the thread-safe locking and declares change notification events.

The locking provides means of synchronization of simultaneous access to the same resource in applications with multiple threads and it works similar to that in **TCanvas** class. For additional information, see Delphi documentation on **TCanvas**.

This class also 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 **TBitmapLayer**) that it was modified and its data has to be repainted to the screen. **TThreadPersistent**, however, does not use or implement automatic change notification itself. This is done in descendants.

Reference

<table>
<thead>
<tr>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>In TRenderThread:</td>
</tr>
<tr>
<td>Create</td>
</tr>
</tbody>
</table>


See Also

OnChange, TBitmap32, TBitmapLayer, TImage32
TRubberbandLayer
Hierarchy

TPersistent
  | TNotifiablePersistent
  | TCustomLayer
  | TPositionedLayer
**Description**

This class implements a rubber band, allowing to move and resize other positioned layers at run-time. The layer paints itself at GDI Overlays stage, using the standard GDI methods.

It is a good idea to keep TRubberbandLayer on top of other layers so that they don't intercept mouse messages.

Note, that this is a preliminary implementation of rubber banding, there is a lot to be done here to make its behavior more consistent, which will be done in future versions of the library. However, most likely I will keep names and general meaning of most properties and methods.

**Reference**

<table>
<thead>
<tr>
<th>Properties</th>
<th>Methods</th>
<th>Events</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>In TRubberbandLayer:</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ChildLayer</td>
<td>Create</td>
<td>OnResizing</td>
</tr>
<tr>
<td>FrameStippleCounter</td>
<td>Destroy</td>
<td>OnUserChange</td>
</tr>
<tr>
<td>FrameStippleStep</td>
<td>SetFrameStipple</td>
<td></td>
</tr>
<tr>
<td>HandleFill</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HandleFrame</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Handles</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HandleSize</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MaxHeight</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MaxWidth</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MinHeight</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MinWidth</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>In TPositionedLayer:</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Location</td>
<td>GetAdjustedLocation</td>
<td></td>
</tr>
<tr>
<td>Scaled</td>
<td>GetAdjustedRect</td>
<td></td>
</tr>
<tr>
<td><strong>In TCustomLayer:</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cursor</td>
<td>BringToFront</td>
<td>OnHitTest</td>
</tr>
<tr>
<td>Index</td>
<td>Changed</td>
<td>OnMouseDown</td>
</tr>
<tr>
<td>LayerCollection</td>
<td>HitTest</td>
<td>OnMouseMove</td>
</tr>
<tr>
<td>LayerOptions</td>
<td>SendToBack</td>
<td>OnMouseUp</td>
</tr>
<tr>
<td>LayerStates</td>
<td>Update</td>
<td>OnPaint</td>
</tr>
<tr>
<td>MouseEventArgs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tag</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Visible</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>In TNotifyablePersistent:</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>UpdateCount</td>
<td>BeginUpdate</td>
<td>OnChange</td>
</tr>
<tr>
<td></td>
<td>EndUpdate</td>
<td></td>
</tr>
</tbody>
</table>
See Also

GDI Overlays, Using Layers
TSelectiveConvolver
Hierarchy

TPersistent
  | TNotifiablePersistent
  | TCustomSampler
  | TNestedSampler
  | TKernelSampler
  | TConvolver
Description

**TSelectiveConvolver** works similarly to **TConvolver**, but it will exclude color samples from the convolution depending on the difference from a local reference sample value. This way we may for instance perform smoothing for regions with small color variance, while still maintaining the high contrast features of the image.

Reference

<table>
<thead>
<tr>
<th>Properties</th>
<th>Methods</th>
<th>Events</th>
</tr>
</thead>
<tbody>
<tr>
<td>In TSelectiveConvolver:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Delta</td>
<td>Create</td>
<td></td>
</tr>
<tr>
<td></td>
<td>GetSample</td>
<td></td>
</tr>
<tr>
<td>In TKernelSampler:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CenterX</td>
<td>Destroy</td>
<td></td>
</tr>
<tr>
<td>CenterY</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Kernel</td>
<td></td>
<td></td>
</tr>
<tr>
<td>In TNestedSampler:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sampler</td>
<td></td>
<td></td>
</tr>
<tr>
<td>In TCustomSampler:</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>FinalizeSampling</td>
<td></td>
</tr>
<tr>
<td></td>
<td>GetSampleBounds</td>
<td></td>
</tr>
<tr>
<td></td>
<td>HasBounds</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PrepareSampling</td>
<td></td>
</tr>
<tr>
<td>In TNotifiablePersistent:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>UpdateCount</td>
<td>BeginUpdate</td>
<td>OnChange</td>
</tr>
<tr>
<td></td>
<td>Changed</td>
<td></td>
</tr>
<tr>
<td></td>
<td>EndUpdate</td>
<td></td>
</tr>
</tbody>
</table>
TSinshKernel
Hierarchy

TPersistent

| TCustomKernel
**Description**

The $\sinh$ kernel is defined as

$$\sinh_{\alpha}(x) = a \frac{\sin(\pi x)}{\sinh(\pi \alpha x)}$$

where alpha is a custom coefficient specified by the **Coeff** property.

**Reference**

<table>
<thead>
<tr>
<th>Properties</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Coeff</strong></td>
<td><strong>Create</strong></td>
</tr>
<tr>
<td><strong>Width</strong></td>
<td><strong>Filter</strong></td>
</tr>
<tr>
<td></td>
<td><strong>GetWidth</strong></td>
</tr>
<tr>
<td></td>
<td><strong>SetWidth</strong></td>
</tr>
</tbody>
</table>

In TCustomKernel:

<table>
<thead>
<tr>
<th><strong>Observer</strong></th>
<th><strong>Changed</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>RangeCheck</strong></td>
</tr>
</tbody>
</table>
See Also

Kernel Example, Resampler Example
TSplineKernel
Hierarchy

TPersistent
  |
 TCustomKernel
Description
This class implements a *B-Spline* interpolation filter.

Reference

<table>
<thead>
<tr>
<th>Properties</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>In TSplineKernel:</td>
<td>Filter</td>
</tr>
<tr>
<td></td>
<td>GetWidth</td>
</tr>
<tr>
<td>In TCustomKernel:</td>
<td>Observer</td>
</tr>
<tr>
<td></td>
<td>Changed</td>
</tr>
<tr>
<td></td>
<td>Create</td>
</tr>
<tr>
<td></td>
<td>RangeCheck</td>
</tr>
</tbody>
</table>
See Also

Kernel Example, Resampler Example
TSuperSampler
Hierarchy

TPersistent
  └ TNotifierPersistent
    └ TCUSTOMSampler
    └ TNESTEDSampler
**Description**

**TSuperSampler** is a *nested sampler* that adds a mechanism for performing super sampling. By adding a super sampler to a chain of samplers, it is possible to increase the sampling density, which will improve the quality of the rasterized output image.

For each input sample \((x, y)\), we find the subsamples by using a rectangular grid, regularly subdivided (according to **SamplingX** and **SamplingY**) within the region \((x - 0.5, y - 0.5, x + 0.5, y + 0.5)\).

**Reference**

<table>
<thead>
<tr>
<th>Properties</th>
<th>Methods</th>
<th>Events</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SamplingX</strong></td>
<td><strong>Create</strong></td>
<td></td>
</tr>
<tr>
<td><strong>SamplingY</strong></td>
<td><strong>GetSample</strong></td>
<td></td>
</tr>
</tbody>
</table>

In **TnestedSampler**:

| **Sampler** |                      |              |

In **TCustomSampler**:

| **FinalizeSampling** | **GetSampleBounds** | **HasBounds** | **PrepareSampling** |

In **TNotifiablePersistent**:

| **UpdateCount** | **BeginUpdate** | **OnChange** | **Changed** | **Destroy** | **EndUpdate** |

|                  |                |              |            |            |              |
See Also

Kernel Example, Resampler Example
TSwizzlingRasterizer
Hierarchy

TCustomRasterizer
  |
TRasterizer
Description

A rasterization method where sample locations are chosen according to a fractal pattern called *swizzling*. An advantage with this pattern is that it may benefit from local coherency in the sampling method used, and moreover optimize the CPU cache usage. To see how this work, take a look at the nested sampling example with the swizzling rasterizer enabled.

Reference

<table>
<thead>
<tr>
<th>Properties</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>In TSwizzlingRasterizer:</td>
<td></td>
</tr>
<tr>
<td><strong>BlockSize</strong></td>
<td></td>
</tr>
<tr>
<td>In TRasterizer:</td>
<td></td>
</tr>
<tr>
<td><strong>Sampler</strong></td>
<td><strong>Rasterize</strong></td>
</tr>
</tbody>
</table>
See Also

Nested Sampling Example
TSyntheticImage32
Hierarchy

TCustomControl

| TCustomPaintBox32 |
**Description**

TPaintBox32 implements the generic behavior introduced in TCustomPaintBox32. It publishes some inherited properties from its ancestors and, in addition, it calls the OnPaintBuffer event when the buffer needs to be repainted.

**Reference**

<table>
<thead>
<tr>
<th>Properties</th>
<th>Methods</th>
<th>Events</th>
</tr>
</thead>
<tbody>
<tr>
<td>TSyntheticImage32:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AutoRasterize</td>
<td>Rasterize</td>
<td></td>
</tr>
<tr>
<td>BitmapAlign</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Buffer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DstRect</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rasterizer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RenderMode</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BufferOversize</td>
<td>Destroy</td>
<td>OnMouseEnter</td>
</tr>
<tr>
<td>BufferValid</td>
<td>DoPaintBuffer</td>
<td>OnMouseLeave</td>
</tr>
<tr>
<td>Options</td>
<td>Flush</td>
<td></td>
</tr>
<tr>
<td>RepaintMode</td>
<td>ForceFullInvalidate</td>
<td></td>
</tr>
<tr>
<td></td>
<td>GetViewportRect</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Invalidate</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Loaded</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Resize</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SetBounds</td>
<td></td>
</tr>
<tr>
<td>TCustomPaintBox32:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BufferOversize</td>
<td>Destroy</td>
<td>OnMouseEnter</td>
</tr>
<tr>
<td>BufferValid</td>
<td>DoPaintBuffer</td>
<td>OnMouseLeave</td>
</tr>
<tr>
<td>Options</td>
<td>Flush</td>
<td></td>
</tr>
<tr>
<td>RepaintMode</td>
<td>ForceFullInvalidate</td>
<td></td>
</tr>
<tr>
<td></td>
<td>GetViewportRect</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Invalidate</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Loaded</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Resize</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SetBounds</td>
<td></td>
</tr>
</tbody>
</table>
See Also

OnPaintBuffer, TCustomPaintBox32
TTesseralRasterizer
Hierarchy

TCustomRasterizer

| TRasterizer
**Description**

This recursive rasterization method uses a divide-and-conquer scheme to subdivide blocks vertically and horizontally. To see how this work, take a look at the [nested sampling example](#) with the tesseral rasterizer enabled.

**Reference**

<table>
<thead>
<tr>
<th>Properties</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Sampler</strong></td>
<td><strong>Rasterize</strong></td>
</tr>
</tbody>
</table>

In TRasterizer:
See Also

Nested Sampling Example
TThreadPersistent
Hierarchy

TPersistent

| TNotifiablePersistent
**Description**

**TThreadPersistent** extends the **TNotifiablePersistent** class with thread-safe locking.

The locking mechanism provides synchronization of simultaneous access to the same resource in applications with multiple threads and it works similar to the implementation found in **TCanvas** class. For additional information, see Delphi documentation on **TCanvas**.

**Reference**

<table>
<thead>
<tr>
<th>Properties</th>
<th>Methods</th>
<th>Events</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>LockCount</strong></td>
<td><strong>Lock</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Unlock</strong></td>
<td></td>
</tr>
<tr>
<td><strong>UpdateCount</strong></td>
<td><strong>BeginUpdate</strong></td>
<td><strong>OnChange</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Changed</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Create</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Destroy</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>EndUpdate</strong></td>
<td></td>
</tr>
</tbody>
</table>
See Also

TNotifiablePersistent
Transformation
Hierarchy
  TObject
**Description**

**TTransformation** is an abstract ancestor for bitmap transformation classes. **TTransformation** objects are passed as parameter in the **Transform** function, which actually performs the bitmap transformation. Custom descendents must at least implement **ReverseTransformFloat** (and **TransformFloat**, if forward transformation is supported), since this is the top precision level in the transformation system of Graphics32. All descendents must override at least this level, but for optimal performance, it is recommended that the integer and fixed versions are implemented, respectively.

**Reference**

<table>
<thead>
<tr>
<th>Properties</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SrcRect</strong></td>
<td><strong>GetTransformedBounds</strong></td>
</tr>
<tr>
<td></td>
<td><strong>HasTransformedBounds</strong></td>
</tr>
<tr>
<td></td>
<td><strong>ReverseTransform</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Transform</strong></td>
</tr>
</tbody>
</table>
See Also

TAffineTransformation, Transform, TTransformation
TTransformer
**Hierarchy**

- TPersistent
  - TNotifiablePersistent
  - TCustomSampler
  - TNestedSampler
**Description**

A *transformer* is a special kind of *nested sampler* that will transform the sampling coordinates using a *transformation* defined by a `TTransformation` descendant.

The transformer is normally used internally by the Transform routine, but it also possible to set it up in an arbitrary chain of samplers.

**Reference**

<table>
<thead>
<tr>
<th>Properties</th>
<th>Methods</th>
<th>Events</th>
</tr>
</thead>
<tbody>
<tr>
<td>In <code>TTransformer</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>Transformation</code></td>
<td><code>GetSample</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>GetSampleBounds</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>HasBounds</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>PrepareSampling</code></td>
<td></td>
</tr>
<tr>
<td>In <code>TNestedSampler</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>Sampler</code></td>
<td><code>Create</code></td>
<td></td>
</tr>
<tr>
<td>In <code>TCustomSampler</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>FinalizeSampling</code></td>
<td></td>
</tr>
<tr>
<td>In <code>TNotifiablePersistent</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>UpdateCount</code></td>
<td><code>BeginUpdate</code></td>
<td><code>OnChange</code></td>
</tr>
<tr>
<td></td>
<td><code>Changed</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>Destroy</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>EndUpdate</code></td>
<td></td>
</tr>
</tbody>
</table>
See Also

Kernel Example, Resampler Example, TTransformation

Copyright ©2000-2007 Alex Denisov and Contributors - Graphics32 v1.8.3 - Build on 4-March-2007
TTwirlTransformation
Hierarchy

TObject

| TTransformation
**Description**

This transformation class implements a twirl transformation. The amount of twirl can be controlled with the twirl property. The image below shows how a checkerboard pattern were transformed with this transformation. The `twirl` parameter were set very low for illustrative purposes. With high twirl frequencies (more extreme settings of the twirl property), it is recommended that antialiasing steps are taken (e.g. supersampling).

![Twirled Checkerboard Pattern](image)

**Reference**

<table>
<thead>
<tr>
<th>Properties</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>In <code>TTwirlTransformation</code>:</td>
<td></td>
</tr>
<tr>
<td><code>Twirl</code></td>
<td></td>
</tr>
<tr>
<td>In <code>TTransformation</code>:</td>
<td></td>
</tr>
<tr>
<td><code>SrcRect</code></td>
<td><code>GetTransformedBounds</code></td>
</tr>
<tr>
<td></td>
<td><code>HasTransformedBounds</code></td>
</tr>
<tr>
<td></td>
<td><code>ReverseTransform</code></td>
</tr>
<tr>
<td></td>
<td><code>Transform</code></td>
</tr>
</tbody>
</table>

Copyright ©2000-2007 Alex Denisov and Contributors  -  Graphics32 v1.8.3  -  Build on 4-March-2007
TVectorMap
Hierarchy

TPersistent
  | TNotifiablePersistent
  | TThreadPersistent
  | TCustomMap
**Description**

The **TVectorMap** class may be used to store two-dimensional vectors, and provides a set of useful indexing and interpolation properties. **TVectorMap** is an ancestor of **TCustomMap** and supports saving and loading (Photoshop .msh format). **TRemapTransformation** wraps TVectorMap, so for a **TTransformation** compatible implementation, this may be a better choice.

**Reference**

<table>
<thead>
<tr>
<th>Properties</th>
<th>Methods</th>
<th>Events</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>In TVectorMap:</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vector</td>
<td>Clear</td>
<td>OnVectorCombine</td>
</tr>
<tr>
<td>VectorCombineMode</td>
<td>Destroy</td>
<td></td>
</tr>
<tr>
<td>Vectors</td>
<td>Empty</td>
<td></td>
</tr>
<tr>
<td></td>
<td>GetTrimmedBounds</td>
<td></td>
</tr>
<tr>
<td></td>
<td>LoadFromFile</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Merge</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SaveToFile</td>
<td></td>
</tr>
<tr>
<td><strong>In TCustomMap:</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Height</td>
<td>Delete</td>
<td>OnResize</td>
</tr>
<tr>
<td>Width</td>
<td>Resized</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SetSize</td>
<td></td>
</tr>
<tr>
<td><strong>In TThreadPersistent:</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LockCount</td>
<td>Lock</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Unlock</td>
<td></td>
</tr>
<tr>
<td><strong>In TNotifiablePersistent:</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>UpdateCount</td>
<td>BeginUpdate</td>
<td>OnChange</td>
</tr>
<tr>
<td></td>
<td>Changed</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Create</td>
<td></td>
</tr>
<tr>
<td></td>
<td>EndUpdate</td>
<td></td>
</tr>
</tbody>
</table>
See Also

Image Warping Example, TCustomMap, TRemapTransformation, TTransformation, Vector Types, Visualization Example
TWindowedSincKernel
Hierarchy

TPersistent

| TCustomKernel
**Description**

Returns the value of the *Sinc* function constrained by a window function. Descendant classes override the **Window** method in order to implement a custom window function.

\[ h(x) = Sinc(x) \times Window(x); \]

Windowed *Sinc* filters are known for producing high quality resampled images. Ideal resampling corresponds to filtering an image with the ideal low-pass filter (i.e. a *Sinc* filter). This is computationally expensive, since the *Sinc* function has infinite extent. Hence a window function is used to constrain the filter.

See topic about [Sampling and Rasterization](#) for further information.

**Reference**

<table>
<thead>
<tr>
<th>Properties</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>In TWindowedSincKernel:</td>
<td></td>
</tr>
<tr>
<td>Width</td>
<td>Create</td>
</tr>
<tr>
<td></td>
<td>Filter</td>
</tr>
<tr>
<td></td>
<td>GetWidth</td>
</tr>
<tr>
<td></td>
<td>SetWidth</td>
</tr>
<tr>
<td>In TCustomKernel:</td>
<td></td>
</tr>
<tr>
<td>Observer</td>
<td>Changed</td>
</tr>
<tr>
<td></td>
<td>RangeCheck</td>
</tr>
</tbody>
</table>
See Also

Kernel Example, Resampler Example, Sampling and Rasterization, Window
TWordMap
Hierarchy

TPersistent
  └── TNotifiablePersistent
  └── TThreadPersistent
  └── TCustomMap
Description

The TWordMap class implements a two dimensional matrix of words. It may be useful for storing 16 bit grayscale data, and in general data that requires or can be represented in 16 bit unsigned integers (words).

Reference

<table>
<thead>
<tr>
<th>Properties</th>
<th>Methods</th>
<th>Events</th>
</tr>
</thead>
<tbody>
<tr>
<td>In TWordMap:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bits</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ValPtr</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Value</td>
<td></td>
<td></td>
</tr>
<tr>
<td>In TCustomMap:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Height</td>
<td>Delete</td>
<td>OnResize</td>
</tr>
<tr>
<td>Width</td>
<td>Empty</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Resized</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SetSize</td>
<td></td>
</tr>
<tr>
<td>In TThreadPersistent:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LockCount</td>
<td>Lock</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Unlock</td>
<td></td>
</tr>
<tr>
<td>In TNotifyablePersistent:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>UpdateCount</td>
<td>BeginUpdate</td>
<td>OnChange</td>
</tr>
<tr>
<td></td>
<td>Changed</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Create</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Destroy</td>
<td></td>
</tr>
<tr>
<td></td>
<td>EndUpdate</td>
<td></td>
</tr>
</tbody>
</table>
See Also

GR32_ByteMaps, TBooleanMap, TIntegerMap
GR32

GR32.pas contains common functions and classes for working with 32-bit device independent bitmaps (DIBs). It also defines some other basic types and operations.
## Contents

<table>
<thead>
<tr>
<th>Classes</th>
<th>Routines</th>
<th>Types</th>
<th>Constants</th>
</tr>
</thead>
<tbody>
<tr>
<td>TBitmap32</td>
<td>AlphaComponent</td>
<td>Arrays</td>
<td>Area Information Flags</td>
</tr>
<tr>
<td>TCustomMap</td>
<td>BlueComponent</td>
<td>Color Types</td>
<td>Color Constants</td>
</tr>
<tr>
<td>TCustomResampler</td>
<td>Color32</td>
<td>Point Types</td>
<td>G32Version</td>
</tr>
<tr>
<td>TCustomSampler</td>
<td>Color32Components</td>
<td>Rectangle Types</td>
<td></td>
</tr>
<tr>
<td>TNotifyablePersistent</td>
<td>Color32ToRGB</td>
<td></td>
<td>TAreaChangedEvent</td>
</tr>
<tr>
<td>TThreadPersistent</td>
<td>Creating Points</td>
<td>TCombineMode</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Creating Rectangles</td>
<td>TDrawMode</td>
<td></td>
</tr>
<tr>
<td></td>
<td>EqualRect</td>
<td>TFixed</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Fixed Point Math</td>
<td>TPixelCombineEvent</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Gray32</td>
<td>TRectRounding</td>
<td></td>
</tr>
<tr>
<td></td>
<td>GreenComponent</td>
<td>TSize</td>
<td></td>
</tr>
<tr>
<td></td>
<td>HSLtoRGB</td>
<td>TStretchFilter</td>
<td></td>
</tr>
<tr>
<td></td>
<td>InflateRect</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Intensity</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>IntersectRect</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>IsRectEmpty</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>OffsetRect</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>PtInRect</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>RedComponent</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>RGBtoHSL</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>SetAlpha</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>SetGamma</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>WinColor</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>WinPalette</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Copyright ©2000-2007 Alex Denisov and Contributors - Graphics32 v1.8.3 - Build on 4-March-2007
GR32_Filters

GR32_Filters.pas provides some basic functions for operating over bitmap data.
## Contents

<table>
<thead>
<tr>
<th>Routines</th>
<th>Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>AlphaToGrayscale</td>
<td>TLogicalOperator</td>
</tr>
<tr>
<td>ApplyBitmask</td>
<td>TLUT8</td>
</tr>
<tr>
<td>ApplyLUT</td>
<td></td>
</tr>
<tr>
<td>ChromaKey</td>
<td></td>
</tr>
<tr>
<td>ColorToGrayScale</td>
<td></td>
</tr>
<tr>
<td>CopyComponents</td>
<td></td>
</tr>
<tr>
<td>CreateBitmask</td>
<td></td>
</tr>
<tr>
<td>IntensityToAlpha</td>
<td></td>
</tr>
<tr>
<td>Invert</td>
<td></td>
</tr>
<tr>
<td>InvertRGB</td>
<td></td>
</tr>
</tbody>
</table>

Copyright ©2000-2007 Alex Denisov and Contributors - Graphics32 v1.8.3 - Build on 4-March-2007
GR32_Math

GR32_Math.pas is a collection of math related functions used in Graphics32.
## Contents

<table>
<thead>
<tr>
<th>Routines</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fixed Ceil</td>
</tr>
<tr>
<td>Fixed Division</td>
</tr>
<tr>
<td>Fixed Floor</td>
</tr>
<tr>
<td>Fixed Multiply</td>
</tr>
<tr>
<td>Fixed Round</td>
</tr>
<tr>
<td>Fixed Square</td>
</tr>
<tr>
<td>Fixed Square Root</td>
</tr>
<tr>
<td>SinCos</td>
</tr>
</tbody>
</table>
GR32_Rasterizers

GR32_Rasterizers.pas provides classes used when rasterizing an image. Rasterization is the process of acquiring a number of samples for the pixels of a destination image.

Rasterizers determine the order in which pixels are updated in the destination bitmap when sampling from an input scene (which can be a reconstructed bitmap, a synthetic sample generator or an image acquisition device).
## Contents

<table>
<thead>
<tr>
<th>Classes</th>
<th>Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>TContourRasterizer</td>
<td>TCombineInfo</td>
</tr>
<tr>
<td>TProgressiveRasterizer</td>
<td></td>
</tr>
<tr>
<td>TRasterizer</td>
<td></td>
</tr>
<tr>
<td>TRegularRasterizer</td>
<td></td>
</tr>
<tr>
<td>TSwizzlingRasterizer</td>
<td></td>
</tr>
<tr>
<td>TTesseralRasterizer</td>
<td></td>
</tr>
</tbody>
</table>

Copyright ©2000-2007 Alex Denisov and Contributors  -  Graphics32 v1.8.3  -  Build on 4-March-2007
GR32_Blend

GR32_Blend.pas includes low-level functions, mostly blending of pixels. The library automatically determines whether the CPU supports MMX instructions, and uses MMX optimized routines when it is possible.
## Contents

<table>
<thead>
<tr>
<th>Routines</th>
<th>Types</th>
<th>Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blend</td>
<td>TBlendLine</td>
<td>MMX_ACTIVE</td>
</tr>
<tr>
<td>BlendEx</td>
<td>TBlendLineEx</td>
<td></td>
</tr>
<tr>
<td>BlendLine</td>
<td>TBlendMem</td>
<td></td>
</tr>
<tr>
<td>BlendLineEx</td>
<td>TBlendMemEx</td>
<td></td>
</tr>
<tr>
<td>ColorAdd</td>
<td>TBlendReg</td>
<td></td>
</tr>
<tr>
<td>ColorDiv</td>
<td>TBlendRegEx</td>
<td></td>
</tr>
<tr>
<td>ColorMax</td>
<td>TCombineMem</td>
<td></td>
</tr>
<tr>
<td>ColorMin</td>
<td>TCombineReg</td>
<td></td>
</tr>
<tr>
<td>ColorModulate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ColorSub</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Combine</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EMMS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lighten</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Merge</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MergeEx</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
GR32_Image

GR32_Image.pas contains visual components for image storing and visualization.
## Contents

<table>
<thead>
<tr>
<th>Classes</th>
<th>Types</th>
<th>Constants</th>
</tr>
</thead>
<tbody>
<tr>
<td>TBitmap32Collection</td>
<td>TPaintBoxOptions</td>
<td>Paint Stage Constants</td>
</tr>
<tr>
<td>TBitmap32Item</td>
<td>TPaintStage</td>
<td></td>
</tr>
<tr>
<td>TBitmap32List</td>
<td>TRepaintMode</td>
<td></td>
</tr>
<tr>
<td>TCustomImage32</td>
<td>TScaleMode</td>
<td></td>
</tr>
<tr>
<td>TCustomImgView32</td>
<td>TScrollBarVisibility</td>
<td></td>
</tr>
<tr>
<td>TCustomPaintBox32</td>
<td>TSizeGripStyle</td>
<td></td>
</tr>
<tr>
<td>TImage32</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TIImgView32</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TIVScrollProperties</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TPaintBox32</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TPaintStages</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Copyright ©2000-2007 Alex Denisov and Contributors - Graphics32 v1.8.3 - Build on 4-March-2007
GR32_OrdinalMaps

GR32_OrdinalMaps.pas is a collection of ordinal type map classes.
## Contents

<table>
<thead>
<tr>
<th>Classes</th>
<th>Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>TBooleanMap</td>
<td>TConversionType</td>
</tr>
<tr>
<td>TByteMap</td>
<td></td>
</tr>
<tr>
<td>TIntegerMap</td>
<td></td>
</tr>
<tr>
<td>TWordMap</td>
<td></td>
</tr>
</tbody>
</table>

Copyright ©2000-2007 Alex Denisov and Contributors - Graphics32 v1.8.3 - Build on 4-March-2007
GR32_Resamplers

GR32_Resamplers.pas provides a comprehensive set of classes that are used within image resampling and rasterization. These classes can roughly be categorized as follows:

- **Resamplers** — classes that implement different resampling algorithms;
- **Kernels** — convolution kernels used by **TKernelResampler**;
- **Nested samplers** — *transformers, super samplers* and *kernel samplers*.

For further information, see topic about [Sampling and Rasterization](#).
<table>
<thead>
<tr>
<th>Classes</th>
<th>Routines</th>
<th>Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>TAdaptiveSuperSampler</td>
<td>BlendTransfer</td>
<td>TKernelMode</td>
</tr>
<tr>
<td>TBitmap32Resampler</td>
<td>BlockTransfer</td>
<td>TPixelAccessMode</td>
</tr>
<tr>
<td>TBlackmanKernel</td>
<td>BlockTransferX</td>
<td></td>
</tr>
<tr>
<td>TBoxKernel</td>
<td>Contract</td>
<td></td>
</tr>
<tr>
<td>TContracter</td>
<td>Dilate</td>
<td></td>
</tr>
<tr>
<td>TConvolver</td>
<td>Erode</td>
<td></td>
</tr>
<tr>
<td>TCosineKernel</td>
<td>Expand</td>
<td></td>
</tr>
<tr>
<td>TCubicKernel</td>
<td>StretchTransfer</td>
<td></td>
</tr>
<tr>
<td>TCustomKernel</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TDilater</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TDraftResampler</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TEroder</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TExpander</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TGaussianKernel</td>
<td></td>
<td></td>
</tr>
<tr>
<td>THammingKernel</td>
<td></td>
<td></td>
</tr>
<tr>
<td>THannKernel</td>
<td></td>
<td></td>
</tr>
<tr>
<td>THermiticKernel</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TKernelResampler</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TKernelSampler</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TLanczosKernel</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TLinearKernel</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TLinearResampler</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TMitchellKernel</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TMorphologicalSampler</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TNearestResampler</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TNearestTransformer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TNestedSampler</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TPatternSampler</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TSelectiveConvolver</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TSinhshKernel</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TSplineKernel</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TSuperSampler</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TTransformer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TWindowedSincKernel</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
See Also

Sampling and Rasterization, TKernelResampler
GR32_Containers

GR32_Containers.pas includes a number of auxiliary container classes.
## Contents

<table>
<thead>
<tr>
<th>Classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>TClassList</td>
</tr>
<tr>
<td>TPointerMap</td>
</tr>
<tr>
<td>TPointerMapIterator</td>
</tr>
<tr>
<td>TRectList</td>
</tr>
</tbody>
</table>

Copyright ©2000-2007 Alex Denisov and Contributors - Graphics32 v1.8.3 - Build on 4-March-2007
GR32_Layers

The GR32_Layers.pas unit contains layer classes, which can be attached to TCustomImage32 class and its descendants.
## Contents

<table>
<thead>
<tr>
<th>Classes</th>
<th>Types</th>
<th>Constants</th>
</tr>
</thead>
<tbody>
<tr>
<td>TBitmapLayer</td>
<td>TRBHandles</td>
<td>Layer Options Bits</td>
</tr>
<tr>
<td>TCustomLayer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TLayerCollection</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TPositionedLayer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TRubberbandLayer</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
See Also

ImgView and Layers Example, TCustomImage32, Using Layers
GR32_Polygons

GR32_Polygons.pas provides functions and classes for drawing polygons and polylines.
<table>
<thead>
<tr>
<th>Classes</th>
<th>Routines</th>
<th>Types</th>
<th>Constants</th>
</tr>
</thead>
<tbody>
<tr>
<td>TBitmapPolygonFiller</td>
<td>Polygon</td>
<td>TAntialiasMode</td>
<td>DefaultAAMode</td>
</tr>
<tr>
<td>TCustomPolygonFiller</td>
<td>Polyline</td>
<td>TFillLineEvent</td>
<td></td>
</tr>
<tr>
<td>TPolygon32</td>
<td>PolyPolygon</td>
<td>TPolyFillMode</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PolyPolygonBounds</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>PolyPolyline</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>PtInPolygon</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
GR32_Transforms

GR32_Transforms.pas provides functions and classes for combining and transforming bitmap images and their fragments.
It also includes affine and projective transformations of bitmaps.
## Contents

<table>
<thead>
<tr>
<th>Classes</th>
<th>Routines</th>
<th>Types</th>
<th>Constants</th>
</tr>
</thead>
<tbody>
<tr>
<td>TAffineTransformation</td>
<td>RasterizeTransformation</td>
<td>TFloatMatrix</td>
<td>IdentityMatrix</td>
</tr>
<tr>
<td>TBloatTransformation</td>
<td>SetBorderTransparent</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TDisturbanceTransformation</td>
<td>Transform</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TFishEyeTransformation</td>
<td>TransformPoints</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TProjectiveTransformation</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TRemapTransformation</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TTransformation</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TWirlTransformation</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
GR32_ExtImage

GR32_ExtImage.pas is an extension to the GR32 image unit. It provides a set of useful classes and routines for threaded rasterization and rendering.
## Contents

<table>
<thead>
<tr>
<th>Classes</th>
<th>Routines</th>
<th>Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRenderThread</td>
<td>Rasterize</td>
<td>TRenderMode</td>
</tr>
<tr>
<td>TSyntheticImage32</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
See Also

GR32_Image
GR32_Lowlevel

GR32_Lowlevel.pas is a collection of RTL extensions and auxiliary low level routines. The unit is primarily used internally in Graphics32.
## Contents

<table>
<thead>
<tr>
<th>Routines</th>
<th>Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clamp</td>
<td>Wrap Procedure Types</td>
</tr>
<tr>
<td>Colorswap</td>
<td></td>
</tr>
<tr>
<td>Constrain</td>
<td></td>
</tr>
<tr>
<td>FillLongword</td>
<td></td>
</tr>
<tr>
<td>FillWord</td>
<td></td>
</tr>
<tr>
<td>Mirror</td>
<td></td>
</tr>
<tr>
<td>MoveLongword</td>
<td></td>
</tr>
<tr>
<td>MoveWord</td>
<td></td>
</tr>
<tr>
<td>Shift Arithmetic Right</td>
<td></td>
</tr>
<tr>
<td>Swap</td>
<td></td>
</tr>
<tr>
<td>SwapConstrain</td>
<td></td>
</tr>
<tr>
<td>TestClip</td>
<td></td>
</tr>
<tr>
<td>TestSwap</td>
<td></td>
</tr>
<tr>
<td>Wrap</td>
<td></td>
</tr>
</tbody>
</table>

Copyright ©2000-2007 Alex Denisov and Contributors - Graphics32 v1.8.3 - Build on 4-March-2007
GR32_RangeBars

GR32_RangeBars.pas provides controls similar to standard TScrollBar and TControlScrollBar. This unit is included into Graphics32 to provide customized scroll bars for TCustomImgView32 and TImgView32 to scroll their contents.
## Contents

<table>
<thead>
<tr>
<th>Classes</th>
<th>Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>TArrowBar</td>
<td>TRBBackgnd</td>
</tr>
<tr>
<td>TCustomGaugeBar</td>
<td>TRBStyle</td>
</tr>
<tr>
<td>TCustomRangeBar</td>
<td></td>
</tr>
<tr>
<td>TGaugeBar</td>
<td></td>
</tr>
<tr>
<td>TRangeBar</td>
<td></td>
</tr>
</tbody>
</table>
See Also

TCustomImgView32, TImgView32
GR32_VectorMaps

The GR32_VectorMaps.pas unit contains the TVectorMap class.
## Contents

<table>
<thead>
<tr>
<th>Classes</th>
<th>Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>TVectorMap</td>
<td>TVectorCombineEvent</td>
</tr>
<tr>
<td></td>
<td>TVectorCombineMode</td>
</tr>
<tr>
<td></td>
<td>Vector Types</td>
</tr>
</tbody>
</table>
See Also

TVectorMap
Overview

Graphics32 is a set of functions, classes, components and controls designed for high-performance graphics programming.

Being highly specialized for 32-bit pixel format, it provides fast operations with pixels and graphic primitives and in most cases Graphics32 considerably outperforms standard \textit{TBitmap/TCanvas} classes.

The library comes with a full source code, documentation and examples.

The following topics are covered in this overview:

- Features;
- License;
- Donate;
- Contacts;
- Naming Convention;
- Changes.
See Also

Changes, Contacts, Donate, Features, License, Naming Conventions
Using TImage32

Starting from v0.99 of the Graphics32 library, implementation of VCL components was changed.
This section describes behavior and properties of TCustomImage32 and TImage32 controls, as well as new realization of layers.
Since TCustomImage32 and TImage32 share the same behavior, I will denote both of them here as TImage32. Most of the discussion concerns TImgView32 as well.
Overview

TImage32 is an image displaying visual component, which also holds a collection of layers.

The image is defined as a TBitmap32 object and there exist several properties which specify how it is scaled and positioned within the control's boundaries. I will denote this image as Bitmap Image here.

The layer is, generally speaking, an entity which 'knows' how to combine itself with the back buffer of the control. TImage32 maintains a collection of layers, see 'Using Layers' for details.

When the image paints itself, it runs through several Paint Stages which determine the order in which layers and the bitmap image are painted. This order is completely customizeable.

To avoid flicker and to speed-up scaling and blending operations, each layer and the bitmap image is painted to a back-buffer of the control. See the reference of TImage32 ancestor TCustomPaintBox32 for details of the back-buffer realization.

Finally, TImage32 supports change notification (OnChange event), which is done similar to change notification in TBitmap32. Basically, it redirects change notification from its layers and the bitmap image.
See Also