November 1998, Volume 4, Number 11

Cover Art By: *Darryl Dennis*

ON THE COVER

FEATURES

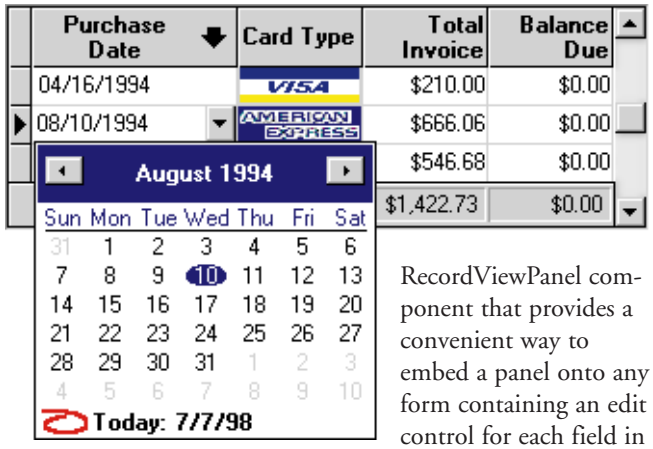
REVIEWS

DEPARTMENTS

Delphi TOOLS

New Products and Solutions

Woll2Woll Ships InfoPower 4



Woll2Woll Software announced that *InfoPower 4*, an upgrade to its visual component suite for Delphi 3 and 4 and C++Builder, is shipping.

InfoPower 4's enhancements include enhanced RichEdit control, which is now based on Microsoft's RichEdit Version 2 and supports embedding of bitmaps and OLE objects, display and automatic opening of Internet URL links, multi-level undo and redo, and database filtering on RichEdit fields; new date and time controls, including a data-bound DateTime picker and a MonthCalendar, support for Year-2000 compliance, formatting masks, and many display customizations for the calendar; enhanced usability, such as auto-filling of the current date and auto-advancing upon a valid month, day, or year; an enhanced grid to support images in both the titles and the data cells, a footer section to display column summary information, and animated column dragging; a new extendable DBNavigator component that supports user-definable images and actions, integration with InfoPower's dialog boxes, flexible control over the layout, and support for multiple rows of icons; a RecordViewPanel component that provides a convenient way to embed a panel onto any form containing an edit control for each field in the table; and an extended validation language to support the IncrementalSearch edit control.

Woll2Woll Software
Price: US$199; InfoPower Professional, US$299 (includes source code and C++Builder compatibility); upgrades for owners of InfoPower 3 are US$99, and US$129 for the professional version.
Phone: (800) WOL2WOL
Web Site: http://www.woll2woll.com

Sandage and Associates Ships CodeBase Components II for Delphi

Sandage and Associates announced CodeBase Components II for Delphi. Version 1 provided VCL support for Sequiter's CodeBase Database Engine. Version 2 offers *TDataSet* encapsulations of the table and query system, making it a true "plug-n-play" replacement for the BDE. With CodeBase Components II for Delphi, developers can connect Delphi's native data-aware controls (plus many third-party controls, including InfoPower) to the CodeBase engine.

CodeBase Components II for Delphi comes with myriad data-aware controls that let you create, index, browse, edit, and update Clipper, FoxPro, and dBASE files. Included are proprietary data-aware controls for Delphi 1, 2, and 3, plus two *TDataSet* descendant controls for Delphi 3, allowing the use of Delphi's native data-aware controls (along with many third-party controls).

CodeBase Components II for Delphi is available for US$210. For more information, call (626) 351-1299 or visit http://www.softsand.com.

D C AL CODA Releases YourTraySpell Words Suite 2.0

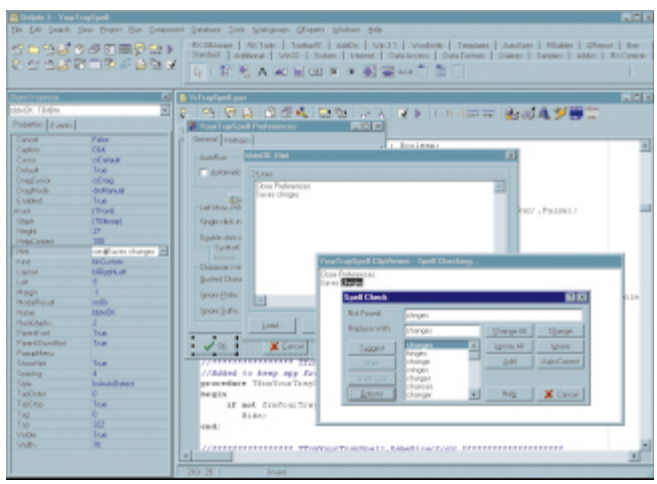
D C AL CODA released version 2.0 of ***YourTraySpell Words Suite*** (YTS), a configurable spell check, thesaurus, dictionary, text editor utility, and optimized word repository.

YTS spell checks all versions of Delphi, including the Object Inspector, IDE Editor, Captions and Hints, and more. A Delphi-specific dictionary is available.

Dictionaries in several languages, including American English, British English, Danish, Dutch, French, German, Italian, Norwegian, Polish, Spanish, and Swedish, put an end to spelling discrepancies, inconsistencies, and errors. Additionally, the 30,000-word *Roget's Thesaurus* and 160,000-word Definitions Dictionary provide access to synonyms and antonyms, as well as the meaning and context of words.

YTS resides in the Windows 95/98/NT System Tray, and can analyze any word, paragraph, or document from any application, text editor, HTML editor, or database. YTS is customizable — set it to run upon Windows startup, choose which characters to ignore, set hotkeys, add words, use the ClipViewer to see words in context, and more.

D C AL CODA
Price: US$29.95 for single-user license; site licenses are available.
Phone: (530) 272-8133
Web Site: http://www.dcalcoda.com



Delphi TOOLS

New Products and Solutions

Engineering Objects Announces Matrix Math Toolkit 4

Engineering Objects Int'l announced Matrix Math Toolkit Version 4 for Delphi 4 (MMTv4), which provides the classes needed to make array, vector, and matrix programming an easy plug-in.

MMTv4 exposes the terms of vectors and matrices through the use of the Delphi default property. Matrix code can now be completely dynamic, yet still use the standard matrix notation.

MMTv4 uses the Delphi 4 zero-based dynamic array as the basis for vectors and matrices, so array shape can be defined at run time, rather than design time. The dynamic array implementation almost eliminates pointer notation, so the code is cleaner and easier to read.

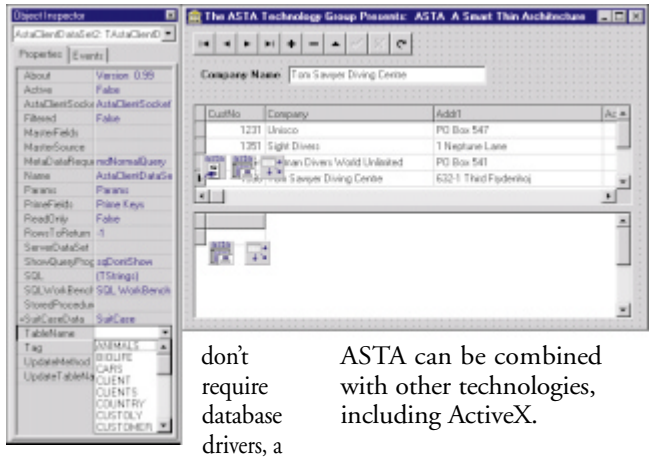
All the classes are persistent (that is, instances can write themselves to, and read themselves from, streams). Vectors and matrices are designed to work together. Along with the source code, the toolkit ships with demonstration programs.

MMTv4 is available for US$79.95 (single license), US$63.96 (up to 10 licenses), or US$51.97 (11 or more). Visit http://www.inconresearch.com/eoi for more information.

ASTA Releases ASTA 1.0

The ASTA Technology Group announced the release of *ASTA 1.0*, a smart thin architecture comprised of native VCL components designed for *n*-tier, thin-client computing in Delphi.

ASTA components behave like native Delphi controls. ASTA's *TAstaClientDataset*, a hybrid *TQuery-TTable* component, delivers design-time data and supports third-party tools, such as InfoPower. It also supports master/detail relationships and cached update functionality. It can be utilized for transitioning existing programs into Internet-ready, three-tier applications.



ASTA-based applications don't require database drivers, a BDE, ODBC, DLLs, DCOM, or client configuration — only the executable. ASTA's thin-client applications are suited for use over any TCP/IP network, including the Internet. ASTA can be combined with other technologies, including ActiveX.

ASTA Technology Group
Price: From US$250 per developer and US$250 for deployed server (US$399 for package); special license packages are available for corporate and vertical market developers.
E-Mail: info@astatech.com
Web Site: http://www.astatech.com

South Pacific Announces TCompress 4.0 and TCompLHA 4.0 for Delphi 4

South Pacific Information Services Ltd. announced the release of Delphi 4 versions of *TCompress* and *TCompLHA* compression component suites.

TCompress 4.0 provides native components for Delphi and C++Builder, supporting easy creation of multi-file compressed archives, and database, file, and in-memory compression using streams.

Included in the TCompress 4.0 set are the *TCompress* component, for general purpose multi-file archive, resource, and stream compression (includes RLE and LZH compression as standard); the COMPONLY unit, a version of *TCompress* for making applications that don't require the Borland Database Engine; the *TCDBImage* component, which uses *TCompress* to compress/expand database image fields; the *TCDBMemo* component, which uses *TCompress* to compress/expand database memo fields; *TCDBRichText* component, which uses *TCompress* to compress/expand rich text database fields (Delphi 3 only); the COMPDEMO application, a full-source, drag-and-drop demonstration of multi-file and database field compression; and source examples.

TCompLHA 4.0 helps create and manage archives compatible with the LHArc and LHA utilities. One-step methods, such as *Scan*, *Compress*, *Expand*, *Delete*, and *Verify* enhance archive management.

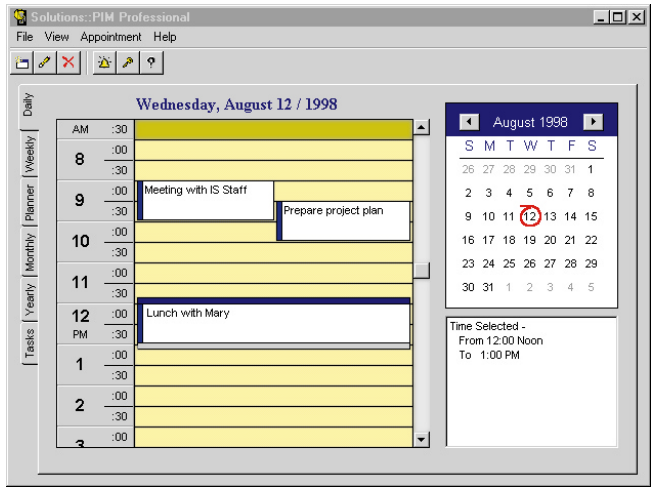
Included in the TCompLHA 4.0 set are the *TCompLHA* component; the *TSegLHA* component, a segmented archive and backup manager component; the *TCompLHAStream* component, a full-source *TStream* derivative for compression to and from any stream; LHADEMO, a full-source, drag-and-drop archive manager demonstration; SFX and MAKEEXE, example projects for making self-extracting/self-installing archives; and SEGDEMO and ADDRESS, full-source applications showing how to create segmented archives, and how to add easy backup/restore functions to any database application.

South Pacific Information Services Ltd.
Price: TCompLHA 4.0 registration and license, US$59; TCompress 4.0 registration and license, US$59.
Web Site: http://www.spis.co.nz

Delphi TOOLS

New Products and Solutions

DBI Technologies Announces Release of Solutions::PIM Professional



DBI Technologies Inc. released *Solutions::PIM Professional*, an upgrade to the company's visual calendaring tool, Solutions::PIM. The collection of 15 ActiveX controls allows developers to add personal information management, calendaring, and scheduling capabilities to Windows applications.

The Solutions::PIM Professional package still includes the yearly, monthly, week-view, and day-view calendars. Also included are the date and time input controls, the alarm control, and a .WAV file player. Enhancements to existing controls include new print methods that allow graphical printing of the displayed information in the visual calendars. New stylings have been added to the visual calendars, such as NT-display-styles and week-of-the-year numbering. New security methods make it possible to lock appointments based on user ID and other logic.

Another addition to the package is the multi-column day-view visual calendar control. Developers can set up ctMDay to view a single person's schedule over several days, or several people's schedules in a single day, showing one to 10 actual columns (and up to 32,000 additional virtual columns). Drag-and-drop is supported to and from the control, as well as between columns within the control.

The new package also includes the Enhanced List control, which fills the gap between insubstantial list controls and complex grids that are excessive for most applications. With ctList, developers can display Microsoft Outlook-style "to-do" lists or e-mail message displays (with sub-text), with sorting, column sizing, and more.

DBI Technologies Inc.
Price: US$349 (32-bit ActiveX controls, online tutorial, and online Help).
Phone: (204) 985-5770
Web Site: http://www.dbi-tech.com

Soletta Announces Standard Delphi Library

Soletta announced the open beta version of the Standard Delphi Library (SDL), a data structure, object persistence, and algorithm library designed for the Delphi environment. SDL is based on the mature design of the Standard Template Library (STL), the container-library standard for C++.

SDL is designed for intermediate to advanced Delphi programmers who need sophisticated data structures or wish to take advantage of SDL's large library of generic algorithms. SDL is also appropriate for programmers experienced with the C++ STL, or ObjectSpace's JGL (Java Generic Library).

SDL offers a number of features not found in any other Delphi class library, including natural storage of atomic data types, allowing SDL containers to be used to hold any Delphi data type (such as Integers, Strings, Extended values) with no special syntax; generic algorithms; integrated persistence; a complete set of data structures; and atomic, associative data structures.

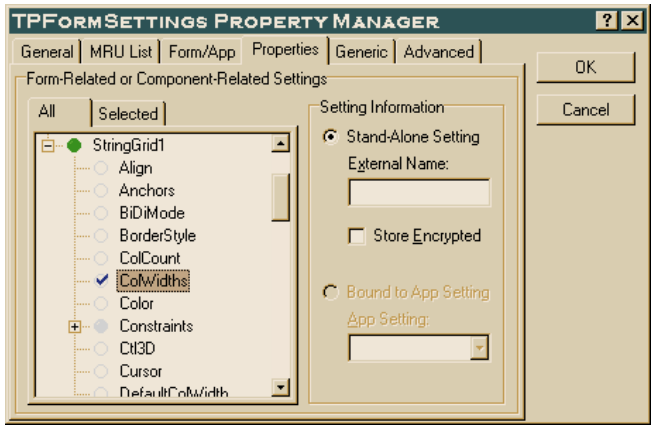
SDL is available for US$75 (SDL binary), US$250 (SDL source). For more information, visit http://www.soletta.com.

Pythoness Releases PSetting 2

Pythoness Software has released *PSetting 2*, a component set for Delphi 2, 3, and 4 that simplifies the process of maintaining application settings between runs.

PSetting 2 aims to automate the management of an application's state, including window positions, dockable toolbar locations, font colors and sizes, etc.

PSetting 2 features several enhancements, including storing hidden properties and complex types (including *TCollection* and *TComponent* descendants); better registry management and control, including saving state information in HKEY_LOCAL_MACHINE under Windows NT; improved MRU list control, including automatic detection of files that no longer exist; automatic application restart after a Windows shutdown; and more events for controlling save and restore options.



Pythoness Software
Price: US$69 (includes source code).
Phone: (208) 359-1540
Web Site: http://www.pythoness.com

News LINE

November 1998

Inprise Announces Strategic Alliance with Sun Microsystems

Denver, CO — Inprise Corp. announced it has entered into a strategic alliance with Sun Microsystems, Inc. to team Inprise's development technologies with Sun's Solaris operating environment. Corporations will be able to take advantage of Inprise's tools for building and running enterprise applications on the Solaris. The alliance is one of several ongoing initiatives between Sun and Inprise.

Through a series of marketing programs, the two companies will continue to migrate Sun customers to Inprise's VisiBroker object request broker (ORB). Earlier this year, Sun announced a migration services agreement to transition users of the Solaris NEO ORB to Visigenic's VisiBroker for Java and VisiBroker for C++ ORB technology. Visigenic Software was subsequently acquired by Inprise in February 1998. Sun and Inprise are now providing consulting, as well as documentation, that map key NEO features to VisiBroker features to streamline the transition between ORBs. Since the beginning of the year, Inprise has marketed VisiBroker to established Sun customers.

Inprise Details Enterprise Application Server Strategy

Denver, CO — Inprise Corp. unveiled the key components of its Inprise Application Server. Scheduled for delivery in late 1998, the Inprise Application Server is an integrated suite of enterprise middleware and development tools that provide a solution for simplifying the development, deployment, and management of distributed applications.

Key components of the Inprise Application Server include visual development tool integration, centralized management of distributed applications, a standards-based infrastructure, reliable transactions in a distributed environment, and enterprise-level security.

For more information on the Inprise Application Server, a detailed executive white paper is available on the Inprise Web site at http://www.inprise.com/appserver/appserver.html.

American Automobile Association Builds Reservation System with VisiBroker for Java

Denver, CO — Inprise Corp. announced the American Automobile Association (AAA) has adopted Inprise's VisiBroker for Java object request broker technology as a key building block for its new travel reservation system, which will go live by the end of 1998. With more than 40 million members, AAA is the largest motoring and travel organization in the world.

AAA is using VisiBroker to develop computing systems that provide a range of travel reservation products and services, such as air, hotel, car rental, cruise bookings, and auto-travel routing requests and assistance. The automobile organization selected VisiBroker as its distributed object infrastructure because of its integrated development tools and its ability to tie in disparate data sources, such as Apollo — a large, global reservation system for the travel industry.

AAA is a federation made up of 94 independent clubs that are part of an overall AAA association. Because each club uses different hardware and software to meet its business needs, AAA was looking to build one application that would enable the integration of each individual club's disparate legacy systems; provide information access to a large number of users throughout the federation; and save time and money on application development.

Inprise Transfers Visual dBASE Development and Marketing Responsibilities to InterBase

Denver, CO — Inprise Corp. announced that its independent subsidiary, InterBase Software Corp., will assume responsibility for the future development and marketing of its Visual dBASE family of Windows database products.

Visual dBASE 7 Professional and Visual dBASE 7 Client/Server Suite are the Windows 95/NT versions of the Xbase database development environment, based on the Borland visual development tools and 32-bit database technology.

Based in Scotts Valley, CA, InterBase delivers InterBase, a high-performance SQL database designed to be embedded into value-added reseller applications.

For more information on InterBase Software Corp. and the Visual dBASE family of Windows database products visit their Web site at http://www.interbase.com.

table of CONTENTS

By Rod Stephens

Picture Perfect

Shrinking, Enlarging, and Rotating Images

Delphi provides a couple of easy ways to resize an image. If you set a *TImage* control's *Stretch* property to True and resize it, the control stretches or shrinks its image to fit. You can also use the *TCanvas* object's *StretchDraw* procedure to resize a graphic and copy it onto a canvas.

The following code stretches the image held in the *imgInput* control to fit into the *imgOutput* control:

```
imgOutput.Canvas.StretchDraw
 (imgOutput.ClientRect,
  imgInput.Picture.Graphic);
```

These methods are fast and easy, but they often give unsightly results. To enlarge an image, these techniques simply duplicate each pixel enough times to fill the new image. If the new image is five times as big as the old one, each pixel is converted into a little five-by-five box. This produces a blocky picture like the one shown in Figure 1. To shrink an image, these techniques remove a fraction of the pixels from the original image. If the new image is half the size of the original, every other pixel is removed from the image. Unfortunately, the pixels Delphi decides to remove may not be the best choices. The removed pixels may contain information that is necessary to convey the shape of the original image.

Figure 1: Enlarging an image using *StretchDraw* produces a blocky result.

At best, the result may be rough and jagged, as shown by the smaller text in Figure 2. At worst, whole pieces of the image may disappear. For example, if the original picture contains a vertical line one pixel wide, shrinking the image may remove every pixel in the line. Similarly in Figure 2, thin parts of the smaller text have disappeared. The text "Rod Stephens" at the bottom contains several gaps. Removing important color information can also cause the strange plaid-like patterns shown in the red text and in the image of the hourglass.

This article explains how you can enlarge and shrink images smoothly in Delphi. Figure 3 shows an image similar to the one shown in Figure 1, but this picture has been enlarged smoothly. The image shows none of the blockiness in Figure 1, but it is much fuzzier. The original image is only a fifth as wide and tall as the picture in Figure 3, so it contains only 1/25th as many pixels. There isn't enough information in the original image to fill all the pixel values in the enlarged image smoothly without some blurring.

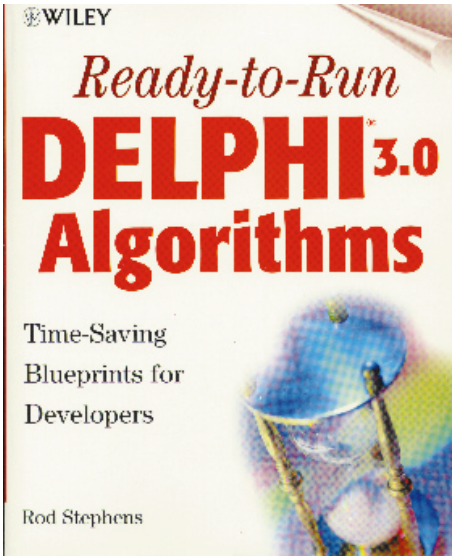


Figure 2: Shrinking an image with *StretchDraw* can produce a rough, jagged result.

Figure 3: Enlarging an image smoothly produces a slightly fuzzy image with no blockiness.

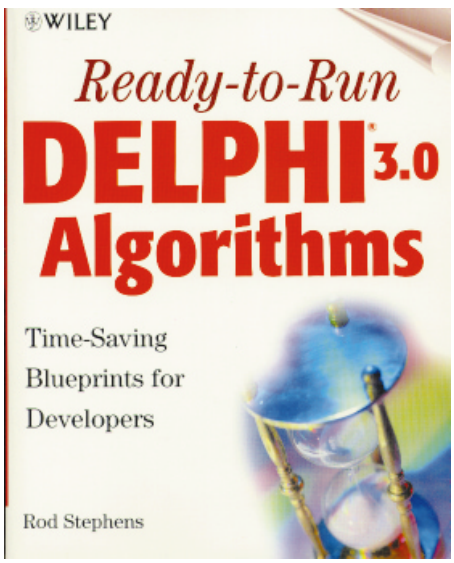


Figure 4: Shrinking an image smoothly removes jagged edges.

The picture in Figure 4 is much smoother than the version shown in Figure 2. There is some blurring in this image, but the effect only smoothes edges that might otherwise be jagged. This image shows no gaps in the small text and does not contain the annoying plaid-like effects in the red text or hourglass.

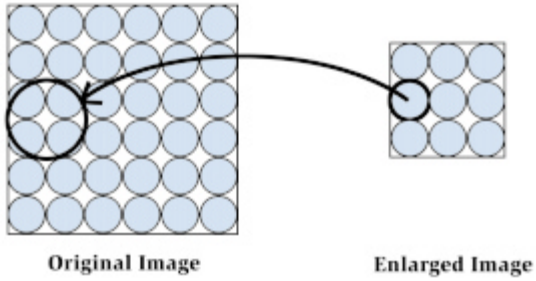


Figure 5: Mapping an output pixel back to the input pixels that determine its value.

Roadmap to Reduction

To shrink an image, *TCanvas.StretchDraw* removes some of the pixels from the original image. The problem with this method is that information contained in the removed pixels is completely lost. If they happen to contain important data, such as the pixels that lie along a thin vertical line, the result can be disappointing.

A better method is to combine nearby pixels by averaging them to produce the pixels in the reduced output image. One way to do this is to consider each pixel in the output image. For each output pixel, the program calculates the pixels in the input image that map to that output pixel. It then averages the red, green, and blue components of those input pixels to produce the output pixel's color value. This process is shown graphically in Figure 5.

Listing One (on page 10) shows the *ShrinkPicture* procedure — a Delphi routine that reduces an image. The function reduces the area *from_x1* <= *x* <= *from_x2*, *from_y1* <= *y* <= *from_y2* in the input canvas *from_canvas*, into the area *to_x1* <= *x* <= *to_x2*, *to_y1* <= *y* <= *to_y2* in the output canvas *to_canvas*.

The key to the code is the function that maps an output pixel back to the input pixels that determine its value. If the image is being scaled by factors of *xscale* and *yscale* in the *X* and *Y* directions, respectively, then the output pixel (*to_x*, *to_y*) is mapped to the rectangle *x1* <= x <= x2, y1 <= y <= y2 where:

```
y1 = Trunc((to_y - to_y1) / yscale + from_y1)
y2 = Trunc((to_y + 1 - to_y1) / yscale + from_y1) - 1
x1 = Trunc((to_x - to_x1) / xscale + from_x1)
x2 = Trunc((to_x + 1 - to_x1) / xscale + from_x1) - 1
```

After the procedure finds the coordinates of the input rectangle, it calculates the average of the red, green, and blue color components of those pixels. It assigns the resulting color components to the output pixel (*to_x*, *to_y*).

The example program Sizer demonstrates this method for reducing images (this program is available for download; see end of article for details). Select **File | Open** to load an image. Enter a scaling factor of less than 1 in the **Scale** edit box. If you click the **Quick Scale** button, the program uses *StretchDraw* to shrink the image by the factor you specified. If you click the **Smooth Scale** button, the program uses the *ShrinkPicture* procedure to shrink the image smoothly. Figure 6 shows the Sizer program in action.

Enlightening Enlargement

To enlarge an image, *StretchDraw* duplicates each pixel in the original image. If the enlarged image is five times as wide and five times as tall as the original, *StretchDraw* turns each pixel into a five-by-five block of pixels in the enlarged image. This gives a blocky result like the one shown in Figure 1.

While shrinking and enlarging an image seem to be very different tasks, a technique similar to the one used by the procedure *ShrinkPicture* described in the previous section allows a program to enlarge an image smoothly.

For each output pixel, the program calculates the point within the input image that maps to the output pixel. Figure 7 shows this mapping graphically. Most of the time, that point doesn't correspond to an integral pixel location. In Figure 7, the output pixel is mapped to a point in the lower-right corner of the upper-left input pixel.

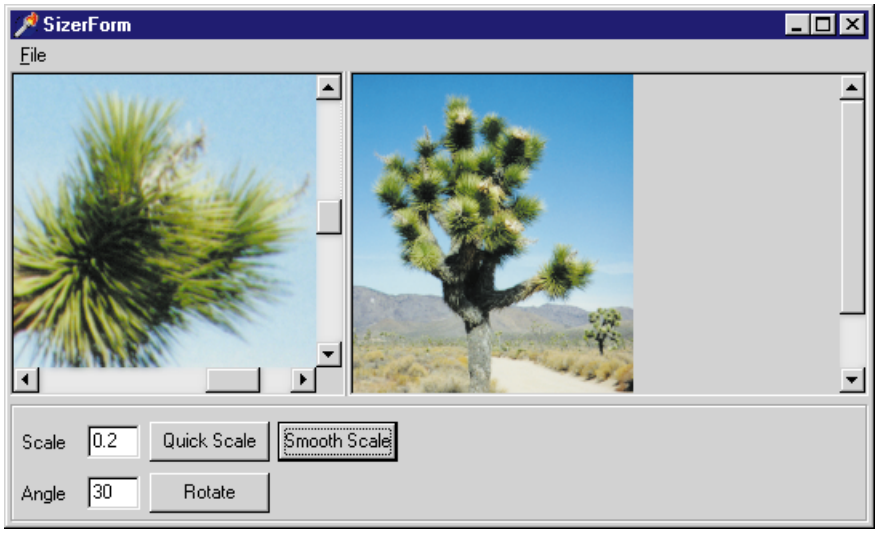


Figure 6: The example program Sizer smoothly shrinking a picture.

If the image is being scaled by factors of *xscale* and *yscale* in the *X* and *Y* directions, then the output pixel (*to_x*, *to_y*) is mapped to the point (*sfrom_x*, *sfrom_y*) where:

```
sfrom_y = (to_y - to_y1) / yscale + from_y1
sfrom_x = (to_x - to_x1) / xscale + from_x1
```

The program then examines the four pixels at integral locations near the computed input point. In Figure 7, those pixels are the four in the upper-left corner. Those pixels have coordinates:

```
(ifrom_x, ifrom_y)
(ifrom_x + 1, ifrom_y)
(ifrom_x, ifrom_y + 1)
(ifrom_x + 1, ifrom_y + 1)
```

where the values *ifrom_x* and *ifrom_y* are:

```
ifrom_y = Trunc(sfrom_y)
ifrom_x = Trunc(sfrom_x)
```

The program examines the red, green, and blue color component values of these four pixels. It uses weighted averages to calculate the components of the output pixel. The average is taken so the input pixels closest to the input point contribute the most to the result.

Listing Two (on page 10) shows the *EnlargePicture* procedure — a Delphi routine that uses this mapping method to enlarge images. If you examine the code closely, you can verify the special case that occurs when the input point corresponds exactly to one of the four input pixels. In that case, the weighting factors for the other three points are zero, so the output pixel's entire value is due solely to the one input pixel. That makes some sense. If an input pixel maps exactly to an output pixel, they should have the same color.

In addition to shrinking images, the Sizer program uses the *EnlargePicture* procedure to enlarge images. Select **File | Open** to load an image. Enter a scaling factor greater than 1 in the **Scale** edit box. If you click the **Quick Scale** button, the program uses *StretchDraw* to enlarge the image by the factor you specified. If you click the **Smooth Scale** button, the program uses the procedure *EnlargePicture* to enlarge the image smoothly. Figure 8 shows the Sizer program enlarging a picture smoothly using *EnlargePicture*.

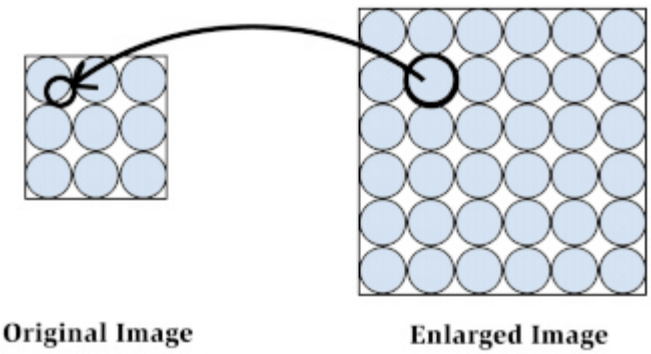


Figure 7: Mapping an output pixel back to a point within the input image.

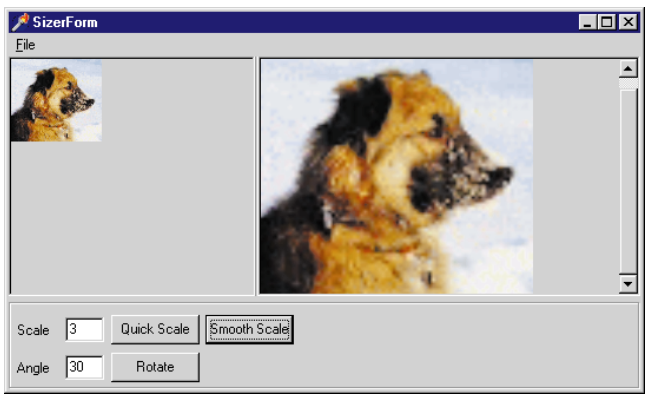


Figure 8: The example program Sizer smoothly enlarging a picture.

Spin Cycle

The equations for shrinking or enlarging an image are fairly

straightforward. In fact, they are simple enough that it's possible to map blocks of pixels from the input image onto the pixels in the output image. The procedures described here do the opposite; they map output pixels back to points in the input image.

Either method will work for enlargement or reduction, but the reverse method described here also makes many other kinds of smooth transformations manageable. For example, you can use reverse mapping to rotate images smoothly.

When you rotate a point (*x*, *y*) through the angle *theta* around the origin (0, 0), the resulting point has coordinates (*x'*, *y'*) where:

x' = x * Cos(theta) + y * Sin(theta)
y' = -x * Sin(theta) + y * Cos(theta)

These equations give the mapping from an input position to an output position.

The inverse of a rotation through the angle *theta* is a rotation through the angle *-theta*. In other words, to map an output pixel back to an input position, you apply the previous equations to rotate the point through the angle *-theta*. If the output pixel is at position (*x'*, *y'*), then the input position is (*x*, *y*) where:

x = x' * Cos(-theta) + y' * Sin(-theta)
y = -x' * Sin(-theta) + y' * Cos(-theta)

Because Sin(*-theta*) = -Sin(*theta*) and
Cos(*-theta*) = Cos(*theta*), these equations simplify to:

x = x' * Cos(theta) - y' * Sin(theta)
y = x' * Sin(theta) + y' * Cos(theta)

Using these equations, a program can rotate an image. For each pixel in the output image, the program uses the previous equations to find the point within the input image that maps to the output pixel. It then uses a weighted average of the four nearest pixels' color components to find the output pixel's color exactly as the procedure *EnlargePicture* does.

Listing Three (on page 11) shows the *RotatePicture* procedure — a routine that uses this technique to rotate an image smoothly. This code varies from the previous discussion slightly, so it can rotate images around their centers, rather than around the origin. This is why the coordinates (*to_cx*, *to_cy*) are subtracted from the output pixel's position before the calculation, and the coordinates (*from_cx*, *from_cy*) are added to the results.

One last rotation detail remains. When a rectangular picture is rotated, the corners stick out, so the result is taller and wider than the original image. The procedure *GetRotatedSize*, shown in Figure 9, calculates the height and width the output image needs. The program can use this procedure to decide how big it should make the output control.

```
// Calculate the height and width of the rotated picture.
procedure TSizerForm.GetRotatedSize(theta: Single;
  old_width, old_height: Integer;
  var new_width, new_height: Integer);
begin
  new_width  := Round(Abs(old_width  * Cos(theta)) +
                      Abs(old_height * Sin(theta)));
  new_height := Round(Abs(old_width  * Sin(theta)) +
                      Abs(old_height * Cos(theta)));
end;
```

Figure 9: Code that calculates the height and width needed for an output image.

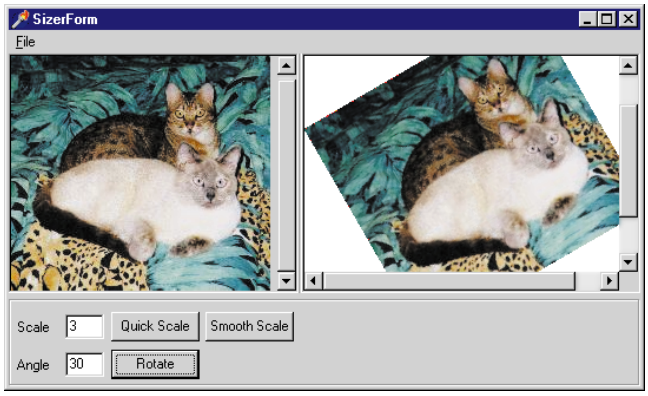


Figure 10: The example program Sizer smoothly rotating a picture 30 degrees.

The Sizer program uses the *RotatePicture* procedure to rotate images. Select **File | Open** to load an image. Enter an angle in degrees in the **Angle** edit box. If you click the **Rotate** button, the program uses *RotatePicture* to smoothly rotate the image. Figure 10 shows the Sizer program after it has rotated a picture 30 degrees.

Get Warped

The technique of mapping output pixels back to input positions and using a weighted average is a powerful one. It lets you shrink, enlarge, or rotate an image fairly easily. It also lets you apply more complicated transformations to an image. For example, you can stretch, twist, or otherwise warp an image to produce strange results. Just reverse the transformation so you can map output pixels back to input positions and take a weighted average.

Try some shape distorting transformations and see what you come up with. If you create a particularly unusual image, drop me a note. If it's interesting enough, I may post it on my Web site for all to enjoy. Δ

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\98\NOV\DI9811RS.

Rod is the author of several books, including *Ready-to-Run Delphi 3.0 Algorithms* [1998] and *Visual Basic Graphics Programming* [1997], both from John Wiley & Sons. He also writes algorithm columns in *Visual Basic Developer* and *Microsoft Office & Visual Basic for Applications Developer*. Rod can be reached via e-mail at RodStephens@vb-helper.com, or see what else he's done at his Web site at http://www.vb-helper.com.

Begin Listing One — *ShrinkPicture*

```
// Shrink the picture in from_canvas and place it in
// to_canvas.
procedure TSizerForm.ShrinkPicture(
  from_canvas, to_canvas: TCanvas;
  from_x1, from_y1, from_x2, from_y2: Integer;
  to_x1, to_y1, to_x2, to_y2: Integer);
var
  xscale, yscale          : Single;
  to_y, to_x              : Integer;
  x1, x2, y1, y2          : Integer;
  ix, iy                  : Integer;
  new_red, new_green      : Integer;
  new_blue                : Integer;
  total_red, total_green  : Single;
  total_blue              : Single;
  ratio                   : Single;
begin
  // Compute the scaling parameters. This is useful if
  // the image is not being scaled proportionally.
  xscale := (to_x2 - to_x1 + 1) / (from_x2 - from_x1);
  yscale := (to_y2 - to_y1 + 1) / (from_y2 - from_y1);

  // Perform the reduction.
  for to_y := to_y1 to to_y2 do begin
    y1 := Trunc((to_y - to_y1) / yscale + from_y1);
    y2 := Trunc((to_y + 1 - to_y1) / yscale + from_y1) -
1;
    for to_x := to_x1 to to_x2 do begin
      x1 := Trunc((to_x - to_x1) / xscale + from_x1);
      x2 := Trunc((to_x + 1 - to_x1) / xscale + from_x1) -
1;

      // Average the values in from_canvas within
      // the box (x1, y1) - (x2, y2).
      total_red   := 0;
      total_green := 0;
      total_blue  := 0;
      for iy := y1 to y2 do begin
        for ix := x1 to x2 do begin
          SeparateColor(from_canvas.Pixels[ix, iy],
                        new_red, new_green, new_blue);
          total_red   := total_red   + new_red;
          total_green := total_green + new_green;
          total_blue  := total_blue  + new_blue;
        end;
      end;
      ratio := 1 / (x2 - x1 + 1) / (y2 - y1 + 1);
      to_canvas.Pixels[to_x, to_y] := RGB(
        Round(total_red   * ratio),
        Round(total_green * ratio),
        Round(total_blue  * ratio));
    end; // End for to_x := to_x1 to to_x2 - 1 loop.
  end; // End for to_y := to_y1 to to_y2 - 1 loop.
end;

// Separate a color into red, green, and blue components.
procedure TSizerForm.SeparateColor(color: TColor;
  var red, green, blue: Integer);
begin
  red   :=  color mod 256;
  green := (color div 256) mod 256;
  blue  :=  color div 65536;
end;

// Combine red, green, and blue color components.
function TSizerForm.RGB(red, green, blue: Integer):
TColor;
begin
  Result := red + 256 * (green + 256 * blue);
end;
```

End Listing One

Begin Listing Two — *EnlargePicture*

```
// Enlarge the picture in from_canvas and place it
// in to_canvas.
procedure TSizerForm.EnlargePicture(
  from_canvas, to_canvas: TCanvas;
  from_x1, from_y1, from_x2, from_y2: Integer;
  to_x1, to_y1, to_x2, to_y2: Integer);
var
  xscale, yscale          : Single;
  sfrom_y, sfrom_x        : Single;
  ifrom_y, ifrom_x        : Integer;
  to_y, to_x              : Integer;
  weight_x, weight_y      : array[0..1] of Single;
  weight                  : Single;
  new_red, new_green      : Integer;
  new_blue                : Integer;
  total_red, total_green  : Single;
  total_blue              : Single;
  ix, iy                  : Integer;
begin
  // Compute the scaling parameters. This is useful if
  // the image is not being scaled proportionally.
  xscale := (to_x2 - to_x1 + 1) / (from_x2 - from_x1);
  yscale := (to_y2 - to_y1 + 1) / (from_y2 - from_y1);

  // Perform the enlargement.
  for to_y := to_y1 to to_y2 do begin
    sfrom_y := (to_y - to_y1) / yscale + from_y1;
    ifrom_y := Trunc(sfrom_y);
    weight_y[1] := sfrom_y - ifrom_y;
    weight_y[0] := 1 - weight_y[1];
    for to_x := to_x1 to to_x2 do begin
      sfrom_x := (to_x - to_x1) / xscale + from_x1;
      ifrom_x := Trunc(sfrom_x);
      weight_x[1] := sfrom_x - ifrom_x;
      weight_x[0] := 1 - weight_x[1];
      // Average the color components of the four
      // nearest pixels in from_canvas.
        total_red   := 0.0;
        total_green := 0.0;
        total_blue  := 0.0;
        for ix := 0 to 1 do begin
          for iy := 0 to 1 do begin
            SeparateColor(from_canvas.Pixels[
              ifrom_x + ix, ifrom_y + iy],
              new_red, new_green, new_blue);
            weight := weight_x[ix] * weight_y[iy];
            total_red   := total_red   + new_red   *
            weight;
            total_green := total_green + new_green *
            weight;
            total_blue  := total_blue  + new_blue  *
            weight;
          end;
        end;

        // Set the output pixel's value.
        to_canvas.Pixels[to_x, to_y] := RGB(
            Round(total_red),
            Round(total_green),
            Round(total_blue));
    end; // End for to_x := to_x1 to to_x2 loop.
  end; // End for to_y := to_y1 to to_y2 loop.
end;
```

End Listing Two

Begin Listing Three — *RotatePicture*

```
// Rotate the picture in from_canvas around its center
// through the angle theta in radians placing the result
// in the center of to_canvas.
procedure TSizerForm.RotatePicture(
  from_canvas, to_canvas: TCanvas; theta: Single;
  from_x1, from_y1, from_x2, from_y2: Integer;
  to_x1, to_y1, to_x2, to_y2: Integer);
var
  sin_theta, cos_theta    : Single;
  from_cx, from_cy        : Single;
  to_cx, to_cy            : Single;
  sfrom_y, sfrom_x        : Single;
  ifrom_y, ifrom_x        : Integer;
  to_y, to_x              : Integer;
  weight_x, weight_y      : array[0..1] of Single;
  weight                  : Single;
  new_red, new_green      : Integer;
  new_blue                : Integer;
  total_red, total_green  : Single;
  total_blue              : Single;
  ix, iy                  : Integer;
begin
  // Calculate the sine and cosine of theta for later.
  sin_theta := Sin(theta);
  cos_theta := Cos(theta);

  // Find the centers of the canvases.
  from_cx := (from_x2 - from_x1) / 2;
  from_cy := (from_y2 - from_y1) / 2;
  to_cx := (to_x2 - to_x1) / 2;
  to_cy := (to_y2 - to_y1) / 2;

  // Perform the rotation.
  for to_y := to_y1 to to_y2 do begin
    for to_x := to_x1 to to_x2 do begin
      // Find the location (from_x, from_y) that
      // rotates to position (to_x, to_y).
      sfrom_x := from_cx + (to_x - to_cx) * cos_theta -
                           (to_y - to_cy) * sin_theta;
      ifrom_x := Trunc(sfrom_x);

      sfrom_y := from_cy + (to_x - to_cx) * sin_theta +
                           (to_y - to_cy) * cos_theta;
      ifrom_y := Trunc(sfrom_y);

      // Only process this pixel if all four adjacent input
      // pixels are inside the allowed input area.
      if (ifrom_x >= from_x1) and (ifrom_x <  from_x2) and
         (ifrom_y >= from_y1) and (ifrom_y <  from_y2) then
      begin
        // Calculate the weights.
        weight_y[1] := sfrom_y - ifrom_y;
        weight_y[0] := 1 - weight_y[1];
        weight_x[1] := sfrom_x - ifrom_x;
        weight_x[0] := 1 - weight_x[1];

        // Average the color components of the four
        // nearest pixels in from_canvas.
        total_red   := 0.0;
        total_green := 0.0;
        total_blue  := 0.0;
        for ix := 0 to 1 do begin
          for iy := 0 to 1 do begin
            SeparateColor(
              from_canvas.Pixels[ifrom_x + ix,
              ifrom_y + iy], new_red, new_green, new_blue);
            weight := weight_x[ix] * weight_y[iy];
            total_red  := total_red   + new_red   * weight;
            total_green:= total_green + new_green * weight;
            total_blue := total_blue  + new_blue  * weight;
          end;
        end;

        // Set the output pixel's value.
        to_canvas.Pixels[to_x, to_y] := RGB(
          Round(total_red), Round(total_green),
          Round(total_blue));
      end;  // End if adjacent pixels in bounds.
    end;  // End for to_x := to_x1 to to_x2 loop.
  end;  // End for to_y := to_y1 to to_y2 loop.
end;
```

End Listing Three

Delphi 2, 3, 4 / Windows 95, 98, NT

By Kevin Bluck

Tray Icons

Implementing Them the Delphi Way

No doubt you've seen them: those little pictures down on the right side of the Windows 95/98/NT Taskbar. They're called Tray Icons, and they provide a very handy place to stash a program that you want to run quietly in the background, offering just enough visual feedback to keep track of it without cluttering up the desktop. Like many aspects of the Windows 95/NT Shell, however, there is precious little information available on how to implement them. This article aims to give you all the information you need to fully understand the tray icon API, and presents a component for implementing them the Delphi way. (The packaged component and a demonstration application are available for download; see end of article for details.)

The Windows API for tray icons is remarkably small. In fact, it consists of only a single function: *Shell_NotifyIcon*. This function, and its accompanying data record, *TNotifyIconData*, are the only tools you'll use to manipulate your icons. As is so often the case with the Windows API, however, this apparent simplicity is deceptive. As usual, the devil is in the details.

The Basics

The *Shell_NotifyIcon* function and its associated data types are defined by Inprise (*nee* Borland) in the ShellAPI unit. This function has a simple interface, with only two arguments:

```
function Shell_NotifyIcon(dwMessage: DWORD;
  lpData: PNotifyIconData): BOOL; stdcall;
```

The first argument, *dwMessage*, is straightforward. There are three things you can do to a tray icon: add, modify, or delete. Accordingly, there are three constants defined for this purpose: NIM_ADD, NIM_MODIFY, and NIM_DELETE. Simply pass the constant appropriate for the desired operation.

The second argument *lpData*, is more complicated. This is a pointer to a record defined by Inprise as *TNotifyIconData*. Let's look at this record's structure, element by element:

```
TNotifyIconData = record
    cbSize:           DWORD;
    Wnd:              HWND;
    uID:              UINT;
    uFlags:           UINT;
    uCallbackMessage: UINT;
    hIcon:            HICON;
    szTip:            array [0..63] of
                        AnsiChar;
  end;
```

cbSize must be set to the size in bytes of the entire record. Because this record is a fixed size, this is easily accomplished by using the *SizeOf* function.

Wnd must be set to the handle of the window that will receive the tray icon's notification messages. These notifications are primarily of mouse events, such as clicks. We'll go into more detail about these later. For now, just remember that a window handle must be associated with every tray icon.

uID is provided so your application and Windows can identify a particular tray icon. This value is included in the notification messages sent to the window identified in the *Wnd* member. Windows identifies each icon in the

tray by the combination of window handle and this ID. If each icon is governed by a different window, you can set *uID* to any value you wish and ignore it (for all practical purposes). If you are using one window to control multiple icons, however, you should assign and keep track of meaningful values in this member, as neither you nor Windows would have any other way of telling which icon is which.

uFlags is used to alert the *Shell_NotifyIcon* function which of the three optional data members have valid values. There are three constants defined for this purpose: NIF_MESSAGE, NIF_ICON, and NIF_TIP. *uFlags* may be set to any or all of these using the **or** operator. As an example, including NIF_MESSAGE in *uFlags* signals *Shell_NotifyIcon* that the *uCallbackMessage* member has a valid value. If this flag is not included, *Shell_NotifyIcon* will ignore any information in *uCallbackMessage*. Similarly, NIF_ICON corresponds to *hIcon*, and NIF_TIP governs *szTip*.

uCallbackMessage is where you specify the value of the notification message sent to the owning window whose handle is in *Wnd*. This value can be any that doesn't correspond to any other message that might be handled by the window. The best way to ensure this is to add some constant value to WM_USER. In fact, just using WM_USER by itself is sufficient. The message value merely needs to be unique for the window, not the entire system, or even throughout your application. Remember, this member is ignored if the NIF_MESSAGE constant is not included in the *uFlags* member.

hIcon accepts the handle of the icon image that you wish to appear in the tray. The easiest way to get at this value in Delphi is to load your icon into a *TIcon* object, and use the *Handle* property of that object. As before, you must set the NIF_ICON flag to alert *Shell_NotifyIcon* that it should interpret the value of this member. It's quite possible to modify the tray icon's appearance after adding it to the tray by submitting a different icon handle via the *hIcon* member in conjunction with a NIM_MODIFY operation. If you've noticed "animated" tray icons before, they work by doing exactly this sort of icon swapping, probably managed by a timer.

szTip is the last member. This is the text of the tooltip, which appears above most tray icons when you rest the mouse cursor over the icon for a second or two. It is, like almost every other string in Windows, a null-terminated string. However, it's important to note that this member is actually defined as a static array of 64 characters, not a character pointer. This means that after you allow for the null terminator, a maximum of 63 characters will fit in the tooltip. This should not be a practical concern for most developers, but there's always somebody who wants to stretch a metaphor a little too far. Like the other two optional data members, the icon's tip will not be updated unless the NIF_TIP flag is set in the *uFlags* member.

Unicode Applications

If you wish to develop Unicode applications, there are wide-character versions of these API elements. The wide version of *Shell_NotifyIcon* is, predictably, *Shell_NotifyIconW*, which takes a data record of type *TNotifyIconDataW*. The only difference between the two record structures is the *szTip* member, which is an:

```
array[0..63] of WideChar
```

in the wide version. All other elements are identical.

Shell_NotifyIcon returns a Boolean value (like most Win32 API functions), which reports if the function succeeded. I've found the errors returned by the *GetLastError* API function are remarkably uninformative when *Shell_NotifyIcon* fails. The only message I could coax it to reveal was, "A Windows API function failed." Gee, thanks for that probing insight, Windows! Our only consolation is that the tray icon API is simple enough that the problem is usually not difficult to find.

Mousing Around

We're still missing one important aspect of tray icons: reacting to mouse input. The only way a user can interact with a tray icon is to use the mouse. Accordingly, we need a way to detect these mouse events. A tray icon is not a window in the normal sense. It's governed by the system tray notification area, which is a special-purpose system window, and over which we have no easy means of control. Because the tray icon is not a window, it requires a window handle to be supplied to the *Wnd* data member in the *TNotifyIconData* record. This gives a place for the system tray window to send notification messages when it determines that mouse activity is occurring over your icon. The value specified by the *uCallbackMessage* member is the actual identifier of the notification message for that icon, so the window that handles the message can recognize it.

It's important to understand that the messages sent to the message handler window are not actual Windows mouse messages. They don't include any of the extra information normally packaged with such messages, such as the mouse position or the keyboard state. They are simply notification messages, which tell you only that a mouse event occurred.

Let's examine the difference between a normal mouse message and a tray icon notification message. All Windows messages have three primary components: the message identifier, a two-byte piece of data commonly known as *wParam*, and a four-byte piece of data known as *lParam*. We'll compare the Windows message that's sent when the left mouse button is pushed down over a normal window, and the message sent when the same action occurs over a tray icon. For the normal window, it receives a message with an identifier equal to the constant integer value WM_LBUTTONDOWN, a *wParam* value encoding the state of the keyboard, and an *lParam* encoding the position of the mouse relative to the window's client area. The window handling the same event for a tray icon, however, receives a message whose identifier is whatever value was specified by the latest valid *uCallbackMessage* value, a *wParam* with a value equal to the *uID* value for the icon, and an *lParam* containing the constant integer value

WM_LBUTTONDOWN. As you can see, all the tray icon message handling window knows is that the button was pressed. It has no idea exactly where it was pressed, nor whether any keys such as Shift or Ctrl were down at the same time. This pretty much eliminates our ability to respond to input any more sophisticated than a simple mouse click.

To further illustrate how mouse input is handled by the message handler window, here's a code snippet from the window procedure of a tray icon's message handler window:

```
// If the message is a tray notification message...
if Msg.Msg = WM_TRAYNOTIFY then begin
  // Check the lParam piece of the message structure to see
  // what happened in the tray.
  case (Msg.lParam) of
    WM_RBUTTONDOWN:   ...;
    WM_MBUTTONUP:     ...;
    WM_LBUTTONDBLCLK: ...;
    WM_MOUSEMOVE:     ...;
```

These four types of notification messages are basically all the message handler will receive from the tray icon. There are a few other obscure ones, such as palette change notifications, which are probably of no use to any but the most unusual development efforts. The messages for mouse button down, up, and double-click come in groups of three — one each for the left, middle, and right buttons. Mouse move messages also arrive, but as there is no good way of determining the mouse's exact position at the time of the message, the rectangle occupied by the tray icon, or any method of setting the tray icon to capture all mouse input, it's difficult to determine when the mouse moves off the tray icon. Yes, it would be possible to call *GetCursorPos* in response to these events, but that won't necessarily produce the mouse position at the time the message was generated. On a slow system with a rapidly moving mouse, the mouse position retrieved from *GetCursorPos* could be quite far from the point where the message was generated after it finally works its way through the message queue. At this time, I have not solved this problem, nor found any information suggesting an answer. Maybe you can.

Creating a Component

Enough of this messy API stuff. Now that you know what to do at the Windows level, let's get started making a component that will eliminate the need for you to remember it.

First, let's define the component's public interface. There are a few obvious things we need to provide for the implementation of a tray icon. An icon seems foremost. Also, tray icons typically have a tooltip hint and a popup menu associated with them. Of course, like all visual interface elements, we'll want a means to show and hide the tray icon.

A couple of less obvious traits come to mind after a bit of reflection on how this component will likely be used. We'll probably want to be able to check it out at design time, without running the application to see our handiwork. On the other hand, when the work is ready to test run from Delphi, we probably won't want to see two identical icons in the tray. Thus, we should provide some means for turning design-time visibility on and off.

One last thing: Tray-icon applications often start with the only visible evidence of their execution being the icon. Furthermore, a tray-icon application often hides itself completely (except for the tray icon), including hiding its taskbar button. It would be convenient for the component user if the component provided some simple means of hiding the entire application from the desktop and taskbar.

All these considerations lead to the following list of published properties:

```
property Hint:              string;
property Icon:              TIcon;
property PopupMenu:         TPopupMenu;
property ShowAtDesignTime:  Boolean;
property ShowApplication:   Boolean;
property Visible:           Boolean;
```

What About Events?

The only things that happen to tray icons without the application's prior knowledge are mouse events. We can't use the standard *TControl* mouse event types, however, because the notification-style message just doesn't provide the same information as the full mouse messages that *TControl* objects receive. As a result, we're pretty much limited to mouse button and click events. Although they're not likely to be of much use without position information, we'll throw in move events just to be complete.

Here's the list:

```
property OnClick:       TkbTrayClickEvent;
property OnDoubleClick: TkbTrayClickEvent;
property OnMouseDown:   TkbTrayMouseButtonEvent;
property OnMouseMove:   TNotifyEvent;
property OnMouseUp:     TkbTrayMouseButtonEvent;
```

Next, we consider the subject of run-time and read-only properties. Although there are a few "internals" that might surface, such as the notification message value, it's hard to imagine what possible use they could be outside the context of the tray icon. There's really no need to expose such pointless detail.

Run-time Methods

The last area of the public interface to consider, the run-time methods, does lend a few candidates for consideration. Almost any component that has a visible interface should provide a *Refresh* method. Also, the component user may want to invoke the popup menu directly. These are the methods:

```
procedure  Refresh;
procedure  ShowPopupMenu;
```

There are, of course, many details to implementing a component beyond the core functions this component encapsulates. There are many other excellent references that cover the minutiae of implementing custom components in Delphi, so this article will not cover them. Instead, it will concentrate

only on those pieces of the component's implementation that directly relate to the specific problem of tray icons.

Shell_NotifyIcon

The implementation of tray icons revolves around the call to *Shell_NotifyIcon*. Everything we do in this component will be in preparation for the moment of that function call. Let's outline how the public properties and methods will relate to that single goal.

The property most directly linked to *Shell_NotifyIcon* is the *Visible* property. Setting this property to True will cause *Shell_NotifyIcon* to be invoked with an operation of NIM_ADD, causing the icon to appear in the tray. As you might guess, setting it to False will call *Shell_NotifyIcon* with an operation of NIM_DELETE, and the icon disappears. One minor complication to this rather simple scenario is the action of *ShowAtDesignTime*. If *Visible* is set to True at design time, it must check to see if *ShowAtDesignTime* is also True before adding the icon. The property handles the logic, but the mechanics of calling *Shell_NotifyIcon* are delegated to another private method, *NotifyTrayIcon*, which will be discussed later. Meanwhile, the code snippet in Figure 1 (from the private property writer method, *SetVisible)* illustrates the logic.

ShowAtDesignTime has a similar problem in reverse. If *Visible* is False, then it doesn't matter what *ShowAtDesignTime* is being set to. If *Visible* is True, however, and we are designing at the moment, then *ShowAtDesignTime* is responsible for seeing that the icon is added and deleted. Again, the call to *Shell_NotifyIcon* is delegated to *NotifyTrayIcon*, in the interest of centralizing that code. Figure 2 shows how the property writer method, *SetShowAtDesignTime*, does its thing.

The *Hint* and *Icon* properties have a similar problem, i.e. how to update the visible properties of the tray icon while it's sitting in the tray. This is exactly the sort of job for which the NIM_MODIFY operation is intended (live updates to an existing icon). As you might guess, the properties don't modify the icon themselves; they update their internal storage and call the *Refresh* method, which calls the now-famous private *NotifyTrayIcon* method.

You might think the *PopupMenu* property is in the same live-update boat as *Hint* and *Icon*, but it's not. The popup menu is generated only on request, so it's sufficient simply to update its internal storage and leave it at that. The new menu will be available the next time the *ShowPopupMenu* function is called.

Managing the Messages

Next, we consider all the events. A window must be available to process all the notification messages generated by the user's interaction with the icon. How best to provide this window? We could use the application's main form, but that would require hooking that form's window procedure, a messy undertaking at best. It seems simplest to generate our

```
...
// If the new value is different from the existing value...
if (NewValue <> Self.FVisible) then begin
  // If the new value is True, and we are either Designing
  // and ShowAtDesignTime is True or else we are not
  // designing at all, add the icon to the tray.
  if ((NewValue) and
     (((csDesigning in Self.ComponentState) and
     (Self.ShowAtDesignTime)) or
     (not (csDesigning in Self.ComponentState)))) then
    Self.NotifyTrayIcon(NIM_ADD);
  // Otherwise, delete the icon from the tray.
  else
    Self.NotifyTrayIcon(NIM_DELETE);
  ...
end;
...
```

Figure 1: Code from the private property writer method, *SetVisible*.

```
...
// If the new value is different from the existing value...
if (NewValue <> Self.FShowAtDesignTime) then begin
  // If we are now designing, and the component is set to
  // Visible...
  if ((csDesigning in Self.ComponentState) and
     (Self.Visible)) then
    // If the new value is True, add the icon to the Tray.
    if (NewValue) then
      Self.NotifyTrayIcon(NIM_ADD);
    // If the new value is False, delete the icon from
    // the Tray.
    else
      Self.NotifyTrayIcon(NIM_DELETE);
  ...
end;
...
```

Figure 2: Code from the property writer method, *SetShowAtDesignTime*.

own invisible window, whose sole purpose is to handle those notification messages, and over whose destiny we have absolute control. This step eliminates problems with conflicting messages and hooks.

Inprise thoughtfully provided a couple of functions to facilitate this scheme. They are *AllocateHWnd* and *DeallocateHWnd*, found in the Forms unit. *AllocateHWnd*'s purpose in life is to generate a handle to an invisible window, using the window message-handling procedure you provide. Exactly what we need! Now, in the constructor, we can get and store a handle to a window that has nothing better to do than manage our icon's messages:

```
// Allocate invisible window to handle tray notification
// messages.
Self.FWindowHandle := AllocateHWnd(Self.WindowProcedure);
```

As you can see, this call is trivial. What's important is the window procedure we passed. This is where the notification messages from the icon are received, and where we have the opportunity to dispatch them. *AllocateHWnd* takes a single *TWndMethod* argument, a class-member procedure that itself takes a single *TMessage* argument. It's up to us to provide that procedure. To give you an idea,

```
procedure TkbTrayIcon.WindowProcedure(var Msg: TMessage);
begin
  // If the message is a tray notification message...
  if Msg.Msg = WM_TRAYNOTIFY then begin
    // Check the lParam piece of the message structure to
    // see what happened in the tray.
    case (Msg.lParam) of
      WM_LBUTTONDBLCLK: Self.DoubleClick(mbLeft);
      ...
    end;
  end
  // If the message is not a tray notification message,
  // then send it to the default window procedure for
  // handling.
  else begin
    Msg.Result := DefWindowProc(FWindowHandle, Msg.Msg,
                                Msg.wParam, Msg.lParam);
  end;
end;
```

Figure 3: An abbreviated version of our component's procedure.

Figure 3 shows an abbreviated version of our component's procedure.

As we discussed earlier, we first check the message identifier to verify that this incoming message is a WM_TRAYNOTIFY message. We ignore all others and send them to default handling, because none of the miscellaneous messages typically broadcast to every window in the system interest us. WM_TRAYNOTIFY is a constant we define; it's not provided by Windows. Setting it equal to WM_USER is the easiest thing to do, and perfectly safe. Once we're satisfied this is indeed a notification message, we decode the *lParam* value to determine which event the message is relating to us, and switch to an event-dispatch method based on that information. Because the message-handler window in our component is only handling a single tray icon, we can safely ignore the *uID* value encoded into the *wParam* data member.

The *NotifyTrayIcon* Method

Now, with all these supporting tasks worked out, we can finally get to the heart of the matter: the long-awaited call to *Shell_NotifyIcon*. It might seem a bit anticlimactic, but this function is found in only one place throughout the entire component. This place, of course, is the *NotifyTrayIcon* private method. Let's work our way through it.

First, set up the *TNotifyIconData* structure:

```
// Set up the tray icon data structure.
IconData.cbSize := SizeOf(IconData);
IconData.Wnd    := Self.FWindowHandle;
IconData.uID    := 0;
IconData.uFlags := NIF_MESSAGE or NIF_ICON or NIF_TIP;
IconData.uCallbackMessage := WM_TRAYNOTIFY;
```

cbSize takes the actual size in bytes of its own structure. *Wnd* takes the handle we created for our private message-handling window in the constructor. *uID* takes 0, because we have no need to use it (thanks to our private window, which handles absolutely nothing other than this icon). We add all three flags to *uFlags*, signifying that all three optional members will contain valid data. Simply updating all three every time is much easier than trying to determine which has changed, and carries no noticeable performance penalty. The message identifier in *uCallbackMessage* never changes; it's always the WM_TRAYNOTIFY constant defined in our unit.

Assigning the Icon

Assigning the icon requires a little fancy footwork. There's really no point inserting a tray icon without an icon, yet the component user might not have assigned an icon to the component, and may never intend to. If this is the case, we'll simply use the *Application* object's icon. The *Application* object always has an icon, even if one wasn't assigned by the developer — even at design time — so it seems the best place to fetch a default. You should know that at design time, the *Application* object refers to Delphi, so the default icon will be Delphi's, not the icon you may have assigned in the Project Options. Any icon you assigned to the *Application* object will appear at run time. Of course, if you have assigned an icon to the tray icon component's *Icon* property, that icon will appear at both design and run time:

```
// If this component has an icon assigned, use that for the
// icon shown in the tray.
if (Self.FIcon.Handle <> 0) then
  IconData.hIcon := Self.FIcon.Handle;
else
  // Otherwise, use the Application's icon.
  IconData.hIcon := Application.Icon.Handle;
```

The hint string also requires a bit of manipulation. Because it's possible to assign a string longer than the maximum tooltip size of 63 characters, we must ensure the property value reflects the text of the tooltip if such an overlong string is assigned:

```
// If the hint string is defined, then load it into the
// icon data structure. Note that the structure can only
// hold 63 characters, so trim it if necessary.
StrPLCopy(IconData.szTip,Self.FHint,High(IconData.szTip));
Self.FHint := StrPas(IconData.szTip);
```

Finally, the moment of truth. We call *Shell_NotifyIcon*:

```
// Instruct shell to perform operation on tray icon based
// on the Operation value and the IconData structure.
Shell_NotifyIcon(Operation, @IconData);
```

The result is a component that makes tray icons almost trivial to implement. You can forget all that Windows API unpleasantness.

How to Use Tray Icons

Now that you have this spiffy component to add tray icons with the greatest of ease, a few words on how to use them to maximum beneficial effect are in order. Like all tools, they can be used for good or evil. It's important to remember that the Taskbar tray is a shared resource. Every application installed by the user has the opportunity to compete for space in that area. Don't go nuts adding icons to the tray, or the useful impact of this excellent metaphor will be diluted. The user may learn to hate the little guys.

Because the tray is a system resource, it's appropriate to use it only to convey information of a global nature, or to provide an outlet for applications that ought to be constantly monitored without interfering with the user's other work. An example of the global sort of icon is the Volume Control provided by Microsoft. This icon allows the user to control the volume of every sound generated by every application. That's probably worth a 16-by-16 pixel square of valuable screen real estate. An example of the monitoring kind of icon would be an e-mail indicator. I have one that came with Netscape Communicator that waves a little flag whenever I have mail to download. That one is also worth 256 pixels.

If your icon is of the status-monitor kind, give some serious thought to how you can present as much information as possible to the user via its appearance alone. Requiring users to physically interact with it to get any useful information defeats the purpose of a status icon; they should have a good idea of the application's status just by looking at the icon. Animation is particularly effective at directing user attention to important events. You don't need a graphics workstation to do this; perfectly good animation effects can be achieved with two or three icons swapped around with a timer.

Generally, a tray icon should support some standard behaviors. These include:

- Mouse hover: Provide a tooltip that, at a minimum, identifies the purpose of the icon. Ideally, the tooltip text should expand on the status information provided by the icon. For example, an e-mail indicator might specify how many messages are waiting.
- Left-click: Provide some sort of popup window to display further information, or offer basic controls to the user. Popup windows are preferable to standard windows, because they can be dismissed by clicking elsewhere. The Volume Control icon, included with Windows, provides a good example of such a popup. This window should appear close to the tray icon, so the user doesn't have to hunt for it. If you have no additional information worth displaying, or the controls you wish to provide are too complex for such a fleeting popup window, then do nothing on a left-click.
- Right-click: Display a popup menu displaying possible operations for the icon. These operations should always include an **Exit** command for users who wish to kill an application altogether. There's nothing more irritating than an application you can't get rid of. The default command on this menu usually should be to show the main form of the icon's application, providing complete access to all functions of the application.
- Double-click: Execute the popup menu's default comand. This should be to show the application's main form.

Conclusion

The system tray is a compact and powerful interface metaphor, providing visual feedback and almost unlimited facility for user interaction into an extraordinarily compact package. With the information presented in this article, you can exploit this compelling feature in your own software. See what you can build — and have fun! Δ

The component and demonstration applications are available on the Delphi Informant Works CD located in INFORM\98\NOV\DI9811KB.

Kevin Bluck is an independent contractor specializing in Delphi development. He lives in Sacramento, CA with his lovely wife Natasha. He spends his spare time chasing weather balloons and rockets as a member of JP Aerospace (http://www.jpaerospace.com), a group striving to be the first amateurs to send a rocket into space. Thanks to Jay Hallett for his valuable assistance in developing the icon component. Kevin can be reached via e-mail at kbluck@ix.netcom.com.

By G. Bradley MacDonald

The Object Repository

An Easy Tool for Sharing and Standardizing Forms

Delphi's Object Repository (OR) is a great method for sharing forms and/or objects among your projects, as well as your developers and their projects. The OR is only available to the machine on which Delphi is installed. So, each developer has his or her own copy of the OR that is inaccessible to other developers. To fully utilize the OR, you may want to consider sharing it among all the developers in your company.

One of the benefits of a shared OR is standardization. For example, you might have a standard header that should be used on all forms created for your company. Rather than have each developer put the header on the forms, you can create a form with the header saved to the shared OR, making it automatically available to all developers to inherit or copy. You can even make it the default form when you create a new form or the main form of new projects. The nice thing about this is that if you need to change the header or add some other object later, you simply change the form in the shared OR. The changes flow through all projects for all developers that inherited from this form the next time they are opened.

```
Forms=
Dialogs=
Projects=
Data Modules=
TimeAcct=

[C:\PROGRAM FILES\BORLAND\DELPHI 3\OBJREPOS\OKCANCL1]
Type=FormTemplate
Name=Standard Dialog
Page=Dialogs
Icon=C:\PROGRAM FILES\BORLAND\DELPHI 3\OBJREPOS\OKCANCL1.ICO
Description=OK, Cancel along bottom of dialog.
Author=Borland
DefaultMainForm=0
DefaultNewForm=0
Ancestor=

[C:\PROGRAM FILES\BORLAND\DELPHI 3\OBJREPOS\OKCNHLP1]
Type=FormTemplate
Name=Dialog with Help
Page=Dialogs
Icon=C:\PROGRAM FILES\BORLAND\DELPHI 3\OBJREPOS\OKCNHLP1.ICO
Description=OK, Cancel, Help along bottom of dialog. This is
  an Inherited Form.
Author=Borland
DefaultMainForm=0
DefaultNewForm=0
Ancestor=C:\PROGRAM FILES\BORLAND\DELPHI 3\OBJREPOS\OKCANCL1
```

Figure 1: A sample Delphi32.dro file.

You can even take this idea to an extreme; you might have an entire maintenance program, with all the logic on one form in the OR. When you want a new program to maintain a particular table, simply inherit from the form in the OR, make the connections to the correct DataSet, drop the fields from the DBExplorer onto the form, and you're done. If you use the inherit option when creating the new form, this would allow you to update all maintenance programs simply by updating the one copy of the form in the shared OR.

What Makes Up the OR?

The OR consists of two main parts: the configuration file and the objects. The configuration file is really nothing more than a plain text file named Delphi32.dro. This file is the heart and soul of the OR, and contains the references to most of the objects shown in the OR dialog box. When you add or remove an object from the OR, you are really adding or removing lines from this file (see

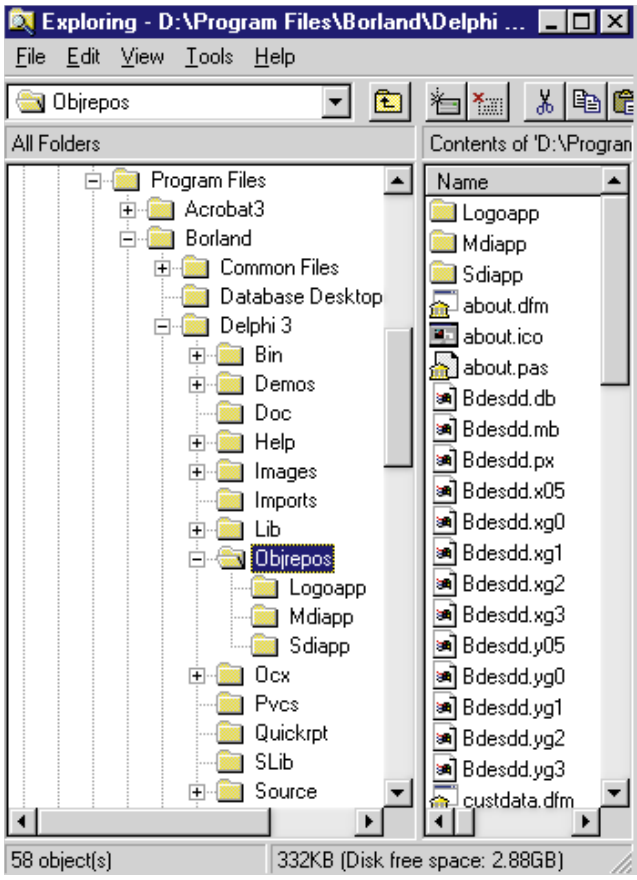


Figure 2: Sample directory structure.

Figure 1). This file, by default, is held in the directory path \Borland\Delphi 3\Bin. All the objects that ship with Delphi are held in the directory \Borland\Delphi 3\Objrepos and its subdirectories (see Figure 2).

Any objects you create and add to the repository don't have to be kept in this directory structure. However, if they are to be shared among your developers, they need to be stored on a shared drive on the LAN.

How to Share the OR

At the bottom of the Preferences tab of the Environment Options dialog box the **Directory** edit box in the **Shared Repository** section allows you to specify the directory you wish to use to locate the shared OR (see Figure 3). When you enter a directory name in this edit box, Delphi will create a Delphi32.dro file there for you if it doesn't exist. The directory you enter must be a shared directory on the LAN, perhaps a shared drive on an NT server. Each Delphi developer must point to the same shared directory. This is a case when you should consider using a UNC name (i.e. \NTServer\ShareName\Delphi\SharedObjRepos) rather than the standard drive:directory naming style (i.e. X:\Delphi\SharedObjRepos).

The reason for the UNC convention is that it references a server and does not depend on what drive letter you have assigned. If you use the drive:directory method, each developer must have the same drive letter assigned to this

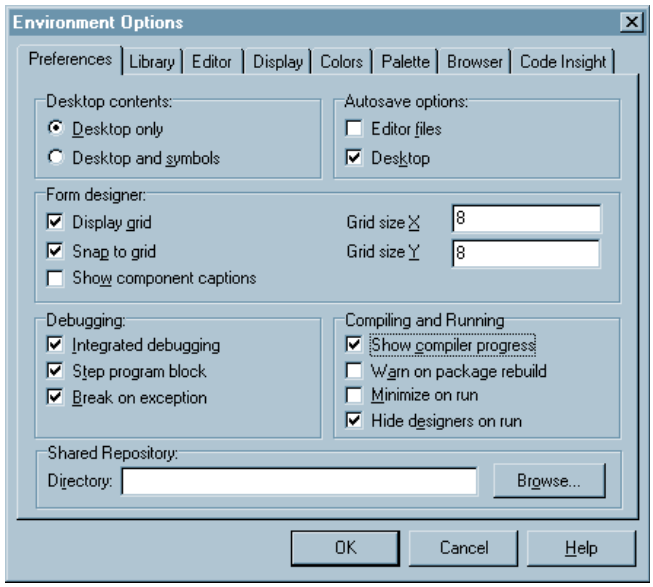


Figure 3: The Environment Options dialog box.

```
Before:
[C:\PROGRAM FILES\BORLAND\DELPHI 3\OBJREPOS\OKCANCL1]
Type=FormTemplate
Name=Standard Dialog
Page=Dialogs
Icon=C:\PROGRAM FILES\BORLAND\DELPHI 3\OBJREPOS\OKCANCL1.ICO
  Description=OK, Cancel along bottom of dialog.
Author=Borland
DefaultMainForm=0
DefaultNewForm=0
Ancestor=

After:
[\\NTSvr1\ShareName\Delphi\ShrdOBJREPOS\OKCANCL1]
Type=FormTemplate
Name=Standard Dialog
Page=Dialogs
Icon=\\NTSvr1\ShareName\Delphi\ShrdOBJREPOS\OKCANCL1.ICO
Description=OK, Cancel along bottom of dialog.
Author=Borland
DefaultMainForm=0
DefaultNewForm=0
Ancestor=
```

Figure 4: Sample change of an entry in the OR.

same share. This is an important point. If you use the drive:directory format and a developer connects to the same server and directory using a different drive letter, they won't be able to access the objects in the shared OR because of the hard-coded reference to the drive letter in the Delphi32.dro file.

Issues of Sharing

The one drawback with simply changing the shared repository location in the Environment Options dialog box is that you lose access to some of the objects that install with Delphi. An easy solution to this is to copy the entire ObjRePos directory structure from the default Delphi install to the new shared directory before changing the shared repository location in the Environment Options dialog box. Then, modify each entry in the Delphi32.dro file to point to the new location (see Figure 4). This allows you to share not only your own objects, but those installed

with Delphi as well. I prefer this method because it provides the most flexibility.

The main concern with sharing the OR is that if a change is made, it's made to all developers who are using the shared OR. All projects for all developers that have inherited from an object in the OR, which is changed, will be affected. Remember that objects that have been copied from objects in the OR are not affected by changing the original object in the OR. The change could be something as simple as a developer changing the default new form or default main form for the shared OR. If one developer changes which form will be the default for all new forms, it affects all users of that shared repository. This change is then forced on all other developers.

While this may be a good way to make global changes, it can be very confusing — and dangerous. For example, developer A changes the default new form before going home on Monday, and developer B comes in Tuesday morning and tries to add a new form to his or her project. However, instead of getting the traditional blank form, developer B gets whatever form developer A selected as the default. If developer B doesn't know what happened, they may lose valuable time trying to determine what is wrong with Delphi. (In my early tinkering with sharing the OR, this exact problem occurred, resulting in a call to Inprise's support line.)

As the preceding example shows, developers that share a LAN-based OR have to be a little more careful about the changes they make. There are times when, as a developer, you want to try out an idea and don't want to expose other developers to any problems that it may cause. In this case, you could simply blank out the **Directory** edit box in the **Shared Repository** section on the Preferences tab of the Environment Options dialog box. This would then point you to the original OR on your machine, and any changes that you make there will only affect you. At this point, you can perform your testing, and when you're finished, you can point back to the shared OR and apply the changes there. I find this useful, as it gives me a 'sandbox' in which to try things out before I change the shared OR.

Conclusion

Sharing the Object Repository is easy, and can be a great tool for standardization. It does require that a little more care be taken on the part of the developers when updating and using it. However, the small amount of risk is worth the immense benefits that can be realized by sharing forms and other objects among all your developers and across your company. Δ

G. Bradley MacDonald is the Technology Planner at the Liquor Distribution Branch of British Columbia, where he is responsible for supporting Delphi and AS/400 developers. He can be reached at Bradley_MacDonald@LDB.GOV.BC.CA or Bradley_MacDonald@BC.Sympatico.CA.

By Cary Jensen, Ph.D.

Delphi Database Development

Part III: The Database Component

In last month's "DBNavigator," we considered the role of the BDEDataSet components. This month's installment continues our extended series on database topics with an introduction to the Database component.

The Database component is responsible for providing BDEDataSet components with information about the nature and location of your data. (*TTable*, *TQuery*, and *TStoredProc* objects are all BDEDataSet components.) Specifically, a Database component stores the location of the data (whether it's local or remote) and what driver to use to access this data. It's also responsible for holding configuration information pertaining to the data access. For example, a Database component can define parameters that control how data is accessed and updated.

There is another, equally important role the Database component plays. It represents your connection to a remote server in a client/server application. Using *TDatabase* methods, you can connect to, or disconnect from, a server; start, commit, and rollback transactions; and store schema information about the files of your database. (Schema information includes data about your database, including the tables, fields, indexes, and so forth.)

A Database component plays a similar role with respect to local data — those file server-based applications that use Paradox or dBASE tables. However, these tables are controlled directly by the BDE (Borland Database Engine). Consequently, no true login connection is required, all transactions are controlled directly by the BDE, and local databases don't store schema information that needs to be read by the database. (Because the BDE controls all access to local tables, schema information is always up-to-date, i.e. it doesn't need to be explicitly read from a remote server.)

Global vs. Local Aliases

You control which Database a particular BDEDataSet uses by assigning an alias to the BDEDataSet's *DatabaseName* property. An alias is either the *DatabaseName* property of a Database component, or the name of a configured database within the BDE configuration file. (You modify your BDE configuration using the BDE Administrator.) An alias that references a database configuration from the BDE is referred to as a "global alias," and one that references the *DatabaseName* property of a Database component is referred to as a "local alias."

(Note: Technically speaking, there is a third value that can be assigned to a BDEDataSet's *DatabaseName* property. If your data is stored in local tables, you can assign the path of your data files to the *DatabaseName* property. While this value is not a true alias, it's more similar to a global alias than to a local alias, in that such a value uses the parameters defined on the Configuration page of the BDE Administrator based on the data type of the file you're accessing.)

Global aliases. A global alias gets its name from the fact that it's available to any application that can use the BDE. For example, any developer using Delphi, C++Builder, or Data Gateway for Java can use a global alias for the access to data. DBDEMOS and IBLOCAL are examples of global aliases.

They're configured during Delphi's installation, and refer to sample data files used by many Delphi sample projects.

An application that uses a global alias is configurable outside of the program logic in your application. Specifically, it's possible to write an application that's completely unaware of the details of the data location, driver type, or connection parameters. Such applications are easily scalable, i.e. you can change the data location, type, and parameters without re-compiling your application. For example, an application that makes use of a global alias can be designed and tested on a stand-alone machine, yet be deployed in a client/server environment without being re-compiled. It's only necessary to update the database configuration for the alias using the BDE Administrator.

Local aliases. In contrast, local aliases are available only to the application in which the corresponding Database component appears. Local aliases are created by adding a Database component to one of your forms or data modules, and then setting the Database's properties to identify the driver and data access parameters.

Normally, you add a Database component that defines a local alias at design time. However, it's perfectly valid, although typically more work, to add one programmatically at run time. This can be done by calling the *TDatabase* class' constructor, and then configuring the properties of the Database object the constructor returns.

Once a Database has been created, the *DatabaseName* property of the Database component is visible to all objects within the application. Specifically, the *DatabaseName* property is stored in the global name space, making this alias visible even to code that appears in units that don't use the unit in which the Database component is referenced (either as a variable or a member field of type declaration).

While a global alias has the advantage of being configurable outside of your application, local aliases also have an important advantage. Using a local alias, your code can control all aspects of the connection. For example, it can be written to define the database driver and access control parameters based on information determined at run time. To determine the location of the data, for instance, your application could read an .INI file on a shared network drive, or it could test the location of the application's executable (using the *Application.ExeName* method, or the *ParamStr(0)* function call).

These two alias types — global and local — aren't mutually exclusive. Indeed, it's not uncommon for the data access information used by a BDEDataSet to come from both alias types at the same time. Specifically, a Database (whose *DatabaseName* property constitutes a local alias) can be configured to read its initial configuration information from a global alias. This is often done for one of two reasons:

- A local Database can selectively override parameters stored in a global alias. This permits your application to leverage the configuration flexibility provided through the BDE configuration, while still maintaining final control over particular parameters. A very simple case of this involves using the parameters stored in a global alias, and then adding or changing one or more parameters, such as a password, in the Database. (Note that a password can't be stored in a BDE configuration file.)
- The second reason for combining global and local aliases is to provide a component for controlling transactions and server connections. Such a local Database may use all the configuration information from a global alias, while providing a convenient component for starting, committing, and rolling back transactions.

Configuring global aliases using the BDE Administrator is beyond the scope of this article. For information on configuring global aliases, please refer to the BDE Administrator's online Help. Configuring local Database components, on the other hand, is the topic of the remainder of this article.

Databases vs. Aliases

It's important to make a clear distinction between a Database component and an alias. A Database component is an instance of the *TDatabase* class. Every BDEDataSet requires a Database component to define and control its access to data. An alias, by comparison, defines either an existing Database component, or a set of parameters that will be applied to an automatically created Database component.

When you attempt to activate a BDEDataSet, it first determines whether it's connected to a Database (via the *DatabaseName* property). If the *DatabaseName* property has been assigned a local alias (the *DatabaseName* property of an existing Database component), the BDEDataSet first checks whether this Database is open. If the Database is open, the BDEDataSet attempts to open itself. If the Database isn't open, the BDEDataSet must first open the Database before opening itself.

When the first BDEDataSet making use of a given global alias attempts to open, it begins by creating an instance of the *TDatabase* class. The parameters used for this Database are drawn from the BDE configuration based on the global alias name. As each additional BDEDataSet that uses the same global alias attempts to open, each will note the Database created by the first BDEDataSet to open, and will attach to that Database.

Configuring a Local Alias

As mentioned earlier, a local alias is one associated with a Database component in your application. Although it doesn't matter how this component is created, the typical technique is to add a Database to a form or data module, and then use the Database Editor dialog box to configure the Database. This technique provides for the automatic creation of the Database component (as part of the creation of the form or data module on which it appears), as well as simplifying the process of configuring a BDEDataSet to use the Database.
This process is demonstrated in the following steps:

1) Create a new project.
2) Add a Data Module by selecting **File | New** from Delphi's

main menu, and then double-clicking the Data Module Wizard on the New page of the Object Repository.

3) Add a Database component to the Data Module.
4) Display the Database Editor (see Figure 1) by double-clicking on the Database, or by right-clicking the Database and selecting **Database Editor**.
5) At **Name** enter `temp`. This value is the local alias name.
6) Move to the **Driver name** drop-down list (we're not going to use the **Alias name** drop-down list in this example). Using the **Driver name** drop-down list, select **STANDARD** (the driver is used with Paradox and dBASE tables).
7) Click the **Defaults** button. This loads the **Parameter overrides** list box with default parameters based on the BDE configuration file. Because a very simple local alias is being created, there are only three parameters: **PATH**, **DEFAULT DRIVER**, and **ENABLE BCD** (Binary Coded Decimal).
8) Leave **DEFAULT DRIVER** and **ENABLE BCD** with their default values. For **PATH**, enter the fully qualified directory path where Delphi stored its sample Paradox and dBASE tables. In Delphi 4 this path is C:\Program Files\Common Files\Borland Shared\Data. In all previous versions, it's the Demos\Data directory under the directory in which Delphi is installed. For example, in Delphi 3 this path is C:\Program Files\Borland\Delphi3\Demos\Data. When you're done, your Database Editor should look like that shown in Figure 2. Click **OK** to accept the dialog box.
9) Ensure the data module is created before the main form. To do this, select **Project | Options** to display the Project Options dialog box. Drag **DataModule2** to the top position in the **Auto-create forms** list. Alternatively, you can simply edit the project (.DPR) file, moving the call to *CreateForm* for the data module to the line before the call to the form's *CreateForm* statement. (Note: Only data modules can appear before the main form in the **Auto-create forms** list. By definition, the main form is the first auto-created form.)
10) Continuing with the building of your main form, return to *Form1*, and add a DBNavigator and DBGrid component to it. Set the DBNavigator's *Align* property to *alTop*, and the DBGrid's *Align* property to *alClient*.
11) Add one DataSource and one Table to the form. Set the DataSource's *DataSet* property to *Table1*. Set the *DataSource* property of both the DBNavigator and the DBGrid to *DataSource1*.
12) We're now ready to configure the Table to use the local alias. Set the Table's *DatabaseName* property to `temp` (the *DatabaseName* property of the Database you entered in step 5). Next, set the Table's *TableName* property to `customer.db`. Finally, set the Table's *Active* property to True.

Run the application. Your form should look as shown in Figure 3.

When you run this application, the data module is created first, making the Database available. Then, the form is created, which causes the creation of the objects that have been placed on it. Once these objects are created, their properties are loaded, including the Table's *Active* property. Before the

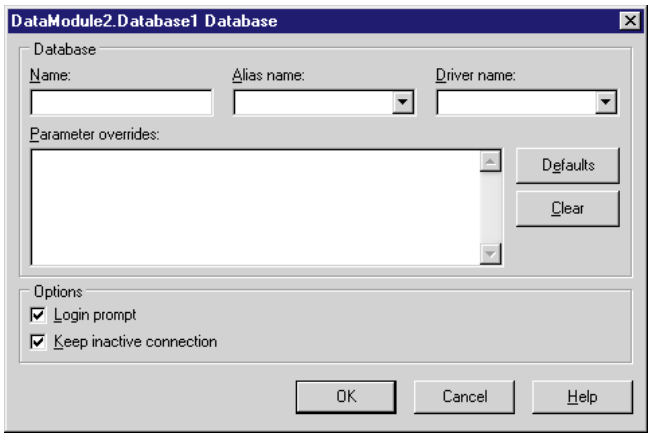


Figure 1: The Database Editor.

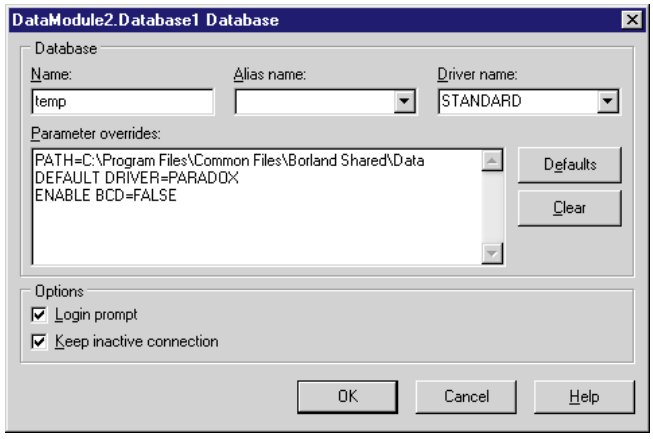


Figure 2: The Database Editor defining a local alias.

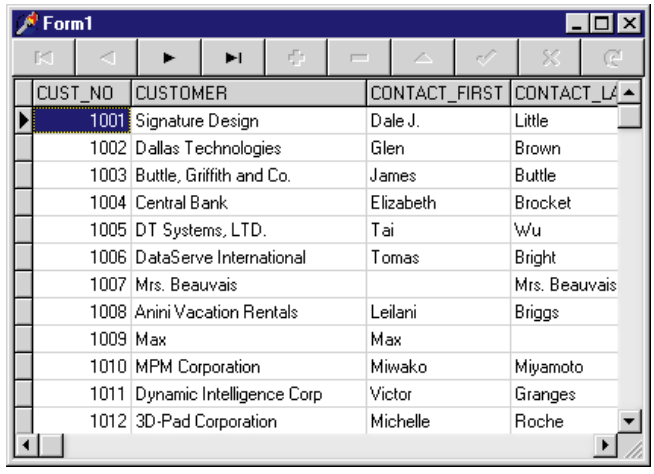


Figure 3: A sample database application that uses a local alias.

Table can open, it locates the Database referenced using the local alias, and attempts to open it. Once the Database is opened, the Table can attempt to open itself. Once the Table is open, the DataSource informs the DBNavigator and the DBGrid to read the data and paint themselves appropriately.

Experienced Delphi database developers who read the previous description will no doubt be wondering why I didn't place the DataSource and the Table on the Data Module along with the Database. The answer is that I wanted to demonstrate that

the local alias was accessible from *Table1*'s **Name** drop-down list, even though the unit in which *Form1* is defined doesn't use the unit in which the data module is defined. If we had placed the Table and/or the DataSource on the Data Module, *Form1*'s unit would be required to use DataModule2's unit.

Controlling Database Parameters at Run Time

In the preceding example, the path to the data was hard coded. As mentioned earlier, it's possible to define the parameters of a Database at run time. To demonstrate this, modify the example project you've created by following these steps:

1) At *Form1*, set *Table1*'s *Active* property to False.
2) At the Data Module, double-click the Database to display its Database Editor. Click the **Clear** button to erase the parameters from the **Parameter overrides** list box.
3) At *Form1*, select **Form1** from the Object Inspector. From the Events page, double-click the **OnCreate** field to generate an *OnCreate* event handler for the form.
4) Edit the *OnCreate* event handler to look like the following:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  DataModule2.Database1.Params.Clear;
  DataModule2.Database1.Params.Add(
    'PATH=D:\Program Files\Common Files' +
    '\Borland Shared\Data');
  DataModule2.Database1.Params.Add(
    'DEFAULT DRIVER=PARADOX');
  DataModule2.Database1.Params.Add('ENABLE BCD=FALSE');
  Table1.Open;
end;
```

5) Finally, with *Form1* selected, select **File | Use Unit** to display the Use Unit dialog box. Select **Unit2** from this list. Note that although the local alias, `temp`, is available to the Table without using *Unit2*, the Database reference (*Database1*) isn't available to *Unit1* unless it's using *Unit2*.
6) Press F9 to run the project. Again, your application should look like the one shown in Figure 3.

Although the actual parameters added to the Database's *Params* property were hard coded in this example, it would have been perfectly acceptable to base these parameters on information determined at run time.

Creating a Local Alias Based on a Global Alias

Earlier in this article you learned that local aliases can be based on global aliases. The following steps demonstrate how this is done:

1) Create a new project.
2) Add a Data Module to the project.
3) Add a Database component to the Data Module.
4) Display the Database Editor for the Database component.
5) Enter `csdemo` as **Name**. This value is the local alias name.
6) Move to the **Alias name** field and select **IBLOCAL**. When you use the **Alias name** field, you're specifying that the parameters of your local alias will be based on the named global alias. If there are any parameters you want to add or override, you specify these using the **Parameter overrides** list box.

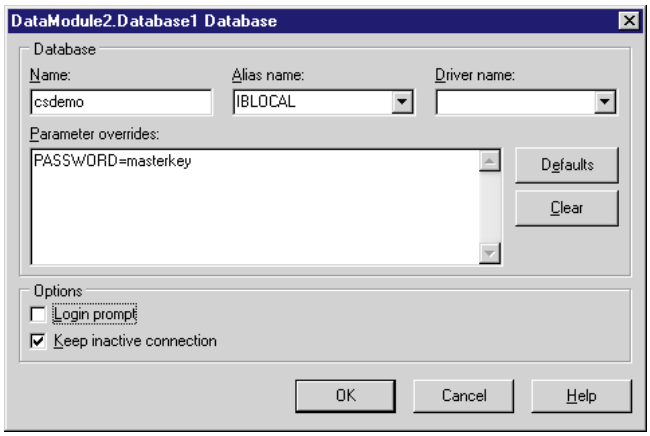


Figure 4: The Database Editor with the password stored.

7) In this case, we'll be adding the password to the local alias. To do this, enter the following line into the **Parameter overrides** list box:

```
PASSWORD=masterkey
```

8) Because the password is now stored with the Database component, it's no longer necessary to prompt the user for the password. To disable the display of the Database Login dialog box, remove the check from the **Login prompt** check box of the Database Editor. The Database Editor should look as shown in Figure 4.
9) The remaining steps of this example are identical to steps 9 through 12 in the example given under the section "Configuring a Local Alias." Consequently, those steps aren't repeated here. However, there is one difference. Instead of adding a Table, use a Query component. To configure the Query, set its *DatabaseName* property to `csdemo`, and its SQL property to the following SQL statement:

```
SELECT * FROM CUSTOMER
```

10) Set the Query's *Active* property to True. The query will execute, and the returned records will be displayed in the DBGrid.

Press F9 to run the project. You'll notice the main form is displayed without prompting for the password. In this case, the local alias uses all parameters of *IBLOCAL*, which identify where the data is located and what driver to use. The Database component, however, adds the password, which is used to establish a connection to the server when the Query component attempts to execute.

This example demonstrates how to override global alias parameters with a local alias. However, it's rarely wise to permit unchallenged access to a database server. Consequently, this technique is normally only used during development, where you would like to avoid having to enter the password each time you test your application. Before deploying such an application, you should return to the Database Editor dialog

box and remove the password parameter from the **Parameter overrides** list box, and enable the **Login prompt** check box.

Conclusion

Database components are used to customize access to a database, as well as to control database connections and transactions. This article demonstrated how to configure a BDEDataSet component to use a Database component, providing you with control over access to the underlying data.

In next month's "DBNavigator" we will take an in-depth look at Data Modules, including when you should use them, and when they should be avoided. Δ

Cary Jensen is President of Jensen Data Systems, Inc., a Houston-based database development company. He is co-author of 17 books, including *Oracle JDeveloper* [Oracle Press, 1998], *JBuilder Essentials* [Osborne/McGraw-Hill, 1998], and *Delphi in Depth* [Osborne/McGraw-Hill, 1996]. He is a Contributing Editor of *Delphi Informant*, and is an internationally respected trainer of Delphi and Java. For information about Jensen Data Systems' consulting or training services, visit http://idt.net/~jdsi or e-mail Cary at cjensen@compuserve.com.

By Peter Dove

The Camera Never Lies

Delphi Graphics Programming: Part VI

It's here! This is the long-awaited finale of the "Delphi Graphics Programming" series that started in January, 1997. In this series, we watched the development of the TGMP 3D-rendering engine, from its inception through several phases of growth (see Figure 1). We're almost done.

In this final installment, we'll cover how to get a camera coordinate system working, adding animated textures, and finally, how to include foreground pictures, i.e. those that allow you to view your 3D objects through a window frame, a cockpit, etc. We haven't been able to include everything we would have liked, but we hope that what follows will be enough to send you on your way to developing your own version.

Smile Please

So far, TGMP allows you to add an object and move it around in 3D space.

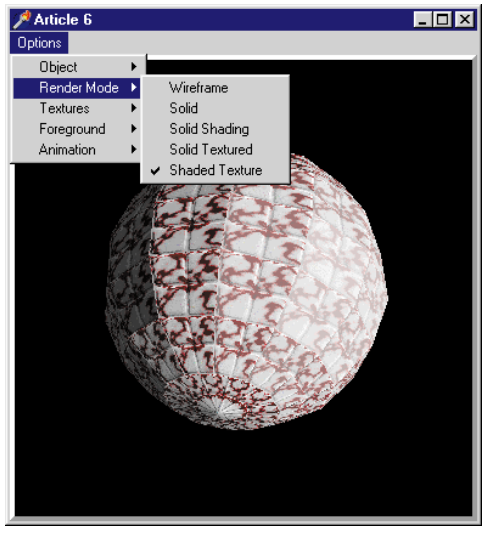


Figure 1: The TGMP application in action.

Unfortunately, you're only able to stand at the center of your 3D universe and look ahead. You can not turn around and see what's behind you, you can not look up or down, and you can not decide to have a walk around. Rather limiting, one would think.

To remedy this situation, we'll be adding a camera coordinate system; we are adding the concept of having a camera that can be moved anywhere in the system by positioning it at coordinates X, Y, and Z, or rotated about its X, Y, and Z axes.

Let's start by adding two variables to the **public** section of the TGMP component:

```
CameraPosition: TPoint3D;
CameraRotation: TPoint3D;
```

Also, we need to add a section to the *TObject3D* record to store the camera coordinates for all the polygons:

```
PolyCamera: array[0..MAXPOLYS] of TPolygon;
```

Adding the camera coordinate functionality works similarly to the way we added the world coordinate system. We created a new method named *WorldToCamera*, which takes the objects after the *LocalToWorld* procedure has processed it.

The explanation of how it works is simple. Imagine you're standing at the center of the 3D universe. You have an object directly in front of you. Now, imagine that the camera (you) moves to the right. If an outside observer saw a snapshot of the scene before and after you moved, they would not be able to tell whether it was the camera (you) that had moved or the object itself. You get the same picture if you move the camera to the right by five units or move the object to the left by five units. The trick of making it look as though we have a camera is based on this principle.

The *WorldToCamera* procedure achieves the trick of camera movement in stages (see Listing Four on page 30). First, the procedure takes the variable *CameraPosition* and subtracts its values from that of all the polygons. Remember that if you move the camera five units to the right (i.e. *CameraPosition.X* is 5), the object looks as though it has been moved five units to the left.

Second, the procedure takes the *CameraRotation* variable and applies the inverse rotations to all the polygons. If, while you look at this page, you rotate your head to the right, it looks as if the page has rotated to the left.

Last, we must remember to apply the same tricks to the light source as well; it also has a position and a direction.

We also need to change two methods for this to work. The first one is *RemoveBackfacesAndShade*. In this method, we need to change all references to *PolyWorld* to *PolyCamera*. The other method we need to change is *RenderNow*. This method must have the statement:

```
WorldToCamera(Object3D);
```

inserted into every section of the **case** statement following the statement:

```
LocalToWorld(Object3D);
```

Also, every reference to *Object3D.PolyWorld[x]* must be replaced with *Object3D.PolyCamera[x]* for the correct polygons to be rendered.

Through the Eye of a Needle

Foreground pictures are nearly as useful as background pictures. By using a foreground picture, you can make your display show the controls of a cockpit in an airplane, or make it seem as though you're looking through a keyhole.

To begin, we need to create two new design-time properties. One is named *ForegroundBitmap*, which is of type *TPicture*. This property allows you to load a bitmap. The second property, *EnableForegroundBitmap*, tells the rendering engine whether to include the foreground in its final image. Both properties need to go into the **published** section of TGMP:

```
property EnableForegroundBitmap : Boolean
  read FEnableForeground write SetEnableForeground;
property ForegroundBitmap : TPicture
  read GetForegroundBitmap write SetForegroundBitmap;
```

The two private members for the properties in the following listing should be placed into the private section of TGMP. Also listed is a support private member named *FForegroundDIB*, which allows a quicker way of putting the bitmap onto the screen:

```
FForegroundDIB : TDib16Bit;     // Stores foreground bitmap.
FForegroundBitmap : TPicture;   // Holds foreground bitmap.
FEnableForeground : Boolean;    // Is foreground enabled?
```

The three private methods that support the new properties are listed in Figure 2, and are straightforward. The last method, *SetEnableForeground*, changes the private member *FEnableForeground*, then calls the *Paint* method to ensure the display keeps up to date with the current state.

The following statements must be added to the *Create* constructor to initialize the new foreground properties:

```
{ ****** Added to support foreground properties ****** }

// Create the foreground Bitmap.
FForegroundBitmap := TPicture.Create;
// Create the foreground DIB.
FForegroundDib := TDib16Bit.Create(ViewHeight, ViewWidth);

{ ****** End of support for foreground properties ****** }
```

A new method, *CopyForeground*, is also created, and is called every time the frame is rendered. *CopyForeground* works in a simple manner (see Figure 3). By the time *CopyForeground*

```
procedure TGMP.SetForegroundBitmap(Value: TPicture);
begin
  FForegroundBitmap.Assign(Value);
end;

function TGMP.GetForegroundBitmap: TPicture;
begin
  result := FForegroundBitmap;
end;

procedure TGMP.SetEnableForeground(Value: Boolean);
begin
  FEnableForeground := Value;
  Paint;
end;
```

Figure 2: These three private methods support *ForegroundBitmap* and *EnableForegroundBitmap*.

```
procedure TGMP.CopyForeground;
var
  X, Y : Integer;
  PixelColor : Word;
begin
  for X := 0 to FForegroundDib.Width - 1 do
    for Y := 0 to FForegroundDib.Height - 1 do begin
      PixelColor := FForeGroundDiB.GetPixel(X,Y);
      if PixelColor <> 0 then
        FDib.SetPixel(X, Y, PixelColor);
    end;
end;
```

Figure 3: The *CopyForeground* method is called every time the frame is rendered.

Figure 4: A sample image with a foreground.

is called, the bitmap you selected as the foreground is stored in *FForegroundDIB*. The scene has been rendered into *FDIB*. TGMP chooses black as its invisible color. It now scans each pixel in the *FForegroundDIB* and transfers it to *FDIB*. If the method comes across the color 0 (Black) in *FForegroundDIB*, it doesn't transfer it to the rendered image.

Figure 4 shows a sample with a foreground image. As you can see, the image has a picture of a frame with a kind of gun scope in the center. If you were to choose this as a foreground image, everything but the black would be transferred onto the window.

There are two more methods that need to be changed to complete the implementation of the foreground bitmap. The first is the *WMSize* method. This has two lines that need to be added to update the height and size of the *FForegroundDIB*:

```
FForegroundDIB.Free;
FForegroundDIB := TDib16Bit.Create(ViewHeight, ViewWidth);
```

The other method is *Paint* (see Figure 5). To transfer the newly chosen background bitmap into *FForegroundDIB* from *FForegroundBitmap*, we to need to create a temporary bitmap in the shape of the local variable *tmpBitmap*. First, we create the temporary bitmap, then we assign the handle of the *FForegroundDIB* to it. This allows us to use the methods belonging to *TBitmap* to write onto the *FForegroundDIB*. We then use the *StretchDraw* method of *TBitmap* to draw the *ForegroundBitmap* into *FForegroundDIB*. After we have finished with *TBitmap*, we release its handle. Releasing the handle this way does not destroy it. We then free it, and the work is done.

Animated Textures

Wouldn't it be great to have a ball of fire with animated flames? That's what got me thinking we should create animated textures. There are many uses; one that springs to mind is spooling video onto 3D objects. We'll implement an animated texturing algorithm that is pretty simple, but we're sure you'll have fun with it. The great thing about this feature is that we get involved with dynamic memory allocation.

```
procedure TGMP.Paint;
var
  tmpBitmap : TBitmap;
begin
  inherited;
  { New ClearBackPage using the DIB class's methods. Notice
    that it now passed a 16-bit color value rather than
    a TColor. }
  FDib.ClearBackPage(CalculateRGBWord(FColor));
  { Create a bitmap. }
  tmpBitmap := TBitmap.Create;
  { Assign the Foreground's Handle to it. }
  tmpBitmap.Handle := FForegroundDIB.GetHandle;
  { Stretch it to the full window. }
  tmpBitmap.Canvas.StretchDraw(
    ClientRect, ForegroundBitmap.Bitmap);
  { Release the handle and free the temporary bitmap. }
  tmpBitmap.ReleaseHandle;
  tmpBitmap.Free;
  { Copy background buffer to main screen. }
  FlipBackPage;
end;
```

Figure 5: The customized *Paint* method.

For this to work, we need only one private variable, *FAnimationList*. Its declaration follows, and should be placed in the **private** section of TGMP:

```
FAnimationList: TList;  // List of all animation frames.
```

We're also going to use three methods to support animated textures. Their declarations should be placed in the public section of TGMP (see Figure 6). Following each method is a section explaining its functionality.

AddAnimationFrame allows the programmer to add a single animation frame to *FAnimationList*. Remember that with this rendering engine, your bitmap size for textures is 128- by-128 pixels. The *AddAnimationFrame* declares a pointer to *TBitmapStorage*. The first line of the method dynamically creates new memory exactly the size of *TBitmapStorage*, and puts the pointer to it in *pBitmapStorage*.

We now add that pointer to *FAnimationList*. (Remember that *TList* simply holds a list of pointers.) After adding the pointer to *FAnimationList*, the next loop copies all the pixels from the parameter of *AddAnimationFrame*, *Bitmap*, into *BitmapStorage*. Notice how we access the *TBitmapStorage* array from the pointer:

```
pBitmapStorage^[x,y]
```

The ^ symbol dereferences the pointer, i.e. it tells the compiler to treat *pBitmapStorage* as type *TBitmapStorage* instead of as a pointer to a *TBitmapStorage* structure. You may have noticed that we're copying the bitmap in upside down (note `[127-y]`). This is because DIBs are stored upside down.

Our next method is *DeleteAnimationFrame*, which accepts one parameter named *Frame* (see Figure 7). This tells the method which frame to delete from *FAnimationList*. Again, we declare a local variable named *pBitmapStorage* as a point-

```
procedure TGMP.AddAnimationFrame(Bitmap : TBitmap);
var
  X, Y : Integer;
  pBitmapStorage : ^TBitmapStorage;
begin
  { Create a dynamic TBitmapStorage. }
  New(pBitmapStorage);
  { Add it to the animation list. }
  FAnimationList.Add(pBitmapStorage);
  { Fill in the data. }
  for X := 0 to 127 do
    for Y := 0 to 127 do
      pBitmapStorage^[x,127-y] :=
        CalculateRGBWord(Bitmap.Canvas.Pixels[X,Y]);
end;
```

Figure 6: Calling three methods that support animated textures.

```
procedure TGMP.DeleteAnimationFrame(Frame: Integer);
var
  pBitmapStorage : ^TBitmapStorage;
begin
  { Check to see the it is a valid frame. }
  if (Frame < 0) or (Frame > FAnimationList.Count) then
    Exit;
  { Get a pointer to the TBitmapStorage. }
  pBitmapStorage := FAnimationList[Frame];
  { Dispose of the memory. }
  Dispose(pBitmapStorage);
  { Decrease the list. }
  FAnimationList.Delete(Frame);
end;
```

Figure 7: The *DeleteAnimationFrame* method accepts the *Frame* parameter.

er to *TBitmapStorage*. The method checks that the *Frame* parameter is valid, then assigns from *FAnimationList* the *n*th (*Frame*) pointer. It then calls the *Dispose* procedure, which releases memory to the exact size of *TBitmapStorage*. The *n*th (*Frame*) entry in the *FAnimationList* is also deleted.

SetCurrentBitmapWithAnimationFrame, the last method in this trio, is easy to figure out:

```
procedure TGMP.SetCurrentBitmapWithAnimationFrame(
  Frame: Integer);
begin
  { Make sure the frame is a valid one. }
  if (Frame < 0) or (Frame > FAnimationList.Count) then
    Exit;
  { Copy the memory of the animation frame
    into FCurrentBitmap. }
  CopyMemory(@FCurrentBitmap, FAnimationList[Frame],
             SizeOf(TBitmapStorage));
end;
```

CurrentBitmap is the texture mapped onto the polygons as they are rendered. So, to create animated textures, we only need to change the *CurrentBitmap* every frame. Obviously, we need a quick way of doing this. The *SetCurrentBitmapWithAnimationFrame* method achieves this by first making sure the *Frame* parameter is valid.

Then, it uses the procedure *CopyMemory*, which accepts three parameters: The first is the destination, and is of type Pointer. I have this as *@FCurrentBitmap*, which means "memory address of *FCurrentBitmap*." The second

```
{ Section 1 - to be used on the OnShow event of the form. }
{ Load animation frames. }
Bitmap := TBitmap.Create;
for X := 1 to 33 do begin
  Bitmap.LoadFromFile(ExtractFilePath(
    Application.ExeName) + 'Frame' + IntToStr(X) + '.bmp');
  GMP1.AddAnimationFrame(Bitmap);
end;
Bitmap.Free;

{ Section 2 - to be used in the Timer event. }
var
  AnimFrame: Integer = 0;
begin
  if Start1.Checked then begin
    GMP1.SetCurrentBitmapWithAnimationFrame(AnimFrame);
    Inc(AnimFrame);
    if AnimFrame > 32 then
      AnimFrame := 0;
  end;
```

Figure 8: A simple example of calling animated frames.

parameter is the source, the place from which we want to copy. This parameter is also of type Pointer. I've used *FAnimationList[Frame]*, a pointer to the *TBitmapStructure* of the *n*th (*Frame*) frame. The last parameter is the amount (in bytes) of the memory to copy, which in our case is *SizeOf(TBitmapStorage)*. The *SizeOf* function returns the size of an object/record/structure in bytes.

To show you how to use animated textures, we've included a snippet from the demonstration application (see Figure 8). The first section shows how to set up the animation, and the second section shows how to animate the textures.

Conclusion

That's it! You now have a basic 3D-rendering engine. Of course, it can be made to go faster and have much more functionality, but you're on the right path to take it further yourself. This series took us through the evolution of the TGMP rendering component using basic techniques, such as properties, events, and property editors, as well as more advanced techniques, including device-independent bitmaps, sprites, and memory management. We covered a lot in this series, so I'm sure you will find many of the techniques useful for your own projects. You should also be convinced that Delphi is worthy of respect as a games and graphics programming language. Δ

[Note: Parts 1 through 5 of this series were published in the following issues of *Delphi Informant* (all in 1997): January, February, March, April, and July.]

The entire sample application referenced in this article is available on the Delphi Informant Works CD located in INFORM\98\NOV\DI9811PD.

Peter Dove is Managing Director of Kortex Systems Limited, based in the United Kingdom. Kortex Systems Limited has a range of software development services that can be seen at http://www.kortex.co.uk. Peter can be e-mailed at peterd@kortex.co.uk.

Begin Listing Four — *WorldToCamera* procedure

```
procedure TGMP.WorldToCamera(var AnObject: TObject3D);
var
  X : Integer;
begin
  { Loops though all polygons, transferring them to
    PolyCamera. }
  with AnObject do begin
    for X := 0 to NumberPolys -1 do begin
      { Additional optimizations. }
      with PolyCamera[x] do begin
        { Subtract camera coordinates from
          world coordinates. }
        Point[0].X :=
          PolyWorld[X].Point[0].X - CameraPosition.X;
        Point[0].Y :=
          PolyWorld[X].Point[0].Y - CameraPosition.Y;
        Point[0].Z :=
          PolyWorld[X].Point[0].Z - CameraPosition.Z;

        Point[1].X :=
          PolyWorld[X].Point[1].X - CameraPosition.X;
        Point[1].Y :=
          PolyWorld[X].Point[1].Y - CameraPosition.Y;
        Point[1].Z :=
          PolyWorld[X].Point[1].Z - CameraPosition.Z;

        Point[2].X :=
          PolyWorld[X].Point[2].X - CameraPosition.X;
        Point[2].Y :=
          PolyWorld[X].Point[2].Y - CameraPosition.Y;
        Point[2].Z :=
          PolyWorld[X].Point[2].Z - CameraPosition.Z;

        Point[3].X :=
          PolyWorld[X].Point[3].X - CameraPosition.X;
        Point[3].Y :=
          PolyWorld[X].Point[3].Y - CameraPosition.Y;
        Point[3].Z :=
          PolyWorld[X].Point[3].Z - CameraPosition.Z;

        { Rotate the points using the camera rotation. }
        RotatePoint(1, 0, 0, -CameraRotation.X, Point[0]);
        RotatePoint(0, 1, 0, -CameraRotation.Y, Point[0]);
        RotatePoint(0, 0, 1, -CameraRotation.Z, Point[0]);

        RotatePoint(1, 0, 0, -CameraRotation.X, Point[1]);
        RotatePoint(0, 1, 0, -CameraRotation.Y, Point[1]);
        RotatePoint(0, 0, 1, -CameraRotation.Z, Point[1]);

        RotatePoint(1, 0, 0, -CameraRotation.X, Point[2]);
        RotatePoint(0, 1, 0, -CameraRotation.Y, Point[2]);
        RotatePoint(0, 0, 1, -CameraRotation.Z, Point[2]);

        RotatePoint(1, 0, 0, -CameraRotation.X, Point[3]);
        RotatePoint(0, 1, 0, -CameraRotation.Y, Point[3]);
        RotatePoint(0, 0, 1, -CameraRotation.Z, Point[3]);

        { Rotate the light source. }
        TransLightSource := LightSource;
        RotatePoint(1, 0, 0, -DegToRad(CameraRotation.X),
                    TransLightSource);
        RotatePoint(0, 1, 0, -DegToRad(CameraRotation.Y),
                    TransLightSource);
        RotatePoint(0, 0, 1, -DegToRad(CameraRotation.Z),
                    TransLightSource);

        NumberPoints  := PolyWorld[X].NumberPoints;
        PolyColor     := PolyWorld[X].PolyColor;
        DibColor      := PolyWorld[X].DibColor;
      end;
    end;
  end;
end;
```

End Listing Four

By Gregory Deatz

Thread-Safe DLLs

Creating Win32 DLLs for Multi-threaded Applications

Multi-threaded applications are now the norm, not the exception. Indeed, the safest assumption to make about a Win32 program is that it uses multi-threading. Therefore, as software houses publish mechanisms for extending an application's functionality, the developer is forced to write thread-safe libraries.

When most of us think about the implementation of a particular procedure or function, we generally think in terms of local and global variables. However, a procedure or function in a DLL can be called by multiple threads simultaneously, and care must be taken to ensure that variables previously understood to be "global" are not actually "thread-local."

Take, for example, functions that must retain "state" between calls. A developer might choose to implement a set of functions like:

```
function GetFirstWord(st: string): string;
function GetNextWord(st: string): string;
```

When a developer needs to parse a string, he or she first calls *GetFirstWord*, which returns the first word of *st*. The developer is unaware that *GetFirstWord* remembers where it stopped scanning *st*, so subsequent calls to *GetNextWord* will return the second word, then the third word, and so on — until *GetNextWord* is forced to return an empty string because there are no more words in *st*. Clearly, these functions can be implemented in a thread-safe manner, and they can also be implemented without taking the possible existence of threads into consideration.

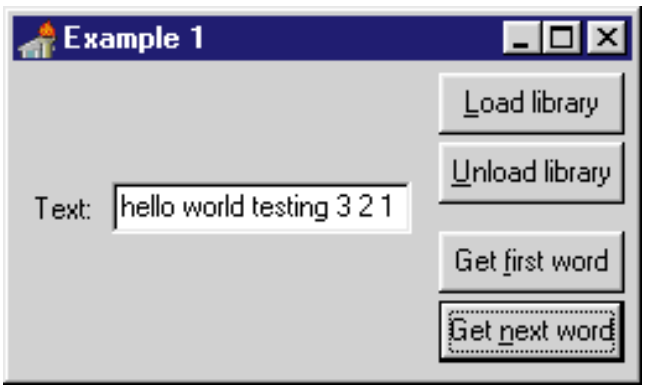


Figure 1: An example of how to declare a valid *DllProc* and how to allow it to be called.

Note: There are many ways to achieve similar *GetNextWord* functionality; this is just one implementation. The key is to recognize that there are types of questions where it's desirable that the function library, and not the calling application, retain state between function calls. For readers generally unfamiliar with management of thread-local variables, and the need for them, the Exe1 program (see Figure 1) and its accompanying DLL (Dll1) are intended to demonstrate the simple case of a single-threaded process and a non-thread-safe DLL. They provide the simple base for all subsequent thread-safe examples, and are left for the reader to study. (The complete source for Exe1, Dll1, and all other demonstration programs discussed in this article are available for download; see end of article for details.)

Delphi's Memory Manager

Before we go too far, we must discuss the Delphi memory manager. Way back when, when Delphi was Borland Pascal, and it was built for DOS, the memory manager did not have to concern itself with threads, so the memory manager Delphi used in version 1 was not thread-safe.

```
unit DllCode;
  ...
implementation
  ...
procedure DllEntry(Reason: Integer);
  ...
initialization
  ...
  DllProc := @DllEntry;
  ...
end.
```

Figure 2: Declaring and setting up *DllProc* so it can be called.

For some strange reason, Borland resolved the problem by introducing a global variable. The variable, *IsMultiThread*, when set to True, tells the memory manager to wrap itself in a Windows-critical section, thus ensuring the memory manager is thread-safe.

When using Delphi's *TThread* class, the developer is assured that this variable is set. However, if the developer uses "raw" Windows threads, or develops a DLL for a multi-threaded application, this variable must be set manually.

This is very important! When developing a DLL for a multi-threaded application, you must set the *IsMultiThread* variable.

Managing Thread-local Variables

The key to proper initialization and finalization of thread-local variables is a procedure named *DllEntryPoint*, which is an inherent part of all Win32 DLLs, and it is called whenever a process or thread attaches to a DLL. Delphi gives the user access to this procedure through a procedure pointer named *DllProc*, which is defined in the System unit.

DllEntryPoint takes a single Integer argument. The possible values of this argument are DLL_PROCESS_ATTACH, DLL_PROCESS_DETACH, DLL_THREAD_ATTACH, and DLL_THREAD_DETACH (more about these in just a bit). Figure 2 shows how to declare a valid *DllProc* and how to get it called.

Note that there is no declaration for *DllEntry* in the **interface** section. This is because the existence of the procedure need only be known to the unit itself. When a process first attaches to Dll2, the **initialization** section of *DllCode* executes, where *DllProc* is set to the address of *DllEntry*.

Now that a process has attached to the DLL, *DllEntry* will be called with DLL_THREAD_ATTACH whenever a thread is created in the calling application. Whenever a thread in the calling application exits gracefully, *DllEntry* will be called with DLL_THREAD_DETACH. When a process is unloading the library, *DllEntry* will be called with DLL_PROCESS_DETACH.

Note that because we were not able to specify our *DllEntryPoint* procedure until the initialization of the DLL, *DllEntry* will never be called with DLL_PROCESS_ATTACH. This might be considered a problem, except that the **initialization** section of a unit in a DLL is exactly equivalent to the call:

```
DllEntryPoint(DLL_PROCESS_ATTACH)
```

Likewise, the **finalization** section of a unit is exactly equivalent to the call:

```
DllEntryPoint(DLL_PROCESS_DETACH)
```

Because process initialization and finalization take place in the obvious places in a Delphi unit, all examples of *DllEntry* will only respond to DLL_THREAD_ATTACH and DLL_THREAD_DETACH.

Two Important Issues

Issue one. Suppose an application starts some threads; it then loads the DLL. The DLL is never explicitly told that those threads are executing, i.e. *DllEntry* will never be called with the DLL_THREAD_ATTACH argument for those threads. However, if those threads exit gracefully, the DLL will be informed they are closing.

This means the DLL is potentially responsible for cleaning up uninitialized data.

Issue two. Suppose an application loads a DLL; it then starts some threads. Before gracefully closing its threads, the application detaches from the DLL. The DLL will be told the process is detaching (with DLL_PROCESS_DETACH), but it won't be explicitly told that each thread is also detaching.

This means that any clean-up procedures associated with the freeing of resources in the DLL won't be called. (It's left as a bit of a brain-teaser to figure this one out.)

So what do these issues mean? In brief: Take care! Unfortunately, there is nothing intuitive about these realities, although they're relatively easy to understand using an example. The example program Exe2a allows you to load Dll2 and watch it call process-level and thread-level initialization and finalization. (To make sure you have a clear understanding of the previous comments, you should make it a point to play around with variations of creating and destroying threads and loading and unloading the DLL.)

But what's to manage here? If you're using thread-local scalars (native Delphi types like Integer) or Delphi strings, you don't have much to worry about. Exe2b demonstrates how Dll2 uses a **threadvar** declaration of an Integer to maintain a thread-local index into the string passed to *GetFirstWord* and *GetNextWord*. (Again, Exe2a and Exe2b are available for download; see end of article for details.)

Can Delphi Burp?

In conjunction with *DllEntry* and the **initialization** and **finalization** sections of the *DllCode* unit, **threadvar** makes it easy to manage thread-local variables. Right? Not quite.

```
procedure DllEntry(Reason: Integer);
begin
  case Reason of
    DLL_THREAD_ATTACH:
      begin
        tlObject := TTestObject.Create;
        DllShowMessage(tlObject.ObjectName);
      end;
    DLL_THREAD_DETACH:
      begin
        if (tlObject = nil) then
          DllShowMessage('Object is nil.')
        else
          DllShowMessage(tlObject.ObjectName);
        tlObject.Free;
      end;
  end;
end;
```

Figure 3: The *DllEntry* procedure.

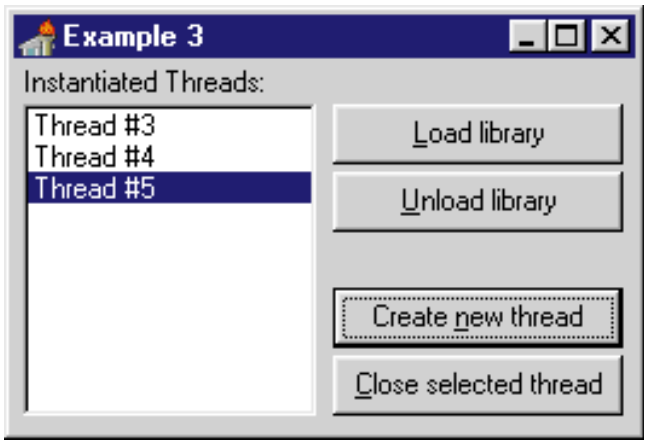


Figure 4: Creating threads in the Dll3 demonstration program.

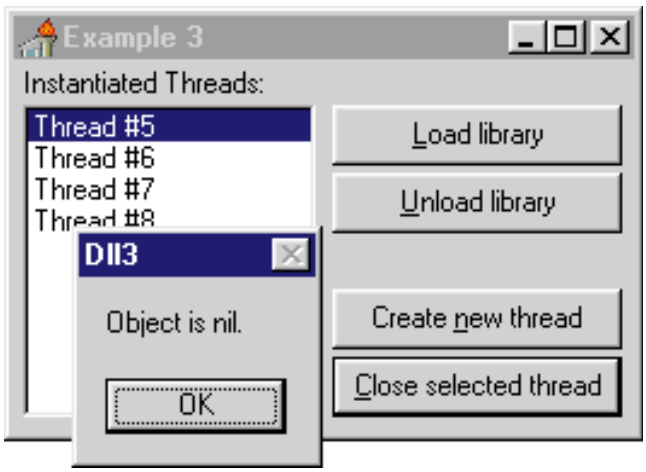


Figure 5: Closing one of the first three threads, i.e. one of those created before the DLL was loaded.

Let's review for a moment. We have navigated process-level and thread-level initialization and finalization of DLLs. It's quite possible that we've exhausted the topic. You should have all the tools you require to go off and write completely thread-safe DLLs, and manage the instantiation and clean-up of thread-local objects. Or should you?

Here's the way thread-local storage is supposed to work: All thread-local storage is zero-initialized. This means that unless you put a value in a thread-local variable, you're guaranteed it's "zeroed out." And, indeed, direct use of Win32's thread-local storage API (discussed later) shows this to be true. However, Delphi has a quirk where its method of thread-local storage has a small gas problem.

The demonstration programs Dll3 and Exe3 demonstrate how we can get Delphi to burp. First, look carefully at the source code for *DllEntry* in Figure 3. The code is simple, and it makes it clear that when a thread attaches to the DLL, it will create a thread-local instantiation of *TTestObject* in *tlObject*. Likewise, when a thread detaches from a DLL, it will free whatever resources it allocated for *tlObject*.

Now, to get Dll3 to burp:

1) Launch Exe3.
2) Start three threads (see Figure 4), noting that because the DLL is not yet loaded, no entry points in the DLL are being called. (Three is an arbitrary number, and is sufficient for demonstrating the problem.)
3) Load the DLL.
4) Start three more threads, noting the messages the DLL displays.
5) Go back to the first thread you loaded; close the thread, noting the fact that *tlObject* is **nil** (see Figure 5). Do this for the next two threads.
6) Now try to close the next thread (see Figure 6). And the next. And the next (see Figure 7).

What happened to the thread-local objects we created? They appear to have been lost in the shuffle. Obviously, something is wrong, as the code is not behaving as expected, and the problem isn't in the code we've written. The code is absurdly simple!

Forgetting about whose bug it is, the next most obvious question is: How do we work around this issue, so that we can maintain thread-local objects?

The Win32 API

We can work around this problem by directly using the Win32 API. *TLSAlloc*, *TLSFree*, *TLSGetValue*, and *TLSSetValue* are the system calls used to manage thread-local storage:

- *TLSAlloc* is used to allocate an index into the thread-local "store." A single, global index refers to a thread-local pointer — a 32-bit value that is local to the thread.
- *TLSFree* is used to free an index that was previously allocated using *TLSAlloc*.
- *TLSGetValue* is used to query the value of the thread-local pointer referred to by the index allocated using *TLSAlloc*.
- *TLSSetValue* is used to set the value of the thread-local pointer referred to by the index allocated using *TLSAlloc*.

A code snippet from Dll4 (shown in Figure 8) demonstrates how each of these functions are used. This is what happens, step by step:

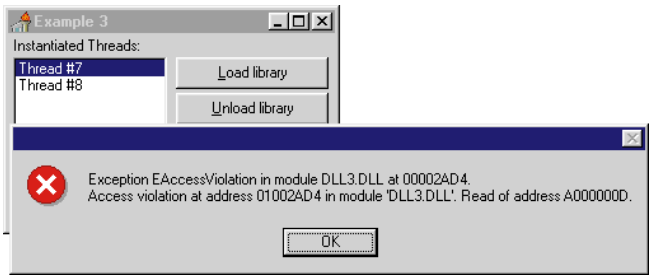


Figure 6: After closing the fourth thread, i.e. one created after the DLL was loaded. There appears to be a problem.

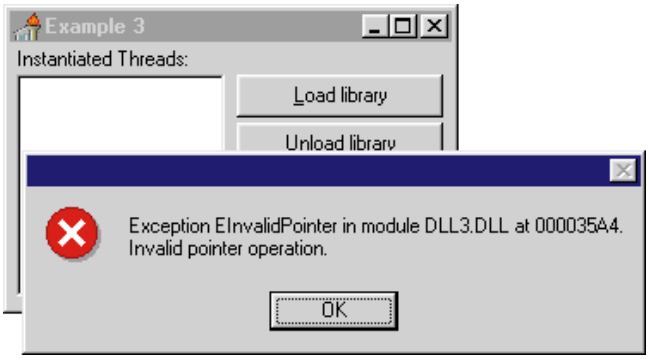


Figure 7: After closing the final thread ... oops!

1) The **initialization** section allocates an index into the thread-local store and saves this index in *tlsObjectIndex*.
2) It then sets *DllProc* to the address of *DllEntry*.
3) Now it calls *CreateObject*, which sets the thread-local pointer referred to by *tlsObjectIndex* to the pointer to the newly created *TTestObject*. *CreateObject* uses *TLSGetValue* to then retrieve the newly created thread-local object, so that its *ObjectName* property can be displayed back to the user.
4) Whenever a thread is created, *CreateObject* is called.
5) Whenever a thread detaches from the DLL, *FreeObject* is called, which first displays a message informing the user either that the object was never initialized, or it displays the *ObjectName* property of the thread-local object. *FreeObject* then frees the object referred to by the thread-local pointer, and it then sets the thread-local pointer back to **nil**.
6) When the library is finally unloaded, it ensures the process-level thread frees up any thread-local storage by calling *FreeObject*.
7) Finally, *tlsObjectIndex* is freed using *TLSFree*.

It should be clear that the semantics of the Win32 system calls and Delphi's **threadvar** are identical. It's just a bit more cumbersome to use the Win32 calls.
You can run Exe4 and follow the burping exercise in the previous section. You'll notice our fourth example runs as expected, and Dll4 does not misbehave during clean-up.

Conclusion

We've just studied thread-local variables and the DLL entry procedure, and we've seen that maintaining a truly thread-safe environment is generally easy, except that management of objects can be a little tricky.

```
unit DllCode;
...
procedure CreateObject;
procedure FreeObject;

var
  tlsObjectIndex: DWord;
  ...
implementation
  ...
procedure CreateObject;
begin
  // First, create the new object and store it in the
  // thread-local slot referred to by tlsObjectIndex.
  TLSSetValue(tlsObjectIndex,
              Pointer(TTestObject.Create));
  // Get object just created and display its ObjectName.
  DllShowMessage(TTestObject(
    TLSGetValue(tlsObjectIndex)).ObjectName);
end;

procedure FreeObject;
begin
  // Get thread-local value and report it's nil, or give
  // the ObjectName property that was stored there.
  if (TLSGetValue(tlsObjectIndex) = nil) then
      DllShowMessage('Object is nil.')
  else
    DllShowMessage(
      TTestObject(TLSGetValue(
      tlsObjectIndex)).ObjectName);
  // Free the object.
  TTestObject(TLSGetValue(tlsObjectIndex)).Free;
  // Set its value in the thread-local store to nil.
  TLSSetValue(tlsObjectIndex, nil);
end;
...
procedure DllEntry(Reason: Integer);
begin
  case Reason of
    DLL_THREAD_ATTACH: CreateObject;
    DLL_THREAD_DETACH: FreeObject;
  end;
end;
...
initialization
  tlsObjectIndex := TLSAlloc;
  DllProc := @DllEntry;
  CreateObject;

finalization
  FreeObject;
  TLSFree(tlsObjectIndex);

end.
```

Figure 8: An example of how the *TLSAlloc*, *TLSFree*, *TLSGetValue*, and *TLSSetValue* functions are used.

Here are some closing thoughts on maintaining a thread-safe environment within DLLs (even in applications):

- *DllEntryPoint* is a nifty way for a DLL to stay aware of active threads in the calling application.
- It's generally safe to use the **threadvar** construct to maintain thread-local scalars, including Delphi's native **string** type.
- It is generally *not* safe to use the **threadvar** construct to maintain thread-local objects. Instead, you should use the Win32 API directly. Although it's a bit more cumbersome, it's still quite simple to use.
- If you are writing an application that must maintain thread-local scalars and objects, it's probably better to

use the Win32 API functions exclusively — if anything, this promotes consistency.

- Before making the decision to maintain thread-local variables, one not-so-obvious question to ask yourself is: Do you really need them? This article was written with the assumption that thread-local variables are sometimes desirable and possibly even necessary. However, many functions, including *GetFirstWord* and *GetNextWord*, can be implemented without global and thread-local variables.

When developing DLLs for third-party applications, the developer generally has no idea of the internal workings of the calling application: the application could be multi-threaded, or it could be single-threaded. This article has explained how to write a thread-safe DLL, even if the developer doesn't know how the calling application uses threads. Along the way, we've discussed the Windows *DllEntryPoint* function, thread-local storage, and appropriate clean-up of "thread-locally" instantiated structures and objects. Δ

All demonstration programs referenced in this article are available on the Delphi Informant Works CD located in INFORM\98\NOV\DI9811GD.

Gregory Deatz is a senior programmer/analyst at Hoagland, Longo, Moran, Dunst & Doukas, a law firm in New Brunswick, NJ. He has been working with Delphi and InterBase for approximately two years and has been developing under the Windows API for approximately five years. His current focus is in legal billing and case management applications. He is the author of FreeUDFLib, a free UDF library for InterBase written entirely in Delphi and hosted at http://www.interbase.com. He can be reached via e-mail at gdeatz@hlmdd.com, by phone at (732) 545-4579, or by fax at (732) 545-4579.

By Yorai Aminov

Is Delphi Running the Code?

Determining If a Program Was Started in the Delphi IDE

How can I tell if code is running under Delphi? It's an important question to shareware authors who want to make their programs available only within the Delphi IDE. At least until they're paid for! A common answer is to look for one of Delphi's standard windows, but that's not a satisfactory method. The fact that Delphi is running doesn't necessarily mean code is being executed under its control.

This article presents a method for detecting if code is running under the control of Delphi, i.e. if it was run from the Delphi IDE. Unfortunately, a great deal of the data needed to determine this can only be obtained by undocumented functions. An additional obstacle is that both the documented and undocumented functions necessary for this task are incompatible across Windows platforms. The method used in this article handles Windows 95 and Windows NT.

This article assumes the reader is familiar with Win32 processes; a full explanation of processes is beyond the scope of this article. If you are not familiar with them, I suggest *Advanced Windows* by Jeffrey Richter [Microsoft Press, 1995]. Another source is the Win32 SDK. An online version of the SDK documentation is available at http://www.microsoft.com/msdn.

The Necessary Information

To determine if it's running under the control of Delphi, the code needs to know two things: It needs to determine if it was started by Delphi, and if it's being debugged. If the answer to both of these questions is yes, the code is running under Delphi's control.

Although the questions are simple, obtaining the answers to them can be complicated. For example, Windows NT supplies the *IsDebuggerRunning* function, which returns True if a process is running in the context of a debugger. Windows 95, however, provides no documented way of determining this information. Similarly, the ToolHelp32 functions provide a simple method for determining the parent of a process, but are implemented only in Windows 95. According to the Microsoft Win32 SDK, these functions are also implemented in Windows NT 5.0. Under earlier versions of NT, however, this information must be obtained through undocumented methods.

Because of these limitations, four separate algorithms must be provided: two for determining the parent process (one for Windows NT and one for Windows 95), and two to check for the presence of a debugger.

Determining the Parent Process

Under Windows 95 (and NT 5.0), the ToolHelp32 functions can be used to provide information regarding processes. The ToolHelp32 functions, constants, and data types are defined in the TlHelp32 import unit, supplied with Delphi 3. Unfortunately, this unit implicitly links the functions to the executable, preventing any code using the unit from running on NT. To make the code portable, explicit linking using the *LoadLibrary* and *GetProcAddress* functions must be employed.

```
type
  PProcessEntry32 = ^TProcessEntry32;
  TProcessEntry32 = record
    dwSize: DWORD;
    cntUsage: DWORD;
    th32ProcessID: DWORD;
    th32DefaultHeapID: DWORD;
    th32ModuleID: DWORD;
    cntThreads: DWORD;
    th32ParentProcessID: DWORD;
    pcPriClassBase: Longint;
    dwFlags: DWORD;
    szExeFile: array[0..MAX_PATH - 1] of Char;
  end;
```

Figure 1: The *TProcessEntry32* record.

```
function GetParentProcessIDForWindows: Integer;
var
  Kernel32: THandle;
  CreateToolhelp32Snapshot: TCreateToolhelp32Snapshot;
  Process32First: TProcess32First;
  Process32Next: TProcess32Next;
  Snapshot: THandle;
  Entry: TProcessEntry32;
  WalkResult: Boolean;
  ID: Integer;
begin
  Result := 0;
  Kernel32 := LoadLibrary('KERNEL32.DLL');
  if Kernel32 <> 0 then begin
    CreateToolhelp32Snapshot :=
      GetProcAddress(Kernel32,'CreateToolhelp32Snapshot');
    Process32First := GetProcAddress(Kernel32,
                                     Process32First');
    Process32Next := GetProcAddress(Kernel32,
                                    'Process32Next');
    if Assigned(CreateToolhelp32Snapshot) and
       Assigned(Process32First) and
       Assigned(Process32Next) then begin
      ID := GetCurrentProcessId;
      Snapshot :=
        CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
      if Snapshot <> -1 then begin
        Entry.dwSize := SizeOf(TProcessEntry32);
        WalkResult := Process32First(Snapshot, Entry);
        while (GetLastError <> ERROR_NO_MORE_FILES) and
              (Result = 0) do begin
          if WalkResult then begin
            if Entry.th32ProcessID = ID then
              Result := Entry.th32ParentProcessID;
          end;
          Entry.dwSize := SizeOf(TProcessEntry32);
          WalkResult := Process32Next(Snapshot, Entry);
        end;
        CloseHandle(Snapshot);
      end;
    end;
    FreeLibrary(Kernel32);
  end;
end;
```

Figure 2: The *GetParentProcessIDForWindows* function accesses the ToolHelp32 functions from KERNEL32.DLL, and walks the process list to retrieve the parent process ID.

The ToolHelp32 functions work by creating a snapshot, then walking through it. A *snapshot* is a description of the requested information at a single moment. Because the information handled by the ToolHelp32 functions — processes, threads, modules, and heaps is constantly changing, a snapshot ensures the data won't change while being examined.

Such a snapshot is required to determine the parent of the current process. The snapshot is retrieved by calling the *CreateToolHelp32Snapshot* function, specifying the TH32CS_SNAPPROCESS flag. The function returns a handle to a snapshot, which must be released later by calling *CloseHandle*.

Once the snapshot is obtained, the *Process32First* and *Process32Next* functions are used to "walk" the snapshot. These functions return True for success, and False for failure. If the function fails, the *GetLastError* function can be used to retrieve error information. When *GetLastError* returns ERROR_NO_MORE_FILES, the end of the process list has been reached. The *Process32First* and the *Process32Next* functions take a *TProcessEntry32* record (see Figure 1) as a parameter. The functions fill this record with information. The only fields of interest to us are *th32ProcessID*, which we compare with our own process ID (obtained by calling the *GetCurrentProcessId* function), and *th32ParentProcessID*, the process ID of the parent process.

Figure 2 lists the *GetParentProcessIDForWindows* function, which returns the parent process ID under Windows 95 and NT 5.0 using ToolHelp32 functions. The function uses *LoadLibrary* and *GetProcAddress* to access the ToolHelp32 functions from KERNEL32.DLL, and walks the process list to retrieve the parent process ID.

Under versions of NT earlier than 5.0, the ToolHelp32 functions are not implemented, nor does any documented way of obtaining the parent process ID exist. The solution is to use the undocumented *NtQueryInformationProcess* function. The function is defined in NTDDK.H in the Microsoft Windows NT Device Driver Kit, along with the data structures it uses. This function can return many types of information regarding a process, but for this article, only the PROCESS_BASIC_INFORMATION class is necessary. The basic process information includes the ID of the parent process. Figure 3 lists the *TProcessBasicInformation* type, translated from the C header file.

The *NtQueryInformationProcess* function is implemented in NTDLL.DLL, and its address can be retrieved by calling *GetProcAddress*. The function takes as a parameter a handle to a process (which we obtain by calling *GetCurrentProcess*), the type of information requested, a pointer to a buffer to receive information, the size of the buffer, and a pointer to a variable that receives the actual size of the output data. *GetParentProcessIDForNT* simply loads NTDLL.DLL, retrieves the address of *NtQueryInformationProcess*, and calls it to retrieve the parent process ID, stored in *TProcessBasicInformation*'s *InheritedFromUniqueProcessID* field.

```
type
  TProcessBasicInformation = packed record
    ExitStatus: Integer;
    PebBaseAddress: Pointer;
    AffinityMask: Integer;
    BasePriority: Integer;
    UniqueProcessID: Integer;
    InheritedFromUniqueProcessID: Integer;
  end;
```

Figure 3: *TProcessBasicInformation*.

```
function GetParentProcessID: Integer;
var
  OSVersionInfo: TOSVersionInfo;
begin
  OSVersionInfo.dwOSVersionInfoSize :=
    SizeOf(TOSVersionInfo);
  GetVersionEx(OSVersionInfo);
  if (OSVersionInfo.dwPlatformId=VER_PLATFORM_WIN32_NT)and
     (OSVersionInfo.dwMajorVersion < 5) then
    Result := GetParentProcessIDForNT
  else
    Result := GetParentProcessIDForWindows;
end;
```

Figure 4: *GetParentProcessID* checks the operating system version and platform, and calls the appropriate function.

The *GetParentProcessID* function (see Figure 4) simply checks the operating system version and platform, and calls the appropriate function to do the work. If the code is running on Windows 95 or NT 5.0 and higher, it uses the ToolHelp32 version; otherwise, it uses *NtQueryInformationProcess*. Once the parent process ID has been obtained, it can be compared with Delphi's process ID (explained later in this article). Even if the IDs match, this is insufficient proof that Delphi controls the code. It's quite possible the code is part of a program started by Delphi, for example, through the **Tools** menu. It's also necessary to make sure the process is being debugged.

Determining the Presence of a Debugger

Again, Windows 95 and NT differ. NT provides the *IsDebuggerRunning* function, which returns True if the current process is being debugged by a Ring 3 debugger, such as Delphi's integrated debugger. Neither NT nor Windows 95 provide a way, documented or not, of determining whether a kernel-mode debugger, such as SoftICE, is running. There are undocumented ways of determining whether a specific kernel-mode debugger is running, but the question itself is irrelevant to this article.

Under Windows 95, there is no documented way of determining if a process is being debugged. Again, this information can only be obtained using undocumented techniques.

When a new process is created under Windows 95, the kernel creates a Process Database (PDB) for it. A process database is an internal object that contains information about a process. In *Windows 95 System Programming Secrets* [IDG Books, 1995], Matt Pietrek describes the PDB in detail. For

```
type
  PProcessDatabase = ^TProcessDatabase;
  TProcessDatabase = packed record
    DontCare1: array[0..7] of Integer;
    Flags: Integer;
    DontCare2: array[0..11] of Integer;
    DebugeeCB: Integer;
    DontCare3: array[0..22] of Integer;
    DontCare4: Word;
  end;
```

Figure 5: The *TProcessDatabase* record.

```
function IsDebuggerPresentForWindows: Boolean;
var
  PDB: PProcessDatabase;
  TID: Integer;
  Obsfucator: Integer;
begin
  Result := False;
  Obsfucator := 0;
  TID := GetCurrentThreadID;
  asm
    MOV     EAX, FS:[18h]
    SUB     EAX, 10h
    XOR     EAX, [TID]
    MOV     [Obsfucator], EAX
  end;
  if Obsfucator <> 0 then begin
    PDB := Pointer(GetCurrentProcessID xor Obsfucator);
    Result := (PDB^.Flags and fDebugSingle) <> 0;
  end;
end;
```

Figure 6: *IsDebuggerPresentForWindows* tests for the presence of a debugger.

the purposes of this article, however, only two fields of the PDB are of interest. Therefore, the *TProcessDatabase* (see Figure 5) record used by the code ignores all other fields. For more information about the PDB, I strongly recommend reading Pietrek's book.

The *Flags* field contains a bit mask describing various properties of the process. The only bit that interests us is the *fDebugSingle* flag ($00000001), which is set when a process is being debugged. The reason the *DebugeeCB* field is also defined in the record is that it appears to be a pointer to the debugee's context block when a process is being debugged. This means that it will be non-zero if the process is being debugged, which provides an alternative method of determining this information.

The tricky part is obtaining a pointer to the PDB. The solution is to use the "Obsfucator" value (the term comes from Microsoft binaries, where it was originally misspelled). The value returned from the *GetCurrentProcessId* function is actually a pointer to the PDB, **xor**'ed with the *Obsfucator* value. The same algorithm is implemented in *GetCurrentThreadId*. To complicate things, this value is calculated upon system startup. This means that to get to the PDB, the value of the *Obsfucator* must be obtained. Fortunately, the rest is simple, because simply **xor**'ing the result of *GetCurrentProcessId* with the *Obsfucator* gives the pointer back to the PDB.

To calculate the *Obsfucator* value, the code uses the result of *GetCurrentThreadId*. *GetCurrentThreadId* returns, much like

```
procedure EnumWindowsProc(Window: THandle;
  LParam: Integer); stdcall;
var
  ClassName: string;
begin
  SetLength(ClassName, 255);
  GetClassName(Window, PChar(ClassName), 255);
  SetLength(ClassName, StrLen(PChar(ClassName)));
  if ClassName = 'TAppBuilder' then
    TList(LParam).Add(Pointer(Window));
end;

function RunningUnderDelphi: Boolean;
var
  List: TList;
  i: Integer;
  ID, ParentID: Integer;
begin
  Result := False;
  ParentID := GetParentProcessID;
  if (ParentID <> 0) and (IsDebuggerPresent) then begin
    List := TList.Create;
    EnumWindows(@EnumWindowsProc, Integer(List));
    for i := 0 to List.Count - 1 do begin
      GetWindowThreadProcessID(Integer(List[i]), @ID);
      if ID = ParentID then begin
        Result := True;
        Break;
      end;
    end;
    List.Free;
  end;
end;
```

Figure 7: The *RunningUnderDelphi* procedure and *EnumWindowsProc* function.

GetCurrentProcessId, a pointer to the thread database (an object similar to the process database), **xor**'ed with *Obsfucator*. Therefore, if we have a pointer to the current process thread database, we can **xor** it with the thread ID to get the *Obsfucator*. The thread database starts 16 ($10) bytes before the thread information block (TIB), which is used a lot by Win32. Both Windows NT and Windows 95 dedicate the FS register to pointing at the TIB for the current thread. Based on this information, the calculation of the address of the thread database is simply the linear address of the TIB, minus $10. The linear address of the TIB is stored at offset $18 in the TIB, so the address of the thread database is FS:[$18] - $10.

The *IsDebuggerPresentForWindows* function (see Figure 6) uses the PDB's *Flags* field to test for the presence of a debugger. It uses assembly code to access the FS register and calculate the *Obsfucator*, and uses the result of this calculation to locate the PDB.

Checking for Delphi's Control

Once the code has determined the presence of a debugger, it should compare the parent process ID with Delphi's process ID. However, it's possible for multiple instances of Delphi to be running, so all instances should be checked. The easiest way to find an instance of Delphi is to look for its main window. Delphi's main form's window class is *TAppBuilder* (at least through version 3). The *RunningUnderDelphi* function and its helper routine, *EnumWindowsProc* (see Figure 7), enumerate through all top-level windows by calling *EnumWindows*, check for the *TAppBuilder* class name, and save the handles of matching windows in a *TList* by storing integers instead of pointers. Each window handle in the list is then checked by obtaining the process ID for the window through the *GetWindowThreadProcessID* function and comparing that ID with the parent process ID. If a match is found, the function returns True and the search is aborted.

Conclusion

This article presents a method of determining if code runs under the control of Delphi. As with all code that uses undocumented methods, there's no guarantee it will work on future versions of Windows. On the other hand, the techniques outlined in this article can be used for many purposes. Although the Win32 API can be quite powerful, there are many cases where a program needs to know more than Windows is ready to tell. The use of undocumented functions and data blocks may be the only way to accomplish a necessary task. Δ

The demonstration application referenced in this article is available on the Delphi Informant Works CD located in INFORM\98\NOV\DI9811YA.

Yorai has been programming for over 10 years — as a professional developer for three years. He uses Delphi as his main programming tool, and writes mostly components, visual controls, Win32 API code, and medium- to large-scale client/server database applications. His previous programming experience includes DOS development in Turbo Pascal, C, and assembly, OS/2 1.x development in C, Windows development in C, and Turbo Pascal for Windows. He has published several articles in *Delphi Developer* (Pinnacle Publishing), dealing with implementation of advanced Delphi and Win32 techniques, and is a member of TeamB. Yorai can be reached via e-mail at yaminov@trendline.co.il.

By Warren Rachele

Wanda the Wizard Wizard

A RAD Tool for Creating Wizards

Wizards are a common feature of the modern Windows landscape. They lead the user through a series of linked — sometimes complex — steps, collecting data and allowing decisions to be made in an orderly, controlled fashion. A wide range of tasks lend themselves to a wizard interface: help systems, product registration, configuration, and training tools are just a few of the obvious choices. Yet creating an effective wizard takes an enormous investment of time and energy to develop. Wanda the Wizard Wizard is a software tool that seeks to make the creation of these interface items and their inclusion into your Delphi program efficient and easy.

Wanda the Wizard Wizard is a RAD tool designed solely to create wizards. Currently in version 1.5.2.2, the tool is produced by Ingeneering Inc. of Ann Arbor, MI. The tightly focused environment and intelligence provides an additional benefit for the developer: The process of building the wizards can be off-loaded to staff more attuned to interface issues, such as an interface designer or trainer.

Using the Tools

Upon starting the Wizard Designer, the developer is immersed in a RAD environment that is immediately familiar. A property sheet takes up the left side of the IDE, with a component toolbar stretching across the top. The rest of the IDE window is devoted to the wizard page (form) being developed. The process of composing the wizard pages follows proven methodologies. A component is selected from the toolbar and dragged onto the page. Once positioned, the developer sets the properties for each component to achieve the desired effect.

Saving the wizard project results in a proprietary .WZX file. Executing the wizard through the IDE is a matter of clicking the Run button. This process engages the runtime engine, WANDA.DLL. The VCL component used to include Wanda technology in a Delphi program is an interface to this DLL, which becomes a part of the distribution file set for the program.

The installation of Wanda the Wizard Wizard runs automatically, providing the programmer with the option of where to install the product and the menu group in which it appears. A complete installation, including the interface objects for all supported products and the example files, consumes just under 4MB of disk space. Installation is a manual process, which adds a new tab to the VCL toolbar and a single component.

So, why bother with a third-party tool to build an object that could be created within the Delphi environment? The answer is: rapid, focused development. Wanda makes it easy to transfer the flow of ideas into action by incorporating wizard-specific properties and methods into its components. Intelligent ideas are everywhere in this tool, starting with the base window, referred to in wizard nomenclature as a "page."

Wanda has borrowed an idea from the page layout profession, allowing the designer to establish a master page. This is a non-visible page that can become the basis for all remaining pages created in the wizard. The designer can establish the size, color, logo, button arrangement, and any other common attribute one time on the master page. Each new page that is created, identifying this object in the *MasterPage* property, inherits all the properties and attributes of the original design. Delphi programmers will immediately see the benefits of this object-oriented design. Any changes that are required of the fixed components or attributes need only be made once. The changes made to the master page immediately cascade through the child pages. Not only does this reduce the possibility of error, it saves the programmer considerable time and effort, especially when dealing with active items, such as buttons and their associated linkages.

Pages other than the master page are also assigned a page type. The choice made (Normal, Start, End) determines the logical flow through the dialog boxes and whether certain button types are enabled. For example, a page type of End requires that the users of the wizard have worked their way to the last page of dialog before a **Done** button will become enabled. Consider the amount of work required in a general-purpose development tool to build this kind of functionality into a series of forms.

The button components and their associated action codes offer a limited, but focused, selection of methods. All button actions, with the exception of **Done** and **Cancel**, are oriented toward the user's navigation of the wizard pages. Buttons can be provided to go to the first, last, previous, or next pages, as well as specific pages within the stack. Initially, the programmer might feel limited by the restrictions imposed by the selection of components, or the fact that no coding is allowed within the wizard. Once the mind-set of the wizard system is acknowledged, however, the programmer realizes that these are the only items and actions needed to implement the ideal wizard, and can focus on the application.

An example of a simple wizard in development is shown in Figure 1. The master page for this wizard shows the common components that will appear on every child page. The button actions have been set as methods of this page and do not need to be modified on the pages derived from it. The designer is now free to concentrate on content and logical flow, rather than the mechanics of getting from page to page and handling user actions that don't go according to plan. Figures 2 and 3 show the two visible pages that make up the registration wizard for Surfside Digital Design. Each of the data elements captured on either page is available for query when the wizard has completed. Figure 3 shows the last page of the wizard containing a list box from which the user will identify the source of the program. This component, as well as the bitmap in the logo component, demonstrates an idiosyncrasy of the Wanda environment.

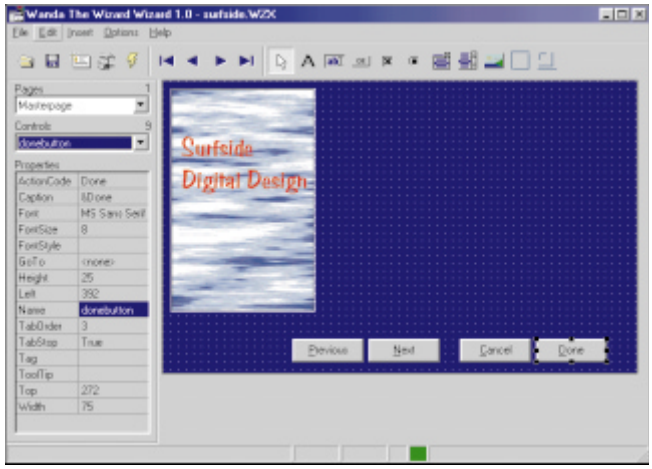


Figure 1: A wizard in development.

The data contained in the list box must be provided as an external text source. In this case, the list was composed in a text editor and saved as a .TXT file. To bring text and bitmap resources into a Wanda wizard, they must be registered with the Resource Manager in the IDE. Once the items have been registered, they become available to the workspace. In the case of the list box, the *Resource* property has been set to a test list registered as *buylist*. Though the data or bitmap will appear right away in the wizard, modification of the underlying data is not automatically reflected within the wizard. The object must be removed from the resource manager and the modified version added back into the resource list. To complete the update, components calling the resource need their properties reset.

Wanda provides a run-time emulator integrated into the IDE that allows the designer/builder to test and debug the wizard. Figure 4 shows the Surfside wizard in "execution mode." The IDE minimizes and is replaced on screen with the wizard form and the Run-Time Emulator (RTEM). The RTEM window is segmented into three boxes: **Page Stack**, **Results**, and **Query**. The **Page Stack** dynamically demonstrates the logical flow from page to page, moving a pointer that follows the user's movement. The **Query** edit box allows the designer to determine the contents or state of any component on the wizard. Each component is assigned a numeric identifier that must be prefaced with the character "c" to be used in the query. When queried, the contents of the selected component are shown in the **Results** list box.

Integration with Delphi

Using the Wanda-created wizard in a Delphi program is as simple as dropping the Wanda component on a form and setting the appropriate properties. When the user needs to run the wizard, the *Active* property is set to True, and the Wanda run-time engine activates and runs the wizard. The Surfside demonstration program shown in Figure 5 is designed to run the registration wizard and query the results for use in the Delphi program. The **Register** button's *Click* method is used to activate the wizard.

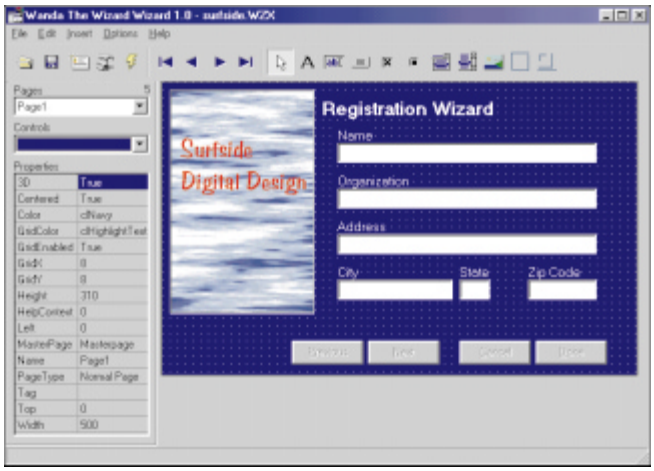


Figure 2: One of two visible pages that make up the registration wizard for Surfside Digital Design.

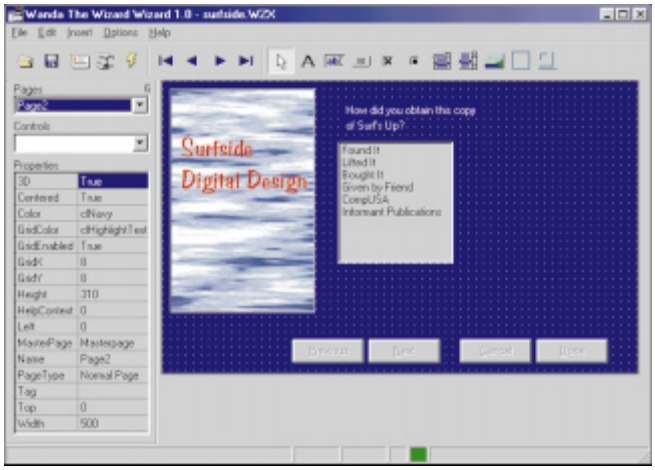


Figure 3: The other visible page for Surfside's registration wizard.

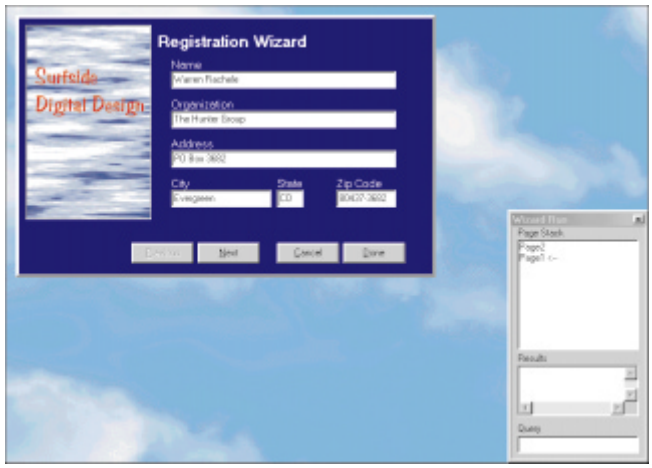


Figure 4: The Surfside wizard in execution mode.

Figure 5: The Surfside demonstration program in the Delphi IDE.

Listing Five (on page 44) demonstrates the behind-the-scenes work necessary to fully exploit the power of the Wanda wizard. Although the component offers the ability to run the wizard through property settings, any truly useful work is going to be accomplished through external function calls to the run-time engine, WANDA.DLL.

The first point to examine is the method called to run the wizard. Setting the *Active* property to True will activate the wizard, but the calling program won't be able to query the wizard's exit status. By calling the *RunWizard* method, the program is able to determine whether the user successfully completed the wizard or canceled out of it. The *RunWizard* method also handles memory management in the event that a *FreeWizard* call was not successful in removing the memory captured by the DLL, or not used at all.

The remaining code is devoted to querying the components on the wizard form to obtain their contents. Contrary to the documentation supplied with the product, it appears that the programmer has a choice of the methods used to set up the query of a component. The statements following the *{ Query Name. }* comment show a call to *SetComponent* before the *QueryString* method. This directs the query method to the particular component by passing the integer value of the component ID to the function, setting up an internal pointer. The *QueryString* method is called with a PChar as the parameter, giving the function a place to put the null-terminated string it will return from the wizard.

Immediately below this set of instructions is a second query to the **Organization** field of the wizard. In this case, the component ID is set through the use of the *Wanda* component's *Query* function, something the documentation appears to warn against. Both approaches worked consistently. Figure 6 shows a view of the executing Surfside demonstration after a return from the wizard. All the fields from the wizard form have been queried, and the data retrieved into the Delphi program.

A disappointment was the handling of the list box query. The query returns a numeric value representative of the list position of the item clicked by the user. Unable to query the contents of the resource directly to capture the text of the item, the programmer would be required to include the text resource in the Delphi program to accomplish this feat, and having the text list in two work areas leads to the possibility of update anomalies.

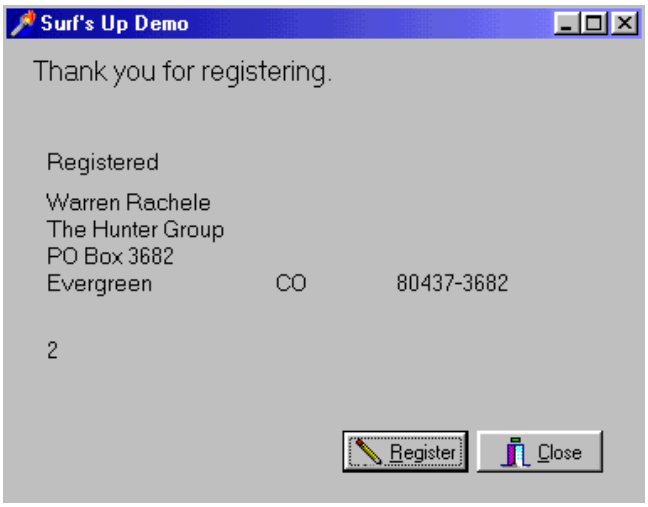


Figure 6: The Surfside demonstration at run time after returning from the Surfside wizard.

The Product

Wanda the Wizard Wizard ships on a single diskette, or it can be downloaded in demonstration form from Ingeneering Inc.'s Web site at http://www.ingeninc.com. It ships with the documentation in an Adobe .PDF (Portable Document Format) file, with a printed version of the manual available from the company at an additional cost.

The documentation is a low point of the product. The organization of the document leads the user to a lot of dead ends without providing answers. An example is the presentation of the tutorials early in the manual. Following the description step by step, wizards are created that deliver many unexpected results, such as buttons never being activated. No explanation is provided for situations such as this, leading to a good deal of frustration for first-time users. The answer to this particular problem is buried as a casual reference to page types much further on in the documentation. The programmer would do well to read the documentation thoroughly before attempting to use the product. A secondary issue with the documentation is that it is shipped incomplete, with numerous references for specific product support labeled as "coming."

The IDE shows a few rough edges as well. For example, components such as buttons and edit fields were found to have mysteriously moved from their original position. Thinking this was a case of a lazy mouse hand, the components were moved back into position where they remained. Then, while reviewing the properties for a component, a button was seen to move as the list of properties was clicked from field to field. Moving up the list in ascending order, the button moved up a notch as each property was clicked.

Also frustrating was the default activity of the Open dialog box. There's no way to set the default directory to the project directory, forcing the user to start in the My Documents folder. Finally, there is no online Help file; the **Help | Topics** menu option directs the programmer back to the .PDF or .PRN file.

INFORMANT FACT FILE

Wanda the Wizard Wizard has some rough edges, particularly when it comes to its documentation. However, intelligent ideas are abundant and it provides rapid, focused development of objects that could be created within Delphi. Given the prevalence of wizards in Windows applications today, Wanda proves to be a promising and practical addition to a Delphi developer's toolbox.

Ingeneering Inc.
1645 Morehead Drive
Ann Arbor, MI 48103

Phone: (734) 662-4646
Fax: (734) 662-4682
E-Mail: info@ingeninc.com
Web Site: http://www.ingeninc.com
Price: US$199.95 (direct from Ingeneering Inc.)

Conclusion

Wanda the Wizard Wizard is a worthwhile tool for the programmer looking to add wizard functionality to a program. As long as its limits are understood and programmer expectations appropriately tempered, the development of wizards to be included in a Delphi program will not be a disappointing experience. In fact, working with the tool leads to new ideas for additional functionality and usefulness. The roughness of the product and the documentation should not dissuade developers and interface designers from exploring the possibility of adding this software to their toolbox. Δ

Warren Rachele is Chief Architect of The Hunter Group, an Evergreen, CO software development company specializing in database-management software. The company has served its customers since 1987. Warren also teaches programming, hardware architecture, and database management at the college level. He can be reached by e-mail at wrachele@earthlink.net.

Begin Listing Five — Surfside Demo

```
unit surfdemo;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, Buttons, WandaRun;

type
  TForm1 = class(TForm)
    WandaRun1: TWandaRun;
    BitBtn1: TBitBtn;
    BitBtn2: TBitBtn;
    lbMessage1: TLabel;
    lbRegistered: TLabel;
    lbName: TLabel;
    lbOrg: TLabel;
    lbAddress: TLabel;
    lbCity: TLabel;
    lbState: TLabel;
    lbZip: TLabel;
    lbPurchased: TLabel;
    procedure BitBtn2Click(Sender: TObject);
  private
    { Private declarations. }
  public
    { Public declarations. }
end;

var
  Form1: TForm1;

implementation

{$R *.DFM}
{ -- Wanda run-time DLL imports -- }
procedure QueryString(s: PChar);
  stdcall; external 'Wanda.DLL';
function SetComponent(id: Integer): LongBool;
  stdcall; external 'Wanda.DLL';
function RunWizard(fn: PChar): LongBool;
  stdcall; external 'Wanda.DLL';
function FreeWizard: LongBool;
  stdcall; external 'Wanda.DLL';
function QueryLong(id: Integer): Integer;
  stdcall; external 'Wanda.DLL';

procedure TForm1.BitBtn2Click(Sender: TObject);
var
  s : array[0..75] of Char;
  lastrun : Boolean;
begin
  lbMessage1.Caption :=
    'Starting the Registration Wizard...';

  { Activate the Wanda run time. }
  lastrun := RunWizard(PChar(WandaRun1.WizardName));
  { Query the exit return value. }
  if not lastrun then begin
    { User clicked on Cancel or pressed Escape. }
    lbRegistered.Caption := 'User Canceled: UNREGIS-
TERED!';
    lbRegistered.Visible := True;
  end
  else begin
    { User completed the wizard and clicked Done. }
    lbMessage1.Caption := 'Thank you for registering.';
    lbRegistered.Caption := 'Registered';
    lbRegistered.Visible := True;

    { Query Name. }
    if not SetComponent(19) then
      raise Exception.Create('Query Error')
    else begin
      QueryString(s);
      lbName.Caption := StrPas(s);
      lbName.Visible := True;
    end;

    { Query Organization. }
    WandaRun1.Query := 'c20';
    QueryString(s);
    lbOrg.Caption := StrPas(s);
    lbOrg.Visible := True;
    { Query Address. }
    WandaRun1.Query := 'c21';
    QueryString(s);
    lbAddress.Caption := StrPas(s);
    lbAddress.Visible := True;
    { Query City. }
    WandaRun1.Query := 'c22';
    QueryString(s);
    lbCity.Caption := StrPas(s);
    lbCity.Visible := True;
    { Query State. }
    WandaRun1.Query := 'c23';
    QueryString(s);
    lbState.Caption := StrPas(s);
    lbState.Visible := True;
    { Query Zip. }
    WandaRun1.Query := 'c24';
    QuerySTring(s);
    lbZip.Caption := StrPas(s);
    lbZip.Visible := True;
    { Query the Listbox. }
    WandaRun1.Query := 'c28';
    lbPurchased.Caption := FloatToStr(QueryLong(28));
    lbPurchased.Visible := True;

    FreeWizard;
  end;
end;

end.
```

End Listing Five

The Joy of Demos

Live technology previews — or "demos" as they're commonly known — are typically nothing more than a *façade*, cobbled together to chug along on the inside, while giving the appearance of grandeur on the outside. Demos are loathed by developers because of the extra work required to simply meet an "artificial" milestone. To make matters worse, a demo is never accounted for in the schedule, forcing the developer to make up the time somehow.

Of course, demos have their merits. Management can get a product in front of a user and start to solicit feedback early in the development process. This lowers development costs and keeps the product's features closer to user expectations. It can also provide positive reinforcement that a course is worth pursuing.

Which takes us to the recent Borland/Inprise conference, held the third week of August, in Denver Colorado. In front of nearly 3,000 people, Chuck Jazdzewski, Delphi's principal architect, demonstrated a pre-release Delphi compiler that produced Java byte code. The result was a thundering round of well-deserved applause.

The demonstration created a *TDatabase* component, and attached it to a JDBC datasource through a new *TDatabase* property. After flushing the sample out to loop through and display records, a replacement command-line compiler, dcc32j, was invoked to produce Java byte code. The intended use of this technology is to allow non-visual applications created in Delphi to run on any platform that has a Java Virtual Machine (JVM). It's an interesting idea: Write your application server for a multi-tier solution in Delphi, and you can run the application server on another platform, e.g. a Sun box.

Although the demonstration was relatively simple, it illustrated Inprise's commitment to create tools for developers that must provide solutions for multiple platforms. Granted, this feature won't be on your desktop tomorrow; there's plenty of work left, but the demonstration was impressive. All the more so considering the venue. How many times have you demonstrated a pre-release compiler to 3,000 people, including the international media? It must have been a serious temptation for Dilbert's Dark Angel of Demos.

The demonstration drove home some additional points as well:

- Inprise is at the forefront of interoperability, and is second-to-none in technical ability.
- You can leverage your Delphi skills and experience to throw your pail and shovel into the Java sandbox.
- The Delphi team has shown that a significant shift in the marketplace doesn't mean the end of Delphi. In fact, as Delphi embraces these technologies and innovations, it gives developers a way to play in that arena quickly and easily. I'll go out on a limb and state that Java won't be the last innovation in our industry. It's nice to know that Inprise has the ability to react when the industry changes.

Like every seasoned developer, your initial reaction is: "Fine. How does this help me today?" With Inprise giving you a sneak peek at tomorrow's vision, you have a unique opportunity to prepare for that future. You'll reap the rewards as you find each piece of software you write becoming available to a larger segment of the computer population.

Who knows where the Delphi compiler will go next? More and more people are asking for things like Delphi/Linux and Delphi/Windows CE. While this technology preview is a far cry from implementing these products, it shows that Inprise could provide these tools if the market demanded.

Advances like MIDAS client for Java further strengthen Inprise's commitment to get "Any Data. Any Time. Anywhere." It looks like it's not just a slogan at Inprise — it's a way of life. Δ

— Dan Miser

Dan Miser is a Design Architect for Stratagem, a consulting company in Milwaukee. He has been a Borland Certified Client/Server Developer since 1996, and is a frequent contributor to Delphi Informant. *You can contact him at http://www.execpc.com/~dmiser.*

The Conference: Borland/Inprise 1998

As usual, this year's Borland/Inprise Conference was outstanding. I had the opportunity to meet and exchange ideas with many readers, get the latest news on Delphi add-on tools, get a preview of Delphi's future, and attend some great sessions.

With this company's new name and its goal to find a niche with leading corporations, an important question remains: "What implications does Inprise's new vision hold for the average developer?" At the same time, with Inprise's proclamations on the importance of its third-party tool makers, these developers are wondering if they can expect an even closer relationship in the future. I'll explore these and other issues, but first I'll share some of my personal highlights from the conference.

Personal highlights. At the center of each conference are the excellent tutorials, presentations, showcases, and birds-of-a-feather sessions. This year, I attended — for the first time — one of the pre-conference tutorials, Cary Jensen's wonderful overview of JBuilder, where he gave a tour of its IDE and explained some of the important differences between JBuilder and Delphi. I also attended Bob Swart's perennial discussion of optimization in which he presented some nice profiling tips and techniques, along with new aspects of Delphi 4.

Mark Miller's session on Class Design was spectacular — insightful, fast paced, humorous, and impressive. Mark is one of the few people with the audacity to perform coding improvisations in front of a large audience.

I attended Ray Konopka's impressive showcase on Raize Software's new debugging tool, CodeSite. The dozen or so developers who showed up at 7 A.M. (yes, A.M.) for the Project Jedi Birds-of-a-Feather session, hosted by Robert Love, proved that this project is alive and well.

I met one of my heroes, Tom Swan, and other writers, including John Ayers. As usual, I spent a lot of time in the exhibition area, learning the latest developments from TurboPower, Eagle Software, Raize Software, Nevrona Designs, Skyline Tools, and some new tool makers I am sure we'll be hearing from in the future.

From Borland to Inprise. Looking ahead, what can we expect from Inprise and Delphi? While I can't promise a definitive answer, I can share some of my perceptions. These are based in part on Inprise's skillful presentation of its vision, juxtaposed with concerns of developers.

Many months before Borland decided to change its name to Inprise, its new CEO, Del Yocum, charted a very new direction. The goal was twofold: find a new, unique niche in the industry, and develop new relationships with leading corporations. The acquisition of Visigenic and the name change were just the latest steps in the mission outlined by Chairman Yocum at the 1997 Borland Conference in Nashville. Furthermore, Inprise's interest in developing and exploiting connections between its tools on the one hand, and its active courting of industry clients, is also not new. What seems new in 1998 is a cohesive game plan to actually make it so.

The vision was presented in an awesome theatrical keynote — à la Camelot — where the wise King Yocum, concerned for the welfare of the subjects of the Land of Bor, leads his brave knights on a quest for the Orb of Knowledge. The Orb's discovery leads to the establishment of a new kingdom: Inprise! Other sessions filled in the details.

As impressive as all of this was, a nagging concern continues to lurk in the background. Most of us would agree that Borland has produced the best Windows development tools on this planet. But, as it grows, will this new incarnation of Borland called Inprise remember the importance of the average developers, the soldiers in the trenches? Without them, the battle will surely be lost. Inprise must continue its commitment to make available entry-level versions of these tools so the ranks of this army will continue to grow.

At the same time, third-party tool makers are an independent militia, fighting under the same banner. While few at this conference would be likely to praise Microsoft as a tool maker, a number of developers and writers made this observation: "Microsoft certainly knows how to keep its third-party tool makers in the loop." Does Inprise do enough of this?

Borland/Inprise: You have produced the fastest and most efficient compilers. You have developed and made available an open tools environment that allows for infinite expansion. You have presented a workable plan to integrate all these tools. But do you realize you don't have to produce every conceivable extension, be it a component or an IDE enhancement?

Consider conducting frank and detailed discussions with your partners — your *real* partners, the third-party tool makers. Let them know what you plan to do and find out where they can make a contribution. And never forget the entry-level developer, without whom Delphi would wither on the vine. King Yocum, with the hope that you will see some wisdom in your humble scribe's suggestions, and that your benevolence will continue to shine on all of your subjects, I leave you with this: "Ask not what your developers can do for you; ask what you can do for your developers." Δ

— Alan C. Moore, Ph.D.

Alan Moore is a Professor of Music at Kentucky State University, specializing in music composition and music theory. He has been developing education-related applications with the Borland languages for more than 10 years. He has published a number of articles in various technical journals. Using Delphi, he specializes in writing custom components and implementing multimedia capabilities in applications, particularly sound and music. You can reach Alan on the Internet at acmdoc@aol.com.