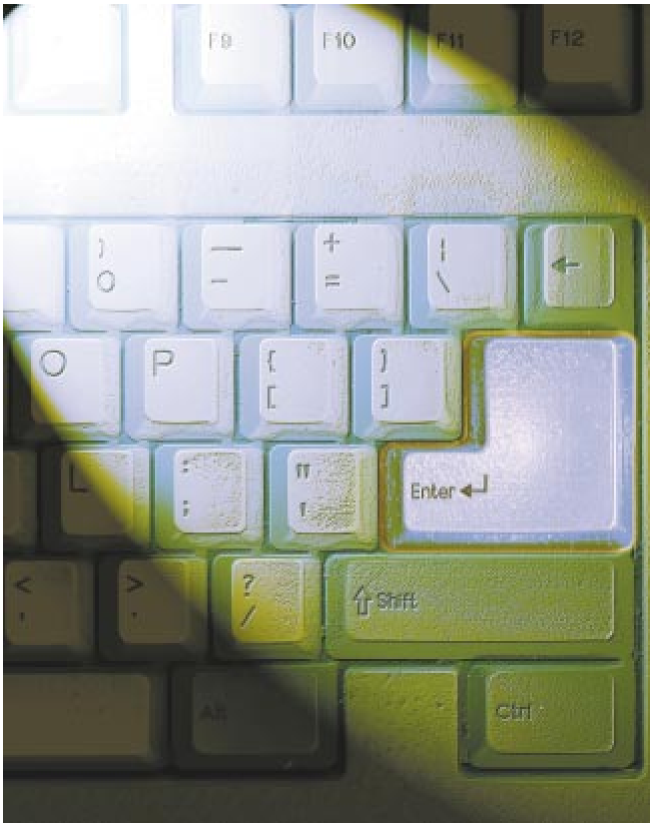
March 1997, Volume 3, Number 3



Cover Art By: *Tom McKeith*

Tweaking the Interface

User-Pleasing Customization Techniques

ON THE COVER

FEATURES

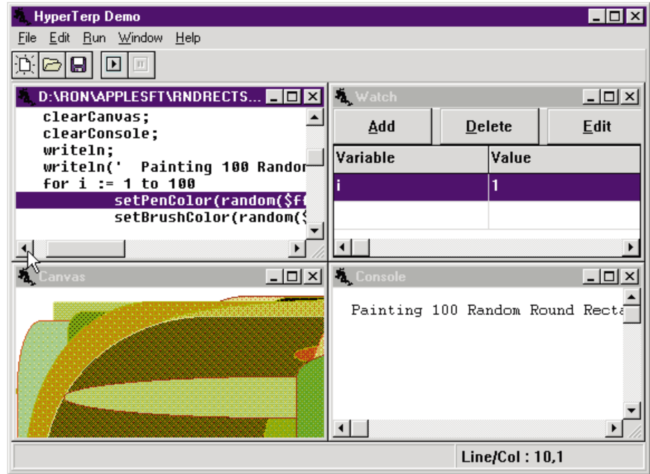
REVIEWS

DEPARTMENTS

Delphi
TOOLS

New Products
and Solutions

HyperTerp 4.0 for Delphi



HyperAct, Inc. released version 4.0 of *HyperTerp*, a drag-and-drop script language VCL component for Delphi.

The features of HyperTerp 4.0 include a plug-in architecture that simplifies the creation of language extensions that work in multiple projects; object-based syntax support; new constructor and destructor definitions for object types; and improved speed.

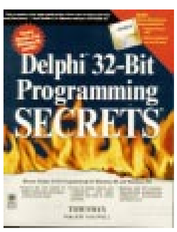
HyperTerp 4.0 supports user forms, procedures and functions, debugger interfaces, arrays, tables, and objects. Additional features include a plug-in architecture allowing users and third-party libraries to create non-application specific custom extensions and object-based syntax. HyperAct also released HyperTerp/PRO, a professional interpreter engine with documented source code, in 16- and 32-bit versions.

Price: HyperTerp 4.0, Standard Version, US$249; upgrade for earlier Standard Version users, US$50; HyperTerp/PRO and source code, US$395; upgrade for PRO users is free.

Contact: HyperAct, Inc., 3437 335th St., West Des Moines, IA 50266
Phone: (515) 987-2910
Fax: (515) 987-2909
E-Mail: rhalevi@hyperact.com
Web Site: http://www.hyperact.com

Delphi Book

Delphi 32-Bit Programming Secrets
Tom Swan with Jeff Cogswell
IDG Books

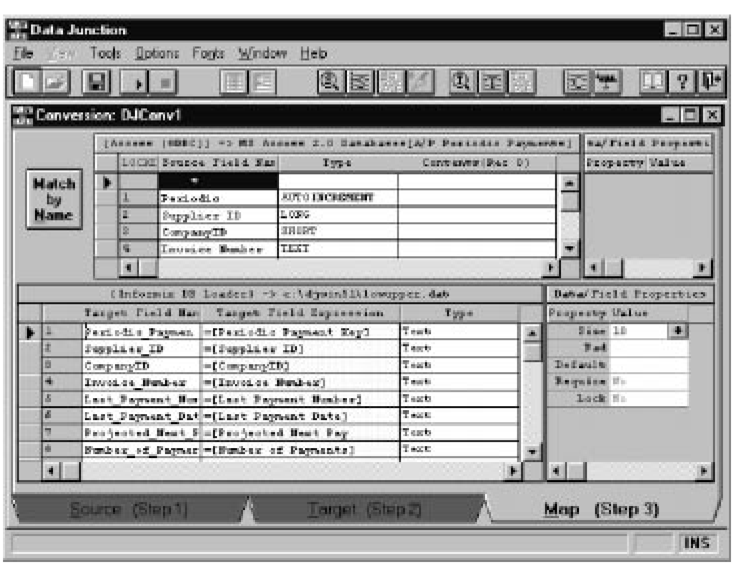


ISBN: 1-56884-690-8
Price: US$44.99
(738 pages, Disk)
Phone: (800) 762-2974

Data Junction Version 5.11 Now Available for Windows

Data Junction Corp., formerly Tools & Techniques, Inc., of Austin, TX, has released *Data Junction Version 5.11* for Windows, a data conversion tool that features additional formats and improved SQL support.

Data Junction provides complete data extraction, filtering, manipulation, and conversion, including legacy mainframe, UNIX, and desktop data. It acts as a universal and neutral junction in the middle of all structured data formats. Additionally, it displays source and target data structures on a single screen, and provides advanced drag-and-drop features to visually map the source data to the target structure. Because it enables users to filter and edit data during the conversion, the output format can be customized.



Users can convert most structured, field, and record-oriented file formats. Visual data browsers and parsers have been designed into the product to assist in defining difficult file types.

Data Junction 5.11 supports Sybase bcp, Informix DB Load, HTML (export), Oracle (native), Informix/SE, Oracle SQL Loader, and IDAPI.

Data Junction 5.11 features improved SQL support via ODBC, reworking of data-type handling for reconciliation of source-to-target variations, ODBC cursor support, and the capability to save partial conversions and point to separate sets of conversions. It also has a richer set of command line parameters, including automated execution and default expressions. Additionally, locked fields are now allowed in target structures.

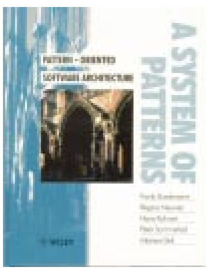
Price: US$299 for a single-user license; US$499 for a two-user network license. Network licenses can be extended for US$100 per user (includes a 30-day money-back guarantee).

Contact: Data Junction Corp., 2201 Northland Dr., Austin, TX 78756
Phone: (800) 580-4411 or (512) 459-1308
Fax: (512) 459-1309
E-Mail: djinfo@datajunction.com
Web Site: http://www.datajunction.com

Delphi TOOLS

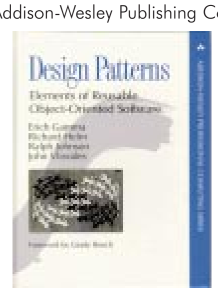
New Products and Solutions

Pattern-Oriented Software Architecture: A System of Patterns
Frank Buschmann, et al.
John Wiley & Sons, Inc.



ISBN: 0-471-95869-7
Price: US$39.95 (457 pages)
Phone: (212) 850-6000

Design Patterns: Elements of Reusable Object-Oriented Software
Erich Gamma, et al.
Addison-Wesley Publishing Co.



ISBN: 0-201-63361-2
Price: US$45.25 (395 pages)
Phone: (617) 944-3700

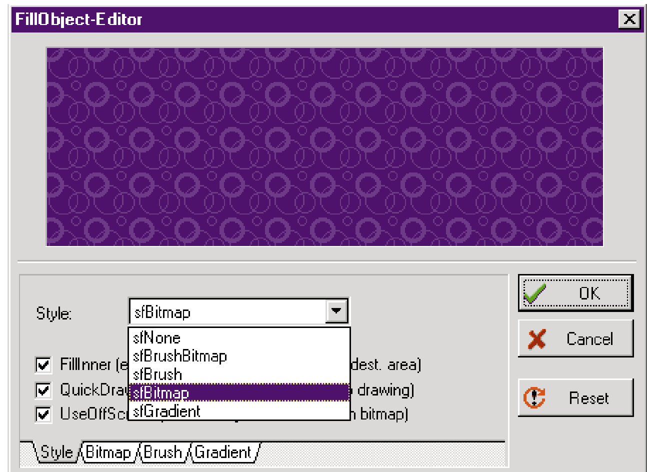
LMD Innovative Releases LMD-Tools Version 2.0

LMD Innovative of Siegen, Germany has released ***LMD-Tools Version 2.0***, a set of native Delphi VCL components and routines for various programming tasks.

The controls cover system, multimedia, visual, and data-sensitive and dialog components. Component editors, which allow visual setting of properties and functions, enable faster access and more efficient work. Most of these editors are available through context menus by right-clicking on a component.

LMD-Tools Version 2.0 is offered in standard and advanced editions. The Standard Edition contains about 30 controls; the Advanced Edition contains about 60 controls (including those in the Standard Edition). Both packages include full source and online Help. The advanced version has more than 30 demonstration projects. Trial versions and demonstrations are available from LMD Innovative's Web site.

Price: Standard Edition, US$69; Advanced Edition, US$135.
Contact: LMD Innovative, Vor der Hohler 17a, 57080 Siegen, Germany
Phone: 49-271-355489
Fax: 49-271-356952
E-Mail: sales@lmd.de
Web Site: http://www.lmd.de



Nevrona Designs Ships AdHocery and Propel for Delphi

Nevrona Designs of Mesa, AZ has released ***AdHocery Version 1.0***, an ad hoc SQL query interface building system, and ***Propel Version 1.0***, a development reuse system.

AdHocery allows the creation of custom user interfaces, and hides tables, fields, and master-detail links from end-users at run time.

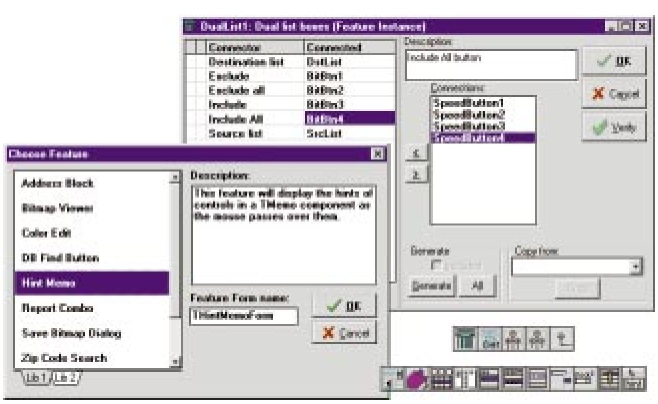
Programmers can use any data-aware components with AdHocery to create QBE-style forms. It also includes: *TAdHocGrid*, which allows queries to be created within a grid style layout with row and cell connectives; *TAdHocOutline* and *TAdHocTreeView*, which display the logic of the query; and *TAdHocLookupGrid*, which allows the selection of multiple items from a list to be entered into a query. Queries can be saved to disk, and query forms created with AdHocery usually require no source code.

AdHocery will work with any other tool that is compatible with *TQuery* components, and can be used as a front-end for reporting systems or on-screen queries.

Propel combines a form's code and components into reusable units, called Features, that can be saved to a local library, shared with other Propel users, or distributed royalty-free as feature components to non-Propel users.

During the creation of Features, the programmer has access to Delphi's IDE, including the form designer, Object Inspector, and debugger. Multiple Features can be used on a single form, and can share components between them. When using a Feature, the programmer can change individual component properties (such as position or size), and can define additional code for each component, or replace components with other types.

Trial versions and product tours are available on Nevrona's Web site.

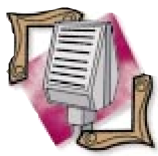


Price: AdHocery Version 1.0, US$149 (US$99 for ReportPrinter Pro users), 16- and 32-bit versions include complete source, printed documentation, and full technical support; Propel Version 1.0, US$199, 16- and 32-bit versions include printed documentation and full technical support.
Contact: Nevrona Designs, 1930 S. Alma School Rd., Suite #B-214, Mesa, AZ 85210-3041
Phone: (888) 776-4765
Fax: (602) 530-4823
E-Mail: info@nevrona.com
Web Site: http://www.nevrona.-com/designs

table of CONTENTS

News
L I N E

March 1997



Borland Adds Informix and DB2 Support to Delphi Client/Server Suite

Scotts Valley, CA — Borland has released Informix and DB2 database drivers for Delphi Client/-Server Suite. The new 32-bit drivers are part of the Borland Database Engine (BDE) with SQL Links 3.5, and include updated support for Oracle, Sybase, Microsoft SQL Server, and Borland InterBase database servers.

Delphi Client/Server maintenance contract customers and Connections members will receive free the new BDE with SQL Links 3.5; other registered Delphi Client/Server Suite owners must call (800) 932-9994. These offers apply only in the United States and Canada. International customers should contact their local Borland office.

For a complete listing of the features in the BDE with SQL Links 3.5, visit Borland Online at http://www.-borland.com/bde/.

Delphi, Delphi Desktop, and Delphi Developer owners interested in the BDE without the SQL Links can download the software from http://www.-borland.com/techsupport/-dbe/utilities.html.

Borland Completes Acquisition of Open Environment Corp.

Scotts Valley, CA — Borland completed its acquisition of Open Environment Corp. in late 1996, and Open Environment is now a subsidiary of Borland. The combined operations of the two companies will be conducted under the Borland name and will be headquartered in Scotts Valley, CA.

Borland's common stock will continue to trade on the Nasdaq National Market System under the symbol BORL. According to the agreement between Borland and Open Environment, each outstanding share of Open Environment stock has been converted into .66 shares of Borland common stock.

A Sneak Peak at Borland's Upcoming Delphi 3

Scotts Valley, CA — Set for release in the second quarter of 1997, Delphi programmers will see many new features and enhancements in Delphi 3, including: a visual multi-tier architecture; an enterprise component foundry; enterprise and Internet client/server solutions; the ability to distribute database information over the Web; and 32-bit optimized native drivers for InterBase, Oracle, Informix, Sybase, SQL Server, IBM DB2, Access, and FoxPro.

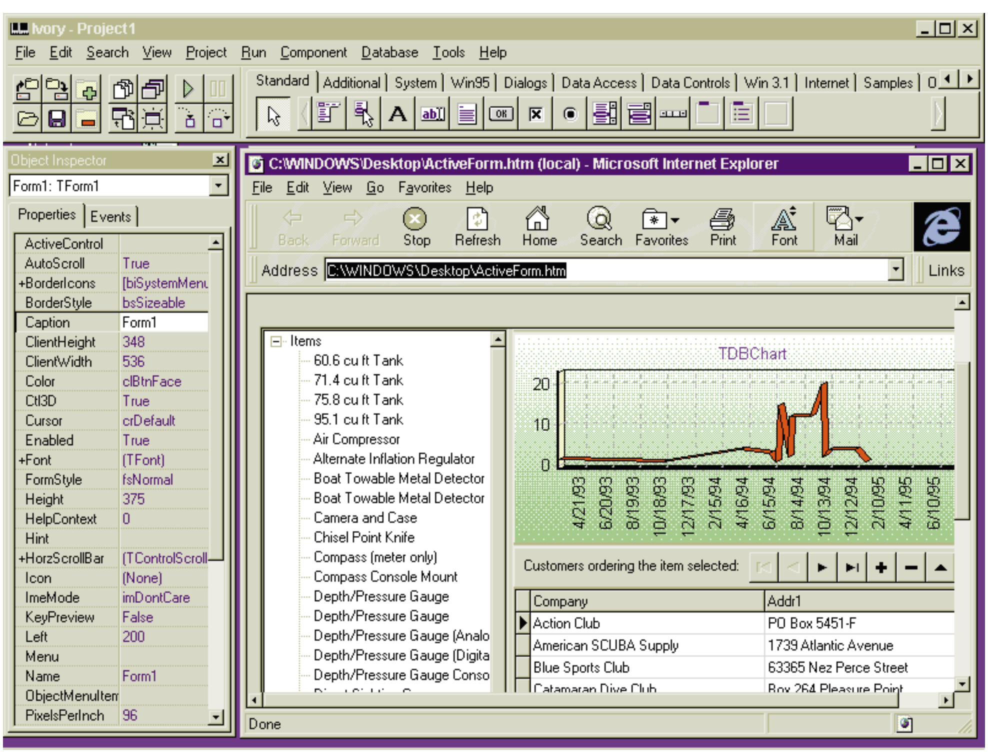
Delphi's new technologies, such as Remote Data Broker, Packages, OLEnterprise, and Internet enablement, allow thin-client applications as small as 15KB to be distributed, configured, and maintained.

Borland has added support for Microsoft standards, including COM and ActiveX.

Delphi 3's ActiveX controls are native machine code compiled, and don't require a distributable run-time environment.

In addition, Delphi 3 uses the ActiveX architecture to deploy thin-client applications over the Web using standard application delivery mechanisms (File, INF, or CAB File delivery).

Delphi 3 is interoperable through COM interfaces with



"*Delphi 3 Sneak Peak*" continued on page 5

News

L I N E

March 1997

ObjectSHOW Launches Online Trade Show

Flower Mound, Texas — ObjectSHOW Inc. has released the Developers' Premier On-Line Trade Show at http://www.-ObjectSHOW.com.

The company offers Delphi products, as well as tools for Visual Basic, Clipper, Java, C/C++, and FoxPro developers. This online trade show enables viewers to see product demonstrations and place orders.

Vendors interested in advertising their products can reserve booth space for a trial three-month period at no cost.

The Solution Works Awarded PSYBT Young Business of the Year

Glasgow, Scotland — The Solution Works, a Greenock information technology company, has been named The Prince's Scottish Youth Business Trust's (PSYBT) Young Business of the Year. Formed in 1994, The Solution Works specializes in Delphi development.

Founders Andrew McKelvie (26), Russell Sneddon (25), and John Shiveral (25) began this venture with a loan from PSYBT. Today the company's clientele includes the Overseas Development Administration, The Environment Agency, Scottish Widows, and Scottish Milk.

For more information call 44+ (0) 4175-743030 or e-mail 100544.3545@compuserve.com.

The Solution Works principals, left to right: John Shiveral, Andrew McKelvie, and Russell Sneddon.

McGraw-Hill Books Online

McGraw-Hill has announced the launch of Beta Books, an online program that makes its computer titles available to the public three months prior to publication. Two new computing books will be posted on the Web each month, in their entirety. The books are trade titles that cover Internet-related topics. Beta Books can be found at McGraw-Hill's Web site at http://www.computing.mcgraw-hill.com.

Yocam Joins Borland as Chairman and CEO

Scotts Valley, CA — Borland has appointed Delbert W. Yocam, 52, a senior executive from Apple Computer, Inc. and Tektronix, Inc., as chairman of the Board of Directors and chief executive officer.

Yocam served as executive vice president and chief operating officer for two years at Apple, and president of Apple Pacific for a year. After Apple, Yocam served as president and chief operating officer of Tektronix, a hardware technology company.

Delphi 3 Sneak Peak (cont.)

C++, Java, Visual Basic, PowerBuilder, JavaScript, and other languages.

In version 3, Delphi forms can become ActiveForms. These ActiveForms are ActiveX controls that use the Delphi form as a container for other Delphi components.

ActiveForms publish ActiveX property pages and type libraries for adding functionality to other development environments, such as Internet Explorer, Visual Basic, Optima, or PowerBuilder. These forms can also be used to deliver applications over the Internet.

To keep abreast of Delphi 3's progress, visit http://www.borland.com.

Delphi 1 / Delphi 2

By Brad Olson

How Much Longer?

Building a Status Gauge Dialog Box Component

In the July 1996 issue of *Delphi Informant*, James Callan explored user interface strategies for maintaining users' patience when your program runs long processes. Callan showed that by using progress indicators and different cursors, you can keep users informed during processing delays.

When showing a user progress status, it is important to remain consistent. Many users want to know only a few things about process status: they want to see text describing *what* is happening, and they want to know *how much* of the process has completed. And finally, they want some way to cancel the process.

One sure way to remain consistent throughout your applications is to use VCL components in your development.

This article presents a way to implement a progress dialog control. It expands upon the concepts espoused in Callan's article. The result is a reusable component for communicating progress status to the user.

Component Requirements

To provide satisfying status information to the user, the component should:

- present a modal form to the user when a process is active,
- allow the program to update the status display, and
- provide the user with a way to cancel the process.

Additionally, the component should be a reusable dialog component similar to those found on the Dialogs page of the Component palette, with an *Execute* method that returns *True* if the process completes or *False* if the user cancels the process.

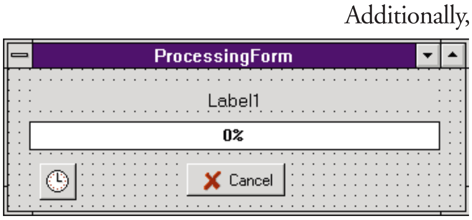


Figure 1: ProcessingForm at design time.

If you've never created a dialog box component, you may want to refer to chapter 13 of Delphi's *Component Writer's Guide*. It's dedicated to creating this kind of component, and provides an easy-to-follow example. We'll use similar steps to create the *TProcessingDialog* component.

Create the Form and Add Components

The first step is to start a new project and add the necessary components to the default form. Then set the form's properties so it will behave like a modal dialog box at run time. To do this, first set the *BorderStyle* property to *bsDialog* and *Position* to *poScreenCenter*. Then set the *Name* property to `ProcessingForm`. Then, add two visual components to the form: a Gauge and a Label. Set the Label's *Alignment* property to *taCenter*.

Next, add a *TBitBtn* to the form and set the *Kind* property to *bkCancel*. Using a *TBitBtn* instead of a regular button will allow us to indirectly control other properties of the button, such as *Caption*, *Glyph*, and *Cancel*, simply by setting the value of *Kind*.

The last control to add to ProcessingForm is a *TTimer* control. *Timer1* will be used to manage the modal state of the form after the process is either completed or canceled. We'll discuss the need for this control later; for now simply set its *Enabled* property to *False*, and its *Interval* property to `500` (milliseconds). Save the unit as PRCDLG.PAS.

```
TStartProcessingProcedure = procedure of object;
  TProcessingDialog = class(TComponent)
  private
    FCanceled : Boolean;
    FFormCaption : string;
    FProgress : Integer;
    FStatusMessage : string;
    FOnStartProcessing : TStartProcessingProcedure;
    FProcessingForm : TProcessingForm;
    procedure SetFormCaption(ACaption: string);
    procedure SetProgress(AProgress: Integer);
    procedure SetStatusMessage(AMessage: string);
  public
    constructor Create(AOwner: TComponent); override;
    function Execute: Boolean;
  published
    property Canceled: Boolean read FCanceled
                                write FCanceled;
    property FormCaption: string read FFormCaption
                                  write SetFormCaption;
    property Progress: Integer read FProgress
                                write SetProgress;
    property StatusMessage: string read FStatusMessage
                                    write SetStatusMessage;
    property OnStartProcessing: TStartProcessingProcedure
      read FOnStartProcessing write FOnStartProcessing;
  end;
```

Figure 2: Declaration for the new component in the PRCDLG.PAS unit, just below the ProcessingForm declaration.

```
function TProcessingDialog.Execute: Boolean;
begin
  Result := False;
  if Assigned(FOnStartProcessing) then
    begin
      FProcessingForm:= TProcessingForm.Create(Application);
      try
        if Assigned(FProcessingForm) then
          with FProcessingForm do begin
            Caption := FormCaption;
            Gauge1.Progress := Progress;
            Label1.Caption := StatusMessage;
            MyProcessingDialogControl := Self;
          end;
        Canceled  := False;
        Result    := (FProcessingForm.ShowModal = mrOk);
        Canceled  := not Result;
      finally
        FProcessingForm.Free;
        FProcessingForm := nil;
      end;
    end
  else
    MessageDlg('No OnStartProcessing handler defined.',
               mtError, [mbCancel], 0);
end;
```

Figure 3: *TProcessingDialog*'s *Execute* method.

Create the Wrapper

Now that we've created ProcessingForm (see Figure 1), we must create the component wrapper. The component, *TProcessingDialog*, will descend from *TComponent*. Create the declaration for the new component in the PRCDLG.PAS unit just below the ProcessingForm declaration, as shown in Figure 2.

The process is controlled in *TProcessingDialog*'s *Execute* method, as shown in Figure 3. *Execute* returns *True* if the process completed, and *False* if the user pressed Cancel. Notice that the control creates a new instance of *TProcessingForm* each time *Execute* is called. The **try..finally** block ensures the instance is disposed of before *Execute* is finished. Also note that the *Execute* method does not show *TProcessingForm* if no event handler is assigned to *OnStartProcessing*.

The control publishes four properties (*Canceled*, *FormCaption*, *Progress*, and *StatusMessage*) and one event (*OnStartProcessing*). The program can check the value of *Canceled* while the process is running to determine if the user has pressed Cancel. *FormCaption* is simply the title of the modal form that is displayed during the process. The default value of *Processing* can be changed by accessing this property.

Progress and *StatusMessage* are the two properties that should be continually updated during a long process. They give the user a visual indication of what is going on and how much has been completed. *StatusMessage* is displayed in *Label1* of ProcessingForm. The *Progress* property is directly passed through to the *Progress* property of *Gauge1*.

The control also publishes an *OnStartProcessing* event handler. The program will implement its process in the handler for this event. *OnStartProcessing* is called indirectly by calling the *Execute* method for the control.

The property access methods (the **read** and **write** specifiers) for each of the properties are shown in PRCDLG.PAS (see Listing One on page 10).

Modify the Form

Before examining how the control operates, we must return to *TProcessingForm* and further modify its definition. One modification is to add two **public** data members:

```
public
  MyProcessingDialogControl : TProcessingDialog;
  DesiredModalResult : TModalResult;
```

MyProcessingDialogControl is used by the form to communicate with the *TProcessingDialog* that created the form. The *DesiredModalResult* data field determines if the user pressed the button on the form.

We need to add some code to initialize the values of the new data members. Both can be set in the *FormCreate* method of the *TProcessingForm*. Create the *OnCreate* event handler for the form as shown here:

```
procedure TProcessingForm.FormCreate(Sender: TObject);
begin
  DesiredModalResult := mrNone;
  { Initial value is set to mrNone; this can only be set if
    the user presses the button on the form. }

  MyProcessingDialogControl := nil;
  { Execute will create and free its own instance of a
    TProcessingDialog. }
end;
```

DesiredModalResult and *Timer1* work together to control the modal state of ProcessingForm. We'll discuss the need for these two pieces in the next section, "Timing Is Everything."

```
procedure TProcessingForm.FormActivate(Sender: TObject);
begin
  if Assigned(MyProcessingDialogControl) then
    MyProcessingDialogControl.OnStartProcessing;

  if (DesiredModalResult = mrNone) then
    { The user did not press the button during the process
      (if so, DesiredModalResult would equal mrCancel). Make
       the button an OK button. }
    BitBtn1.Kind  := bkOk;
  Timer1.Enabled  := True;
end;

procedure TProcessingForm.BitBtn1Click(Sender: TObject);
begin
  if Assigned(MyProcessingDialogControl) then
    begin
      DesiredModalResult := BitBtn1.ModalResult;
      MyProcessingDialogControl.Canceled :=
        (DesiredModalResult = mrCancel);
    end;
end;
```

Figure 4: The two additions to *TProcessingForm*'s definition: assign a procedure to the form's *OnActivate* event handler; assign a procedure to BitBtn1's *OnClick* event handler.

There are two other additions that must be made to *TProcessingForm*'s definition. One is to assign a procedure to the form's *OnActivate* event handler. The other is to assign a procedure to BitBtn1's *OnClick* event handler. Both implementations are shown in Figure 4. These routines, along with the *Execute* method, work in concert to create the action of the processing dialog box. When the user calls the *Execute* method, the form is created and displayed via the call to *ShowModal*.

One of the effects of calling *ShowModal* is that it calls the form's *OnActivate* event handler, which, in this case, is *FormActivate*. The first thing *FormActivate* does is call the *OnStartProcessing* event handler. After *OnStartProcessing* has returned, *FormActivate* changes the BitBtn to an **OK** button. It then sets the *Enabled* property of *TTimer1* to *True*.

Timing Is Everything

So what are the timer control and the *DesiredModalResult* field used for? The answer lies in the VCL source. If you can, look at the file FORMS.PAS and search for the *ShowModal* method for a *TForm*. Forms have a property called *ModalResult* which is used to terminate a modal form. Setting *ModalResult* to any non-zero value ends the form's modal state. The value assigned to *ModalResult* becomes the return value of the *ShowModal* function call which displayed the modal form.

By default, *ModalResult* is set to zero when a form is displayed using *ShowModal*. The problem is that this value is set to zero *after* the call to the *OnActivate* event handler. So setting the value of *ModalResult* during the process loop (which is called during *OnActivate*) doesn't work.

Assigning a value to *DesiredModalResult* helps solve this problem. If the user presses the button on the processing form, *DesiredModalResult* is assigned the value intended for the form's *ModalResult* property. Because we can't set the value of *ModalResult* during our process, we need a way of transferring the value of *DesiredModalResult* to *ModalResult* after the process is complete. We can accomplish this by enabling the timer after we return from our processing loop. Then the *OnTimer* event handler transfers the value to *ModalResult*, which forces the form's modal state to end:

```
procedure TProcessingForm.Timer1Timer(Sender: TObject);
begin
  if Assigned(MyProcessingDialogControl) then
    DesiredModalResult := BitBtn1.ModalResult;
  ModalResult := DesiredModalResult;
end;
```

Compile and Install

Before compiling the new component, verify that your code matches PRCDLG.PAS as shown in Listing One. Remember to add the following registration procedure to the unit before trying to install the component:

```
procedure Register;
begin
  RegisterComponents('Dialogs',[TProcessingDialog]);
end;
```

Install the component on your palette. *TProcessingDialog* should appear on the Dialogs page of your palette.

Sample Program

After installing the component, create a new project. On the default form add a *TButton* and a *TProcessingDialog*. Double-click on the button and define the following method:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if ProcessingDialog1.Execute then
    ShowMessage('Process complete.')
  else
    ShowMessage('Process CANCELED!');
end;
```

Now define a test process. Place a ProcessingDialog control on the form, and specify the following code for its *OnStartProcess* event handler:

```
procedure TForm1.ProcessingDialog1StartProcessing;
var
  i : Integer;
begin
  i := 1;
  while (i < 1000) and not ProcessingDialog1.Canceled do
    begin
      Inc(i);
      Application.ProcessMessages;
      ProcessingDialog1.StatusMessage := 'Please wait...';
      ProcessingDialog1.Progress      := (i*100) div 1000;
    end;
end;
```

Let's examine the process in more detail. Notice that one of the tests for the loop is on the value of *ProcessingDialog1.Canceled*. This test will allow your process to stop prematurely. *Canceled* becomes *True* when the user presses the button on the form or hits Esc. If your loop is more complicated than the one shown here, you may want to check *Canceled* in several places to be more responsive to the user.

A key part of the loop is the call to *Application.ProcessMessages*. This call is crucial for the process to work correctly — it allows Windows to handle messages in its message queue. This means that if the user moves the mouse over the dialog box and presses Cancel, the program will respond.

Calling *ProcessMessages* is necessary, but be careful! The performance of your loop can be adversely affected by calling it too often. You must judge how often you want to make the call. Depending on the process, you may only want to relinquish control to Windows every fifth or sixth time through the loop. However, if one iteration of the loop takes a relatively long time, you may want to call *ProcessMessages* on every iteration.

Run several tests on your process to see how often you should call *ProcessMessages*. The interval should be long enough to allow your process to complete in a reasonable amount of time. However, it must be short enough so the user has adequate opportunity to cancel the process. It's frustrating to a user to press Cancel and wait a minute or so before the request is acknowledged.

Modifying the *OnStartProcessing* event handler for your application should be simple if you use a looping structure for your process. Remember to follow the same basic pattern for the processing loop:

```
procedure TForm1.ProcessingDialog1StartProcessing;
begin
  while { Your condition } and
        not ProcessingDialog1.Canceled do begin
    { Your application's process here }

    Application.ProcessMessages;
    ProcessingDialog1.StatusMessage  := 'some status';
    ProcessingDialog1.Progress       := { Some percentage };
  end;
end;
```

After you've assigned an event handler, call the *Execute* method of the control. The result will be *True* if the process completed, and *False* if the user pressed Cancel.

Not So Fast!

One characteristic of ProcessingDialog is that upon completion of the process, the form disappears and the program continues. However, this behavior can be modified to keep the form on the screen after the process finishes, as well as present the user with an OK button to acknowledge the end of the process. To do this, add a property to *TProcessingDialog* called *AutomaticEnd*:

```
private
  FAutomaticEnd : Boolean;

published
  property AutomaticEnd: Boolean read FAutomaticEnd
                                 write FAutomaticEnd;
```

Then change the **if** statement in the *Timer1Timer* event handler as follows:

```
if Assigned(MyProcessingDialogControl) then
  if MyProcessingDialogControl.AutomaticEnd then
    DesiredModalResult := BitBtn1.ModalResult;
```

The value of *AutomaticEnd* determines if ProcessingForm remains on the screen after the process is finished. If *AutomaticEnd* is *True* when the process completes, the form disappears and the program continues. If *False*, the BitBtn will change from Cancel to OK and wait for the user's response.

Take It from Here

This implementation of a process dialog box can be modified considerably to accommodate a variety of needs. You may want to change the *TLabel* to a *TMemo* to display more information during your process. If you are using Delphi 2, you may want to use a Windows 95-style progress bar instead of a *TGauge*. You can even configure your dialog box to look similar to the compiler progress dialog box for Delphi, which has many separate labels to show the state of different values.

When using this control and assigning an *OnStartProcess* event handler, remember three things:

- Check the *Canceled* property value often during your loop.
- Call *Application.ProcessMessages* during the process often enough to appear responsive to your user.
- Display progress reports quickly. Choose controls for the form that don't take a long time to draw or update.

Conclusion

Creating a consistent user interface is a fundamental requirement of any application. When presenting the user with delays in processing, it is important to present them with the same fundamental information and options every time. At a minimum, your program should exhibit text describing the process status, an indication of percentage complete, and a means for canceling the process. By using the ProcessingDialog component presented here, you can create a consistent way of communicating process status. ∆

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\97\MAR\DI9703BO.

Brad Olson is Vice President of Systems Design for Olson Research Associates, Inc. in Columbia, MD. His company has been developing its Financial Planning Model for Commercial Banks, using Borland products since 1986. He is currently using Delphi 2 for all new application development. You can reach Brad at (410) 290-6999 or by e-mail at 76646.2306@compuserve.com.

Begin Listing One — The prcdlg Unit

```
unit prcdlg;

interface

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

type
  TProcessingDialog = class;
  TProcessingForm = class(TForm)
    BitBtn1: TBitBtn;
    Gauge1: TGauge;
    Label1: TLabel;
    Timer1: TTimer;
    procedure FormCreate(Sender: TObject);
    procedure FormActivate(Sender: TObject);
    procedure BitBtn1Click(Sender: TObject);
    procedure Timer1Timer(Sender: TObject);
  public
    MyProcessingDialogControl : TProcessingDialog;
    DesiredModalResult : TModalResult;
  end;

  TStartProcessingProcedure = procedure of object;
  TProcessingDialog = class(TComponent)
  private
    FAutomaticEnd : Boolean;
    FCanceled : Boolean;
    FFormCaption : string;
    FProgress : Integer;
    FStatusMessage : string;
    FOnStartProcessing : TStartProcessingProcedure;
    FProcessingForm   : TProcessingForm;

    procedure SetFormCaption(ACaption: string);
    procedure SetProgress(AProgress: Integer);
    procedure SetStatusMessage(AMessage: string);

  public
    constructor Create(AOwner: TComponent); override;
    function Execute: Boolean;

  published
    property AutomaticEnd: Boolean read FAutomaticEnd
                                   write FAutomaticEnd;
    property Canceled: Boolean read FCanceled
                               write FCanceled;
    property FormCaption: string  read FFormCaption
                                  write SetFormCaption;
    property Progress: Integer read FProgress
                               write SetProgress;
    property StatusMessage: string read FStatusMessage
                                   write SetStatusMessage;
    property OnStartProcessing: TStartProcessingProcedure
      read FOnStartProcessing write FOnStartProcessing;
  end;

procedure Register;

implementation

{$R *.DFM}

constructor TProcessingDialog.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  FAutomaticEnd  := True;
  FCanceled      := False;
  FFormCaption   := 'Processing';
  FProgress      := 0;
  FStatusMessage := 'Please wait...';
end;

procedure TProcessingDialog.SetFormCaption(
  ACaption: string);
begin
  FFormCaption := ACaption;
  if Assigned(FProcessingForm) then
    FProcessingForm.Caption := FormCaption;
end;

procedure TProcessingDialog.SetProgress(AProgress: Integer);
begin
  FProgress := AProgress;
  if Assigned(FProcessingForm) then
    FProcessingForm.Gauge1.Progress := Progress;
end;

procedure TProcessingDialog.SetStatusMessage(AMessage:
                                             string);
begin
  FStatusMessage := AMessage;
  if Assigned(FProcessingForm) then
    FProcessingForm.Label1.Caption := StatusMessage;
end;

function TProcessingDialog.Execute: Boolean;
begin
  Result := False;
  if Assigned(FOnStartProcessing) then
    begin
      FProcessingForm:= TProcessingForm.Create(Application);
      try
        if Assigned(FProcessingForm) then
          with FProcessingForm do begin
            Caption          := FormCaption;
            Gauge1.Progress  := Progress;
            Label1.Caption   := StatusMessage;
            MyProcessingDialogControl := Self;
          end;
        Canceled  := False;
        Result    := (FProcessingForm.ShowModal = mrOk);
        Canceled  := not Result;
      finally
        FProcessingForm.Free;
        FProcessingForm := nil;
      end;
    end
  else
    MessageDlg('No OnStartProcessing handler defined.',
               mtError, [mbCancel], 0);
end;

procedure TProcessingForm.FormCreate(Sender: TObject);
begin
  DesiredModalResult := mrNone;

  MyProcessingDialogControl := nil;
end;

procedure TProcessingForm.FormActivate(Sender: TObject);
begin
  if Assigned(MyProcessingDialogControl) then
    MyProcessingDialogControl.OnStartProcessing;
  if (DesiredModalResult = mrNone) then
    BitBtn1.Kind := bkOk;
 Timer1.Enabled := True;
end;

procedure TProcessingForm.BitBtn1Click(Sender: TObject);
begin
  if Assigned(MyProcessingDialogControl) then
    begin
      DesiredModalResult := BitBtn1.ModalResult;
      MyProcessingDialogControl.Canceled :=
        (DesiredModalResult = mrCancel);
    end;
end;

procedure TProcessingForm.Timer1Timer(Sender: TObject);
begin
  if (DesiredModalResult = mrNone) then
    if MyProcessingDialogControl.AutomaticEnd then
      DesiredModalResult := BitBtn1.ModalResult;
  ModalResult := DesiredModalResult;
end;

procedure Register;
begin
  RegisterComponents('Dialogs', [TProcessingDialog]);
end;

end.
```

End Listing One

By Derek Davidson

Go Your Own Way

Customizing DBNavigator Component Buttons

As a Delphi trainer, a question I'm often asked is: "How do I use my own images on the DBNavigator component?" The simple answer, of course, is that you can't; Borland doesn't provide a property that lets you do it. With Delphi however, there's always a way — in this case, more than one.

Quick and Dirty

The simplest way to use custom images with a DBNavigator component is to edit the Delphi database controls resource file, DBCTRLS.RES, found in \PROGRAM FILES\BORLAND\DELPHI 2.0\LIB (the path names used in this article are based on a default installation of Delphi 2). Use the Delphi Image Editor (or other resource editor) to examine DBCTRLS.RES. You'll see the default buttons used in the DBNavigator component. Simply replace the default bitmaps with your own. Remember to keep the same image names; Delphi uses these to load the images. Now, whenever you use a DBNavigator component, you'll see your own images in place of the Delphi defaults.

That's the easy — and inflexible — answer. What if you need more flexibility? For example, what if you need to change these images at design time? The only realistic option is to write a custom component and add an image property.

A Better Way

Unfortunately, when Borland produced the DBNavigator component, they made it extremely difficult to add any additional functionality, especially using the preferred, OOP style of sub-classing. This means we must either produce a custom component from scratch, or rewrite the Borland offering. In this article, we'll rewrite some of the Delphi code to produce our own DBNavigator. If you have the VCL source, you'll likely find the file we're to modify in \PROGRAM FILES\BORLAND\DELPHI 2.0\SOURCE\VCL\DBCTRLS.PAS.

Before we proceed, a quick word of caution: Component writing can damage your Delphi Component Library. Before working with — or even installing — components, it's wise to make a backup of your current library: \PROGRAM FILES\BORLAND\DELPHI 2.0\BIN\CMPLIB32.DCL. Should your library become corrupt, you can recover it by simply substituting your saved copy.

Let's start by making a copy of the source code. Create a directory anywhere on your system and copy the DBCTRLS.PAS file into it. Next, fire up Delphi, make sure no other files are open, then use **File | Open** to load the copy of the source code. Select **File | Save As** and save the source code as TDM.PAS. You can, of course, give it any name you like, but let's keep "TDM" for now.

One of the first things we'll do is remove all the code that isn't pertinent. This makes reference simpler, and prevents duplicating code that appears in the original DBCTRLS.PAS file. I'll refer to line numbers as we go through this stage; in case your source code doesn't perfectly match mine, I'll also give text-quoted line references. Refer to the table in Figure 1 and remove all code from (and including) the start point, up to (but *not* including) the end

Start Text	Start Line	End Text	End Line
`type`	19	`const`	64
`{ TDBLookup Control`	142	`implementation`	416
`{ TPaintControl`	148	`{ TDBNavigator`	1938
`{ TDataSourceLink`	622	`end.`	1712

Figure 1: Trim your copy of DBCTRLS.PAS by removing these lines of code, including the start lines, but *not* the end lines.

point. If all goes well, the TDM unit will contain only 622 lines — certainly easier to work with than the original 4406.

Next, we need to change the name of our component. To do this, select **Search | Replace**. In the dialog box, search for `TDBNavigator` and replace it with `TDMNavigator`. Make sure you have the **Global** radio button selected, then perform the replacement.

Before our first test, we'll comment out the "include resource" directive. Go to line 146, which contains:

```
{$R DBCTRLS.RES}
```

and change it to:

```
//{$R DBCTRLS}
```

Let's try to install the component to ensure the work we've done is right. To do this, we need to add a *Register* procedure to our code. *Register* tells Delphi to add a component to its component library, and which page of the Component palette the component should appear on. First, we forward-declare the *Register* procedure in the interface part of the code. Insert the following code on line 141 of your source code (just before the line containing the keyword **implementation**):

```
procedure Register;
```

Next, add the *Register* procedure at the end of the source code:

```
procedure Register;
begin
  RegisterComponents('TDM',[TDMNavigator]);
end;
```

This code adds a component called *TDMNavigator* to the TDM page of your Component palette. You probably don't yet have a TDM page — don't worry, Delphi will create it automatically.

The Moment of Truth

From the Delphi menu, select **Component | Install**. Click **Add**, then **Browse**. Select your TDM file (TDM.PAS) and click **OK**. Click **OK** once more, and Delphi will attempt to rebuild the library. If all has gone well, you'll see a new tab, titled TDM, on your Component palette. If you click on the tab, you should see a single option consisting of a circle, square, and triangle (the default Delphi component image). Pass the mouse pointer over this button, and you should see the hint "DMNavigator". As a final test, select **File | New Form** from the Delphi menu and attempt to add your new component to the form. If everything has worked, you'll see a DBNavigator component on the form, but it's actually your DMNavigator. Once you've tested this, dispose of the test unit and the form to which it relates, leaving only your component code.

If anything has failed thus far, your best bet is to start from scratch. Select **Component | Install** from the Delphi menu to display the Install Components dialog box. Highlight the `TDM` component in the **Installed Units** list box, then click **Remove**. Delphi will now attempt to rebuild your library, omitting the *TDMNavigator* component. If unsuccessful, reinstall the library you saved earlier and start again.

We're almost ready to start coding, but first let's cover how to implement the image properties.

Nuts and Bolts

It's important that our component match the look and feel of the original Delphi component. To that end, let's examine how it allows customization by the user, taking the *Hints* property as an example.

TDBNavigator features a set of default hints that can be overwritten by a user entering hints into a *TStrings* editor. The positioning of these strings is important: The first string is applied as a hint to the first button in the component, the second string is applied to the second button in the component, and so on. To emulate this style, we'll use a *TImageList* component. *TImageList* is a form of container object in that it holds a series of images in a manner similar to the way a *TStrings* component holds a series of strings. Each image in *TImageList* is referred to by ordinal position, so we can use this just as we would user-provided hints.

Having decided on this approach, let's add a new property to our component. First, we're going to make an entry in the **private** section of the *TDMNavigator* class:

```
FButtonGlyphs: TImageList;
```

Place it on line 48, between the following lines:

```
FConfirmDelete: Boolean;
function GetDataSource: TDataSource;
```

To allow access to the property at design time, we must declare it as published in the *TDMNavigator* class declaration. So, in the **published** section of the *TDMNavigator* source code, insert:

```
property ButtonGlyphs: TImageList read FButtonGlyphs
                                  write SetButtonGlyphs;
```

Let's quickly discuss what the **published** property is doing for us. First, we declare the property with the name *ButtonGlyphs*. This is the name the Delphi developer will see in the Object Inspector at design time; it's also the name the developer will use to refer to the property at run time, if appropriate.

Second, the **read** section tells us that to obtain the current value of the *ButtonGlyphs* property, we must examine *FButtonGlyphs*. You'll recall that *FButtonGlyphs* was declared

```
procedure TDMNavigator.SetButtonGlyphs(Value: TImageList);
var
  n        : Integer;      // General looping variable
  x        : TNavigateBtn; // Local variable button type
  ResName  : string;       // Local copy of resource name
begin
  { Set FButtonGlyphs to passed value }
  FButtonGlyphs := Value;

  { Cycle through Navigator Buttons available in current
    TDMNavigator component, and set button image to correct
    image resource loaded from the DBCTRLS resource file. }
  for x := Low(TNavigateBtn) to High(TNavigateBtn) do begin
    FmtStr(ResName,'dbn_%s',[BtnTypeName[x]]);
    Buttons[x].Glyph.Handle  :=
      LoadBitmap(HInstance, PChar(ResName));
    Buttons[x].NumGlyphs     := 2;
  end;

  { If ButtonGlyphs property is set, cycle through images
    and apply images to buttons in the TDMNavigator.  }
  if ButtonGlyphs <> nil then
    begin
      { Loop through all available Navigator buttons and set
        their images according to images in the TImageList
        component pointed to by FButtonGlyphs. }
      for n := 0 to Lesser(FButtonGlyphs.Count - 1,
                           Ord(High(TNavigateBtn))) do
        begin
        FButtonGlyphs.GetBitmap(
          n,Buttons[TNavigateBtn(n)].Glyph);
        Buttons[TNavigateBtn(n)].NumGlyphs := 1;
      end;
    end;
end;
```

Figure 2: The *SetButtonGlyphs* procedure.

earlier in the **private** section of the *TDMNavigator* class declaration. Declaring it as private means that access to it is available only from within the class. Why didn't we simply declare *FButtonGlyphs* as public and make it available for all to examine? Because the **write** section tells us that we'll use *SetButtonGlyphs* to write the value to *FButtonGlyphs*.

Now let's write the *SetButtonGlyphs* procedure (see Figure 2). This code is heavily commented to guide you. It may raise as many questions as it answers, but I'll address those shortly. For now, simply enter the code as shown. I placed mine just after the *TDMNavigator.Destroy* method, but you can, of course, place it anywhere you wish within the TDM unit's **implementation** section. As with all procedures in Delphi, we also need to forward-declare it, so add the following line in the **private** section of the *TDMNavigator* class:

```
procedure SetButtonGlyphs(Value: TImageList);
```

Remember the earlier question about declaring *FButtonGlyphs* as public? Using that style, we couldn't use a method to read or write the value and, as the above method shows, writing a value to a class-instance variable often means more than just assignment. Your customer will use a property called *ButtonGlyphs* and may not know that a lot of processing is being done in the background. If, in subsequent releases of your component, you change the code outlined above, your Delphi developer will never know — his or her code will compile exactly as before.

To make our testing of the component a little easier, and to prevent us from having to recompile the library every time we test it, we'll create a new application. Select **File | New Application** from the Delphi menu. (Your *TDMNavigator* source will disappear. If you're prompted to save it, select **Yes**.) Choose **View | Project Manager** and press the **Add Unit** button. Select your TDM.PAS file and click **OK**. Go to your application source code file (probably titled Unit1) and add `TDM` to the unit's **uses** clause:

```
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, TDM;
```

Add a *TDMNavigator* component to your form and compile the application. There will be two errors relating to a missing method named *Lesser*. Let's remedy this by adding, to our component source code, a general function titled — unsurprisingly — *Lesser*. It accepts two parameters and returns a value matching the lower of the two. We use it to ensure that we don't attempt to write images to non-existent buttons in *TDMNavigator* (thereby creating run-time exceptions). The code for *Lesser* is:

```
function Lesser(NumberA, NumberB : Integer) : Integer;
begin
  if NumberA < NumberB then
    Result := NumberA
  else
    Result := NumberB;
end;
```

Add it to the end of your TDM unit — and don't forget the forward declaration:

```
function Lesser(NumberA, NumberB : Integer ) : Integer;
```

This would normally be added immediately prior to the forward declaration for the *Register* procedure, which we covered earlier.

If you recompile now, all should be well. If so, rebuild the library by selecting **Component | Rebuild Library**. (Select **Yes** if prompted to save any changes to the project.) When complete, examine your application again; you should notice that *TDMNavigator* now has a new property titled *ButtonGlyphs*.

Up and Running — I Think

Well, we haven't come this far not to test it, so let's try the property. Drop a *TImageList* component onto the form (you'll find it on the Win95 page). Then set the *ButtonGlyphs* property for *TDMNavigator* to *TImageList*. Not a lot will happen, as we haven't added any images to *TImageList*; let's do that now. Double-click the *TImageList* component and add some images. You'll find some useful images in PROGRAM FILES\BORLAND\DELPHI 2.0\IMAGES\BUTTONS.

When you've finished adding images, you'll notice no change to the navigator. Remedy this by recycling the selection of the *TImageList* component (that is, reselect *TImageList* in *TDMNavigator*'s *ButtonGlyphs* property). Voilà! We now have a *TDMNavigator* component capable of accepting user-defined images at design time.

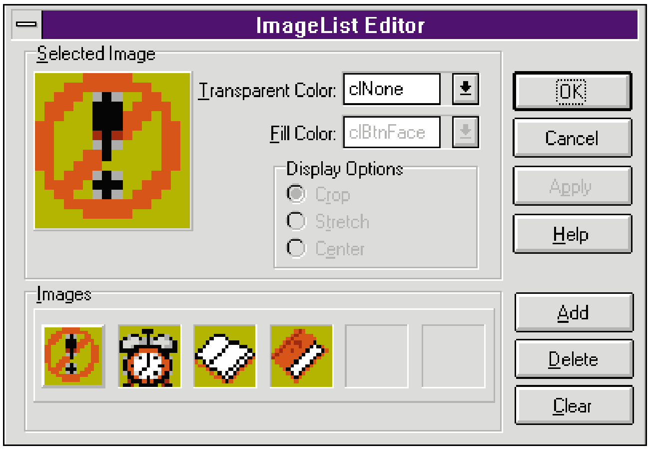


Figure 3: *TChangeLink* notes any changes to *TImageList*, like the four images added here ...

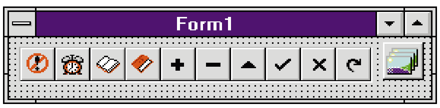


Figure 4: ... and affects the changes in registered objects like *Form1*.

We could leave the component as-is, but let's make it more professional. Any changes to *TImageList* ought to be reflected immediately within *TDMNavigator*. So how do we achieve this? Delphi provides a couple of interesting methods. They are *RegisterChanges* and *UnRegisterChanges*, methods of the *TImageList* component. They allow any object to register any changes to *TImageList*. When changes are made, it transmits the fact to all registered objects, which can then act appropriately. Next question: "How do we do that?" Enter the *TChangeLink* object.

The *TChangeLink* Object

The *TChangeLink* object is used by *TImageList* to notify registered objects of changes (see Figures 3 and 4). *TChangeLink* itself must contain the name of a procedure to call when changes are made to the *TImageList*. In our case, we'd provide the name of a method within our *TDMNavigator* component. So now, when *TImageList* is changed, a method in our *TDMNavigator* is called, within which we can update our *ButtonGlyphs*.

We'll implement this next. (You may find that you need to re-open your TDM unit to get to the source code. A good short cut for this is to place the cursor on the word `TDM` in your Unit1 **uses** clause, then right-click and select **Open File At Cursor** from the pop-up menu.)

Let's write the necessary code for the *TChangeLink* object. Add an entry in the **private** section of the *TDMNavigator* class code as follows:

```
FImageChangeLink : TChangeLink;
```

Next, we need to instantiate the *TChangeLink* object, so add the following code to the constructor of *TDMNavigator* (after the inherited *Create* call):

```
FImageChangeLink          := TChangeLink.Create;
FImageChangeLink.OnChange  := ImageListChange;
```

```
procedure TDMNavigator.ImageListChange(Sender: TObject);
var
  n         : Integer;
  x         : TNavigateBtn;
  ResName   : string;
begin
  for x := Low( TNavigateBtn ) to High( TNavigateBtn ) do
    begin
      FmtStr(ResName,'dbn_%s',[BtnTypeName[x]]);
      Buttons[x].Glyph.Handle  :=
        LoadBitmap(HInstance, PChar(ResName));
      Buttons[x].NumGlyphs     := 2;
    end;

  for n := 0 to Lesser(FButtonGlyphs.Count - 1,
                       Ord(High(TNavigateBtn))) do begin
    Buttons[TNavigateBtn(n)].Glyph := nil;
    FButtonGlyphs.GetBitmap( n, Buttons[TNavigateBtn(n)].Glyph );
    Buttons[TNavigateBtn(n)].NumGlyphs := 1;
    Buttons[TNavigateBtn(n)].Refresh;
  end;
end;
```

Figure 5: The *ImageListChange* procedure.

Of course, if we create an object, we must also destroy it. Add the following code to the destructor of *TDMNavigator* (before the inherited *Destroy* call):

```
FImageChangeLink.Free;
```

You'll notice that in the constructor, we set the *FImageChangeLink.OnChange* event to point to a *TDMNavigator* method titled *ImageListChange*, which we've yet to produce. The code we need to add is shown in Figure 5 (you'll notice some similarities with the *SetButtonGlyphs* method we produced earlier). As usual, don't forget to forward-declare the method. Place the following code in the **private** section of the *TDMNavigator* component:

```
procedure ImageListChange(Sender: TObject);
```

This is an opportune moment to try recompiling. If all is well, your application should compile without problems. Now we need only use the *RegisterChanges* and *UnRegisterChanges* methods, in the *SetButtonGlyphs* method that we wrote earlier. Expand *SetButtonGlyphs* to match the code shown in Figure 6. I've attached extra comments to explain the added code, and highlighted the new portions.

Compile your application to check the code you just added, then rebuild the library. Test that the ImageList works correctly. Note that changes aren't notified to *TDMNavigator* until you press the **OK** button on the ImageList Editor dialog box. These should be shown immediately at design time, without having to recycle the *ButtonGlyphs* property.

So there we have it: a complete, commercial-quality addition to our Delphi toolbag that will increase the speed and ease with which we can produce our code.

One Last Thing

Currently, if you use a *TImageList* component, set the *ButtonGlyphs* property, then delete the *TImageList* compo-

```
procedure TDMNavigator.SetButtonGlyphs(Value:  TImageList);
var
  n        : Integer;       // General looping variable
  x        : TNavigateBtn;  // Local variable for button type
  ResName : string;         // Local copy of resource name
begin
  { Check if ButtonGlyphs property has been set. If so,
    sever our interest in changing ImageList encapsulated
    in ButtonGlyphs. }
  if ButtonGlyphs <> nil then
    ButtonGlyphs.UnRegisterChanges(FImageChangeLink);
  // Set FButtonGlyphs to passed value.
  FButtonGlyphs := Value;

  { Cycle through Navigator Buttons available in current
    TDMNavigator component, and set button image to correct
    image resource loaded from the DBCTRLS resource file. }
  for x := Low(TNavigateBtn) to High(TNavigateBtn) do begin
    FmtStr(ResName,'dbn_%s',[BtnTypeName[x]]);
    Buttons[x].Glyph.Handle :=
      LoadBitmap(HInstance, PChar(ResName));
    Buttons[x].NumGlyphs    := 2;
  end;

  { If the ButtonGlyphs property is set, cycle through
    images and apply images to buttons in TDMNavigator. }
  if ButtonGlyphs <> nil then
    begin
      { Record that we're interested in changes to the
        TImageList, by notifying any changes to
        FImageChangeLink. }
      ButtonGlyphs.RegisterChanges(FImageChangeLink);
      { Loop through all available Navigator buttons and
        set their images according to images in TImageList
        component pointed to by FButtonGlyphs. }
      for n := 0 to Lesser(FButtonGlyphs.Count - 1,
                           Ord(High(TNavigateBtn))) do
        begin
        FButtonGlyphs.GetBitmap(
          n,Buttons[TNavigateBtn(n)].Glyph);
        Buttons[TNavigateBtn(n)].NumGlyphs := 1;
      end;
    end;
end;
```

Figure 6: The expanded *SetButtonGlyphs*.

nent, you'll get an Access Violation error reported in the Object Inspector against the *ButtonGlyphs* property (see Figure 7).

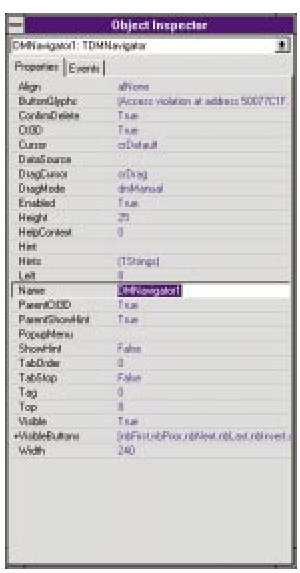


Figure 7: An ounce of prevention avoids this error in *ButtonGlyphs*.

Needless to say, this is a situation we should prevent programmatically, but how do we notify the *TDMNavigator* that the *TImageList* component has been deleted? With the *FreeNotification* method of the *TComponent* class.

The *FreeNotification* method accepts a single parameter of type *TComponent*. For the sake of discussion, let's call this parameter *AComponent*. By calling the *Notification* method in the passed component, *FreeNotification* ensures that the component passed as the parameter is notified when the component to which *FreeNotification* refers is to be destroyed. In our example, we could show this with the pseudo-code:

```
TImageList.FreeNotification(TDMNavigator);
```

This would call our notification method for the *TDMNavigator* component when *TImageList* will be destroyed. To employ this functionality, add the following line of code to the *SetButtonGlyphs* method:

```
// Value is the passed TImageList
Value.FreeNotification(Self);
```

Insert this immediately after the following statement:

```
ButtonGlyphs.RegisterChanges(FImageChangeLink)
```

Now, inside our *Notification* method, we reset our *ButtonGlyphs* property. Expand the *Notification* method by inserting code to match that shown in Figure 8.

```
procedure TDMNavigator.Notification(AComponent: TComponent;
                                    Operation: TOperation);
begin
  inherited Notification(AComponent, Operation);
  if (Operation = opRemove) and
     (FDataLink <> nil)     and
     (AComponent = DataSource) then
    DataSource := nil;
    { New code added here to cope with
      destruction of TImageList. }

  if (Operation = opRemove) and
     (AComponent = ButtonGlyphs) then
    ButtonGlyphs := nil;
end;
```

Figure 8: The expanded *Notification* method.

Conclusion

Now, compile your application to prove your code, then rebuild the library. You now have a useful component you can easily drop on any form. As a flourish, you might provide a separate icon for the Visual Component Library. ∆

The TDM.PAS file referenced in this article is available on the Delphi Informant Works CD located in INFORM\97\MAR\DI9703DD. Mr Davidson's TDBNavigateCollate *component, which provides all the functionality described in this article, as well as offering a collation capability, refresh capability, and incremental search capability for* TQuery*-based datasets, is also available from the Delphi 2 forum on CompuServe (GO BDELPHI32).*

Derek Davidson is a part-time technical author and full-time Delphi user who specializes in client/server applications. He currently works for Micro Key Software in Kissimmee, FL. Derek can be contacted on CompuServe at 100432,1360, or via the Internet at 100432.1360@compuserve.com.

Delphi 2

By *Alex Sumner*

Key Issues

Modifying Standard Keyboard Behavior

When a Windows version of an application replaces a DOS counterpart, users must learn new ways of working with it. This is not always welcome, particularly if users prefer the old version. One common complaint concerns the use of the Enter key. For many years, and on many platforms, it has been standard for users to press Enter when they have completed entering one item of information and wish to move on to the next.

Windows has a different standard: Tab is used to move to the next item on a form; Enter means all items have been completed, and usually has the same effect as clicking an OK button. In Delphi, a *TButton* or *TBitBtn* has a property called *Default*; if this is set to *True*, then pressing Enter when the button's form is active will have the same effect as clicking on that button. Usually, OK is the default button on a form.

Nearly every Windows user must at some time have experienced this: a dialog box is displayed with several items of information to enter, they fill in the first one, press Enter to go to the second, and the dialog box vanishes — they have inadvertently activated the OK button. The most common response to this complaint is "Using Tab to move from control to control is the Windows standard; get used to it."

While conforming to a common standard is generally wise, the complaint in this case arises because we have departed from a previous standard. This article will address alternative approaches to the problem. We'll begin by reviewing some widely-known solutions (using keyboard event handlers), and demonstrate what can and cannot be achieved using these "standard" approaches. We'll then present a more powerful, object-oriented solution. In doing so, we'll look in-depth at how Delphi applications handle keyboard input, and learn other useful tactics, even for developers who have no intention of departing from the standard Windows handling of Enter.

Standard Approaches

The easiest solution to the previous example of the disappearing dialog box is to have the OK button disabled until entry of all required items of information. Take for example a modal dialog box with several edit boxes to be filled. The OK button is not enabled until all the boxes contain some text, and only then will Enter activate OK and hide the dialog box. While this is an improvement, two problems remain:

1) This approach is not always applicable. There may be circumstances when some items in a dialog box are optional, so OK must be enabled even when these items have not been supplied.
2) Even when this approach can be used, it doesn't accomplish what the user wants. Using the example of a dialog box with several edit boxes to fill, when the user initially sees the dialog box, the first edit box already has focus. The user supplies the required information, and presses Enter. The dialog box doesn't disappear, but the focus doesn't move either. The computer simply beeps, because Enter has

returned focus to the first edit box, which cannot accept it. This can be annoying; the application is not misinterpreting the key press, it's rejecting it. The user obviously wants to move to the next item — and they want to know why the application can't do it.

If our application is to respond more intelligently to Enter, we must write some code. One obvious location for this code is in an *OnKeyPress* event handler. Figure 1 shows an *OnKeyPress* event handler that responds to Enter by moving the focus to the next control in the tab order, then setting the *Key* value to 0, thus achieving what the user wants *and* preventing further processing of the key. However, if you try this, you'll discover it doesn't work when there is a default button enabled on the form. It only works in conjunction with disabling **OK**, as previously described.

Of course, we could simply not include a default button, but there is still a drawback to our approach. To be consistent, we must attach our code to the *OnKeyPress* event handler of all our edit boxes, and perhaps of other controls, on all the forms in our application. This is rather messy and redundant. Every time we add a new component, we must remember to attach our code to its *OnKeyPress* event handler. We might try to simplify matters by creating our own customized edit box component, which could have the required focus-moving behavior built in. Unfortunately, this will commit us to the potentially open-ended task of modifying VCL components. Having customized the edit box, we may find we also need a customized *TMaskEdit*, or perhaps a customized *TComboBox*.

To find a more self-contained solution, we must approach the problem from a different perspective. The words "self contained" provide a clue as to how we may proceed. Our problems occur in part because moving the focus is not something that concerns *only* the control which currently has the focus. Moving the focus also affects the control which is to receive the focus, as well as the form on which both controls are placed. To find a self-contained solution, we must work at the form level.

A Delphi *TForm* has a property called *KeyPreview*, which is of type *Boolean* and has a default value of *False*. If *KeyPreview* is set to *True*, the key events (*OnKeyDown*, *OnKeyPress*, and *OnKeyUp*) go to the form's event handler before moving to the event handler of the control with focus. By setting *KeyPreview* to *True* and attaching our event handler to the form's *OnKeyPress* event handler, we can achieve the same effect, but with much greater economy of effort. Now we must only remember to attach code to the *OnKeyPress* event handler each time we add a new form, rather than each time we add a component to an existing form. This is certainly an improvement, but we still have a problem with the default **OK** button. If you try this, you'll find that no matter what code you attach to a form's *OnKeyPress* (or *OnKeyDown* or *OnKeyUp*) event handler, you cannot prevent an enabled default button from being activated by Enter.

```
procedure TMyForm.KeyPress(Sender: TObject; var Key: Char);
begin
  { If it's an Enter key ... }
  if Key = #13 then
    begin
      { Move the focus ... }
      SelectNext(ActiveControl,True,True);
      { and prevent further processing of the key. }
      Key := #0;
    end;
end;
```

Figure 1: An *OnKeyPress* event handler.

We could work around this by being careful when we enable or disable default buttons. Alternatively, we could avoid the problem altogether by not setting any buttons as the default button on a form. After all, if we don't want them to *behave* like default buttons, there is little reason to *make* them default buttons.

Taking Stock

Now we're in a position to make our application respond in any one of three ways to Enter:

1) We can have standard Windows key handling, with default buttons.
2) We can keep the default buttons, but change our forms by setting *KeyPreview* to *True* and supplying a suitable event handler for the *OnKeyPress* event handler. If we are careful about when we enable the default buttons, we can then simulate a non-Windows user interface in many circumstances, while still conforming to the Windows standard (i.e. Enter will activate an enabled **OK** button).
3) We can depart from the Windows standard altogether by removing the default buttons. Of course, we'll want to retain the **OK** buttons, but we can set their *Default* properties to *False*. This way our application will behave in a manner familiar to DOS users: Pressing Enter will only activate an **OK** button if the focus is already on that button.

But Wait, There's More

We could stop at this point. If we're confident we know what our users require, and that those requirements will not change, we have a solution. Unfortunately, this solution introduces dependencies. The required behavior only emerges when different properties of both our forms and the buttons on them are set in a suitable combination. The weakness of this design becomes apparent when we try to make changes, as is often the case with design problems. Suppose we've implemented the third option of emulating a DOS user interface, but a new customer requests standard Windows key handling. In this case, we must go through all our forms, removing the *OnKeyPress* event handlers and setting the **OK** buttons as default buttons again. Worse still, suppose we have a multi-user application, and each user wants the capability to specify which key-handling standard they prefer, and have the application thereafter respect their wishes. We must either add event handlers, or identify default buttons at run time, as each form is loaded.

Our original problem was simple and self-contained. We'd like to find a solution requiring modifications to be made in only one location. If this is what we want, or even if we are curious as to why the *OnKeyPress* approach cannot do it, we must look in more detail at the way our application receives and responds to keyboard input.

Keyboard Handling in Delphi 2

When our example dialog box is displayed with the focus on its first edit box and the user presses a key, a complicated chain of events begins. How the application responds depends on several factors, such as which key was pressed, and whether there is an enabled default button on the form. If the key is a character key, that character will be appended to the text in the edit box. If Tab is pressed, the focus will be moved. If Enter is pressed, then either the default button will be activated, or if the default button is not enabled, a warning beep will sound. To successfully alter this behavior, we must make our changes at the appropriate point in the chain of events. If we take action too late, we'll find we must make changes in many locations to consistently catch all occurrences.

On the other hand, if we take action too soon (e.g. in a handler for the application's *OnMessage* event), we may prevent something else from working because we've intercepted Enter before any of the normal processing for it has occurred. Take for example grid controls, which allow in-place editing of their data. They can require Enter to initiate editing a value. If we've already intercepted Enter for our own purposes, this will no longer work.

To find an appropriate location to make our changes, we should first find the point at which Tab normally causes the focus to move. To do this, follow the process from the moment the user presses a key. (Note: The following description is accurate for Delphi 2, and the solution has only been tested under Delphi 2.)

The first action when the user presses a key is that Windows creates a *WM_KEYDOWN* message for the window with focus (in our example, the edit box), and posts it to our application's message queue. This message contains the details of which key was pressed. In a Delphi application, this role is performed by the Application object (which is found in the Delphi VCL source file FORMS.PAS). The Application object also sits in a loop taking messages from the queue.

However, the Application object doesn't always simply dispatch these messages. Under some circumstances it will process a message itself, or even send an alternative message to the window. A *WM_KEYDOWN* message is one such circumstance. Before deciding whether to dispatch the message to the WinControl for which it is intended, the Application object first sends a corresponding *CN_KEYDOWN* message to the WinControl. *CN_KEYDOWN* is a Delphi-defined message declared in the source file CONTROLS.PAS. (In the remainder of this article, we will refer to several other Delphi-defined messages. Each will have a prefix of CN_ or CM_, as opposed to the WM_ prefix of standard Windows messages.) The *CN_KEYDOWN* message contains the same information about which key was pressed as did the original *WM_KEYDOWN* message.

The *CN_KEYDOWN* message is then processed by the Delphi message handling system, and ultimately arrives at the *CNKeyDown* message handler that all WinControls inherit from *TWinControl* (which is also found in the source file CONTROLS.PAS). *CNKeyDown* processes the message in five steps:

1) *CNKeyDown* checks if the key is a menu key for a pop-up menu, the WinControl's parent form's menu, or the main form's menu (the latter involves sending the Application object a *CM_APPKEYDOWN* message). If any of these apply, the key is processed and the *CNKeyDown* message handler exits with the message result field set to 1. Otherwise, proceed to step 2.
2) The WinControl's *CNKeyDown* message handler sends itself a *CM_CHILDKEY* message by calling its *Perform* method. (*Perform* is a static method all Delphi controls inherit from *TControl*. It is used to send a message directly to a control without going via Windows. A control's *Perform* method takes three parameters, with which it constructs a message; it then sends this message to itself by directly calling its own *WndProc* method.) The *CM_CHILDKEY* message then goes through the Delphi message handling system to arrive at the WinControl's *CMChildKey* message handler. This passes the message on to its parent by directly calling its parent's *WndProc* method, unless it has no parent, in which case it returns immediately without performing any processing of the message. The result navigates through the chain of parents until it runs out of parents, but on the way it may find one to process the key by overriding the *CMChildKey* message handler. The only VCL component that processes keys this way is the *TDBControlGrid*, which processes Tab, F2, Esc, and Return, and sets the *CM_CHILDKEY* message result field to 1 to indicate it has done so. In any case, it eventually arrives at a WinControl that has no parent, then returns down the chain of *CMChildKey* message handlers to the original *CNKeyDown* handler. This inspects the result field of the *CM_CHILDKEY* message, which now has the value 1 if the key has been processed, and 0 otherwise. If the key has been processed, the *CNKeyDown* handler sets its message result field to 1 and exits. The third, fourth, and fifth steps occur only if one of the following keys is pressed: Tab, Esc, an arrow key, Return, Execute, or Cancel. If the key is not one of these and has not been processed in steps 1 or 2, the *CNKeyDown* handler exits with its result field left as 0.
3) The WinControl's *CNKeyDown* handler checks if the WinControl itself wants the key by sending itself a *CM_WANTSPECIALKEY* message. Again, it does this by calling its own *Perform* method. Only two VCL components indicate they want keys in response to this message. The first is a *TCustomGrid* with *goEditing* in its *Options* set, which requests Enter. The other is a *TMaskEdit* that has had its contents modified, which will request Esc. In

any case, the key is not processed at this stage; a component that wants the key merely indicates this by setting the *CM_WANTSPECIALKEY* message result field to 1; all other WinControls leave the result field as 0. If the key is desired, the *CNKeyDown* handler prevents any other processing of the key at this stage by exiting with its message result field left at 0. Otherwise, proceed to step 4.

4) The WinControl's *CNKeyDown* handler further verifies if the *TWinControl* itself wants the key by sending itself a *WM_GETDLGCODE* message. Again, it does so by calling its own *Perform* method. *WM_GETDLGCODE* is a standard Windows message, and unless a particular WinControl supplies or inherits a handler for it, it will eventually be processed by the default window procedure for the WinControl's window class. Some VCL components do supply handlers for the *WM_GETDLGCODE* message (e.g. a *TDBNavigator* will request arrow keys, and a *TCustomMemo* will request Tab if its *WantTabs* property is *True*). Wherever the *WM_GETDLGCODE* message is handled, its result field indicates which types of keys are required. As in step 3, no processing of the key is performed at this stage; the WinControl merely indicates which types of key it wants to receive. The *CNKeyDown* handler inspects the *WM_GETDLGCODE* result field to see whether the keys required by the WinControl correspond to the key it has. If the key is required, the *CNKeyDown* handler prevents any other processing of the key at this stage by exiting with its message result field left at 0. Otherwise, proceed to step 5.

5) The WinControl's *CNKeyDown* event handler now sends its parent a *CM_DIALOGKEY* message by calling its parent's *Perform* method. At this stage Tab may cause the focus to be moved. If the parent is a *TForm*, its *CMDialogKey* message handler will respond to Tab or an arrow key by moving the focus, and indicate it has done so by setting the *CM_DIALOGKEY* message result field to 1. If the key is not Tab or an arrow key, the *TForm* will pass the message to the *CMDialogKey* handler, which it inherits from *TWinControl*. This will broadcast the message, which means it will send the message to each control on the form (by directly calling their *WndProc* methods), verifying each to see if the key has been processed. It is at this stage that a default button on the form will respond to Enter, which it will do by calling its own *Click* method. If any of the controls do process the key, they indicate this by setting the *CM_DIALOGKEY* result field to 1 and the broadcast stops. Finally, the *Perform* method of the WinControl's parent will return with a result of 1 if the key has been processed, and 0 otherwise. The WinControl's *CNKeyDown* event handler will then exit, also with a message of 1 if the key has been processed, and 0 otherwise.

Now, back in the Application object, sending the *CN_KEYDOWN* message returns a result of 1 if the key has been processed, and 0 otherwise. If the key has been processed, the Application object stops all further processing of the original *WM_KEYDOWN* message. If the key has not been processed, the Application object translates and dispatches the original *WM_KEYDOWN*. Now we can see why using *OnKeyPress* or *OnKeyDown* event handlers could never prevent a default button from responding to Enter. We have passed the point at which a default button responds to Enter, but we have not seen any *OnKeyDown* or *OnKeyPress* events along the way. These occur only later, and only if the Application object translates and dispatches the *WM_KEYDOWN* message — which it won't do if a default button has already processed Enter. We could follow the process further, but we have probably seen enough for now. Certainly we now know enough to see where to put our code to modify the application's response to Enter.

EMFForm

Looking back at the Delphi VCL's processing of a *CN_KEYDOWN* message, we see that steps 1 through 4 are a series of checks that give various components an opportunity to process the key, or at least to indicate that they wish to do so. It is only at step 5, if none of these checks has found a taker for the key, that the WinControl's parent form will respond to Tab by moving the focus. It is also in step 5 that an enabled default button will respond to Enter. Therefore, we should place our own modifications to this behavior as close to step 5 as possible. To act before then may stop some other component from working properly by denying it access to Enter. To act after step 5 will be difficult because we may have to make changes in many different places, and we will be acting after any enabled default button has already received Enter.

Because focus shifts in response to Tab are performed in the parent form's *CMDialogKey* message handler, this is the place to intervene. By overriding the *CMDialogKey* handler, we can substitute our own responses to Enter, and by calling the inherited handler we can also carry out standard processing in the majority of cases.

Listing Two (on page 21) shows the code for a descendant of *TForm* called *TEMFForm* (EMF stands for Enter Moves Focus). EMFForm differs from a standard *TForm* in only two respects. First, it supplies a new message handler for the *CM_DIALOGKEY* message. Second, it introduces two new Boolean properties: *StandardKeyHandling* and *DOSEmulation*. EMFForm's key-handling behavior can be modified by setting one of these two properties to *True*. The values of these properties are stored in two private fields, *FStandardKeyHandling* and *FDOSEmulation*. Both properties are public and have a default value of *False*.

Before we discuss EMFForm's key-handling behavior in detail, we should note that it also supplies a new constructor: its *Create* method. This constructor does nothing except call the constructor inherited from *TForm*. When an object is created in Delphi, its fields are all initialized to 0, which means *False* in the case of a field of type *Boolean*. Because the fields *FStandardKeyHandling* and *FDOSEmulation* will both be set to *False*, and because these are the default values we want, there is no need to take any special action to initialize them in the EMFForm constructor. In fact, EMFForm's new constructor is completely unnecessary as it stands; it does nothing the constructor inherited from *TForm* doesn't do. However, we'll be making some changes to EMFForm's constructor in the next section.

The real substance of EMFForm is in the new *CMDialogKey* message handler which it supplies. In the default configuration, with both *StandardKeyHandling* and *DOSEmulation* turned off, EMFForm's *CMDialogKey* handler first calls the handler inherited from *TForm*. If the key hasn't been processed after that, and if it is Enter, then the focus is moved (and the message result field is set to 1 to indicate that the key has been processed). In this way, Enter will generally move the focus, unless there is an enabled default button on the form. If there is an enabled default button, then it will process Enter during the inherited *CM_DIALOGKEY* handling, and no focus move will occur. If we use modal dialog boxes with the **OK** button initially disabled, and only enable it when the last edit box has been filled, we can produce behavior on data entry forms that is unfamiliar to DOS users. However, Windows users shouldn't notice any significant change; Enter will still "click" an enabled default button, so they can still use Tab to navigate, and Enter to close a modal dialog box. The disadvantage of this approach is that pressing Enter on an edit box doesn't always have the same effect; the effect depends on whether the **OK** button has been enabled.

If *DOSEmulation* is set to *True*, EMFForm responds to Enter by moving the focus and setting the *CM_DIALOGKEY* message result field to 1. The only circumstance under which EMFForm will not do this is if the control which currently has the focus is a button. Presumably if a user presses Enter on a button, they mean to click it. The inherited *CM_DIALOGKEY* message handler is called only if the key is not Enter, or if the control which has the focus is a button. Note that this means a default button on the form will not be clicked by pressing Enter unless it already has the focus. This might not be what a Windows user expects, but it is close to the behavior of many DOS applications.

If *StandardKeyHandling* is set to *True*, EMFForm's *CMDialogKey* message handler does nothing except call the handler inherited from *TForm*. This means it will perform standard Windows key processing in exactly the same way as *TForm*. Because *StandardKeyHandling* and *DOSEmulation* are mutually exclusive options, setting one property to *True* will automatically set the other to *False*. However, they can both be *False* (the default configuration).

Using EMFForm

If we place EMFForm in the Delphi repository and use it — or forms descended from it — throughout our application, then we'll have an application that has three modes of keyboard handling built into it. It would be nice if we could make *DOSEmulation* and *StandardKeyHandling* published properties, rather than public ones. Then we could set EMFForm's behavior at design time. Unfortunately, Delphi doesn't support the addition of published properties to a descendant of *TForm*, so we can't do that. However, there are other possibilities.

The simplest thing we can do is make a minor alteration to EMFForm's constructor. Figure 2 contains the code for an alternative constructor that makes *DOSEmulation* the default mode for the form. It would, of course, be possible to make *StandardKeyHandling* the default in the same way. We can now have a complex application, or suite of applications, which supports from a common code base multiple customers with differing key-handling requirements. The only differences between the applications the various customers have will be in one line of code in the constructor of our base form. This will certainly make for simpler maintenance than would the approach we discussed earlier, where we modified the form's behavior by means of *OnKeyPress* event handlers.

```
constructor TEMFForm.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  DOSEmulation := True;
end;
```

Figure 2: An alternative constructor for *TEMFForm*.

We are now finally able to tackle the multi-user scenario we described earlier. Suppose that each user is allowed to specify which of the key-handling modes they prefer. A user's preference may be looked up when they log in to our application, and stored in some manner that makes it accessible to EMFForm. This may simply be done by means of a globally accessible variable, or perhaps the application will create a User object which maintains other information about the user besides their key-handling preference. Possibly security information will also be maintained by the User object; not all users may be permitted access to all parts of the application. Exactly how the information is made available to EMFForm is immaterial for our purposes. As an example, Figure 3 contains the code for a constructor which verifies a user's key-handling preferences by referring to a User object, and configures EMFForm to suit the current user.

```
constructor TEMFForm.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  if User.WantsDOSEmulation then
    DOSEmulation := True
  else if User.WantsStandardKeyHandling then
    StandardKeyHandling := True;
end;
```

Figure 3: A multi-user version of the *TEMFForm* constructor.

Conclusion

After some searching we have found the ideal place in the VCL to modify the default behavior of Enter. In the process, we have seen how the VCL handles keyboard input — knowledge that may be useful in many other circumstances. If, for example, we want to write a component that needs to ensure that it receives Tab, arrow keys, or Esc, we need only look at steps 3 and 4 in the processing of the *CN_KEYDOWN* message handler. Or if we want to write a container component that needs to know whenever a component that it contains has been sent a particular key, the answer lies in step 2.

By placing our changes in the most appropriate place we gain two advantages. The first advantage is safety. We know we are intervening only *after* all the normal checks Delphi performs, so we won't prevent some other component from receiving Enter. The second advantage is ease of maintenance; we do not need to duplicate our efforts in multiple locations. In a few lines of code, we have implemented a form which supports three modes of keyboard handling, and can dynamically switch between these modes. This should be enough to please most users. ∆

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\97\MAR\DI9703AS.

Alex Sumner is an independent software developer specializing in Delphi development. He can be contacted via CompuServe at 100405,3112.

Begin Listing Two — The *TEMFForm* Class

```
interface

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

type
  TEMFForm = class(TForm)
  private
    FDOSEmulation: Boolean;
    FStandardKeyHandling: Boolean;
    procedure CMDialogKey(var Message: TCMDialogKey);
      message CM_DIALOGKEY;
  protected
    procedure SetDOSEmulation(DoDOSEmulation: Boolean);
      virtual;
    procedure SetStandardKeyHandling(
      DoStandardKeyHandling: Boolean); virtual;
  public
    constructor Create(AOwner: TComponent); override;
    property DOSEmulation: Boolean
      read FDOSEmulation write SetDOSEmulation;
    property StandardKeyHandling: Boolean
      read FStandardKeyHandling write
        SetStandardKeyHandling;
  end;

implementation

{$R *.DFM}

constructor TEMFForm.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
end;

procedure TEMFForm.CMDialogKey(var Message: TCMDialogKey);
var
  IsEnterKey: Boolean;
begin

  with Message do begin
    { Check if it's an Enter key, don't count Alt Enter
      or Ctrl Enter. }
    IsEnterKey := (Charcode              = VK_RETURN)and
                  (GetKeyState(VK_MENU)    >= 0)       and
                  (GetKeyState(VK_CONTROL) >= 0);

    if not (FStandardKeyHandling or FDOSEmulation) then
      begin
        { Default behavior: First see if inherited handler
          processes it. }
        inherited;
        { If it's an Enter key and it hasn't been used for
          anything else. }
        if IsEnterKey and (Result = 0) then
          begin
            { Move the focus to the next control, or
              previous control for Shift Enter. }
            SelectNext(ActiveControl,
                       GetKeyState(VK_SHIFT) >= 0, True);
            Result := 1;
          end;
      end
    else if FDOSEmulation then
      begin
        { DOS emulation: Enter will not function as a
          Windows user expects. }
        if IsEnterKey and
           not (ActiveControl is TButton) then
          begin
            { Move the focus to the next control, or
              previous control for Shift Enter. }
            SelectNext(ActiveControl,
                       GetKeyState(VK_SHIFT) >= 0, True);
            Result := 1;
          end;
        if Result = 0 then
          { If not an Enter key, or we're on a button,
            call the inherited handler. }
          inherited;
      end
    else
      { Standard Windows key handling; same as Tform. }
      inherited;

  end; { with message do }

end;

procedure TEMFForm.SetDOSEmulation(DoDOSEmulation: Boolean);
begin
  FDOSEmulation := DoDOSEmulation;
  { Can't have DOS Emulation and Standard Key Handling. }
  if FDOSEmulation then
    FStandardKeyHandling := False;
end;

procedure TEMFForm.SetStandardKeyHandling(
  DoStandardKeyHandling: Boolean);
begin
  FStandardKeyHandling := DoStandardKeyHandling;
  { Can't have Standard Key Handling and DOS Emulation. }
  if FStandardKeyHandling then
    FDOSEmulation := False;
end;

end.
```

End Listing Two

By Ken Jenks

Database-Driven Web Sites

Managing Web Sites with Delphi

Although it's fairly easy to create HTML files, it's hard to build a good Web site. It's even harder when the site grows and the number of files increases. Keeping your site up-to-date as you add and delete files can be a real chore. Luckily, Delphi can make this job easier.

A Web site consists of multiple files on a Web server, including HTML files, video clips, sound files, Java code, CGI programs, data files, and .GIF and .JPG images. There's an image file for every button, banner, horizontal rule, and picture on the site. To keep track of these files, you can build a database, then create an interface from the database to the Web server. Creating interfaces to databases is a job Delphi does particularly well.

An Online Art Gallery

As an example, we'll build an art gallery. Each *object d'art* will have a title, an artist, a small "thumbnail" .GIF image, a large full-color .JPG image, a description, and a price. In our Web art gallery, we'll provide four indices for locating the artwork: artist, title, thumbnail, and price. We'll use these to produce four "views" of our database. (Note: In this article, the term *view* doesn't refer to a SQL view, although the result is similar.)

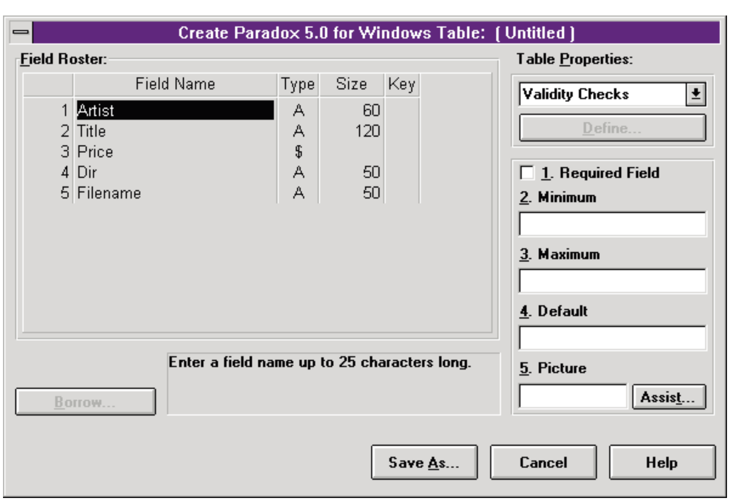


Figure 1: The table structure.

First, use the Database Desktop to create a simple Paradox table, artwork.db, with the five fields shown in Figure 1. Save it in a new directory, C:\dbweb.

Second, populate your database with information about the artwork in your gallery (see Figure 2).

Generating Views

From this database, we'll generate four views and save them as HTML files. Then we'll discuss how to send these HTML files from your PC to the Web server. To generate those four views, we'll generate four reports from our database, including the appropriate HTML tags, saving the reports on the PC.

In this example, two fields, Dir and Filename, refer to external files also stored on your PC. The .GIF thumbnail and the full-color, full-sized .JPG images have the same base filename (e.g. "night") but different extensions (.GIF and .JPG). Those image files are stored in the artist's directory on the PC (e.g. C:\dbweb\vangogh\). The description of the artwork is also kept in an external file with the same base name, but using an .HTM extension. The directory structure is shown in Figure 3.

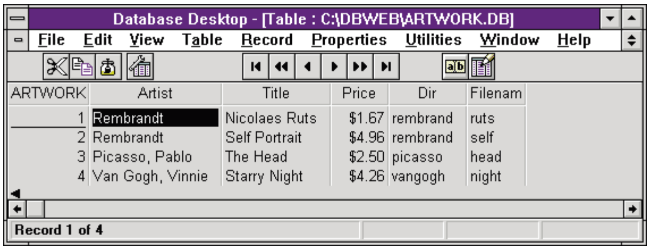


Figure 2: Table contents.

C:\dbweb\	Home directory for the project. Contains source code, .EXE, and database. Target directory for generating HTML files.
C:\dbweb\rembrand\ ruts.gif ruts.htm ruts.jpg self.gif self.htm self.jpg	Contains six files for artist "Rembrandt": Thumbnail image of ruts.jpg Description of "Nicolaes Ruts" Full-size, full-color image Thumbnail image of self.jpg Description of "Self Portrait" Full-size, full-color image
C:\dbweb\picasso\ head.gif head.htm head.jpg	Contains three files for artist "Picasso, Pablo": Thumbnail image of head.jpg Description of "The Head" Full-size, full-color image
C:\dbweb\vangogh\ night.gif night.htm night.jpg	Contains three files for artist "Van Gogh, Vinnie": Thumbnail image of night.jpg Description of "Starry Night" Full-size, full-color image

Figure 3: The directory structure.

Now we'll build the Delphi application to automatically generate Web pages from the table, images, and description files.

Building the Application

Use Delphi to create a new SDI application with one form. Save the new project as C:\dbweb\dbweb.dpr and save the Unit as C:\dbweb\dbwebu.pas. Onto the form, drop a Table component (named *tblArtwork* in our sample), a DataSource component (named *dsArtwork*), a Query (named *qArtwork*), a DBGrid, and a DBNavigator. Link the DataSource to the Table, link the DBGrid and the DBNavigator to the DataSource, and link all of these with your database, setting the Table properties' *DatabaseName* to `c:\dbweb`, *TableName* to `ARTWORK.DB`, and *Active* to *True*. Your data should appear in the DBGrid (see Figure 4).

Now add a button to the form. Change the *Name* of the button to `btnGenerate` and the *Caption* to `&Generate`. Double-click on the **Generate** button and add the code in Listing Three (beginning on page 24) to generate Web pages. The code will create five HTML files, one for each of the four views, plus an index file. We'll define some local procedures to generate HTML files and call the procedures from *TDBWebForm.btnGenerateClick*.

Notice how we create text files to contain our HTML code, then use *WriteLn* to generate our reports. There are many other ways of generating reports from databases in Delphi, but this is the simplest for short reports.

You can run this program to generate Web pages from a database. Use your Web browser to open the local file,

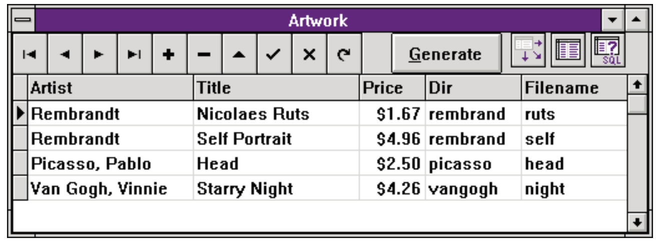


Figure 4: The artwork data in the DBGrid.

C:\dbweb\index.htm, then look at the other Web pages — they're simply pre-generated, static views of the database.

Next, you must move the pages to the Web server. Depending on your Web server, this could be as simple as writing the HTML files to the Web server's document directory, or it could require you to FTP the files to the Web server. (In this case, you could use an FTP component in Delphi to automatically transfer the files to the Web server.) Note that the .GIF and .JPG images are stored in different directories than the HTML files. When you move the HTML files from your PC to the Web server, you must create these directories and copy the images as well.

A Corporate Example

Mind's Eye Fiction sells short stories on the Web. At Mind's Eye Fiction, Delphi is used to generate static indices (similar to those previously described) from databases of stories, allowing users to view the story database by author, title, genre, and illustration. Author information such as name, address, phone number, social security number, e-mail, Web site, etc. is kept in a separate database, linked to each story by author name. Title information includes number of words in the story, and price. Delphi generates the Web pages and canned e-mail notes to send to authors at various stages of review and publication. Delphi also sets up the requests to the electronic commerce systems used at Mind's Eye Fiction.

Conclusion

This little application is a starting point that you can build on, using your own database. As previously noted, it can be augmented with an FTP component to automatically transfer your files and images to your Web server or delete them from the server when they are no longer needed. You can also loop through a table to generate a list of items in radio buttons for HTML forms. After you start using Delphi's database power to build Web pages, your Web site management will be much simpler. ∆

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\97\MAR\DI9703KJ.

Ken Jenks is owner and Editor-in-Chief of Mind's Eye Fiction (http://tale.com), a Delphi-driven, Web-based publishing company specializing in short stories. He has used computers for 21 years, the Internet for 10 years, and Borland Pascal since version 2.0 for CP/M. He can be reached by e-mail at MindsEye@tale.com.

Begin Listing Three — dbwebu.pas

```
procedure TDBWebForm.btnGenerateClick(Sender: TObject);
  { To assist in creating Web pages from text files, we'll
    define a local procedure to copy a file from disk to an
    open text file. Notice this procedure is not a method of
    TDBWebForm, but local to the btnGenerateClick method. }
  procedure CopyAFile(var F : TextFile; FN : string);
  var
    Temp  : TextFile;
    Ch    : Char;
  begin
    try
      AssignFile(Temp,FN);
      Reset(Temp);
      while not System.Eof(Temp) do begin
        Read(Temp, Ch);
        Write(F, Ch);
      end;
      CloseFile(Temp);
    except
      MessageDlg(
        'Problem reading file. Did you UNZIP with -d?',
        mtInformation,[mbOk], 0);
    end;
  end;

  { Here's another local procedure to generate a standard
    header for each HTML file. If you ever want to change
    the headers for all pages on your Web site, change it
    here, then regenerate the pages. }
  procedure BannerToFile(var F : TextFile);
  begin
    WriteLn(F,'<HTML>');
    WriteLn(F,'<HEAD>');

    WriteLn(F,'<TITLE>Art Gallery</TITLE>');
    WriteLn(F,'</HEAD>');
    { You can set up background colors
      and images in the <BODY> tag. }
    WriteLn(F,'<BODY BGCOLOR="#FFFFFF">');
    WriteLn(F,
      '<CENTER><IMG SRC="banner.gif" HEIGHT=60 WIDTH=400 ',
      'ALT="Art Gallery"></CENTER><P>');
  end;

  { This procedure will generate index.htm. Since index.htm
    doesn't change if you change the database, you don't
    really need to use a program to generate it, but this
    shows the basic technique of using WriteLn to send text
    to the HTML file. }
  procedure GenerateIndex;
  var
    F: TextFile;
  begin
    try
      AssignFile(F,'index.htm');
      Rewrite(F);
      BannerToFile(F);
      WriteLn(F,'<H1>Art Gallery Index</H1>');
      WriteLn(F,'You can browse the gallery in four ways:');
      WriteLn(F,'<UL>');
      WriteLn(F,'<LI><A HREF="artists.htm">by Artist</A>');
      WriteLn(F,'<LI><A HREF="title.htm">by Title</A>');
      WriteLn(F,'<LI><A HREF="price.htm">by Price</A>');
      WriteLn(F,'<LI><A HREF="gallery.htm">MicroGallery</A>');
      WriteLn(F,'</UL>');
      WriteLn(F,'</BODY>');
      WriteLn(F,'</HTML>');
    finally
      CloseFile(F);
    end;
  end;

  { This procedure will generate artists.htm from the
    database, sorted by artist. }
procedure GenerateArtists;
var
  F: TextFile;
begin
  try
    { To sort by artist, we'll execute a query against
      the Artwork table and "order by" the Artist field. }
    qArtwork.Close;
    qArtwork.SQL.Clear;
    qArtwork.SQL.Add('SELECT * FROM Artwork ORDER BY ' +
                     'Artwork.Artist');
    qArtwork.Open;
    { Now that the query is complete, we'll step
      through each record of the results, starting
      at the first. }
    qArtwork.First;
    AssignFile(F,'artists.htm');
    Rewrite(F);
    BannerToFile(F);
    WriteLn(F,'<H1>Art Gallery by Artist</H1>');
    { Loop through query results. }
    while (not qArtwork.EOF) do begin
      { There are several ways to access a particular
        field in Delphi. FieldByName works on TQuery
        objects as well as on TTable objects. }
      WriteLn(F,'<B>',
             qArtwork.FieldByName('Artist').AsString,
             '<B><BR>');
      { The non-breakable space ( ) allows us to
        indent these lines in the HTML file. }
      WriteLn(F,'   ',
        qArtwork.FieldByName('Title').AsString,'<BR>');
      WriteLn(F,'   $',
        qArtwork.FieldByName('Price').AsString,'<BR>');
      { Note: next statement is Write instead of WriteLn. }
      Write(F,'   ');

      { Copy description file from directory and
        filename specified in table. We could have
        done this using a Memo field, but this
        demonstrates how to incorporate external
        files in your Web pages. }
      CopyAFile(F,qArtwork.FieldByName('Dir').AsString +
               '\' +
               qArtwork.FieldByName('Filename').AsString
               + '.htm');
      { At the end of each record, we'll place a
        horizontal ruler <HR> in the HTML code. }
      WriteLn(F,'<HR>');
      WriteLn(F);
      { When done with this record, move to the next. }
      qArtwork.Next;
    end; { While }

    { At end of HTML document, close the <BODY> and
      <HTML> tags. This isn't mandatory, but is good
      form. It's also a good idea to put your e-mail
      address on each Web page; this is a good place
      for that. }
    WriteLn(F,'</BODY>');
    WriteLn(F,'</HTML>');
  finally
    CloseFile(F);
  end;
end;


{ This procedure generates the price.htm file using
  the same techniques as above. }
procedure GeneratePrices;
var
  F: TextFile;
begin
  try
    qArtwork.Close;
    qArtwork.SQL.Clear;
```

```
    { Note that we're sorting by Price this time. }
    qArtwork.SQL.Add( 'SELECT * FROM Artwork ORDER BY ' +
                      'Artwork.Price');
    qArtwork.Open;
    qArtwork.First;
    AssignFile(F,'price.htm');
    Rewrite(F);
    BannerToFile(F);
    WriteLn(F,'<H1>Art Gallery by Price</H1>');

    while (not qArtwork.EOF) do begin
      WriteLn(F,
        '<B>$',qArtwork.FieldByName('Price').AsString,
        '<B><BR>');
      WriteLn(F,'   ',
        qArtwork.FieldByName('Title').AsString,'<BR>');
      WriteLn(F,'   by ',
        qArtwork.FieldByName('Artist').AsString,'<BR>');
      Write(F,'   ');
      CopyAFile(F,qArtwork.FieldByName('Dir').AsString +
               '\' +
               qArtwork.FieldByName('Filename').AsString
               + '.htm');
      WriteLn(F,'<HR>');
      WriteLn(F);
      qArtwork.Next;
    end;
    WriteLn(F,'</BODY>');
    WriteLn(F,'</HTML>');
  finally
    CloseFile(F);
  end;
end;

{ Generate both titles.htm and gallery.htm. }
procedure GenerateTitlesAndThumbs;
var
  F,G: TextFile;
begin
  try
    qArtwork.Close;
    qArtwork.SQL.Clear;
    { Since titles.htm and gallery.htm are both sorted by
      title, we can use the same procedure to generate
      them. This saves us from doing another query and
      looping through another time--potentially a large
      savings for a large database. }
    qArtwork.SQL.Add( 'SELECT * FROM Artwork ORDER BY ' +
                      'Artwork.Title');
    qArtwork.Open;
    qArtwork.First;
    AssignFile(F,'title.htm');
    Rewrite(F);
    BannerToFile(F);
    WriteLn(F,'<H1>Art Gallery by Title</H1>');
    AssignFile(G,'gallery.htm');
    Rewrite(G);
    BannerToFile(G);
    WriteLn(G,'<H1>Micro Gallery</H1>');


    while (not qArtwork.EOF) do begin
      WriteLn(F,'<B>',
              qArtwork.FieldByName('Title').AsString,
              '<B><BR>');
      WriteLn(F,'   by ',
              qArtwork.FieldByName('Artist').AsString,
              '<BR>');
      WriteLn(F,'   $',
              qArtwork.FieldByName('Price').AsString,
              '<BR>');
      Write(F,'   ');
      CopyAFile(F,qArtwork.FieldByName('Dir').AsString+'\'
               +qArtwork.FieldByName('Filename').AsString
               +'.htm');
      WriteLn(F,'<HR>');
      WriteLn(F);
      WriteLn(G,'<A HREF="',
              qArtwork.FieldByName('Dir').AsString,'/',
              qArtwork.FieldByName('Filename').AsString,
              '.jpg">','<IMG SRC="',
              qArtwork.FieldByName('Dir').AsString,'\',
              qArtwork.FieldByName('Filename').AsString,
              '.gif" ALIGN=LEFT></A>');
      WriteLn(G,'   ',
              qArtwork.FieldByName('Title').AsString,
              '<BR>');
      WriteLn(G,'   by ',
              qArtwork.FieldByName('Artist').AsString,
              '<BR>');
      WriteLn(G,'   $',
              qArtwork.FieldByName('Price').AsString,
              '<BR>');
      Write(G,'   ');
      CopyAFile(G,qArtwork.FieldByName('Dir').AsString+'\'
               +qArtwork.FieldByName('Filename').AsString
               +'.htm');
      WriteLn(G,'<HR>');
      WriteLn(G);
      qArtwork.Next;
    end;
    WriteLn(F,'</BODY>');
    WriteLn(F,'</HTML>');
  finally
    CloseFile(F);
    CloseFile(G);
  end;
end;

begin { The btnGenerateClick method. }
  { Disable button and change cursor to hourglass while
    pages are being generated. }
  btnGenerate.Enabled := False;
  Screen.Cursor       := crHourglass;
  try
    GenerateIndex;
    GenerateArtists;
    GeneratePrices;
    GenerateTitlesAndThumbs;
    Screen.Cursor := crDefault;
    MessageDlg('HTML files created!',
               mtInformation,[mbOk],0);
  finally
    Screen.Cursor       := crDefault;
                                       { Change cursor back. }
    btnGenerate.Enabled := True;       { Re-enable button.   }
  end;
end; // btnGenerateClick method
```

End Listing Three

By Cary Jensen, Ph.D.

The TApplication Class

Bonus Functionality Right under Your Nose

In last month's DBNavigator, you learned of three special instance variables that Delphi automatically declares for your applications. These variables, named Application, Screen, and Session, correspond to instances of the *TApplication*, *TScreen*, and *TSession* classes, respectively. To accompany last month's discussion of the Screen instance variable, this month's installment takes a closer look at the Application variable by considering a number of interesting techniques that employ the *TApplication* class methods and properties.

Unlike the Screen instance variable, whose existence is not obvious, the presence of the Application variable is hard to ignore; all applications contain at least two references to it in the project source (three in Delphi 2). Specifically, every form-based application calls, at a minimum, two *TApplication* methods, *CreateForm* and *Run* (Delphi 2 applications may also call the *TApplication* method *Initialize*). *CreateForm* is used to create the main form of the application (in addition to all other auto-created forms), *Run* executes the application, and *Initialize* calls an *InitProc*, if one is defined.

Properties	Methods	Events
Active	*Create*	*OnActivate*
ComponentCount	*CreateForm*	*OnDeactivate*
ComponentIndex	*Destroy*	*OnException*
Components	*FindComponent*	*OnHelp*
ExeName	*Free*	*OnHint*
Handle	*HandleException*	*OnIdle*
HelpFile	*HandleMessage* *	*OnMessage*
Hint	*HelpCommand*	
HintColor	*HelpContext*	
HintHidePause *	*HelpJump*	
HintPause	*InsertComponent*	
HintShortPause *	*MessageBox*	
Icon	*Minimize*	
MainForm	*NormalizeTopMost*	
Name	*ProcessMessages*	
Owner	*RemoveComponent*	
ShowHint	*Restore*	
Tag	*RestoreTopMost*	
Terminated	*Run*	
Title	*ShowException*	
UpdateFormatSettings *	*Terminate*	

Figure 1: Properties, methods, and events of *TApplication* (* Delphi 2 only).

The *TApplication* class, which is declared in the Forms unit, declares the properties, methods, and event properties shown in Figure 1. The use of the *TApplication* class is demonstrated in this article through a series of example projects. These projects show you how you can identify the directory in which your application is running, how and why to process Windows messages within your application, how to respond to your application's loss of focus, and how to replace your application's default exception handler.

Pinpointing Your Application

The *ExeName* property of the *TApplication* class returns a string that identifies the fully qualified path of your application. Using the *ExtractFilePath* function, defined in the SysUtils unit, you can identify the directory in which your application is stored. The EXENAME.DPR project demonstrates the use of the *ExeName* property. The form shown in

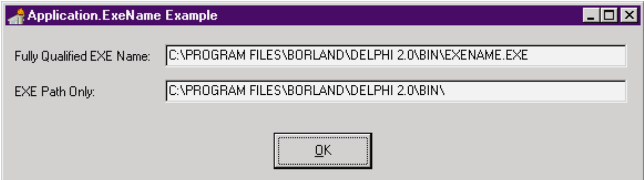


Figure 2: The EXENAME.DPR project.

Figure 2 contains two Labels and two Edits. The following code is associated with this form's *OnCreate* event handler:

```
Edit1.Text := Application.ExeName;
Edit2.Text := ExtractFilePath(Application.ExeName);
```

As a result, *Edit1* displays the fully qualified name of the running application, while *Edit2* displays the path-only part.

Using the *ExeName* property is particularly useful in local database applications where you store your data either in the same directory as your executable, or in a subdirectory of that directory. For example, the following code will point the Table component named *Table1* to a table named APPDATA.DB, located in a "Data" subdirectory that resides immediately under your application's directory; it will then open that table:

```
Table1.DatabaseName :=
  ExtractFilePath(Application.ExeName) + 'Data';
Table1.TableName := 'APPDATA.DB';
Table1.Open;
```

Using *ProcessMessages*

You use the *TApplication* method *ProcessMessages* to temporarily suspend the execution of your application so that Windows has an opportunity to process messages in its message queue. This statement can play an important role under Windows 3.1x, which is a cooperative multitasking environment. Here, calling *Application.ProcessMessages* permits the processing of messages that apply to the current application, as well as those intended for other running applications. This use of *Application.ProcessMessages* is often referred to as a *yield*.

Since Windows 95 and Windows NT are preemptive operating systems (where the OS ensures other applications receive CPU time without your application specifically yielding control), you might think this method is obsolete. However, *Application.ProcessMessages* remains very useful.

Contrary to the way *Application.ProcessMessages* works under Windows 3.1x, the use of this method under Win32 does not affect the execution of other applications. However, it still serves to instruct Windows to respond to all messages in the Windows message queue for the current application. Consequently, this method is invaluable when your current application is performing a processor-intensive operation, but needs to respond to Windows messages such as those that repaint your application's forms.

The following example demonstrates an appropriate use of *Application.ProcessMessages* in a Delphi 2 application.

Imagine that your client/server application displays a splash screen during startup. If you use the *TForm* method *Show* to display your splash screen immediately before attempting to open the Main form of the application, Windows won't complete the painting of your splash screen before beginning the server login process. The result is that your splash screen won't get painted properly. To remedy this problem, call *Application.ProcessMessages* immediately after calling the splash screen's *Show* method, but before calling the *Application.CreateForm* method for the application's main form.

This technique is demonstrated in the project named SPLASH:

```
program Process;

uses
  Forms,
  Process1 in 'PROCESS1.PAS' {Form1},
  Process2 in 'PROCESS2.PAS' {Splash};

{$R *.RES}

begin
  Splash := TSplash.Create(Application);
  Splash.Show;
  Application.ProcessMessages;
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.
```

The preceding code serves to display the splash screen as a non-modal form. The following code, which is attached to the main form's *OnActivate* event handler, closes, then releases (destroys), the splash screen:

```
procedure TForm1.FormActivate(Sender: TObject);
begin
  Splash.Close;
  Splash.Release;
end;
```

Using the *OnDeactivate* Event Property

Similar to the *TScreen* class, *TApplication* publishes several event properties that permit you to define code that's automatically executed in response to events. One of the lesser known, but useful event properties for an Application component, is the *OnDeactivate* event property. An event handler assigned to this property is executed when your application loses focus — that is, when the user makes another application active.

You can do two particularly interesting things from within an *OnDeactivate* event handler. The first is to minimize your application. This can be achieved simply by calling the Application component's *Minimize* method from within the *OnDeactivate* event handler:

```
procedure TForm1.Deactivate(Sender: TObject);
begin
  Application.Minimize;
end;
```

This technique is particularly effective in Windows 95 and Windows NT, using the compatibility shell. When the user moves to another application, your Delphi application minimizes, becoming a button on the Taskbar.

The second useful thing to do with an application is to terminate it. Obviously, this is only appropriate for a particular class of application — those that provide system information and are of very short duration. A user who finishes viewing such information might appreciate the ability to terminate the application by switching to another. This can be accomplished using the *TApplication* method *Terminate*. For example, you can achieve this effect by adding the following line of code to your Application's *OnDeactivate* event handler:

```
Application.Terminate;
```

Note: While you might be inclined to call the procedure *Halt* to exit an application, this is not the recommended technique. Using *Application.Terminate* is a cleaner, gentler way to terminate an application. However, *Terminate* is not always immediate, because it permits the appropriate event handlers to execute normally. *Halt*, on the other hand, is the procedure you should call when immediate termination is required. Use *Terminate* in most cases; but if you need to abnormally end an application and return an exit code (an error code that sets the DOS errorcode variable), use *Halt*.

While the preceding two examples require very little code, they do require some preparation. Because you can't access a *TApplication* object at design time, you must create the header and implementation for the *OnDeactivate* event handler manually. Specifically, you must add a procedure to the **published** or **public** section of your form class:

```
procedure Deactivate(Sender: TObject);
```

Also, you must implement this procedure in the **implementation** section of your form's unit. The following is an example of an *OnDeactivate* event handler that will terminate your application when the user switches to another application. This event handler is associated with the TERMINAT.DPR project:

```
procedure TForm1.Deactivate(Sender: TObject);
begin
  Application.Minimize;
end;
```

Finally, you must assign your defined event handler to the *Application.OnDeactivate* property. Typically, this is done within your main form's *OnCreate* event handler:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Application.OnDeactivate := Deactivate;
end;
```

For an example of an application that automatically minimizes when the user switches to another application, download MINIMIZE.DPR (see end of article for details).

Creating an Application-Level Exception Handler

Under normal circumstances, when an exception is raised either by a run-time error encountered by Delphi, or through your use of the reserved word **raise** within your code, your application branches to the nearest exception-handling routine. If you have not declared an explicit exception-handling routine (using **try-except**), the exception is handled by the application-level exception handler. This default exception handler acts to display the exception message to the user, then destroys the exception object.

Using the Application object, you can replace the default exception handler with your own by assigning an event handler to the *TApplication OnException* event property, which is a *TExceptionEvent* method pointer. This event handler has the following syntax:

```
TExceptionEvent =
  procedure (Sender: TObject; E: Exception) of object;
```

Using this event property, you can define exactly what happens when an exception occurs. For example, you can choose to display a message in a status bar, as opposed to displaying a modal dialog box, or you can write information concerning the exception to an error log (a database or file that stores details concerning encountered errors).

Just as with the *OnDeactivate* event handler, if you want to create an *OnException* event handler, you need to declare it in your form's **type** declaration, as well as implement it in the unit **implementation** section. In addition, you must execute code to assign this event handler to the Application object's *OnException* property. For example, to create an alternative event handler for the *TForm1* class, you need to add a *TExceptionEvent*-type procedure to the form's **type** declaration, similar to the following:

```
procedure ExceptHandler(Sender: TObject; E: Exception);
```

To assign this exception handler to *Application.OnException*, you might use code similar to the following in your main form's *OnCreate* event handler:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Application.OnException := ExceptHandler;
end;
```

The following is an example of a replacement application-level exception handler. This exception handler tests for custom-declared exceptions. If the exception is of the type *ECustomWarningException* (a custom superclass declared to organize non-critical exceptions), a sound is generated and the exception message is displayed in a status bar (*Panel2* in this example). Alternatively, if the exception is of the type *ECustomCriticalException* (a superclass to all custom critical exceptions), the standard exception dialog box is displayed using the *TApplication ShowException* method. In this code, all Delphi-generated exceptions are also displayed using *ShowException*:

```
procedure TForm1.AppException(Sender: TObject;E: Exception);
begin
  if E is ECustomException then
    if E is ECustomWarningException then
      { Custom warning exception }
      begin
        Panel2.Caption := E.Message;
        MessageBeep(MB_ICONEXCLAMATION);
      end
    else
      { Custom critical exception }
      Application.ShowException(E)
  else
    { Delphi exception }
    ShowMessage(E.Message);
end;
```

As you might have already realized, this type of code is meaningful only to the extent that you pre-declared custom exceptions — and consistently used only those exceptions you declared — when you needed to explicitly raise an exception from within your code. The following is an example of a **type** declaration that demonstrates how the custom exceptions used in the preceding code might look:

```
type
  ECustomException = class(Exception);
  ECustomCriticalException = class(ECustomException);
  EMissingLocalTable = class(ECustomCriticalException);
  { Additional custom critical exceptions declared here... }
  ECustomWarningException = class(ECustomException);
  EFieldException = class(ECustomWarningException);
  ERecordException = class(ECustomWarningException);
  { Additional custom warning exceptions declared here... }
```

It's important to note that even when you completely replace your application's *OnException* event handler, the exception object is automatically destroyed by the event. Also, whenever you replace the default exception handler by creating an *OnException* event handler, you're responsible for writing code to display the exception to the user. In other words, the exception dialog box that's normally displayed by the default application-level exception handler is not automatically displayed from within an *OnException* event handler.

Conclusion

The Application object, like other run-time-only components, offers many powerful capabilities that extend and enhance your Delphi applications. While this component can be controlled only at run time, as opposed to the design-time manipulation afforded by those components that appear on the Component palette, the extra work of run-time manipulation is almost always worth the advantages provided by *TApplication*. ∆

The demonstration files referenced in this article are available on the Delphi Informant Works CD located in INFORM\97\MAR\DI9703CJ.

Cary Jensen is President of Jensen Data Systems, Inc., a Houston-based database development company. He is author of more than a dozen books, including *Delphi In Depth* [Osborne/McGraw-Hill, 1996]. He is also Contributing Editor to *Delphi Informant*. Cary is a member of the Delphi Advisory Board for the 1997 Borland Developers Conference. You can reach Jensen Data Systems at (713) 359-3311, or via CompuServe at 76307,1533.

By Keith Wood

Random Thoughts

A Look at Generating Random Numbers with Delphi

Random numbers are used in many different applications; they add an element of unpredictability to games, ensuring that something different occurs each time. They can also be used in approximations and simulations when trying to model some process.

Although computers cannot generate truly random numbers, they can produce seemingly random sequences that, nevertheless, follow a pattern. From these values, a uniform probability distribution can be obtained, and from that, any given distribution can be simulated.

Random-Number Generators

Because a computer is deterministic (i.e. you can determine its next state given its current state and the program that's running), it cannot produce a true random sequence, which is, by definition, non-deterministic. Even so, computers can generate number sequences that appear to be random — so-called pseudo-random sequences. A typical way of doing this is to generate numbers according to the formula:

$$r_{i+1} = (r_i \times a) \bmod b$$

```
const
  iRandMultiplier: LongInt = 31415927;
  iRandModulus: LongInt = 27182819;
var
  iRandSeed: LongInt;

{ Return a random value in the range 0 <= x < 1.
  Home-grown version using formula x(i+1) =
  x(i) * a mod b. }
function MyRandom: Real; far;
begin
  if iRandSeed = 0 then
    iRandSeed := 1;
  iRandSeed  := (iRandSeed*iRandMultiplier) mod iRandModulus;
  Result     := iRandSeed / iRandModulus;
end;
```

Figure 1: A home-grown random-number generator.

where the next number is the modulus (remainder after dividing) of the product of the current value and a constant *a*, with a second constant *b*. By choosing appropriate values for the constants, a reasonable random sequence occurs. Ideally, the constants should have no common factors. The values produced are in the range 0 to b-1 as a result of the modulus operation, and are converted to a real value in the range 0 to 1 (but not including 1) by dividing by the second constant.

A random-number generator that follows this model is implemented in the demonstration program shown in Figure 1. It uses the values 31415927 and 27182819 for the constants *a* and *b*. You probably noticed these are derived from π (pi) and *e* (natural logarithms). A special case occurs when the current random value is zero, shifting the value to 1. Otherwise the sequence isn't random at all!

Delphi also provides a built-in random-number generator, *Random*. It can be used as a function in one of two ways: On its own, it returns a real value in the range 0 to 1 (excluding 1); and when passed an integer as a parameter, it returns an integer value in the range 0 to one less than the parameter.

The demonstration program allows us to compare the randomness of these two generators by plotting their distribution across a number of values (see Figure 2). For a truly

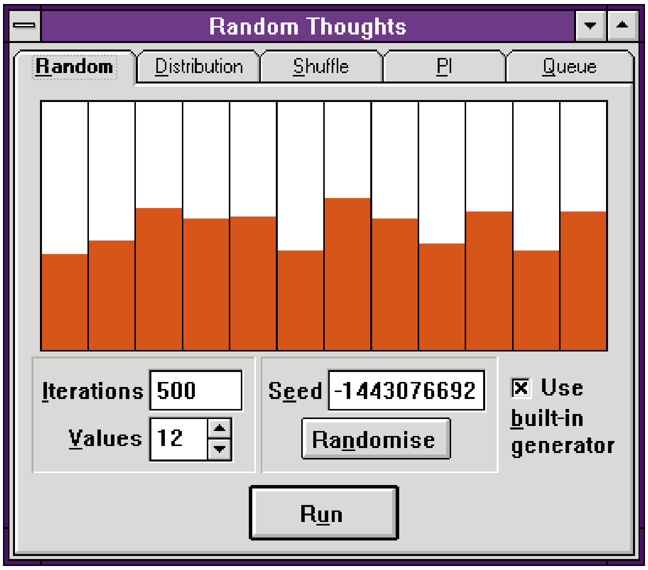


Figure 2: Testing the randomness of a random-number generator.

random generator and a sufficiently large sample, all values should be seen an equal number of times.

The checkbox on the right in Figure 2 determines which of the generators is being used. It sets a variable, *fnRandom*, to point to one of the two functions. Because the built-in generator is a special case (allowing for different modes of calling), a proxy function has been defined, which can be passed as a parameter, *BuiltInRandom*. This function simply returns the value from the built-in version. The appropriate generator can then be referenced throughout the rest of the code via this variable, without knowing which generator it is. This can be extended to use any other random-number generator, provided it returns a real value in the range 0 to 1 (excluding 1). Note that the function must be declared with the **far** directive to allow it to be passed as a parameter.

The fields on the left in Figure 2 indicate the number of sample values to generate, and the number of values to spread the distribution across. A larger number of iterations should smooth out the differences between the columns.

To translate a random real value less than 1 to an integer in a given range, the *RandomInt* function (from the RandUtil unit) is used. It takes two parameters, the lower and upper limits of the range. The computation then returns the corresponding integer value:

```
{ Return a random integer in the range iLower to iUpper,
  inclusive, with equal probability for each value. }
function RandomInt(iLower, iUpper: LongInt): LongInt;
begin
  if iUpper < iLower then
    raise ERandom.Create(
      'Range for random integer is invalid');
  Result := Trunc(fnRandom * (iUpper-iLower+1)) + iLower;
end;
```

The center section of Figure 2 shows the current value of the seed used to start the random sequence. Pressing the **Randomise** button causes this value to be selected at random. The home-grown version takes the last seven digits of the current time in reverse order. The built-in generator has its own *Randomize* procedure to achieve a similar effect. A particular "random" sequence is repeatable by simply starting at the same seed. This can be useful during testing, when it can provide a known "random" sequence.

After running some comparisons between the two generators, the built-in one seems to give a better spread of values, whereas the home-grown version appears to be slightly biased toward the bottom end of the range. This bias becomes more obvious when the number of values being distributed across is reduced. Because of this apparent bias, the remainder of the demonstration program uses the built-in generator exclusively.

Distributions

The numbers produced by the built-in generator are real values in the range 0 to 1 (excluding 1), with each value having an equal chance of appearing. This is a *uniform probability distribution*. Many other types of probability distributions can be used to alter the likelihood of any particular value being selected. Each can be described by a distribution function that maps from a uniformly distributed value in the range 0 to 1, to the desired values. The *RandomDistribution* function in the RandUtil unit allows for this mapping to take place, and returns a value from the specified distribution.

It accepts an array of points that closely approximate the required distribution. These points are expected to be monotonically increasing, i.e. the values at each successive point are at least as large as the preceding ones. When plotted on a graph, these functions never curve downward.

One component of each of these points is the cumulative probability of its happening, i.e. the probability that this value, or one less than it, is chosen. As such, the probabilities of the first and last elements in the array must be equal to 0 and 1, respectively. The other component is the corresponding value that has this cumulative probability.

To map from one to the other, the function generates a uniformly distributed value in the range 0 to 1. It then finds the segment of the distribution function that contains this probability, and interpolates between the values of the end points in a linear fashion (see Figure 3).

Of course, this doesn't give a completely accurate result, because we are approximating the function by a series of straight line segments. However, this can be improved by increasing the numbers of these segments, to more closely follow the curve.

One of the most common distributions is *normal distribution*, which has a characteristic bell-shaped curve. It is described by the equation:

```
{ Select a real value from a specified
  probability distribution. }
function RandomDistribution(
  recDist: array of TProbDistPoint): Real;
var
  i: Integer;
  rRandom: Real;
  bFound: Boolean;
begin
  if (recDist[Low(recDist)].Prob <> 0.0) or
     (recDist[High(recDist)].Prob <> 1.0) then
    raise ERandom.Create(
      'Limits of distribution function must be 0 and 1');

  rRandom := fnRandom;
  bFound := False;
  for i := Low(recDist) + 1 to High(recDist) do begin
    if (recDist[i].Prob  < recDist[i - 1].Prob) or
       (recDist[i].Value < recDist[i - 1].Value) then
      raise ERandom.Create(
        'Distribution function must increase monotonically');
    if not bFound and (rRandom >= recDist[i - 1].Prob) and
                      (rRandom < recDist[i].Prob) then
      begin  { Interpolate within this range. }
        bFound := True;
        Result := (recDist[i].Value - recDist[i-1].Value) *
                  (rRandom - recDist[i - 1].Prob) /
                  (recDist[i].Prob - recDist[i - 1].Prob) +
                  recDist[i - 1].Value;
      end;
  end;
end;
```

Figure 3: Returning a random value in the given distribution.

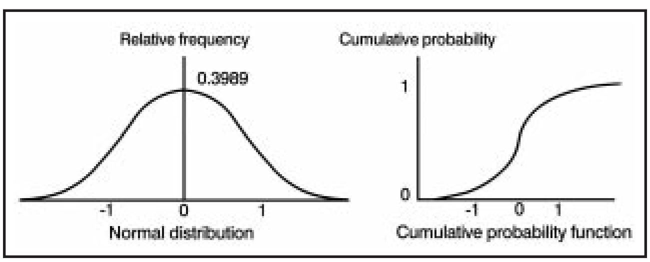


Figure 4: The normal distribution and its corresponding probability function.

$$f(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}}$$

for a mean of 0 and a standard deviation of 1. The mean is the average value, while the standard deviation measures the spread of the values.

Figure 4 graphs the distribution, based on the previous formula and its corresponding probability distribution function.

The Distribution tab in the demonstration program allows us to define a probability distribution, then generate sample values from this distribution for comparison (see Figure 5). The choices of function include the uniform, normal, and Poisson distributions, or we can define our own. By dragging the red boxes up or down, we can alter the underlying distribution.

Note that the function always retains its monotonic character by forcing following points to be at least as large as the one being dragged, and preceding points to be at least as small. As

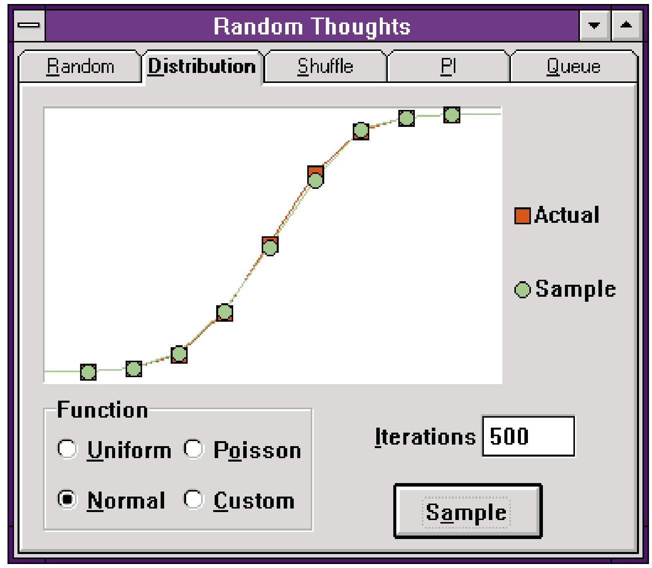


Figure 5: Generating random values to a given distribution.

before, with a larger number of samples, the random values should more closely follow the specified distribution.

The RandUtil unit also includes the *NormalDistribution* function, which takes two parameters: the mean and the standard deviation of the distribution. It returns a random real value from the specified distribution. Internally, it uses the *RandomDistribution* function, passing it a pre-defined distribution function. This value is then adjusted for the mean and standard deviation given:

```
{ Select a random value from a normal distribution. }
function NormalDistribution(rMean, rStdDev: Real): Real;
begin
  if rStdDev <= 0.0 then
    raise ERandom.Create(
      'Standard deviation must be greater than 0');
  Result := RandomDistribution(rNormalDist) * rStdDev+rMean;
end;
```

Shuffling

A common application of random numbers is in games where a random ordering of certain objects is required, such as games that use cards or Mah Jong tiles.

A first attempt at shuffling a deck of cards might involve stepping through each position in each hand, and picking a card at random from the deck. We would then need to ensure that the card chosen had not already been used before adding it to the current hand. Unfortunately, this strategy is not guaranteed to terminate. As we get closer to the end of the hands, the chances of picking a card that hasn't been used lessens, and the process takes longer and longer.

We want a way of obtaining the random ordering in a fixed amount of time. This can be done by imitating the process of shuffling a deck.

First, we order the cards in an array (the deck). We then step through each position in the array, and select another position at random in the remainder of the array. We can

```
{ Deal the cards randomly. }
procedure TfmRandom.btnShuffleClick(Sender: TObject);
const
  sValues: String[13] = 'A23456789TJQK';
  sSuits: array [1..4] of string[8] =
           ('Hearts','Diamonds','Spades','Clubs');
var
  i, j, iCard, iHand, iSuit, iValue: Integer;
  iCards: array [1..52] of Integer;
begin
  { Initialise the deck. }
  for i := 1 to 52 do
    iCards[i] := i;
  { Shuffle the cards. }
  for i := 52 downto 2 do begin
    { Select one at random from those left. }
    j := RandomInt(1, i);
    { And swap with the current one. }
    iCard := iCards[i];
    iCards[i] := iCards[j];
    iCards[j] := iCard;
  end;

  { And display. }
  for iSuit := 1 to 4 do
    for iHand := 1 to 4 do
      TLabel(FindComponent('lbl' + sSuits[iSuit] +
                           IntToStr(iHand))).Caption := '';
  for i := 1 to 52 do begin
    iHand := (i + 12) div 13;
    { Get the hand, 1-4. }
    iSuit := (iCards[i] + 12) div 13;
    { And the suit, 1-4. }
    iValue := (iCards[i] - 1) mod 13 + 1;
    { And the value, 1-13. }
    with TLabel(FindComponent('lbl' + sSuits[iSuit] +
                              IntToStr(iHand))) do
      Caption := Caption + sValues[iValue];
  end;
end;
```

Figure 6: Shuffling and dealing a deck of cards.

then swap the positions of these two cards. At the end of 51 steps (no point in processing the last position), we have a shuffled deck that can be dealt to the players.

This process is shown in Figure 6. First the array of cards is filled with all the cards. These are represented by integers from 1 to 52, where the first 13 are the hearts from Ace to King, then diamonds, spades, and clubs. We then shuffle the deck by swapping a randomly selected card with the current one, stepping through each position in turn.

Finally, the cards are dealt, with the first 13 cards going to the first hand, the second 13 to the second hand, etc. The card values from these positions in the deck are mapped on a representation, and displayed on the screen. This is all demonstrated on the Shuffle tab in the accompanying demonstration program (see Figure 7). Press the button to deal the cards, and watch how they seem to fall randomly. Similar processing can be performed for any sequence that requires a random ordering.

Approximations

Another use for random numbers is in approximating values using a random sample. The idea is that we sample a volume

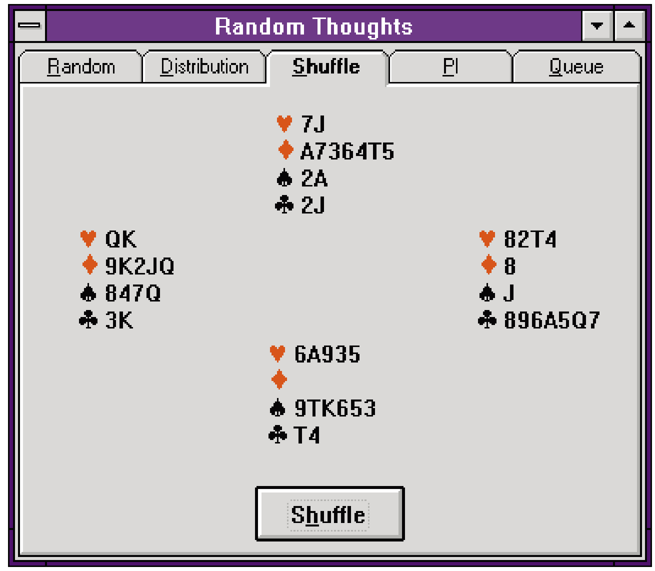


Figure 7: Dealing the cards.

containing the required value, counting those points that satisfy some condition directly related to it. We can then calculate an approximation as the ratio of those points that satisfy the condition to the total number of samples.

As an example, we can compute an approximate value of π by observing that the area of a circle of radius 1 is equal to π. If we place this circle in a bounding square, which has an area of 4, we have the situation previously outlined, and shown in Figure 8. We can now pick random points within the square, calculate their distance from the center, and count them if less than or equal to 1, i.e. if they fall within the circle. Comparing this count with the total number of sample points, and multiplying by 4 gives us an approximate value of π.

Choosing a random point within the square is straightforward. If we assume the square is centered on a Cartesian coordinate grid, then it extends from -1 to +1 in both the x and y directions. The random-number generator returns a real value in the range 0 to 1 (excluding 1), which can be mapped on the required range by multiplying it by 2 and subtracting 1. In more general terms, we multiply by the difference between the upper and lower bounds of the range, then add the value of the lower bound. This is encapsulated in the *RandomReal* function:

```
{ Return a random real value such that
  rLower <= x < rUpper. }
function RandomReal(rLower, rUpper: Real): Real;
begin
  if rUpper < rLower then
    raise ERandom.Create(
      'Range for random real is invalid');
  Result := fnRandom * (rUpper - rLower) + rLower;
end;
```

Generating two values in the range -1 to +1 gives us a point in the square. We then calculate its distance from the center by adding the squares of the two values, and count it if it falls inside the circle.

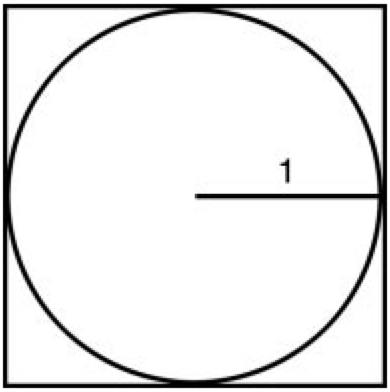


Figure 8: Approximating π using a random sample of points in a square with sides of length 2.

The PI tab in the demonstration program illustrates this process (see Figure 9).

After entering the number of sample points to test, the program calculates an approximation as previously described, and displays the results. It also provides visual feedback of the process by plotting the points on the screen. Those inside the circle are colored red, while those outside are black.

As the number of sample points grows larger, the value computed for π should converge to 3.14159. At smaller numbers of points, the approximation can vary widely.

Simulations

Simulation is our final example of using random numbers. Almost any process can be simulated on a computer, allowing us to model a situation, and by changing certain parameters, alter its outcomes.

The process we are simulating is that of a queue of people waiting to be served at a number of outlets, such as tellers in a bank.

The parameters that control this process are the distribution of the arrival time of the people, the distribution of their service times, and the number of service points available.

Arrival and service times can be fairly accurately simulated by a Poisson distribution. This distribution gives the probability of a certain number of events happening within a given time period, or alternatively, the probability of having to wait a given length of time for an event to occur.

It is governed by the equation:

$$\pi\,(x;\mu) = \frac{e^{-\mu}\mu^{x}}{x!}$$

which returns the probability of x events happening for a distribution with μ as the average. This clusters values around the average, but does make it possible for very small or very large values to occur.

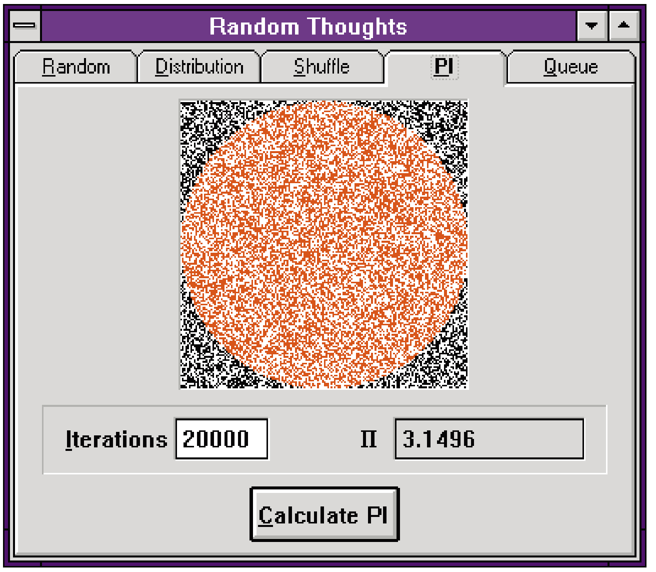


Figure 9: Approximating π with a random-number generator.

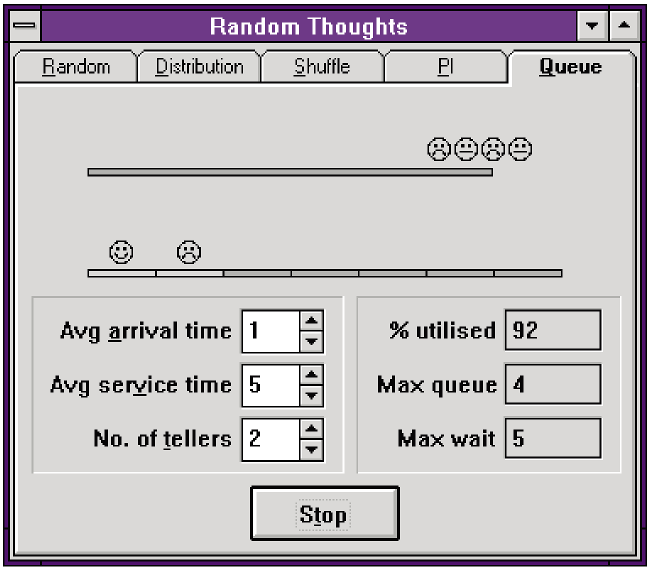


Figure 10: Simulating a queue of people at a bank.

The demonstration program gives you an opportunity to test your theories on queues.

Go to the Queue tab (see Figure 10). It allows the average time between arrivals, the average time spent being served, and the number of service points to be set before starting the simulation. Then watch as the people arrive, wait in line if necessary, get served, and leave.

Statistics are accumulated as the process continues, measuring the percentage of time all service points are being used, the maximum length of the queue, and the maximum time that any one person had to wait.

Obviously, we want the utilization to be as high as possible, with the other two values being as low as possible. Try different combinations of the parameters to simulate an early-morning lull, or a lunch-time rush.

Conclusion

Random numbers are used in a wide variety of applications, providing a degree of unpredictability — or at least the appearance of it — in an otherwise deterministic process. The demonstration program shows several applications of random numbers in differing situations.

Delphi, or more correctly Object Pascal, provides a built-in random-number generator, which can be manipulated to produce any probability distribution that we want to use.

The RandUtil unit that accompanies this article implements many of the common requirements when working with random numbers, and can easily be included in other projects. Take a chance. ∆

The demonstration project referenced in this article is available on the Delphi Informant Works CD located in INFORM\97\MAR\DI9703KW.

Keith Wood is an analyst/programmer with CSC Australia, based in Canberra. He started using Borland's products with Turbo Pascal on a CP/M machine. Occasionally working with Delphi, he has enjoyed exploring it since it first appeared. You can reach him via e-mail at kwood@netinfo.com.au or by phone (Australia) 6 291 8070.

By Peter Dove and Don Peer

Parentage and Texture Mapping

Delphi Graphics Programming: Part III

Our last installment discussed polygon filling, flat shading, directional light sources, vectors, normals, bit shifting, and backface removal — features that take you a step closer to building a 3D, rendered component. This time, we'll cover two basic concepts that will move us toward the same end. First, we'll change our parents. Then we'll add a new graphics feature to our 3D component: texture mapping, or the ability to map textures onto polygons.

Changing Your Parents

This month, we'll make a fundamental change to the *TGMP* object; that is, we'll change the object from which *TGMP* is inherited. Originally, *TGMP* was inherited from *TComponent*. This was fine except that the form displayed only a small button, making it quite difficult to visualize, at design time, how the screen would look. Wouldn't it be great to simply drop a *TGMP* window onto the form, rather like a *TImage*? Well, we can; all we need to do is inherit from class *TCustomControl*.

TCustomControl combines two things: a window with a handle and a *TCanvas*. This solves our visualization problem and gives the component much more flexibility. We can now place the component on a panel and align it just as we would align panels to a form, giving us all the drawing functionality we need. Figure 1 shows the required deletions, additions, and modifications to the *TGMP* class and its constructor. We've also added a few new procedures and record structures that we'll explain as we go.

```
TGMP = class(TCustomControl) { Was TComponent }
  private
    { FFrontBuffer removed - TGMP has own window & canvas }
    { FWindowHandle removed - TGMP has own window handle }
    procedure SetBackColor(Value : TColor);
      { Sets the background color and then calls Paint }
    procedure Paint; override;
      { The Paint procedure is overridden, so we can fill
        the background color into the backbuffer, then
        flip it onto the TGMP canvas. This is necessary
        whenever the component gets uncovered by another
        component. }
  published
    { Published declarations }
    property Align;
      { The Align property's functionality was declared in
        its 'great grandparent' class of TControl. We just
        moved it from the public section to published. }
end;

constructor TGMP.Create(AOwner : TComponent);
begin
  { Add the following }
  Height := 200;
  Width  := 200;
  { Get the height and width of the window }
  ViewHeight := Height;
  ViewWidth  := Width;
end;
```

Figure 1: The *TGMP* class and its constructor.

Next, we need to handle a Windows message to manage some aspects of resizing the control:

```
procedure TGMP.WMSize(var Message: TWMSize);
begin
  inherited;
  { Get height and width of window }
  ViewHeight := Height;
  ViewWidth  := Width;
  { Set bitmap's height }
  FBackBuffer.Height := ViewHeight;
  FBackBuffer.Width  := ViewWidth;
  { Set up viewport }
  HalfScreenHeight  := ViewHeight div 2;
  HalfScreenWidth   := ViewWidth  div 2;
end;
```

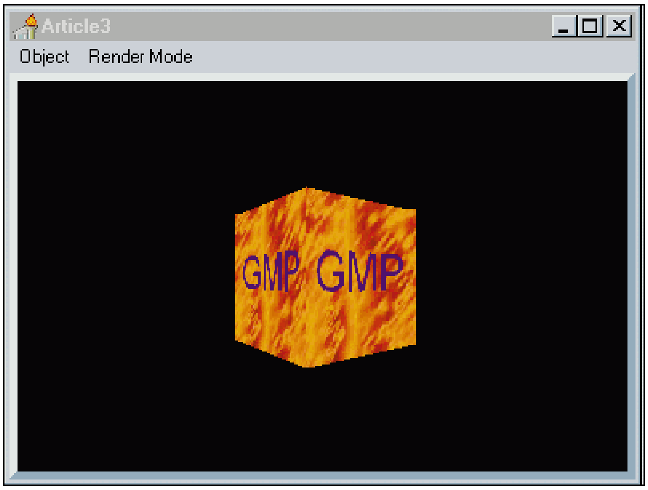


Figure 2: Freshen your drink? A texture-mapped cube.

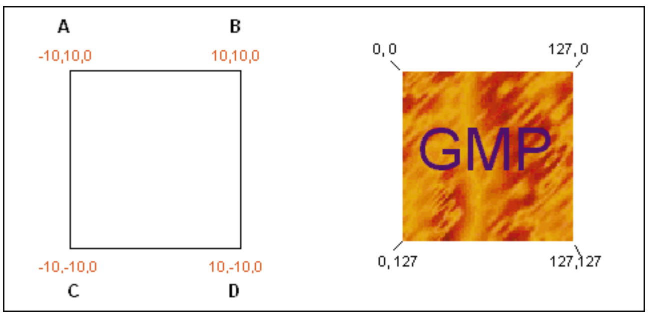


Figure 3: By the numbers; mapping texture coordinates onto polygons.

TGMP is a window that can be resized. It also has a back-buffer that mirrors the size of the front window; every time the user resizes the window, we have to resize our backbuffer to match. Notice the call to **inherited**; this makes sure that the resize message gets passed on for any default handling done by its ancestors. That's all there is to it! Of course, messaging can get a little more complicated; if you want to learn more about the intricacies of messaging, read *Developing Custom Delphi Components* by Ray Konopka [Coriolis Group Books, 1996]. This book offers good insights into using messages in relation to your own components.

Taking the Rough with the Smooth

Texture mapping is an efficient way of simulating complicated objects by minimizing the number of polygons. Every polygon you use has to go through many calculations — as demonstrated in the previous installment — involving normals, lighting, and rotation. See Figure 2 for an example of a texture-mapped cube.

The gaming community and the authors of books about 3D graphics make a big deal over the difficulty and computational expense of texture mapping. Well, the good news is that texture mapping is actually easy — and reasonably fast. The bad news is that the example we'll show you runs quite slowly because we've reached the limits of the Graphical Device Interface (GDI). This will all be rectified in next month's article, when

```
{ Types }
TRenderMode =
  (rmWireframe,rmSolid,rmSolidShade,rmSolidTexture);
TBitmapStorage = array[0..127,0..127] of TColor;
{ Private data member }
FCurrentBitmap : TBitmapStorage;
{ Public declarations }
procedure SetCurrentBitmap(Bitmap : TBitmap);
{ Implementation }
procedure TGMP.SetCurrentBitmap(Bitmap : TBitmap);
var
  X, Y : Integer;
begin
  with Bitmap.Canvas do
    for X := 0 to 127 do
      for Y := 0 to 127 do
        FCurrentBitmap[x,y] := Pixels[X,Y];
end;
```

Figure 4: The modified *TRenderMode*.

```
{ Add to var statement }
TextureStart, TextureEnd : TPoint;

{ Add after rmSolidShade part of case statement }
SolidTexture :
begin
  RemoveBackfacesAndShade(Object3D);
  OrderZ(Object3D);
  for X := 0 to Object3D.NumberPolys - 1 do
    with Object3D.PolyStore[x] do
      begin
        { If backface then don't bother rendering }
        if not visible then
          Continue;
        ClearYBuckets;
        //***** Texturing for triangle *********
        if NumberPoints = 3 then
          begin
            TextureStart.X  := 63;  TextureStart.Y  := 0;
            TextureEnd.X    := 0;   TextureEnd.Y    := 127;
            DrawLine3DTexture( Point[0], Point[1],
                               TextureStart, TextureEnd);
            TextureStart.X  := 0;   TextureStart.Y  := 127;
            TextureEnd.X    := 127; TextureEnd.Y    := 127;
            DrawLine3DTexture( Point[1], Point[2],
                               TextureStart, TextureEnd);
            TextureStart.X  := 127; TextureStart.Y  := 127;
            TextureEnd.X    := 63;  TextureEnd.Y    := 0;
            DrawLine3DTexture( Point[2], Point[0],
                               TextureStart, TextureEnd);
          end
        else
          //***** Texturing for Quad *********
          begin
            TextureStart.X  := 127; TextureStart.Y  := 0;
            TextureEnd.X    := 0;   TextureEnd.Y    := 0;
            DrawLine3DTexture( Point[0], Point[1],
                               TextureStart, TextureEnd);
            TextureStart.X  := 0;   TextureStart.Y  := 0;
            TextureEnd.X    := 0;   TextureEnd.Y    := 127;
            DrawLine3DTexture( Point[1], Point[2],
                               TextureStart, TextureEnd);
            TextureStart.X  := 0;   TextureStart.Y  := 127;
            TextureEnd.X    := 127; TextureEnd.Y    := 127;
            DrawLine3DTexture( Point[2], Point[3],
                               TextureStart, TextureEnd);
            TextureStart.X  := 127; TextureStart.Y  := 127;
            TextureEnd.X    := 127; TextureEnd.Y    := 0;
            DrawLine3DTexture( Point[3], Point[0],
                               TextureStart, TextureEnd);
          end;
        { Scan Convert Texture Buckets }
        RenderYBuckets;
      end;  { End of with block }
end; { End of rmSolidTexture block }
```

Figure 5: Adding two new *TPoint* variables.

```
procedure TGMP.DrawLine3DTexture(var StartPoint, EndPoint :
  TPoint3D; var TextStart, TextEnd : TPoint);
var
  NewStartPoint, NewEndPoint : TPoint;
begin
  NewStartPoint.X := HalfScreenWidth + Round(StartPoint.X *
    ViewingDistance / (StartPoint.Z + ZDistance));
  NewStartPoint.Y := Round(HalfScreenHeight - StartPoint.Y *
    ViewingDistance / (StartPoint.Z + ZDistance));
  NewEndPoint.X := HalfScreenWidth + Round(EndPoint.X *
    ViewingDistance / (EndPoint.Z + ZDistance));
  NewEndPoint.Y := Round(HalfScreenHeight - EndPoint.Y *
    ViewingDistance / (EndPoint.Z + ZDistance));
  case RenderMode of
    rmSolidTexture : DrawLine2DTexture(NewStartPoint,
                     NewEndPoint,TextStart,TextEnd);
  end;
end;

procedure TGMP.DrawLine2DTexture(var StartPoint, EndPoint,
  TextStart, TextEnd : TPoint);
var
  CurrentX, XIncr : Single;
  TextX, TextY, TextXIncr, TextYIncr : Single;
  Y , Length : Integer;
  TempPoint : TPoint;
begin
  { No point in drawing horizontal lines! The rest of the
    polygon will define the edges. }
  if StartPoint.Y = EndPoint.Y then
    Exit;
    { Swap if Y1 is less than Y2, so we are always drawing
      from top to bottom }
  if EndPoint.Y < StartPoint.Y then
    begin
      TempPoint   := StartPoint;
      StartPoint  := EndPoint;
      EndPoint    := TempPoint;
      TempPoint   := TextEnd;
      TextEnd     := TextStart;
      TextStart   := TempPoint;
    end;
  Length := (EndPoint.Y - StartPoint.Y) + 1;
  { Xincr is how much the X must increment
    through each Y increment }
  XIncr     := (EndPoint.X - StartPoint.X) / Length;
  CurrentX  := StartPoint.X;
{ Work out the TextX Increment and TextY Increment }
  TextXIncr  := (TextEnd.X - TextStart.X) / Length;
  TextYIncr  := (TextEnd.Y - TextStart.Y) / Length;
  TextX      := TextStart.X;
  TextY      := TextStart.Y;
  { Loop through the Y values and fill the YBuckets }
  for Y := StartPoint.y to EndPoint.y do begin
    { YBuckets := 0 to 479 must not be greater than 479 }
    if Y > 479 then
      Break;
      { YBuckets := 0 to 479 must not be less than 0 }
      if Y >= 0 then
        { All YBuckets are initialized to -16000 }
        if YBuckets[Y].StartX = -16000 then
          begin
            YBuckets[Y].StartX := Round(CurrentX);
            YBuckets[Y].EndX   := Round(CurrentX);
            TextureBuckets[Y].StartPosition.X :=Round(TextX);
            TextureBuckets[Y].StartPosition.Y :=Round(TextY);
            TextureBuckets[Y].EndPosition.X   :=Round(TextX);
            TextureBuckets[Y].EndPosition.Y   :=Round(TextY);
          end
        else
          begin
            if CurrentX > YBuckets[Y].EndX then
              begin
                YBuckets[Y].EndX := Round(CurrentX);
                TextureBuckets[Y].EndPosition.X :=
                  Round(TextX);
                TextureBuckets[Y].EndPosition.Y :=
                  Round(TextY);
              end;
            if CurrentX < YBuckets[Y].StartX then
              begin
                YBuckets[Y].StartX := Round(CurrentX);
                TextureBuckets[Y].startPosition.X :=
                  Round(TextX);
                TextureBuckets[Y].startPosition.Y :=
                  Round(TextY);
              end;
          end;
      CurrentX  := CurrentX  + XIncr;
      TextX     := TextX     + TextXIncr;
      TextY     := TextY     + TextYIncr;
    end;
end;
```

Figure 6: The *DrawLine3DTexture* and *DrawLine2DTexture* methods.

we move away from the GDI and into Device Independent Bitmaps (DIBs), which will allow us to draw directly into their memory, resulting in much faster texture mapping.

There are two basic types of texture mapping: linear and perspective. Perspectively correct texture mapping is the mathematically pure version, in which every pixel is divided by its *z* value to obtain the correct perspective for every pixel along the line. As you may recall from the previous articles, we worked out the 2D points by dividing the *x,y* value by the *z* value, then interpolated between those converted points to draw the line. This is fine with solid drawing, but results in slight warping of a texture when the polygon is large and a lot of interpolation occurs. On the other hand, hardly any difference is apparent between linear and perspective texture mapping on small polygons. The *TGMP* rendering component will use linear texture mapping, because it delivers quality results at a reasonable speed (but not with the GDI).

As you may have realized, our *TGMP* rendering component allows only for quads and triangles; so we'll first explain how to texture-map quads. This is the easiest place to start — mapping a square, 2D texture onto a 3D quad, and applying the mapping after the 3D quad has been converted to 2D screen coordinates. Therefore, we want to introduce the idea of texture coordinates. Initially, we'll limit our texture to 128 by 128 pixels. (In the next article, we'll explain how to support any size, although we suspect that you'll work that out for yourself.) Figure 3 shows a texture on the right and a quad polygon on the left. It also shows the *x, y, z* coordinates on the polygon.

Let's say that we "knew" the distance on the screen between polygon point A and point B was 10 pixels, and that we wanted to map the texture on the right directly on the polygon. We would pick 10 textured pixels evenly spaced from 0, 0 to 127, 0 and apply them to the polygon. We'll demonstrate this with some programming

```
procedure TGMP.RenderYBuckets ;
var
  Y, I, Length : Integer;
  TextX, TextY, TextXIncr, TextYIncr : Single;
begin
  if RenderMode <> rmSolidTexture then
    begin
      for Y := 0 to 479 do
        if YBuckets[Y].StartX <> -16000 then
          DrawHorizontalLine(Y, YBuckets[Y].StartX,
                                    YBuckets[Y].EndX);
     end
  else
    // ********** This deals with texturing *************
    for Y := 0 to 479 do
      if YBuckets[Y].StartX <> -16000 then
        begin
          { Work out how long the scan line is }
          length := (YBuckets[Y].EndX-YBuckets[Y].StartX)+1;
          { Work out X, Y increments to go from the start
            texture coordinate to end texture coordinate }
          TextXIncr := ((TextureBuckets[Y].EndPosition.X -
            TextureBuckets[Y].StartPosition.X))/length;
          TextYIncr := ((TextureBuckets[Y].EndPosition.Y -
            TextureBuckets[Y].StartPosition.Y))/length;
          { The current coordinates }
          TextX := TextureBuckets[Y].StartPosition.X;
          TextY := TextureBuckets[Y].StartPosition.Y;
          { Step through texture coordinates and draw them }
          for I := YBuckets[Y].StartX to YBuckets[Y].EndX do
            begin
              SetPixel(FBackBuffer.Canvas.Handle,I, Y,
                FCurrentBitmap[Round(TextX),Round(TextY)]);
              TextX := TextX + TextXIncr;
              TextY := TextY + TextYIncr;
            end;
        end;
end;
```

Figure 7: The *RenderYBuckets* procedure.

```
TForm1 = class(TForm)
  public
    MyBitmap : TBitmap;
end;

procedure TForm1.FormShow(Sender: TObject);
begin
  { Allocate memory for bitmap }
  MyBitmap := TBitmap.Create;
  { Load bitmap }
  MyBitmap.LoadfromFile('tgmp.bmp');
  GMP1.SetCurrentBitmap(MyBitmap);
end;

procedure TForm1.FormDestroy(Sender: TObject);
begin
  { Free memory allocated for the bitmap }
  MyBitmap.Free;
end;
```

Figure 8: The bitmap loads after memory is allocated in the *FormShow* procedure.

examples, but first we need to define a record where we can store the texture coordinates as we draw the lines of the polygon:

```
TTextureBucket = record
  StartPosition, EndPosition : TPoint;
end;
```

We also need to place this data member in the **private** section of *TGMP*:

```
{ 480 is maximum screen height }
TextureBuckets : array[0..479] of TTextureBucket;
```

This data member is similar to the YBuckets we discussed last month. This similarity is required because we'll want to record the changes in the texture coordinates as we move through the *Y* scan lines. Now we can add a new texture mode that will include the texturing and also create a structure to hold the texture map. Figure 4 shows the modified type *TRenderMode* with the new mode added, and the type *TBitmapStorage*, which is a holder for the texture map. This is followed by the **private** declaration of the bitmap data member *FCurrentBitmap* and the **public** declaration for the procedure *SetCurrentBitmap*, which is used to set the bitmap. Beneath the listed declarations is the implementation of the *SetCurrentBitmap* procedure.

The procedure *SetCurrentBitmap* takes a bitmap and assigns all the pixels into the array *FCurrentBitmap*. The reason we decided to do this, rather than take a pointer to the bitmap, is that it's much faster to access an array than to get the individual pixels via *Bitmap.Canvas.Pixels[X,Y]*. By creating this array in advance, we can expedite our access to the pixel colors at the cost of only a small area of memory. Next, we must add two new variables of type *TPoint* — namely *TextureStart* and *TextureEnd* — to the procedure *RenderNow*, then add additional code to *RenderNow* for processing that will deal with the texture mapping mode (see Figure 5).

In Figure 5, we have a new procedure that's called from within *RenderNow*: *DrawLine3DTexture*, which converts the 3D coordinates to 2D screen coordinates and passes them, along with the texture coordinates, to *DrawLine2DTexture*. This is the procedure that does all the work of mapping the texture coordinates to the polygon. The code for *DrawLine3DTexture* and *DrawLine2DTexture* is shown in Figure 6.

As you can see, *DrawLine2DTexture* does a lot of work: It figures out the beginning of the 2D line and determines the *X* increment it must make through every *Y* step; then it determines the *X* and *Y* increments it must get from the start texture coordinate to the end texture coordinate in the required number of *Y* steps.

If you've been following this series, you can see the logical progression from the previous article's *DrawLine2DSolid* procedure. Last, we need to look at the procedure that actually draws the polygons on the screen, called *RenderYBuckets* (see Figure 7). This, of course, has been changed to cope with the texturing, but it hasn't been changed so much that it's incompatible with the previous version in Part II. As you can see from the section that deals with texturing, the math is simple. And that's all there is to texture mapping.

Our Third Application

Our third application uses basically the same code as that in Part II. The only major modifications were made to provide the **Solid Texture** menu option and to declare *MyBitmap*. As you can see in Figure 8, the bitmap is loaded after memory is allocated in the *FormShow* procedure. The memory that's allocated for the bitmap is eventually freed in the *FormDestroy* procedure. The complete source listing for our third application, developed with the *TGMP* component, is available on this CD.

Hints and Harbingers

Remember to set your display driver to 16-bit color rather than 256 colors; otherwise you'll think we've applied a "mud" texture! We've deliberately stuck with 16-bit color this month so that we don't have to worry about palettes; and we'll stay with 16-bit color for a while before we move on. The benefit of moving to a 256-color platform is speed; there's a lot less image data to move around, although considerably more effort is required to apply textures.

Next month, we'll create a 3D data file reader for polygon files and move on to using DIBs. This will make *TGMP* supersonic and remove the current GDI bottleneck. DIBs will also take us into the realm of using pointers with Delphi and bit manipulation. Finally, we'll rewrite some of the drawing routines, using our own pointers to draw directly into memory, and close by adding shading to this month's texture mapping algorithm. ∆

References

LaMothe, A., *Black Art of 3D Game Programming* [Waite Group Press, 1995].

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\97\MAR\DI9703DP.

Peter Dove is a Technical Associate with Link Associates Limited and a partner in Graphical Magick Productions. He can be reached via the Internet at peterd@graphicalmagick.com.

Don Peer is a Technical Associate for Greenway Group Holdings Inc. (GGHI) and a partner in Graphical Magick Productions. He can be reached via the Internet at dpeer@graphicalmagick.com.

By Robert Vivrette

Wildcard Specs, etc.

Delphi Tips and Techniques

How can I delete files based on a wildcard specification?

You may have already noticed that Delphi's principal file management routines generally only allow file operations on a single file at a time, e.g. the *DeleteFile* function will only delete one file.

The key to this trick is to use the *FindFirst* and *FindNext* functions to loop through all files in a directory that match a certain wildcard specification. The procedure in Figure 1 accepts a wildcard specification, and a Boolean value to indicate whether the user should be prompted for each file deletion. The procedure uses *FindFirst* to look for the first file that matches the wildcard. If it finds one, it shows a confirmation dialog box (if the *Prompt* Boolean is set). If the file is to be deleted, then a call is made to *DeleteFile* to erase the file. The *FindNext* function is called to get the next file, and so on, until no files are found.

```
procedure DeleteFileWithMask(Path: string; Prompt: Boolean);
var
  SearchRec  : TSearchRec;
  JustPath   : string;
  Found      : Integer;
  Response   : Integer;
begin
  JustPath := ExtractFilePath(Path);
  Found    := FindFirst(Path,faArchive,SearchRec);
  while Found = 0 do
    with SearchRec do begin
      Response := mrYes;
      if Prompt then
        Response :=
          MessageDlg('Delete File?'+#13+JustPath+Name,
                     mtWarning,mbYesNoCancel,0);
      case Response of
        mrCancel  : Break;
        mrYes     : if not DeleteFile(JustPath+Name) then
                      MessageDlg('Error Deleting File',
                                 mtError,[mbOK],0);
      end;
      Found := FindNext(SearchRec);
    end;
  FindClose(SearchRec);
end;
```

Figure 1: A procedure to delete files based on a wildcard specification.

Note that with the user prompt turned off, this routine could easily delete all files in a directory in a fraction of a second. Therefore, care should be taken when implementing any facility for mass deletion of files within a program.

A call to the procedure would look something like this:

```
procedure TForm1.BitBtn1Click(Sender: TObject);
begin
  DeleteFileWithMask(Edit1.Text,
                     CheckBox1.Checked);
end;
```

See Figure 2 for a sample of its use.

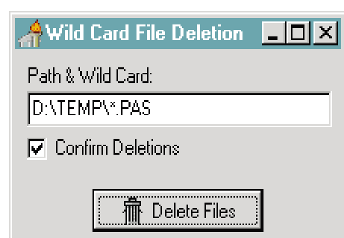


Figure 2: The Wild Card File Deletion dialog box.

How do I determine the name of a key that has been pressed?

Occasionally, you might want to obtain a textual representation of a key that has been pressed. This can be easily accomplished with the *GetKeyNameText* API call. Simply trap the *WM_KEYDOWN* message on a form as indicated in Figure 3.

When a key is pressed, the message is handled by the *WMKeyDown* method. We extract the *KeyData* field from the message and pass it into *GetKeyNameText*, which fills a

```
unit Unit1;

interface

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

type
  TForm1 = class(TForm)
    Label1: TLabel;
  private
    procedure WMKeyDown(var Message: TWMChar);
      message WM_KeyDown;
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.WMKeyDown(var Message: TWMKeyDown);
var
  KeyName : array[0..255] of Char;
begin
  if GetKeyNameText(Message.KeyData,KeyName,
                    SizeOf(KeyName)) > 0 then
    Label1.Caption := KeyName;
  inherited;
end;

end.
```

Figure 3: Getting the name of a key that has been pressed.

buffer we have provided. To show what key was pressed, this return value is assigned to a label on our sample form.

Figure 4: The result of *GetKeyNameText*.

Note that the value returned will depend on the keyboard layout currently defined in Windows. Figure 4 shows the result of *GetKeyNameText*.

When I make many rapid changes to a table, my data-aware controls flicker. How can I stop this?

Often, you may be quickly traversing through the records of a database to perform some calculations, addition/deletion of records, etc. Each time the record changes, data-aware controls are automatically updated by the BDE. Sometimes you aren't interested in the appearance of the controls until you are done.

To solve this problem, you can use the *DisableControls* and *EnableControls* methods of the *TTable* (and also the *TQuery* and *TStoredProc*) objects. *DisableControls* turns off updates to data-aware controls linked to the table and *EnableControls* turns updates back on.

To illustrate this, the sample application in Figure 5 adds 1,000 records with random values to a table. A DBGrid is used to show the records of the table. Without using *DisableControls*, the grid's scrollbar will flicker as new

```
procedure TForm1.AddRecords;
var
  a : Integer;
begin
  for a := 1 to StrToInt(edtItemsToAdd.Text) do begin
    Table1.Append;
    Table1.FieldByName('Value1').AsInteger := Random(1000);
    Table1.FieldByName('Value2').AsInteger := Random(1000);
    Table1.FieldByName('Value3').AsInteger := Random(1000);
    Table1.Post;
  end;
end;

procedure TForm1.ClearRecords;
begin
  Table1.First;
  while not Table1.Eof do
    Table1.Delete;
end;

procedure TForm1.btnFillClick(Sender: TObject);
begin
  if chkDisable.Checked then
    begin
      Table1.DisableControls;
      AddRecords;
      Table1.EnableControls;
    end
  else
    AddRecords;
end;

procedure TForm1.btnClearClick(Sender: TObject);
begin
  if chkDisable.Checked then
    begin
      Table1.DisableControls;
      ClearRecords;
      Table1.EnableControls;
    end
  else
    ClearRecords;
end;
```

Figure 5: Using *DisableControls* to make multiple changes to a table occur faster.

records are added or cleared. The process often takes many times longer with the controls enabled (and therefore updating after each change). When the controls are disabled, all the controls are updated only once, at the end of all the changes. Figure 6 shows a use of the *DisableControls* method in the DisableControls Demo. Δ

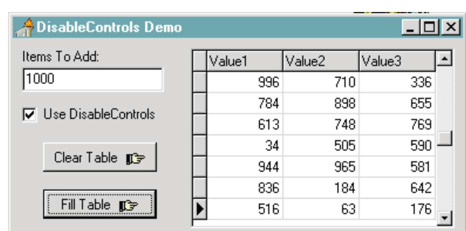


Figure 6: Using *DisableControls* to expedite numerous changes to data-aware controls.

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\97\MAR\DI9703RV.

Robert Vivrette is a contract programmer for Pacific Gas & Electric, and Technical Editor for *Delphi Informant*. He has worked as a game designer and computer consultant, and has experience in a number of programming languages. He can be reached via e-mail at 76416.1373@compuserve.com.

By Robert Vivrette

BoundsChecker 4.0

NuMega's Debugging Product for Delphi

If you're a Delphi programmer, I'll make this very easy for you. Put down this magazine. Find your checkbook. Order a copy of BoundsChecker for Delphi. If you need a little convincing, read on.

BoundsChecker 4.0 for Delphi is one of several debugging products developed by NuMega Technologies. In the past, their products have been principally for C++ programmers. However, with version 4.0 of BoundsChecker, NuMega has moved into the Delphi arena. BoundsChecker's principal purpose is simple: detect and locate program errors. And believe me, there's not much that will evade BoundsChecker's scrutiny.

Installation is simple, and the package integrates itself nicely into the **Tools** menu of Delphi's IDE. After configuring a few preferences, you're ready to go.

So Go

Running BoundsChecker on a Delphi application is equally straightforward.

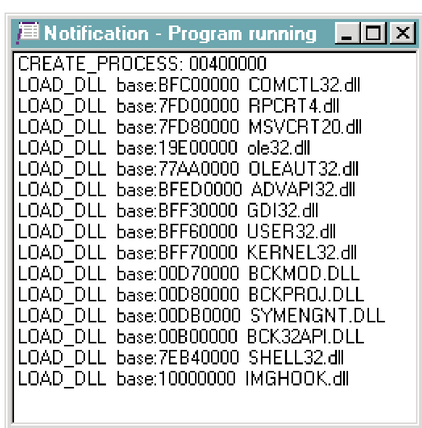


Figure 1: BoundsChecker makes its presence felt by displaying its Notification dialog box listing the modules your program needs to start.

First, from Delphi's menu, select **Options | Project** to display the Project Options dialog box. Now select the Compiler page. Click the **Debug Information** and **Stack Frames** check boxes. Then go to the Linker page and click on **Include TDW debug info**. That's it! Rebuild your program and select **Tools | BoundsChecker**.

BoundsChecker hooks into the debugging information generated during compilation, and monitors virtually every aspect of your program. First, you'll see a window that identifies the initialization steps your code is taking (see Figure 1). The information presented in the Notification dialog box includes the DLLs that must be loaded to execute the program. This window displays the modules your application is relying on to run properly — many are part of Windows (e.g. GDI32.DLL and USER32.DLL).

Your application appears next. It will run exactly as if it were running outside BoundsChecker, except it'll run a little slower.

As you put your program through its paces, BoundsChecker watches everything that's going on. It checks API calls for appropriate parameters, memory that's allocated and not released, and other bad things your program might be doing. Once you're done exercising all your program's features, quit your application to return control to BoundsChecker. Now the fun begins!

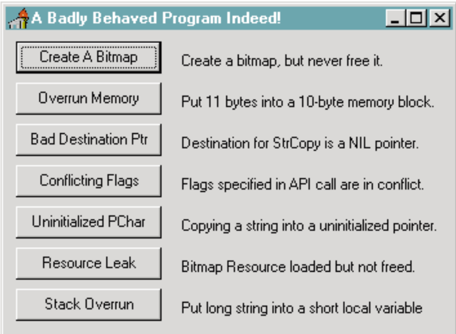


Figure 2: A badly-behaved demonstration program.

Buttons Behaving Badly

BoundsChecker takes all the information it collected during the execution of your program and prepares a detailed report of all abnormal occurrences (bugs). Each entry will show the line of code where the bug occurred, and a simple description of the problem.

To give you a better idea of how BoundsChecker works, I built the simple, badly-behaved, program shown in Figure 2. Each button in this application exhibits some unacceptable behavior indicated by the comment on the right of each button. Figure 3 shows the code behind these buttons.

Usually, BoundsChecker will trap errors and report them when the application terminates. For example, the code behind the **Create A Bitmap** button really isn't a bug until the application quits. For all the program knows, the application might clean this up later.

Sometimes however, BoundsChecker will trap and report events as the program is running. When this occurs, BoundsChecker comes to the foreground and asks you what you want to do about the error. For example, the code behind the **Overrun Memory** button immediately generates an error (see Figure 4). BoundsChecker identifies the Dynamic memory overrun, and even knows that an attempt was made to copy 11 bytes of data into 10 bytes of memory. It also shows the offending source in the bottom pane.

The center pane shows the sequence of events leading to the error. Say for example, we didn't know how the application got to *Button2Click*. (Okay, I suppose the name would pretty much give it away, but let's assume for the sake of argument it was named something else.) The information in the center pane shows that we got to the procedure by means of a button click, and that the execution path traced through several points in the Controls and StdCtrls units.

Acknowledge or Suppress?

There are several buttons on this dialog box. The **Acknowledge** button tells BoundsChecker to log this error and continue; **Info** displays information from the BoundsChecker Help file on this error; **Note** allows you to attach a descriptive note to the error; and **Halt** terminates the program.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  MyBitmap : TBitmap;
begin
  MyBitmap := TBitmap.Create;
end;

procedure TForm1.Button2Click(Sender: TObject);
var
  A : PChar;
begin
  GetMem(A,10);
  FillChar(A^,11,#0);
  FreeMem(A);
end;

procedure TForm1.Button3Click(Sender: TObject);
var
  A : array[0..20] of Char;
  B : Pointer;
begin
  B := nil;
  StrCopy(B,A);
end;

procedure TForm1.Button4Click(Sender: TObject);
begin
  SetPriorityClass(GetCurrentProcess(),
    IDLE_PRIORITY_CLASS or NORMAL_PRIORITY_CLASS);
end;

procedure TForm1.Button5Click(Sender: TObject);
var
  A : PChar;
  B : PChar;
begin
  A := StrNew('Hi There Folks!');
  StrCopy(B,A);
  StrDispose(A);
end;

procedure TForm1.Button6Click(Sender: TObject);
var
  A : HBitmap;
begin
  A := LoadBitmap(hInstance,'SAMPLE');
end;

procedure TForm1.Button7Click(Sender: TObject);
var
  A : array[0..10] of Char;
begin
  StrCopy(A,'This string is longer than 10 characters.');
end;
```

Figure 3: The code behind the buttons in the sample application.

The **Suppress** button is one of the unique features of BoundsChecker. It allows you to tell BoundsChecker to suppress (to varying degrees) further reporting of an error. When you click it, you will be asked whether to suppress the error:

- just in this function,
- in this source file,
- in this application or DLL, or
- everywhere.

The value of suppressing errors becomes apparent when you're dealing with a third-party control or DLL to which you don't have the source. This control or DLL might

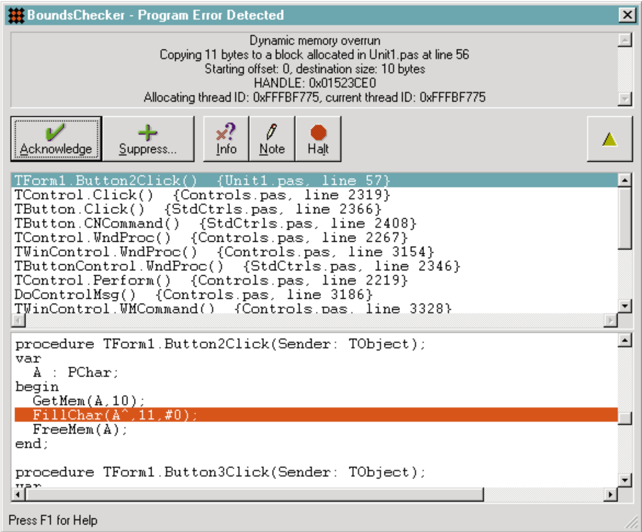


Figure 4: BoundsChecker detected the Dynamic memory overrun error raised by selecting the **Overrun Memory** button.

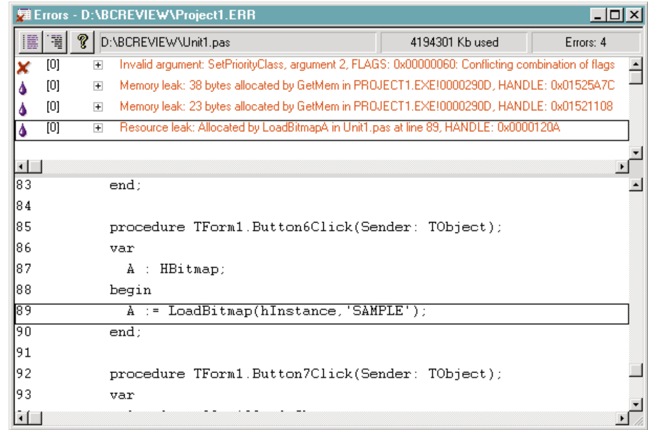


Figure 5: BoundsChecker identifies a resource leak in the sample program.

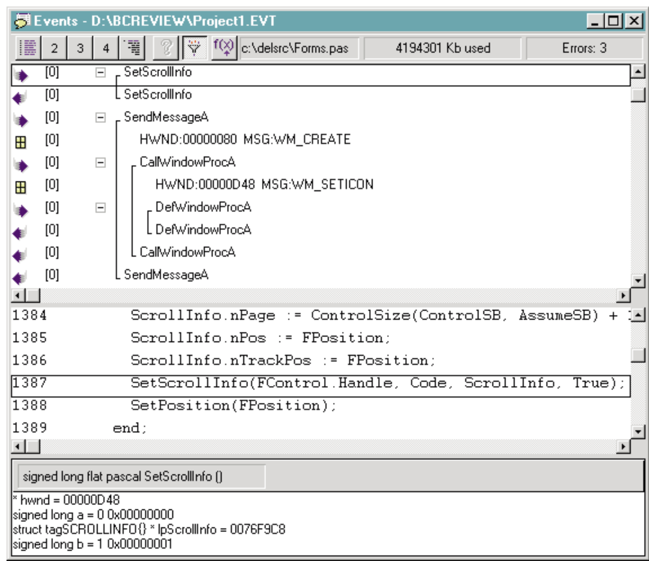


Figure 6: The BoundsChecker Events dialog box displays a log of all the program's events.

repeatedly generate an error. Suppressing the error allows you to concentrate on your code without wading through someone else's error messages.

Then, armed with this information, you contact the third-party developer and tell them what BoundsChecker reported. They'll be delighted to hear from you — or they'll hang up on you!

The error suppression information you enter can be saved to disk and, as an example, distributed to other members of your development team. NuMega provides suppression libraries with BoundsChecker for common "anomalies" in the Delphi VCL.

After quitting your application, BoundsChecker prepares a summary report that identifies all trapped errors (see Figure 5). The information presented is similar to that previously described, but there will probably be memory or resource leaks thrown in that aren't normally detected until an application is closed.

Some of the principal error detecting "techniques" in BoundsChecker for Delphi include:

- APICheck — Checks arguments passed to and received from over 5,000 API functions, including Win32, DirectX, WinSock, and Internet APIs. Errors detected include: bad pointers, conflicting flags, invalid flags, missing arguments, out-of-range arguments, uninitialized fields, and uninitialized structure size fields.
- WriteCheck — Detects the overwriting of dynamically allocated memory, local or stack memory, and global or static memory.
- LeakCheck — Detects memory and resource leaks.
- PointerCheck — Detects invalid pointer operations such as: operations on null pointers, pointers that don't point to valid data, pointer comparison errors, errors in the use of function pointers, and attempts to free handles without unlocking them.

And BoundsChecker doesn't stop there. Besides detecting run-time errors, BoundsChecker also allows you to log every detail of what your program is doing when it runs. BoundsChecker refers to these as "events." When they're enabled, you'll probably get more information about your program than you want.

For example, Figure 6 shows only a minuscule segment of all the events that transpired in the course of launching and quitting my test application. I would estimate BoundsChecker placed well over 8,000 events into the top list box. Note that the item selected in the top pane is a call to *SetScrollInfo* (I randomly stopped here in the list). The arrows in the left margin are visual clues to show you where the application is going into a procedure or function, and where it leaves. The middle pane again shows the source code involved in this event (in this case it's the Delphi FORMS.PAS unit). At the bottom of the screen, you get information on all the parameters that were passed to this function call. The level of detail in this portion of BoundsChecker can be frightening at times. Fortunately, the program provides various "find" and "filter" features that help manage the incredible mass of information.

Conclusion

Before using BoundsChecker, I was a devout fan of TurboPower's Sleuth program (previously named MemMonD for Delphi 1). Although I still use Sleuth on occasion, BoundsChecker is a far more comprehensive solution to debugging Delphi applications.

The version of BoundsChecker that I reviewed was the Standard Edition for Delphi. The list of errors caught by this version (as opposed to the C++ version) is quite a bit shorter, but all the important ones are there. I am quite certain that future versions of BoundsChecker for Delphi will narrow the differences between the two products.

I have very little to say that can be considered negative. Nothing is perfect, however, so I will mention the only two issues I have had to-date with BoundsChecker.

INFORMANT FACT FILE

BoundsChecker 4.0 for Delphi (Standard Edition) is an effective, thorough debugging tool that integrates with the Delphi IDE. It uses the debugging information generated at compile time to monitor nearly every aspect of a program. It checks for bad pointers, uninitialized structure size fields, and arguments passed to and received from API functions; detects the overwriting of memory; detects memory and resource leaks; and detects invalid pointer operations, just to name a few. It's a must-have utility for the serious Delphi developer.

BoundsChecker is a platform-specific tool. When ordering, please specify your current working platform (Windows 95 or Windows NT). NuMega recently released BoundsChecker 4.2. Current owners of version 4.0 can upgrade by purchasing the CD-ROM, or by downloading the upgrade from NuMega's Web site.

NuMega Technologies Inc.
9 Townsend West
Nashua, NH 03063

Phone: (603) 889-2386
Fax: (603) 889-1135
Web Site: http://www.numega.com
Price: Standard Edition, US$329; Professional Edition, US$549. To upgrade from version 4.0 to 4.2 via CD-ROM, the cost is US$150.

First, occasionally, I would debug a program and BoundsChecker wouldn't show any source code for certain units (specifically my units). After tweaking the program in about three or four areas, BoundsChecker eventually showed me what I wanted.

Although it's nothing I couldn't solve on my own, it would have been nice to have a little more "hand-holding" in such a situation. For example, I didn't know if the source file was missing; if the program was not compiled with the proper debug information in it; or if the preferences for BoundsChecker was causing it to look in the wrong directories.

The second minor issue I had was that it would be nice if BoundsChecker detected that my program needed compiling and did it for me when I select the **BoundsChecker** option from the **Tools** menu. (I know this can be done because TurboPower's Sleuth does it.) As it is, you need to ensure that the program is completely compiled and that all source files are saved before launching BoundsChecker.

As you can see, my only two complaints are fairly minor and easily dismissed given the tremendous capabilities provided by BoundsChecker.

Now you can put down this magazine, find your checkbook, and order a copy. ∆

The badly-behaving program referenced in this article is available on the Delphi Informant Works CD located in INFORM\97\MAR\DI9703BC.

Robert Vivrette is a contract programmer for Pacific Gas & Electric, and Technical Editor for *Delphi Informant*. He has worked as a game designer and computer consultant, and has experience in a number of programming languages. He can be reached via e-mail at 76416.1373@compuserve.com.

Business Objects: The Sequel

My column about business objects (in January's *Delphi Informant*) seems to have struck a chord for many readers. Several of you wrote noting your interest in the subject, as well as detailing how you're already implementing business objects in Delphi. Take, for example, Peter Roth of Engineering Objects International, who employs business objects in the engineering field. He notes, "The most gratifying experience was changing the solution algorithm of a nonlinear dynamic solver without affecting the rest of the program. When we used to try this in Fortran, we failed utterly."

We've already discussed some of the benefits of business objects, but let's dive deeper by looking at some of the key technical issues involved with using them in Delphi.

Not rocket science. There's really nothing special about the code for a business object. It's like any other object you've used. Take, for example, the *TCustomer* object shown here:

```
TCustomer = class(TComponent)
  private
    FID: string;
    FHomeStore: string;
    FLastName : string;
    FFirstName : string;
    FTitle: string;
    FSignUpDate: TDateTimeField;
  public
    constructor Create( AOwner : TComponent ); override;
    destructor Destroy; override;
    property ID : string read FID;
    property HomeStore : string read FHomeStore;
    property LastName : string read FLastName;
    property FirstName : string read FFirstName;
    property Title : string read FTitle;
    property SignUpDate : TDateTimeField read FSignUpDate;
    function FullName : string;
    procedure PurchaseGoods;
end;
```

TCustomer's properties and methods are based on the "real world," but it's defined in code just like a system-level object.

Therefore, the problem isn't in the architecture of a business object — it's in the storage of the object's data once it has been instantiated.

Storage options. One common method of storing object data is to use streams. In doing so, you can save the state of an object instance to a file and reload it later. Roth has found streams to be an effective way to save data, and now requires each class that he writes "to be able to save and restore itself." And while stream-based solutions work well in some contexts, they don't in a typical client/server database application that includes a large amount of data. Two people cannot simultaneously access the same objects, so a database manager is needed to handle data sharing issues.

Alternatively, you can use a relational database to store business objects. In the Delphi world, one of the best examples I've seen is in Ray Konopka's book, *Developing Custom Delphi Components* [Coriolis Group Books, 1996]. Konopka shows an innovative way to create and link business components to Delphi's data access components. Indeed, this method works well for objects that can map to one *and only one* table in a database. (For example, the *TCustomer* object shown earlier could be mapped to a single record in a Customer table.)

Nonetheless, while some business objects are simple, many aren't that easy to handle: namely, composite objects that contain pointers to other objects. For instance, suppose our *TCustomer* object has *Orders* and *Payments* as properties, both of which are arrays of orders or payments the customer has made. Because of the complexity of the interrelationships between these elements, we couldn't use the simple object solution to store this data.

In the real world, it's not uncommon for objects to span many tables and require a large number of complex joins to reassemble them when called. I have yet to see an adequate generic solution that uses Delphi to store complex objects using a relational database as the object store.

Lastly, you can opt to use an object database — something that until recently has been out of reach for Delphi developers. POET Software Corp.'s POET 4.0 bridges this gap by providing an OCX interface, enabling Delphi to store instantiated objects in a POET object database. However, while this is among the better solutions available in terms of OOP design, you can't use it alongside Delphi's built-in data-access components. Which begs the question: If you wish to build business objects in Delphi, do you need to abandon Delphi's data-aware components? From my experience, I would say so.

Besides the technical challenges we've talked about, Matthew Raffel, an independent consultant from Cumming, GA, noted his major problem is "management's lack of motivation to take the steps necessary to successfully implement business objects." He adds, "Many managers see design as too time consuming and thus, never go beyond screen and database design. Many [managers] have told me, 'We don't have the time for that kind of design.' Yet, so much time is lost in rewriting because of a lack of design around business objects. How can we get our mangers to see the benefits of designing business objects?"

— Richard Wagner

Richard Wagner is the Chief Technology Officer of Acadia Software in the Boston, MA area, and Contributing Editor to Delphi Informant. *He welcomes your comments at rwagner@acadians.com.*