February 1997, Volume 3, Number 2

Delphi ROCKS!

Delphi Games Programming

Cover Art By: *Doug Smith*

Delphi

TOOLS

New Products and Solutions

NuMega Technologies Releases BoundsChecker 4.2

NuMega Technologies, Inc. of Nashua, NH has released ***BoundsChecker 4.2***, an error detection tool for Windows. BoundsChecker 4.2 expands the number of Windows APIs supported by BoundsChecker, including ActiveX and ODBC, as well as supporting Windows NT 4.0.

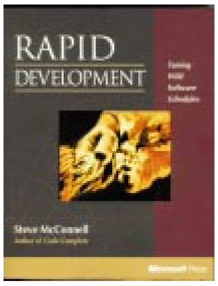
BoundsChecker helps developers detect errors within and between components. In addition, it provides developers with support for Win32, COM, ActiveX, ODBC, DirectX, the CryptoAPI, Winsock, and other Windows-specific APIs. This enables developers to detect API and OLE interface errors between components, and between components and the operating system.

BoundsChecker 4.2 is a tool for developers creating and shipping ActiveX controls. For developers building client/server applications, it checks many function calls specific to ODBC 2.5. This allows developers to check calls from an application in C, C++, or Delphi to its ODBC-compliant database.

BoundsChecker provides support for more than 100 new Windows NT 4.0 API functions. Developers can also determine the portability of their applications across all Win32 platforms.

Price: BoundsChecker 4.2 Standard Edition, US$329; upgrades for existing customers and corporate site license options are available.
Contact: NuMega Technologies, Inc., #9 Townsend West, Nashua, NH 03063
Phone: (800) 468-6342 or (603) 889-2386
Fax: (603) 889-1135
E-Mail: info@numega.com
Web Site: http://www.numega.com

Rapid Development
Steve McConnell
Microsoft Press



ISBN: 1-55615-900-5
Price: US$35 (647 pages)
Phone: (212) 886-9222

SQA Upgrades its Windows Client/Server Testing Tool

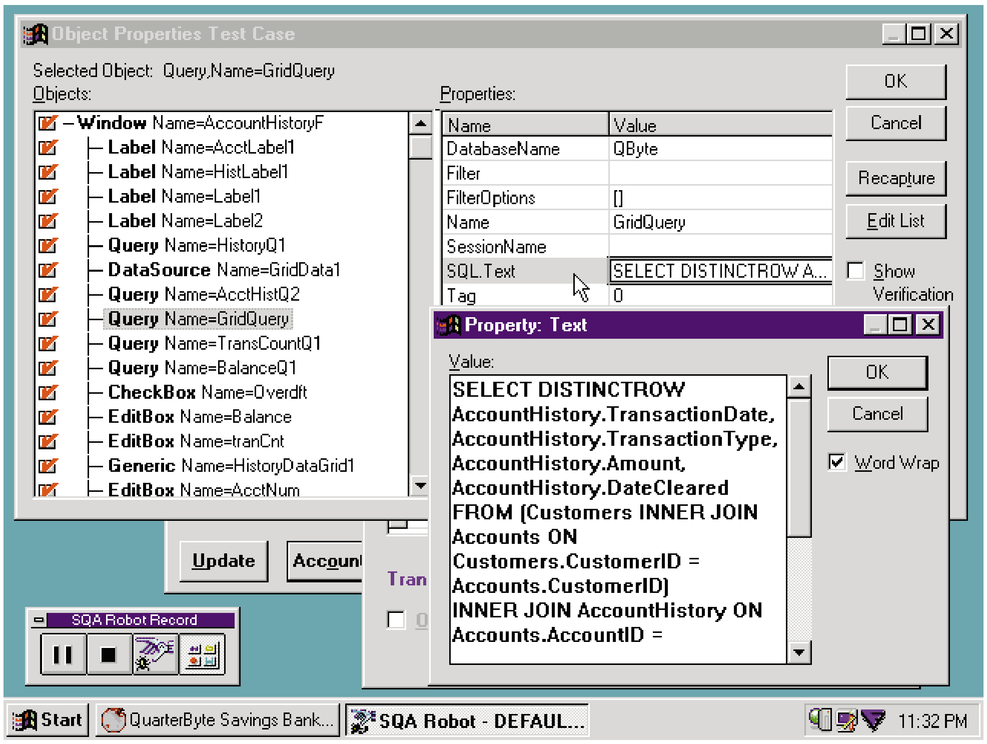
SQA, Inc. has announced ***SQA Suite 5.1***, a new version of its Windows client/server testing tool. SQA Suite 5.1 assists developers that build, deploy, and/or migrate applications across Windows platforms.

SQA Suite 5.1 can capture data and properties of hidden components, allowing objects to be manually tested in a GUI. The suite includes an SQA Manager WebEntry; cross-Windows testing solutions; a scalable, server-based test repository; portable scripts; OCX and ActiveX testing; and load and stress testing from a single location.

SQA has also expanded its Object Testing technology to test objects without a GUI. Hidden Object Testing enables users to test the properties and data of objects that cannot be manually tested. Examples include Delphi DataSource components, PowerBuilder DataStores, and hidden OCX/ActiveX controls.

SQA Suite features automated testing of enterprise-level 16- and 32-bit Windows NT and Windows 95 client/server applications, as well as 16-bit Windows 3.x applications.

SQA Suite is comprised of three products: SQA RobotJ26, SQA Manager, and SQA LoadTest.



Object Properties Test Case

Selected Object: Query,Name=GridQuery

Objects:
- Window Name=AccountHistoryF
  - Label Name=AcctLabel1
  - Label Name=HistLabel1
  - Label Name=Label1
  - Label Name=Label2
  - Query Name=HistoryQ1
  - DataSource Name=GridData1
  - Query Name=AcctHistQ2
  - Query Name=GridQuery
  - Query Name=TransCountQ1
  - Query Name=BalanceQ1
  - CheckBox Name=Overdft
  - EditBox Name=Balance
  - EditBox Name=tranCnt
  - Generic Name=HistoryDataGrid1
  - EditBox Name=AcctNum

Properties:

Name	Value
DatabaseName	QByte
Filter	
FilterOptions	[]
Name	GridQuery
SessionName	
SQL.Text	SELECT DISTINCTROW A...
Tag	0

Property: Text

Value:
```
SELECT DISTINCTROW
AccountHistory.TransactionDate,
AccountHistory.TransactionType,
AccountHistory.Amount,
AccountHistory.DateCleared
FROM (Customers INNER JOIN
Accounts ON
Customers.CustomerID =
Accounts.CustomerID)
INNER JOIN AccountHistory ON
Accounts.AccountID =
```

Price: SQA Suite, single user license, US$3,295; SQA Robot 5.1, single user license, US$2,695; SQA Manager 5.1, single user license, US$1,395.
Contact: SQA, Inc., One Burlington Woods, Burlington, MA 01803
Phone: (800) 228-9922 or (617) 229-3500
Fax: (617) 229-3780
E-Mail: info@sqa.com
Web Site: http://www.sqa.com

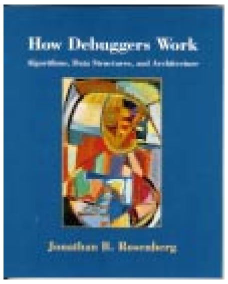
Delphi

T O O L S

New Products and Solutions

New Delphi Book

How Debuggers Work
Jonathon B. Rosenberg
Wiley Computer Publishing



ISBN: 0-479-14966-7
Price: US$34.95 (256 pages)
Phone: (212) 850-6336

MITI Announces Upgrade to Developer's Toolset

MITI of Menlo Park, CA has upgraded its SQR family of products, including ***SQR Server 4.0*** and ***SQR Workbench 4.0*** for Windows, a production-level report developer's toolset. The new versions can export production reports to HTML, as well as offer Year 2000 support, International support, and decimal match functionality.

SQR Servers provide native data access and manipulation capabilities for more than 80 database/operating system combinations. SQR Servers can produce multiple reports and database updates from a single pass through a local or remote database. SQR servers can be used with most databases, including Oracle, Sybase, Informix, IBM DB2, Ingres, Microsoft SQL Server, and Centura SQLBase, and across many operating systems, including MVS, AIX, HP-UX, VAX/VMS, SunOS, Solaris, Windows NT, Windows 3.1, and Macintosh.

SQR Workbench enables developers to create production reports, execute those reports on any server in the network, and print, fax, e-mail, or file transfer the reports within the enterprise.

SQR Workbench can perform calculations on large data sets, move and scrub data among multiple databases, warehouses, and data marts, or help back up and restore production systems.

Price: SQR Server 4.0 begins at US$10,030 per platform; SQR Workbench 4.0, US$795.
Contact: MITI, 1080 Marsh Rd., Menlo Park, CA 94025
Phone: (800) 505-4399 or (415) 326-5000
Fax: (415) 326-5100
E-Mail: info@miti.com
Web Site: http://www.miti.com

SELECT Software Tools Announces Interface for Delphi

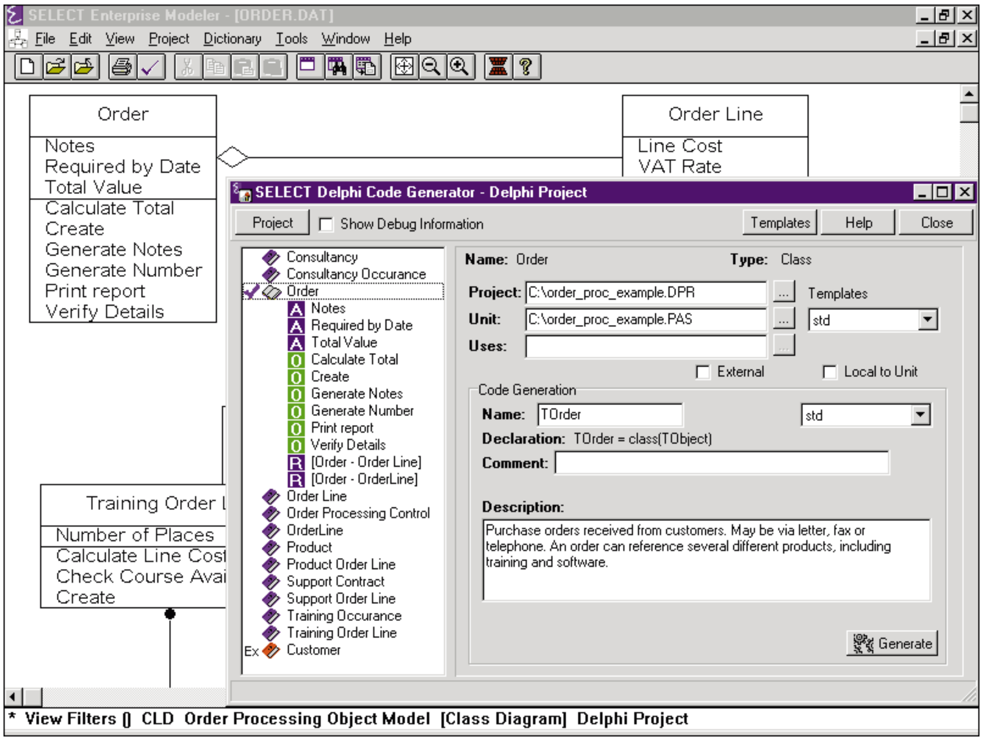
SELECT Software Tools, Inc. and **FMI Ltd.** have built ***SELECT Enterprise for Delphi***. The new object-oriented modeling toolset features analysis and design capabilities, automated process support, and an integrated code generator for Delphi 2.

SELECT Enterprise for Delphi will manage and automate the development of applications in Delphi from an object model, allowing users to work on the model and target implementation. Using the SELECT Delphi Generator, developers can ensure the model and target development areas remain consistent.

SELECT Enterprise for Delphi features initial code generation, allowing developers to generate a template-driven code framework for the application. It can also generate code from abstract models held within SELECT. The code produced is governed by user-definable templates.

Any implementation-specific information can be stored within the repository and used later for regeneration and to match against when reverse engineering.

Using SELECT's Delphi Generator, consistency between the design/modeling and implementation of Delphi code is maintained, in addition to inheritance, attributes, operations, and roles and associations.



Name: Order **Type:** Class
Project: C:\order_proc_example.DPR
Templates: std
Unit: C:\order_proc_example.PAS
Uses:
Name: TOrder std
Declaration: TOrder = class(TObject)
Comment:
Description:
Purchase orders received from customers. May be via letter, fax or telephone. An order can reference several different products, including training and software.

* View Filters [] CLD Order Processing Object Model [Class Diagram] Delphi Project

Price: US$2,995
Contact: SELECT Software Tools, Inc., 1524 Brookhollow Dr., Santa Ana, CA 92705
Phone: (800) 577-6633 or (714) 825-1050
Fax: (714) 825-1090
E-Mail: julieb@selectst.com
Web Site: http://www.selectst.com

Borland Announces New Java Development Tool

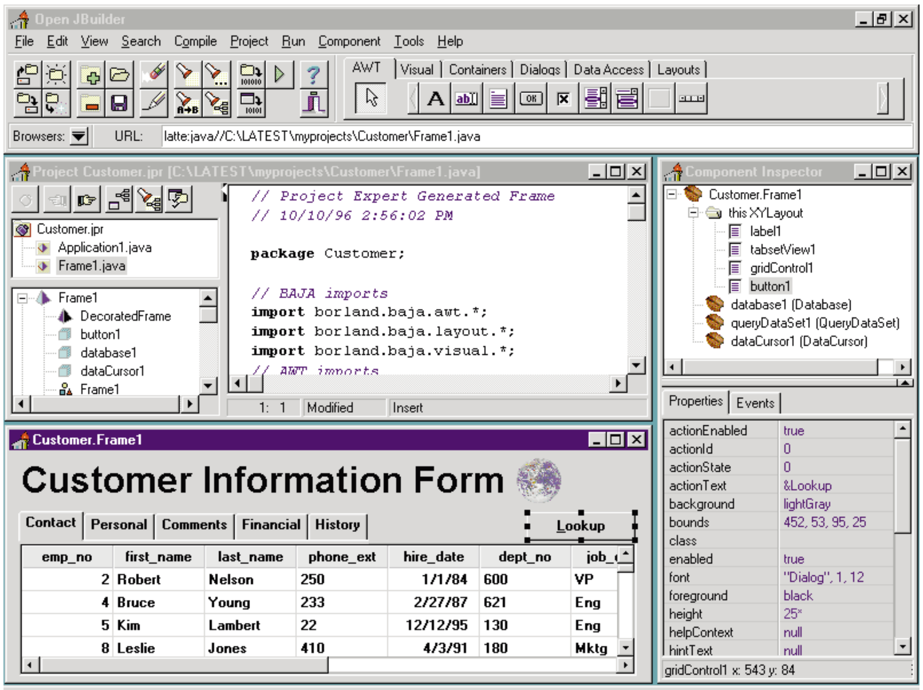
Washington, D.C. — Formerly code-named Latté, Borland has named its unreleased Java application development tool Open JBuilder.

Open JBuilder provides next-generation visual component-based development tools for Java-based, cross-platform development. Open JBuilder is a development environment for projects ranging from Web-delivered applets and applications to enterprise-wide, distributed computing solutions. It combines visual development tools, a reusable Java Beans component architecture, and database connectivity.

Customers can access detailed information about the product, including white papers, product fact sheets, and an evaluator's guide from Borland's Web site at http://www.borland.com. Open JBuilder is scheduled to ship in early 1997. Final pricing has not yet been determined.

Borland has also announced its Open JBuilder Partner Program. The program is designed to give Java developers several products and services for creating, testing, and marketing JavaBeans components to Open JBuilder customers. It will provide selected independent software vendors with pre-release versions of Open JBuilder, access to online technical support, a free copy of Open JBuilder when it's released, electronic access to the Open JBuilder development team, and contact with Borland's developer relations department. Members will also be able to participate in a number of co-marketing activities.

For more information on Borland's Partner Program for JavaBeans, visit http://www.borland.com/-internet/OpenJBuilderinfo.-html.


Open JBuilder
File Edit View Search Compile Project Run Component Tools Help
AWT Visual Containers Dialogs Data Access Layouts
Browsers: URL: latte:java//C:\LATEST\myprojects\Customer\Frame1.java
Project Customer.jpr [C:\LATEST\myprojects\Customer\Frame1.java]
Customer.jpr
Application1.java
Frame1.java
Frame1
DecoratedFrame
button1
database1
dataCursor1
Frame1
1: 1 Modified Insert
Component Inspector
Customer.Frame1
this XYLayout
label1
tabsetView1
gridControl1
button1
database1 (Database)
queryDataSet1 (QueryDataSet)
dataCursor1 (DataCursor)
Properties Events
gridControl1 x: 543 y: 84
Customer.Frame1
Customer Information Form
Contact Personal Comments Financial History
Lookup

```
// Project Expert Generated Frame
// 10/10/96 2:56:02 PM

package Customer;

// BAJA imports
import borland.baja.awt.*;
import borland.baja.layout.*;
import borland.baja.visual.*;
// AWT imports
```

emp_no	first_name	last_name	phone_ext	hire_date	dept_no	job_
2	Robert	Nelson	250	1/1/84	600	VP
4	Bruce	Young	233	2/27/87	621	Eng
5	Kim	Lambert	22	12/12/95	130	Eng
8	Leslie	Jones	410	4/3/91	180	Mktg

actionEnabled	true
actionId	0
actionState	0
actionText	&Lookup
background	lightGray
bounds	452, 53, 95, 25
class	
enabled	true
font	"Dialog", 1, 12
foreground	black
height	25*
helpContext	null
hintText	null

Borland InterBase for Windows NT on the PowerPC Platform

Scotts Valley, CA — Borland and Motorola Computer Group have released Borland InterBase 4.1 for Windows NT on the PowerPC platform.

InterBase for the PowerPC contains technology for large enterprise client/server environments, and provides lock-free transactions on the PowerPC.

InterBase 4.1 for Windows NT PowerPC is available for US$850 for five users. For more information, contact Motorola at (602) 438-3481 or Borland at (408) 431-1000.

Borland and TCIS will Deliver Delphi for AS/400

Scotts Valley, CA — Borland announced it has signed a Letter of Intent to license AS/400-compatible connectivity and development software from Traitement Cooperatif & Integration de Systeme (TCIS) of Paris, France. Under this agreement, Borland will license ClientObjects/400 and ScreenDesigner/400 from TCIS to connect Delphi Client/Server to AS/400s.

In addition, the agreement will allow Borland to resell TCIS' software with its Delphi Client/Server Suite. For more information, contact Borland at (408) 431-1000 or visit their Web site at http://www.borland.com.

Attention Winshoes Users

Phoenix Business Enterprises (PBE) has ceased dealings with NetMasters of Nashua, NH, because of a breach of contract. Registered users of Winshoes (Internet Suite, Internet Pros), should contact Shoreline Software at support@shoresoft.com to verify their product is registered.

Errors & Omissions

In the September issue of *Delphi Informant*, the winners of the 1996 RAD Race at Software Development 96 were incorrectly identified. The winners were Jason Vokes and Colin Ridgewell of Dunstan Thomas Ltd.

We apologize for any inconvenience this may have caused.

table of CONTENTS

ON THE COVER

Object Pascal / Delphi 2

By John Ayres

Delphi ROCKS!

A Fast Action Asteroids Clone

Delphi is an impressive tool for creating database applications of all types and complexities. It's excellent for creating business applications; with Delphi, programmers are more productive, creating quality applications in less time than it would take with other languages.

There is one arena however, in which Delphi has yet to make its mark. The language of choice for the software entertainment industry has long been C. Given Delphi's incredible power and ability to interface with DirectX and other gaming APIs, can Delphi compete with other languages when it comes to making state-of-the-art games? The answer is a resounding "Yes!"

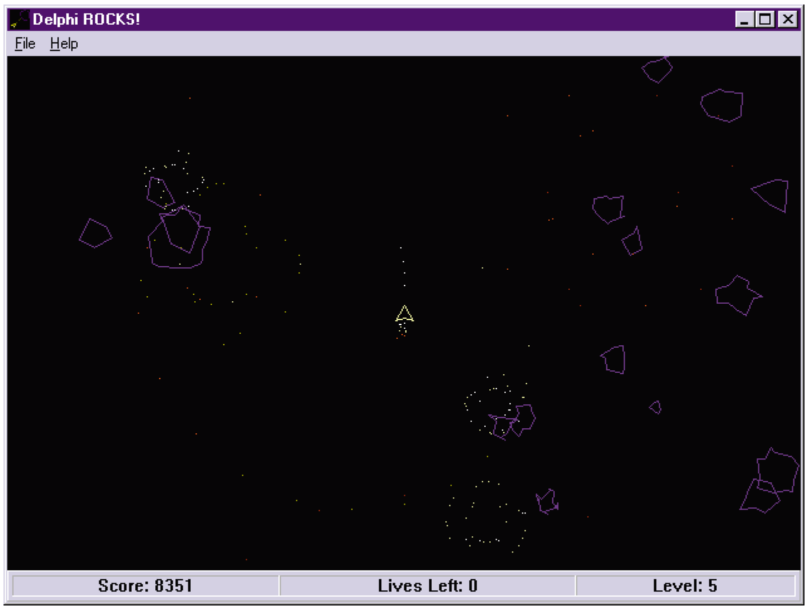
Object Pascal supports inline assembly and pointer arithmetic, probably the two most important language requirements in developing complex gaming engines. C may have a few tricks to which Delphi doesn't have a direct correlation, but with clever programming, Delphi can do anything C can. Even the Delphi development team has said that they can't manually write optimized assembly code better than the code the compiler generates.

Additionally, Delphi 2 now uses the same back-end linker as Borland C++, so the programs generated will run just as fast as anything done in C. If Delphi can do anything that C can, and C is the industry standard in the gaming industry, then Delphi should be a viable development platform for creating games.

In this article, we'll use Object Pascal and a few Windows API functions to create Delphi ROCKS!, a fast, arcade-style Asteroids clone (see Figure 1).

Although most games use assembly language and other advanced optimization techniques to improve game speed, Delphi ROCKS! uses only a few simple techniques.

Assembly language and other optimizations tend to make the code difficult to follow. By omitting these types of optimizations, the code will be more accessible to inexperienced Delphi programmers. Using only simple optimization techniques will also demonstrate just how powerful Delphi is, because it can handle 2D sprite animation and polygon graphic manipulation with reasonable speed.



Score: 8351 Lives Left: 0 Level: 5

Figure 1: Delphi ROCKS! in progress.

The Game Engine Subsystems

From a high-level standpoint, Delphi ROCKS! can be thought of as a complex animation. At every turn through the game, a new image is created and shown on the screen, similar to traditional cartoons. Therefore, we need:

1) A technique to display one frame of animation after another, flicker free and as fast as possible;
2) Some specialized trigonometric functions and rotation equations, because most of the graphics are polygons capable of rotating to any angle;
3) A way to track and control the motion, velocity, shape, and other properties of all the moving images on the screen, such as the *sprites* (the asteroids and the player's ship); and
4) A main control loop to bring these subsystems together into a cohesive, fast action game.

The animation system is probably the most important part of any game. Because the game's speed is directly affected by how fast the next frame of animation can be displayed on the screen, the animation system has to be fast, without any visible transition from one frame to the next. So we'll use an animation technique known as *double buffering*.

A double-buffering system typically has an off-screen buffer the same size as the final image to be displayed on the screen. In the main control loop, fill the off-screen buffer with black to erase the previous image; then draw each sprite's image on this off-screen buffer. Finally, copy the entire buffer to the main form (see Figure 2). This technique both erases the previous frame of animation and draws the new frame at the same time, resulting in a flickerless animation. Every image that must be displayed on the screen will first be drawn on the off-screen buffer. Although this isn't the fastest method of animation, it's more than adequate for this game.

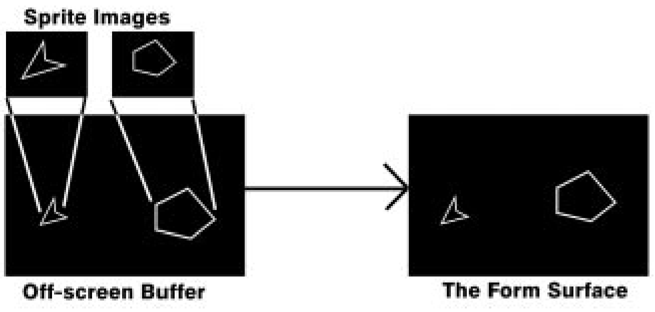


Figure 2: Illustrating the use of an off-screen buffer.

Simple Math

Most of the game's sprites will be represented by unfilled polygons that can rotate to any orientation. To accomplish this, certain trigonometric functions are required. Fortunately, you don't really need to know how they work — just that they work for the techniques described. Specifically, we'll use the sine and cosine functions to determine an X and a Y coordinate based on an angle. This coordinate will be used when determining the velocity of a sprite. We will also use two trigonometric functions to rotate a point around its origin by a specified angle. These functions will be used when rotating a polygon to a new orientation.

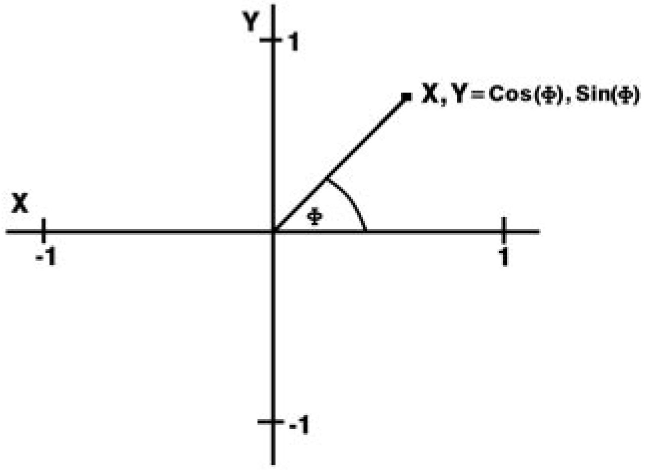


Figure 3: The relationship between sine and cosine and an X,Y coordinate.

The relationship between the sine and cosine functions and an X,Y coordinate is relatively simple. Given any angle, the X value can be determined by taking the cosine of the angle, which will be between -1 and 1. The Y value is the sine of the angle, which will also be between -1 and 1 (see Figure 3). This is really the heart of the trigonometric functions we need. In fact, this relationship alone is all we need to determine how to move a sprite in a specific direction.

Let's say we have an asteroid that's moving at a 45 degree angle. To move this sprite to its new location, we must add values to its X and Y positions. We simply take the cosine of a 45 degree angle and add this to its X position, then take the sine of a 45 degree angle and add this to its Y position. If we multiply the values returned from these functions, we can make the asteroid move even faster. Therefore, the basic sine and cosine functions will be used to determine the velocity of all the sprites in the game.

The rotation equations are simply an extension of this sine-cosine relationship. These equations will rotate an X and Y coordinate around the origin (0,0) by the given angle (see Figure 4). (Although a detailed explanation of how these equations are derived is beyond the scope of this article, rest assured that they work as described.) Given a two-dimensional point and the angle to which you want this point to be rotated, the new X and Y coordinates can be generated from the following formulas:

```
NewXPosition = X * Cosine(Angle) - Y * Sine(Angle)
NewYPosition = X * Sine(Angle) + Y * Cosine(Angle)
```

The polygons for our sprites are stored as an array of points, one for each vertex in the polygon, with the origin in the center of the sprite. When an asteroid or the player's ship needs to be rotated, we simply feed each of the points from the array to these equations, specifying the angle to which we want the new point to be rotated. Through the magic of math, we have a new polygon at the correct orientation.

Although many optimizations have been omitted (so the code would be readable), we must discuss one math-specific opti-

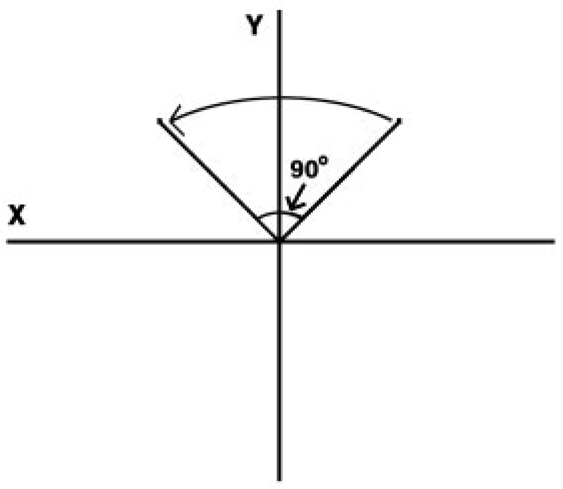


Figure 4: Equations will rotate an X and Y coordinate around the origin (0,0) by the given angle.

mization. The built-in trigonometric functions are, by their very nature, incredibly slow. You wouldn't want to use such functions in a real-time game like this, as it would slow things down considerably.

A useful optimization technique for many game elements is to employ a lookup table. Basically, for any mathematical function with inputs in a known range, the result for each set of inputs can be determined during the game initialization and stored in an array. Instead of computing the mathematical functions on-the-fly, we simply index to an array of values based on the normal inputs to the math function. This results in an incredible decrease in processor time for most functions, and will speed up code tremendously. This is what we've done for the *Sin* and *Cos* functions. There is one array for each function and each array has 360 elements, one for every angle. When we need to know the sine or cosine of any angle, we simply index to the array at that angle (e.g. `CosineArray[45]`for the cosine of a 45 degree angle) to get the appropriate value.

```
{ Moves sprites according to velocity }
procedure TSprite.MoveSprite;
begin
  { Add X and Y velocities to sprite's current position }
  XPos := XPos + XVel;
  YPos := YPos + YVel;
  { Check for screen sides }
  if XPos > 632 then XPos := 0;
  if XPos < 0   then XPos := 632;
  if YPos > 409 then YPos := 0;
  if YPos < 0   then YPos := 409;
end;

{ Accelerates the sprite within a maximum velocity }
procedure TSprite.Accelerate;
begin
  { Adjust the X velocity }
  XVel := XVel + CosineArray^[Angle];
  { Check for maximum limits }
  if XVel > 10 then
    XVel := 10
  else if XVel < -10 then
    XVel := -10;

  { Adjust the Y velocity }
  YVel := YVel+SineArray^[Angle];
  { Check for maximum limits }
  if YVel > 10 then
    YVel  := 10
  else  if YVel < -10 then
    YVel := -10;
end;
```

Figure 5: The *MoveSprite* function repositions sprites.

Probably the most efficient way to control sprites is to think of them as individual objects (again, see Figure 1). When examining the general properties of each sprite, we can conclude that each sprite will need information on its current position, its velocity, and the color in which it's drawn. The velocity will ultimately determine the direction of motion. Because we're dealing with rotatable, polygon-based graphics, we need a way to track the angle to which the polygon is rotated. We also need a way to track if the object is *alive* in the world or has been destroyed. In addition, we need methods that act on these properties, specifically a way to move the object on the screen by modifying its position according to its velocity, and a way to accelerate the object, which operates on the velocity alone. Therefore, our base sprite class will be:

```
TSprite = class
    { World coordinates relative to center of sprite }
    XPos, YPos: Real;
    { World velocities }
    XVel, YVel: Real;
    { If it's alive, it'll be moved }
    Living: Boolean;
    { The color the sprite will be drawn in }
    Color: TColor;`
    { Direction the sprite is facing }
    Angle: Integer;
    { Moves the sprite based on velocity }
    procedure MoveSprite;
    { Increases velocity }
    procedure Accelerate;
  end;
```

The *MoveSprite* function repositions sprites by simply adding the X and Y velocities to the X and Y positions, respectively, and wrapping the object to the other side of the screen if it has gone beyond the edges (see Figure 5). The *Accelerate* procedure uses the *Sin* and *Cos* functions as described earlier. Because we want to accelerate in the direction the object is facing (to simulate thrust), we simply take the cosine of the object's *Angle* property and add that to the X velocity, and the sine of the *Angle* property and add that to the Y velocity. Depending on the direction of the angle, the acceleration could be positive or negative, so we perform a check to limit the object to a positive and negative velocity range. All the sprites in the game need this basic functionality and will descend from this base class.

Ammunition and Dust

Bullets, explosion particles, and exhaust particles are all represented by a single dot on the screen, and are the simplest sprites in the game. In addition to the basic functionality, we need a way to track the lifespan of these objects, because bullets have a specific range, and explosion and exhaust particles should fade over time. For bullet objects, we need a way to track its bounding box, something that is used in collision detection (we'll discuss

```
{ Moves and draws a particle }
procedure TParticle.Draw;
var
  { A placeholder for the particle color }
  ParticleColor: Tcolor;
  { Used in determining color of particle }
  ColorIndex: Real;
begin
  { Move this particle }
  MoveSprite;
  { Decrease the LifeSpan, as another animation
    frame has gone by }
  Inc(LifeSpan, -1);
  { If the lifespan has run out, kill this particle }
  if LifeSpan = 0 then
    begin
      Living := False;
      Exit;
    end;

  { If the particle is still alive, then draw it.
    Determine where the color will fall in the
    color array }
  ColorIndex := MaxLife/5; { We have 5 colors }
  { And the particle color is ... }
  ParticleColor := FadeColors[Trunc(LifeSpan/ColorIndex)];
  { Draw this particle to our off-screen buffer }
  AsteroidForm.FOffscreenBuffer.Canvas.
    Pixels[Round(XPos),Round(YPos)] := ParticleColor;
end;
```

Figure 6: The *Draw* procedure.

this shortly). We also need to implement a procedure to draw the sprite on the off-screen buffer. Thus, our class for particles and bullets will be:

```
TParticle = class(TSprite)
    { Current life span, expressed in number of frames }
    Lifespan: Integer;
    { Particle's maximum lifespan, used to determine color }
    MaxLife: Integer;
    { Bounding box for bullets }
    ColDelta: Integer;
    { Draws the particle and decreases its life }
    procedure Draw;
  end;
```

The *Draw* procedure for this class simply calls the inherited *Move* method to update its position, decreases the *Lifespan* property, then checks to see if it's still alive. If so, *Draw* performs a calculation to see the color in which the particle or bullet should be drawn. To get the "fade out" effect, we defined a static array of five colors. The color of the particle or bullet is based on the maximum lifespan of the sprite and its current lifespan. As the current lifespan reaches its maximum lifespan, we use progressively darker colors (see Figure 6). Finally, we draw a single dot on the off-screen buffer in the appropriate color.

Lastly, we need a class to track the sprites represented with polygons. Obviously, these sprites will need a property to track the vertices of the polygon, so we need an array of points. We'll assume a maximum of 20 points, but since we won't use all of them in every sprite, we need a property to track the number of used points in the polygon. For the rotation equations to work, the points in the polygon will assume that the origin is in the center of the polygon. We will also need: a property to track the bounding box of the sprite, as used in collision detection (described later); a method to draw these objects; and a method by which to rotate them. Therefore, our basic class for polygon graphic sprites will be:

```
TPolygonSprite = class(TSprite)
    { Used to determine a bounding box relative
      to the center of the sprite }
    ColDelta: Integer;
    { The points for drawing the shape, relative
      to the center of the polygon }
    ThePolygon: array[0..19] of TRPoint;
    { The number of vertices used in ThePolygon }
    NumVertices: Integer;
    { Draws and rotates the polygon }
    procedure Draw;
    { Modifies the direction that the sprite is facing }
    procedure Rotate(Degrees: Integer);
  end;
```

As it turns out, the player's ship needs one extra property, that of a shield life, and the asteroids need one extra property, that of a rotation rate. Therefore, this class is a base from which the player's ship and asteroids descend.

The *Rotate* method is simply a way to see if the polygon went beyond a full rotation. The *Draw* procedure is where the real work is. To draw the polygon shape, we'll use a Windows API function, *PolyLine*, which takes an array of *TPoints*. Now, the property that stores the polygon is an array of *TRPoints*. The difference is that *TPoint* is a record with two members, *X* and *Y*, of type Integer, where *TRPoint* is a record with X and Y members of type Real. A point record with members of type Real was used to get a more accurate polygon rotation. For the *PolyLine* function to work, we need an array of *TPoints* that represents the polygon in the proper orientation and in the proper position on the screen. So we must iterate through each element in the array, *ThePolygon*. Each *TRPoint* in the array is rotated using the trigonometric rotation equations described earlier. The *XPos* and *YPos* of the sprite is added to the X and Y results, and these values are stored in a temporary array of *TPoints*. This final array is used in the *PolyLine* function, drawing the final, rotated polygon at the correct position in the off-screen buffer (see Figure 7).

The particle explosion and ship exhaust effects are relatively simple. As previously described, each moving dot is a sprite of type *TParticle*. An array of *TParticle* is used for each explosion and the ship exhaust. During the main control loop, a function is called that iterates through each particle in the array, moving and drawing it as needed. This is a simplistic effect based on what we've already covered.

Of course, a method for determining when one sprite contacts another is needed, specifically when a bullet hits an asteroid, and when an asteroid hits the player's ship. This is known as *collision detection*, and the *ColDelta* property of each sprite is used for this purpose. If we assume that a box surrounds the entire sprite, we can use the Windows API function *IntersectRect* to determine if these bounding boxes are touching. If they are, a collision has occurred.

```
{ Modify the sprites current facing }
procedure TPolygonSprite.Rotate(Degrees: Integer);
begin
  { Modify the angle }
  Angle := Angle + Degrees;
  { Check for boundaries }
  if Angle>359 then
    Angle := Angle - 359;
  if Angle<0    then
    Angle := 360 + Angle;
end;

{ This procedure rotates the points in the polygon to the
  current facing and draws it to the off-screen buffer }
procedure TPolygonSprite.Draw;
var
  { A Polyline compatible polygon point array }
  TempPoly: array[0..19] of Tpoint;
  { General loop control variable }
  Count: Integer;
begin
  { Rotate the polygon by the angle using point rotation
    equations from trigonometry & translate its position }
  for Count := 0 to NumVertices-1 do begin
    TempPoly[Count].X := Round((ThePolygon
      [Count].X*CosineArray^[Angle] -
      ThePolygon[Count].Y*SineArray^[Angle]) + XPos);
      TempPoly[Count].Y := Round((ThePolygon[Count].X*
      SineArray^[Angle] + ThePolygon[Count].Y*
      CosineArray^[Angle]) + YPos);
  end;

  { Adjust the pen and the brush of the
    off-screen buffers' canvas }
  AsteroidForm.FOffscreenBuffer.Canvas.Pen.Color    :=
    Color;
  AsteroidForm.FOffscreenBuffer.Canvas.Brush.Style :=
    bsClear;
  AsteroidForm.FOffscreenBuffer.Canvas.Brush.Color :=
    clBlack;
  { Draw the polygon using a Windows API function }
  Polyline(AsteroidForm.FOffscreenBuffer.Canvas.Handle,
          TempPoly,NumVertices);
end;
```

Figure 7: The *Rotate* method.

ColDelta is an offset from the center of the sprite. If you take this value and add and subtract it from the *XPos* and *YPos* values, respectively, you will have the coordinates for the sprite's bounding box. In Figure 8, you can see this technique isn't always accurate. One way to improve the accuracy is to reduce the size of the bounding rectangle so that *most* of the sprite is included. The downside to this is that sometimes a collision will happen that is not detected. However, this technique is more than accurate enough for a fast-action shooter like Delphi ROCKS!

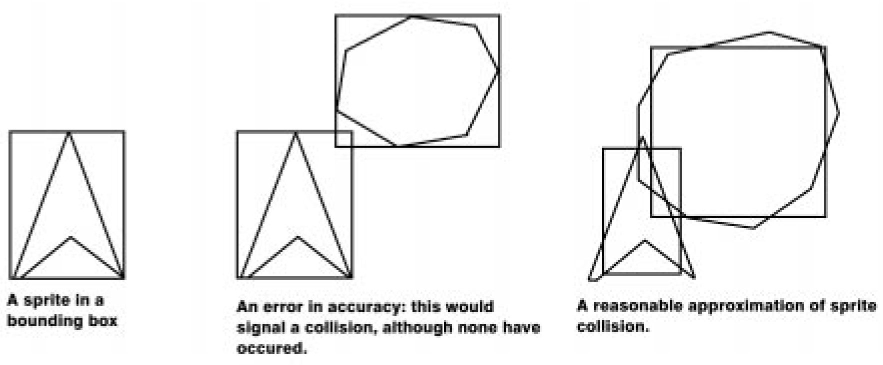

A sprite in a bounding box
An error in accuracy: this would signal a collision, although none have occured.
A reasonable approximation of sprite collision.

Figure 8: The sprites' bounding boxes.

Putting It All Together

Realistically, only a limited amount of bullets, asteroids, and explosions will be on screen at one time. Thus, one-dimensional arrays can be used to track the bullets, explosions, and asteroids.

We could have used linked lists to track sprites, but the code would become more complex. Also, as the size of the linked list increases, the game will slow down, as there are more sprites to move and more collision tests to make. Additionally, there are several general procedures for moving and starting the various sprites. Movement is easy; we simply loop through each element in the appropriate array of sprites, calling the *Move* method if its *Living* property is *True*. The logic for starting a new asteroid or bullet is also fairly simple. First, start iterating through the appropriate sprite array. At each element in the array, check the *Living* property. If it's *True*, skip this element, but if it's *False*, we can use it. We set the *Living* property to *True*, and then generate new values for the appropriate sprite.

Another crucial aspect needed in any game is a function to translate input from the user. Although mouse or joystick support could have been included for this game, we'll use the keyboard for simplicity. The *OnKey* events of the main form could have been used for user input, but these events get their input from the keyboard buffer. The result is that it cannot process multiple keys simultaneously. For instance, if the player wanted to turn and fire at the same time, the *OnKey* method might first get an arrow key, starting the turn. It would then get the spacebar, firing a bullet. The event is called for each key press, so it will only process one at a time. The keyboard repeat rate also affects the input, causing a pause if a key is held down.

Instead, through every iteration of the main control loop, we call the *ProcessUserInput* function, which is a series of *GetAsyncKeyState* API functions (see Figure 9). These functions indicate if the specified key is being depressed at the time of the function call. We can test specifically for each key that will translate to an on-screen action, thus providing the ability to perform multiple actions at once, such as turning and firing. Of course, no game would be complete without a scoring method, a count of the number of lives the player has left, and an indication as to how far in the game the player has progressed. A score is tallied when an asteroid is struck by a player's bullet, based on the size of the asteroid. This score is kept in

```
{ We use this routine instead of the OnKey events of the
  form so that we can determine if more than one key is
  being held down. This allows us to rotate, accelerate,
  and fire all at the same time. }
procedure TAsteroidForm.ProcessUserInput;
begin
  { Counterclockwise rotation }
  if (GetAsyncKeyState(VK_LEFT) and $8000) > 0 then
    PlayerShip.Rotate(-15);
  { Clockwise rotation }
  if (GetAsyncKeyState(VK_RIGHT) and $8000) > 0 then
    PlayerShip.Rotate(15);
  { Fire a missile }
  if (GetAsyncKeyState(VK_SPACE) and $8000) > 0 then
    AsteroidForm.StartMissle(PlayerShip.XPos,
      PlayerShip.YPos,PlayerShip.Angle,
      PlayerShip.XVel,PlayerShip.YVel);
  { Acceleration }
  if (GetAsyncKeyState(VK_UP) and $8000) > 0 then
    begin
      StartShipExhaustBurst;
      PlayerShip.Accelerate;
    end;
  { Turn the shields on }
  if (GetAsyncKeyState(VK_RETURN) and $8000) > 0 then
    PlayerShip.ShieldLife := SHIELDLIFESPAN;
end;
```

Figure 9: The *ProcessUserInput* procedure.

a global variable, and displayed as the caption in a panel at the bottom of the main form (again, see Figure 1). The number of lives a player has left is also tracked, with three lives being the maximum and one being lost each time the player's ship is struck by an asteroid. Finally, a variable to track the level is incremented each time all of the asteroids on the screen have been destroyed. The number of random asteroids generated at the start of each level is affected by this variable; more asteroids are created for higher levels.

The Main Control Loop

We finally have everything we need to assemble the game. From a high-level standpoint, we need a main control loop that will move all the sprites and draw each frame of animation as quickly as possible (see Listing One beginning on page 11). You will notice that this is implemented as a **while** loop. A Timer object could be used, but even with an interval of one, timers are too slow for fast animation, and a **while** loop will run as fast as the processor allows.

The main control loop can be considered to have three states:
1) the active playing state, where the player is controlling the ship and blowing away space rocks;
2) the intermission state, where the player has cleared a level and the next level is being prepared; and,
3) the demo state, a "between-games" state where some randomly generated asteroids are moved about the screen.

With this in mind, the control loop is implemented with a **case** statement, with each game state having its own series of instructions. This is the largest procedure in the program, and as such, it's too large to explain in detail. However, the pseudocode in Figure 10 shows the flow of logic for the main control loop.

The technique of controlling the game through a state-driven loop should provide the ability to easily expand or modify the

```
while (Still Playing The Game) do begin
  case GameState of
    Playing:
      begin
        if (The Player Has No More Ships) then
          GameState:=Demo
        if (There Are No More Asteroids) then
          GameState:=Intermission
          Erase The Offscreen Buffer
        if (The Player Is Still Alive) then
          Process User Input Move The Players Ship
        else
          Start A New Ship, Decreasing Number Of Lives Left
          Draw The Players Ship, Including Shields
          Draw The Ship Exhaust
          Move And Draw All Bullets
          Move And Draw All Asteroids
            And Perform Collision Detection
          Move And Draw All Explosions
          Copy The Next Frame Of Animation To The Form
          Process Any Windows Messages
    end
    Intermission:
    begin
      Clear Any Moving Sprites (Explosions,
        Exhaust, Bullets, Etc.)
      Increase The Current Level
      Erase The Offscreen Buffer
        And Display The Next Level Number
      Copy The Next Frame Of Animation To The Form
      Pause For Four Seconds
      GameState:=Playing
      Restart The Players Ship In The Middle
        Of The Playing Field
      Generate A New Set Of Asteroids
    end
  Demo:
    begin
      Erase The Offscreen Buffer
      Move And Draw Random Asteroids
      Display The Program's Title On Top Of The Asteroids
      Copy The Next Frame Of Animation To The Form
      Process Any Windows Messages
    end
  end
end
```

Figure 10: The psuedocode that shows the flow of logic for the main control loop.

overall game behavior. For example, we could easily add a bonus stage by simply plugging in another game state and a series of instructions. We could also modify a game state's behavior by rearranging the functions within that state. A few paragraphs back, I indicated that a Timer to control the speed of the game is too slow for efficient animation. How then do you manage the pacing of the game? Obviously a 166MHz machine will run the game much faster than a 75MHz machine. The solution is to have the computer *waste* a certain amount of time between each cycle in the game. This is achieved through the following code at the end of the Playing section of the *MainLoop* procedure:

```
PacingCounter := GetTickCount;
repeat
  Application.ProcessMessages;
until (GetTickCount-PacingCounter) > 50;
```

The *GetTickCount* function is part of the Windows API and returns the number of milliseconds that have elapsed since the current Windows session was started. All we do is call this function and save the result, then go into a message processing loop and don't come out until 50 milliseconds have elapsed. The *Application.ProcessMessages* call ensures that other processes in Windows get serviced so other programs remain responsive while the game is running. To make the game run faster, you would simply lower the number of milliseconds specified. Even better, you could put a game option that specifies the speed of play, with that option controlling the value in the **repeat** loop.

Room for Improvement

This is the type of game that has a lot of potential for interesting improvements. A space background with stars and galaxies instead of just dull black would really dress it up. The most obvious feature this game lacks is sound. It would add a lot to the game play to hear some explosions, the thrust of your ship, the whine of the shields going up and down, and photon blasts.

A high score feature would also be nice, to track the 10 highest scores, storing them in an .INI file between games. The original Asteroids also had an enemy ship that would come out periodically; adding this would break up the monotony of simply shooting asteroids. The original game also gave the player only a limited amount of shields, and making the shield effect last longer but depleting a maximum shield store every time would add another strategic element. Powerups could also be added to extend the maximum shield store, provide a limited invincibility, add another life, or even provide new weapons, such as fire and forget missiles that never miss their target. Add all of this, plus a multi-player capability, and you might just have a hot-selling game on your hands!

Although Delphi ROCKS! would make a good base from which to start, some optimizations are needed to make it a competitive commercial game. For starters, instead of using bitmaps for everything, device-independent bitmaps would be faster. We can directly access the bits of the image, and writing custom polygon drawing methods would certainly increase the animation speed.

Also, instead of drawing the entire off-screen buffer to the screen, we should use another animation technique, *Dirty Rectangle Animation*. Simply put, this technique copies to the screen only the combined area of the off-screen buffer that has changed since the last frame. This is generally smaller than the entire off-screen buffer, thus resulting in a faster animation as less information needs to be copied to the form surface. The use of DirectX would also improve things considerably, as you would have almost direct access to the video memory buffer, and direct control of what video resolution the user is running.

Conclusion

Although Delphi ROCKS! is nowhere near the complexity of a state-of-the-art, three-dimensional, light-sourced, first-person shooter, Delphi is capable of producing such a game, and this article will hopefully open some eyes. Delphi is just as capable of making quality games as any other development platform. Maybe, through the actions of hopeful and energetic Delphi programmers, some company will take the plunge and develop the first best-selling game written in Delphi. ∆

Further Reading

Unfortunately, most game programming books on the market these days focus on C. However, the theories and algorithms presented can be used in Delphi with little or no modification. If you would like to learn more about game programming, check out the following books:

LaMothe, A., *Black Art of 3D Game Programming* [Waite Group Press, 1995].
LaMothe, Ratcliff, Seminatore, and Tyler, *Tricks of the Game Programming Gurus* [SAMS Publishing, 1994].
Lampton, Christopher, *Flights of Fantasy* [Waite Group Press, 1993].
Norton, Michael, *Spells of Fury* [Waite Group Press, 1996].

References

The material in this article is based on a vector graphic Asteroids clone presented in the book *Teach Yourself Game Programming In 21 Days* by Andre LaMothe [SAMS Publishing, 1994].

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\97\FEB\DI9702JA.

John Ayres is currently working as a software engineer for 7th Level, creating the next generation of computer games. He is also on the board of directors for the Delphi Developers of Dallas, one of the largest Delphi users groups in the United States. John has over 7 years of programming experience in C, Assembly, and Pascal, and keeps himself busy writing three new Delphi programming books. Look for *The Delphi 97 Windows API Guide, DirectX Game Programming for Delphi 97*, and *Interface Design With Delphi 97* in the fall of next year. He can be reached on CompuServe at 102447,10.

Begin Listing One — The Main Control Loop

```
{ This is the main control loop for the entire program.
  This controls the action from a high level standpoint,
  and is itself controlled by a state variable that
  determines what should be happening. We want to do this
  inside of a regular loop as opposed to putting this on a
  timer event. This is because even with an interval of 1,
  the timer is too slow, and a loop will give us the best
  performance. }
procedure TAsteroidForm.MainLoop;
var
  LevelPauseCounter: TDateTime;
  { For timing how long the level intermission has lasted }
  Count: Integer;
  { General loop control }
begin
  while FDoLoop do
    { Continue this loop until the user closes program }
  case GameState of
    Playing:

    { We are actively playing the game }
    begin
    { If the player does not have any ships left, end the
      game and go into demo mode }
    if NumShipsLeft < 0 then EndGame;
```

```
{ If the player has killed all of the asteroids, go to
   intermission and increase the level }
   if not AnyAsteroidsMoved then
     GameState := Intermission;
   { Erase the previous frame in the off-screen buffer, so
     we can begin a new one }
   with FOffscreenBuffer.Canvas do begin
     Brush.Color := clBlack;
     Brush.Style := bsSolid;
     Fillrect(Cliprect);
   end;

   { If the player is still alive, get user input
     and move the player's ship }
   if PlayerShip.Living then
     begin
       ProcessUserInput;
       PlayerShip.MoveSprite;
     end
   else
     { We died, start us over }
     StartPlayer(TRUE);
   { Draw the player's ship to the off-screen buffer }
   PlayerShip.Draw;
   { If shields are on, draw them around the player's ship}
   if (PlayerShip.ShieldLife > 0) then
     with FOffscreenBuffer.Canvas do begin
       Pen.Color    := clGreen;

       Brush.Style := bsClear;
       { Shields are represented by a simple circle }
       Ellipse(Round(PlayerShip.XPos -
                     PlayerShip.ColDelta - 5),
               Round(PlayerShip.YPos -
                     PlayerShip.ColDelta - 5),
               Round(PlayerShip.XPos +
                     PlayerShip.ColDelta + 5),
               Round(PlayerShip.YPos +
                     PlayerShip.ColDelta + 5));
       { Decrease the shield life span }
       Inc(PlayerShip.ShieldLife, -1);
     end;

   { Draw ship exhaust effect }
   DrawShipExhaust;
   { Move all active missiles }
   MoveMissles;
   { Move all asteroids and check for collisions }
   MoveAsteroids;
   { Draw any explosions that have just occurred }
   DrawExplosions;
   { Copy the next frame to the screen }
   AsteroidForm.Canvas.Draw(0, 0, FOffscreenBuffer);
   { Display Score Changes }
   Panel2.Caption := 'Score: ' + IntToStr(Score);
   { Process any pending Windows messages for 50
     milliseconds }
   PacingCounter := GetTickCount;
   repeat
     Application.ProcessMessages;
   until (GetTickCount-PacingCounter) > 50;
 end;

 Intermission:
   { This does a slight pause in between levels }
 begin
   { Kill any moving sprites }
   ClearAll;
   { Increase the level }
   Inc(CurLevel);
```

```
{ Erase the former frame, so we can begin a new one }
   with FOffscreenBuffer.Canvas do begin
     Brush.Color := clBlack;
     Brush.Style := bsSolid;
     Fillrect(Cliprect);
   end;

   {  Draw the level on the off-screen buffer }
   with FOffscreenBuffer.Canvas do begin
     SetTextAlign(Handle, TA_CENTER);
      Font.Name    := 'Arial';
     Font.Size    := 30;
     Font.Color   := clRed;
     Font.Style   := [fsBold];
     Brush.Style := bsClear;
     { Display the text centered in the off-screen buffer
}
     TextOut(FOffscreenBuffer.Width  div 2,
            (FOffscreenBuffer.Height div 2)-
            (TextHeight(‘ABC’) div 2),'LEVEL' +
             IntToStr(CurLevel));
   end;

   { Copy the next frame to the screen }
   AsteroidForm.Canvas.Draw(0, 0, FOffscreenBuffer);
   { Process any ending Windows messages }
   Application.ProcessMessages;
   { Show this intermission for approximately 4 seconds }
   LevelPauseCounter := Time;

   repeat
     Application.ProcessMessages;
   until (Time-LevelPauseCounter) > 0.00004;
   { Intermission is over, we are actively playing
     the game again }
   GameState := Playing;
   { Start the player in the middle of the screen, with
     shields on }
   StartPlayer(FALSE);
   { Display the game values }
   Panel2.Caption := 'Score: ' + IntToStr(Score);
   if NumShipsLeft>-1 then
     { Don't want to display a negative amount of ships }
     Panel3.Caption := 'Lives Left: ' +
                       IntToStr(NumShipsLeft);
   Panel4.Caption := 'Level: ' + IntToStr(CurLevel);
   { Now, generate some new asteroids,
     based on our current level }
   for Count := 0 to Random(5)+CurLevel do
     StartAsteroid(Random(631),Random(408),Random(359),
       Random(10) + 20);
   { This must be set so we enter the main playing loop }
   AnyAsteroidsMoved := True;
 end;
 Demo:
   { We are not playing the game, so let's show some
     general animation }
 begin
   { Erase the previous frame, so we can begin a new one }
   with FOffscreenBuffer.Canvas do begin
     Brush.Color := clBlack;
     Brush.Style := bsSolid;
     Fillrect(Cliprect);
   end;

   { Move the random asteroids }
   MoveAsteroids;
   { Draw a message on the off-screen buffer }
   with FOffscreenBuffer.Canvas do begin
```

```
      SetTextAlign(Handle, TA_CENTER);
      Font.Name    := 'Arial';
      Font.Size    := 30;
      Font.Color   := clRed;
      Font.Style   := [fsBold];
      Brush.Style := bsClear;
      { Display the text centered in the off-screen buffer }
      TextOut((FOffscreenBuffer.Width div 2) -
              (TextWidth('Delphi') div 2),
              (FOffscreenBuffer.Height div 2) -
              (TextHeight('ABC')div 2),'DELPHI');
      Font.Name   := 'Times New Roman';
      Font.Style := [fsBold,fsItalic];
      TextOut((FOffscreenBuffer.Width div 2) +
              (TextWidth('ROCKS!') div 2) + 23,
              (FOffscreenBuffer.Height div 2) -
              (TextHeight('ABC')div 2) - 1,'ROCKS!');
    end;

    { Copy the next frame to the screen }
    AsteroidForm.Canvas.Draw(0,0,FOffscreenBuffer);
    { Process any pending Windows messages }
    Application.ProcessMessages;
  end;
  end;
end;
```

End Listing One

Delphi / Object Pascal

By Keith Wood

A Game of Decode

Developing a "Mastermind" Clone in Delphi

Writing a game with Delphi isn't just an exercise in frivolity. It's a lesson in DrawGrids, StringGrids, resource files, random numbers, drag-and-drop, and .INI files — all of which are useful outside of games, but more fun to learn while developing one.

The Game

The game we'll develop is based on Mastermind — a game in which players decipher a hidden pattern of colored pegs by proposing solutions that are automatically scored. The scoring informs players how many pegs match exactly, and how many pegs are the right color, but in the wrong position. From this information, players try to solve the hidden pattern (see Figure 1).

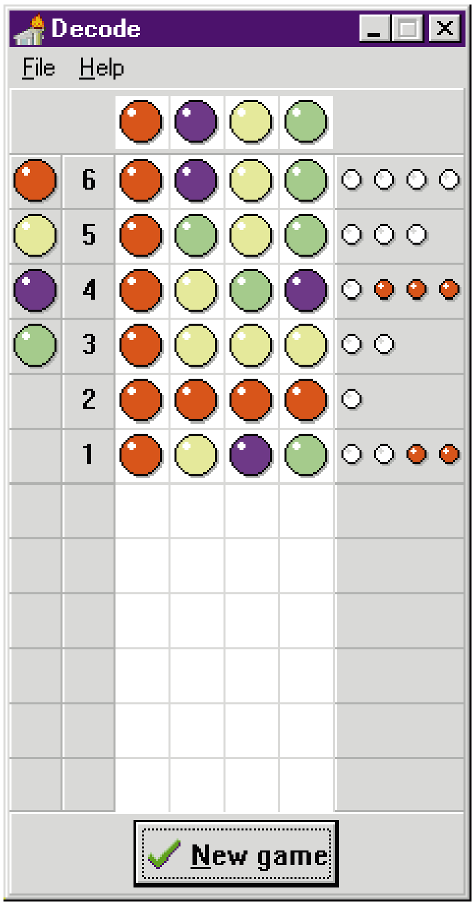


Figure 1: A completed game of Decode.

To add variation to the game, players can alter the size of the hidden pattern, the number of colors that can be used and/or repeated, and the maximum number of solution attempts. Preset combinations are also available. We'll also want to track the best scores and preferred configurations. Both are achieved using an .INI file.

First Steps

The playing board consists of several regions: the solution; the pegs available for placement; the turn numbers; the board showing the attempts; the scoring display; and an end-of-turn button. Because the window changes size when the parameters of the game are altered, these regions are controls that correctly size themselves. The solution and end-of-turn button reside on panels aligned to the top and bottom respectively.

The remaining areas are DrawGrids or StringGrids, generally aligned with the left or right edges of the form. In each turn, players drag or double-click available pegs onto the board. After all the spots are filled, the turn is completed by pressing the **Done** button. The guess is scored and shifted down a row, unless the puzzle was solved. This allows the current "move" to remain close to the solution for easy comparison at the end of the game.

Separate forms are required to enable us to alter the game's parameters and display the best scores. We'll also add an About dialog box.

Icon Resources

To make the game a "good" Windows program, we set the form's colors based on the Delphi constants that represent the player's preferences. This means that the background on which we draw the pegs is unknown as we write the game. If we simply draw the peg images (as bitmaps) over this, the backgrounds may not match. So we'll use icons that have a transparent color already defined and blend in with the color scheme in use.

We could load the icons into the Image components on the main form and use them from there. A better way, however, is to use a resource file. This contains all the images required for the colored pegs that make up our guesses and the solution, as well as the scoring pegs. (The file is easily created using Delphi's Image Editor.)

To get these into the program, we must use an API call, *LoadIcon*, which takes as parameters a handle for an executable containing the icon and the name of that icon. Delphi provides the handle to our application in the form of the *HInstance* variable. The function returns a handle to the icon, or zero if no handle could be found. To use these in the program, we declare an array of *TIcon* and create them before assigning their handles to those returned by the function call. See Figure 2 for the code that loads the peg icons.

Of course, we must include the resource file in the program before we can access it like this. The `$R` compiler directive does the job. It's already being used in our program to include the resource file for the form itself, the .DFM file, and we can add our extra file following it.

Unfortunately, to work with both Delphi 1 and 2, we must have two resource files, 16- and 32-bit respectively. The inclusion of the appropriate file can be automatically handled with conditional directives and the pre-defined conditional symbol *WIN32*, which is only defined for Delphi 2:

```
{$IFDEF WIN32}
{$R DECODE32.RES}
  { Icons/cursor for this application - 32 bit }
{$ELSE}
{$R DECODE16.RES}
  { Icons/cursor for this application - 16 bit }
{$ENDIF}
```

A New Cursor

For the drag-and-drop features described later, we want to indicate that a drag is in operation. As expected, we use the *DragCursor* property of the appropriate component to change the cursor when it's dragging. The default is the *crDrag* cursor, an arrow with a page attached. In our case, however, we're dragging pegs around, so it would be nice to have the cursor reflect this.

```
{ Load peg icons from resource file }
for i := 1 to iMaxColours do begin
  icnPegs[i] := TIcon.Create;
  try
    StrPCopy(pIcon, 'PEG' + IntToStr(i));
    icnPegs[i].Handle := LoadIcon(HInstance, pIcon);
    if icnPegs[i].Handle = 0 then
      raise Exception.Create
        ('Icon PEG' + IntToStr(i) + 'not found');
  except on e: Exception do
    begin
      MessageDlg('Error in loading icon '+ IntToStr(i)
                 + #13#10 + e.Message, mtError, [mbOK],
                 0);
     StrDispose(pIcon);
     Close;
    end;
  end;
end;
```

Figure 2: Loading icons from a resource file.

```
{ Determine new code }
if bRepeats then
  { Repeats of colours allowed }
  for i := 1 to iPegs do
    iSolution[i] := Random(iColours) + 1
      { Select from all available colours }
else
  begin
    { No repeats allowed }
    for i := 1 to iColours do
      { Initialise with all colours }
      iWorkColour[i] := i;
    for i := iColours downto 2 do begin
      j := Random(i) + 1;
      { Randomly select a colour from remainder }
      k := iWorkColour[i];
      { And swap with the current one }
      iWorkColour[i] := iWorkColour[j];
      iWorkColour[j] := k;
    end;
    for i := 1 to iPegs do
      { Then copy into solution }
      iSolution[i]   := iWorkColour[i];
  end;
```

Figure 3: Randomizing the hidden pattern.

Delphi maintains an array of cursors available for use in the *Cursors* property of the *Screen* object. The values we used to refer to these cursors, such as *crDrag*, are actually the indexes into this array. New cursors can be added to the array, then used like the standard ones.

First, we draw the cursor and place it in our resource file, along with the icons. Then we define a constant to be used as the index for our cursor, making sure it doesn't conflict with any existing constants. Standard constants are indexed by zero or a negative number, so any positive value should do:

```
const
  crDragPeg = 1; { Cursor when dragging a peg }
```

Next, load the cursor from the resource file into the *Cursors* array. This is done similarly to the previous loading of the icons. Finally, assign the new cursor, via its index, to the appropriate components:

```
Screen.Cursors[crDragPeg] :=
  LoadCursor(HInstance,'DRAG_PEG');
drgBoard.DragCursor := crDragPeg;
```

Getting Random

The game wouldn't be interesting if it always generated the same pattern, so we use Delphi's random number generator to introduce the required variation. Even then, however, the generator produces exactly the same sequence of values every time the program is run. It's necessary to initialize the generator so it doesn't repeat itself. This is done in the creation of the main form as follows:

```
Randomize;
```

Every time a new game is started, we need to establish a new pattern to be deciphered. Now, we give the player the option of having colors repeat, and this drastically affects how we generate the solution. If repeats are allowed, we simply select any of the available colors at random in each position, regardless of what happens in any other position. We use the *Random* function, which returns a random integer in the range of zero to the parameter value minus one.

If repeats aren't allowed, we must randomize the sequence of available colors before making our selection, ensuring that at most, one of each color appears in the solution. One way is to set up an array of all the possible values. This array is then randomized by stepping through it and picking an entry in the remainder of the array to swap with the current one, including itself. When completed, we can copy the first few elements into the final solution and be assured they have been selected randomly without repetition (see Figure 3).

Drag-and-Drop

The pegs can be placed using drag-and-drop. This implementation involves several steps. First, we must put the control into drag mode when the player clicks and drags the mouse over it. This can be done automatically by setting the *DragMode* property to *dmAutomatic*. Because we only want to be able to drag valid pegs and allow the player to double-click on a peg to put it into play, we can't take this route. Instead we must capture the *OnMouseDown* events themselves and start the dragging from there.

In the *OnMouseDown* event, we check that the left mouse button has been pressed, and if so, determine which cell was selected. The coordinates are saved in the *ptOrig* variable because we need to know them again when the peg is dropped. If a valid peg has been chosen, then we begin the dragging with the *BeginDrag* method. The parameter to this method determines whether the dragging starts immediately. In this case, we don't want it to start right away, so that we can respond to double-clicks as well. Finally, a flag is set to indicate that this cell should be highlighted when redrawn during the drag.

The next step is to allow the dragged peg to be dropped somewhere. The *OnDragOver* event for the target does this. It takes as parameters the source control from which we are dragging, the current mouse coordinates, the drag state, and a flag indicating whether this control accepts the dragged object. This last value is set to show that the dragged object can be dropped here. In our case, we only accept pegs from the pegs display or elsewhere on the main game board.

```
{ Save starting point for drag }
procedure TfmMain.drgPegsMouseDown(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y:
  Integer);
var
  iCol, iRow: LongInt;
begin
  if bGameOver then
    { Do nothing }
    exit;

  if Button = mbLeft then
    begin
      ptOrig := Point(X, Y);
      drgPegs.MouseToCell(X, Y, iCol, iRow);
      if iRow < iColours then
        { Peg selected }
        begin
          drgPegs.BeginDrag(False);
          bDragPeg := True;
        end;
    end;
end;

{ Accept a dragged peg }
procedure TfmMain.drgBoardDragOver(Sender, Source: TObject;
  X, Y: Integer; State: TDragState; var Accept: Boolean);
begin
  Accept := not bGameOver and (Source is TDrawGrid) and
            (TDrawGrid(Source).Name = 'drgPegs') or
            (TDrawGrid(Source).Name = 'drgBoard'));
end;

{ And make the move }
procedure TfmMain.drgBoardDragDrop(Sender, Source: TObject;
  X, Y: Integer);
var
  iCol, iRow, iPegCol, iPegRow: LongInt;
begin
  drgBoard.MouseToCell(X, Y, iCol, iRow);
  TDrawGrid(Source).MouseToCell(ptOrig.X, ptOrig.Y,
                                iPegCol, iPegRow);
  if TDrawGrid(Source).Name = 'drgPegs' then
    { From pegs }
    iAttempts[1, iCol + 1] := iPegRow + 1
  else if (iPegRow <> iRow) or
       (iPegCol <> iCol) then
    { From board }
    begin
      iAttempts[1, iCol + 1] :=
        iAttempts[iPegRow + 1,iPegCol + 1];
      if (iPegRow = 0)      and
         (iPegCol <> iCol) then
        iAttempts[1, iPegCol + 1] := 0;
    end;
  CheckDone;
end;

{ Finished with selection }
procedure TfmMain.drgEndDrag(Sender, Target: TObject;
  X, Y: Integer);
begin
  bDragPeg   := False;
  bDragBoard := False;
  drgBoard.Invalidate;
  drgPegs.Invalidate;
end;
```

Figure 4: Implementing drag-and-drop from the pegs grid to the main game board.

When the player releases the mouse button, we must make the indicated play. This is achieved through the *OnDragDrop* event for the target. We determine which cell was originally dragged from the *ptOrig* variable, and, based on its source, either copy or move the specified peg onto the board in the current column. The *CheckDone* procedure enables the **Done** button if all the spots in the current turn are filled.

Finally, the original control receives an *OnEndDrag* event, allowing it to tidy up. Here we reset the dragging flags so that no cells are highlighted, and redraw the peg and main board grids to reflect their new configuration. This event fires even when the object is dropped somewhere that doesn't accept it. Code for the entire drag-and-drop process is shown in Figure 4.

DrawGrids and StringGrids

The game's visual interface is comprised of DrawGrid and StringGrid components that display the available pegs, the turn number, the guesses so far, the score, and the solution.

By default, the DrawGrids only display the background color and any grid lines requested. To make them do more, we must set their *DefaultDrawing* property to *False* and attach the drawing code to the *OnDrawCell* event.

This event is then called for each cell as it is drawn, passing the coordinates of the current cell, the rectangle in which to draw within the grid, and the state of the cell — whether it's focused, selected and/or fixed.

Looking at the main playing board, we want to draw the appropriate peg icon in each cell. First we fill the cell's rectangle with the background color, and if a peg exists in this cell, we draw the icon for that peg over the top. The positioning of the pegs is kept in the *iAttempts* array — a zero indicates a blank cell, and a positive value indicates that a peg is present.

Note that the cells of the DrawGrid start their numbering at zero, both vertically and horizontally. The code for drawing the playing board cells is shown in Figure 5.

Additionally, we want to highlight the selected cell when it is dragged (as a reminder of which one we picked). This is achieved by combining the state of the cell with a flag indicating whether we are currently dragging. The cell is highlighted by changing the background color to *clBtnShadow* instead of the default *clWindow*. The code for the remaining DrawGrids is similar.

The StringGrid is designed to display the text held in its *Cells* property in the grid without our intervention. This is fine, provided we only want left-justified text.

To generate any other formatting, we must take over the drawing of the cells, just as we did for the DrawGrids. As before, we set the *DefaultDrawing* property to *False* and respond to the *OnDrawCell* events.

```
{ Draw the pegs placed so far }
procedure TfmMain.drgBoardDrawCell(Sender: TObject;
  Col, Row: Longint; Rect: TRect; State: TGridDrawState);
begin
  with drgBoard.Canvas do begin
    if(gdSelected in State)   and
       bDragBoard             then
       Brush.Color := clBtnShadow
    else
      Brush.Color := clWindow;
    FillRect(Rect);
    if iAttempts[Row + 1, Col + 1] <> 0 then
      Draw(Rect.Left + 1, Rect.Top + 1,
           icnPegs[iAttempts[Row + 1, Col + 1]]);
  end;
end;
```

Figure 5: Drawing a cell in a draw grid.

.INI File Sections

As mentioned earlier, we're keeping track of our preferred configuration, along with the best scores, in an .INI file. The former is standard .INI file manipulation, whereas the latter is a bit different: we don't know ahead of time what the entries will be. A separate section is required for the best scores to allow us to deal with them more easily.

The entries record the combination of parameters used in the game, the number of turns taken to solve it, and the name of the player who achieved this feat. Each entry in this section has the following format:

```
PppCccRr=nnxxx...
```

where *pp* is the number of pegs in the solution; *cc* is the number of colors; *r* is a "Y" or "N" indicating if repeats are allowed; *nn* is the number of turns taken, and *xxx...* is the name of the player who achieved this score. For example, the value:

```
P05C06RY=07Keith
```

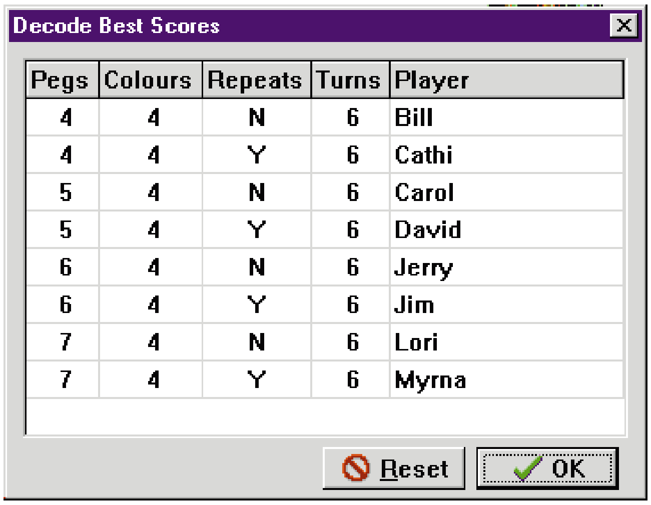
indicates that Keith took seven guesses to solve a puzzle with five pegs and six colors, and that repeats were allowed.

On the best scores form, we want to read all the existing entries and display the results in a StringGrid (see Figure 6). First, we must find out what the current values are, which is done with this statement:

```
IniFile.ReadSection(sScores, slScores);
```

This loads all the identifiers in the nominated section into the specified string list. The string list has its *Sorted* property set to *True*, so the entries are automatically placed in a meaningful order.

We then step through each of these identifiers and extract their corresponding value from the .INI file as usual. Both the identifier and its value are broken up and loaded into the



Pegs	Colours	Repeats	Turns	Player
4	4	N	6	Bill
4	4	Y	6	Cathi
5	4	N	6	Carol
5	4	Y	6	David
6	4	N	6	Jerry
6	4	Y	6	Jim
7	4	N	6	Lori
7	4	Y	6	Myrna

Figure 6: Displaying the best scores in a string grid.

grid's next row for display. The code for this is shown in Figure 7.

The **Reset** button on the scores form clears all the current best scores, allowing new champions to emerge. Existing entries in the .INI file are easily deleted using:

```
IniFile.EraseSection(sScores);
```

Additional code then tidies up the StringGrid to reflect the loss of these details.

Conclusion

Delphi enables us to write all sorts of programs, from multi-user database applications to single-user utilities, from mission-critical systems to amusements. More than just fun, games have their uses in exploring different aspects of Delphi we may not normally investigate. ∆

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\97\FEB\DI9702KW.

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.

```
{ Load the best scores from the .INI file, section
  [SCORES]. Entries expected in format PppCccRr=nnxxx...,
  where pp is number of pegs, cc is number of colours,
  r is repeats flag, nn is number of turns,
  and xxx... is player's name. }
procedure TfmScores.FormCreate(Sender: TObject);
var
  slScores: TStringList;
  i, j, k: Integer;
  sValue: string;
begin
  slScores := TStringList.Create;
  with stgScores do
    try
      try
        { Read game parameters from .INI file }
        slScores.Sorted := True;
        fmMain.IniFile.ReadSection(sScores, slScores);
      except
        { Ignore }
      end;

      { Set grid size }
      k := 1;
      if slScores.Count = 0 then
        RowCount := 2
      else
        RowCount := slScores.Count + 1;

      { And headers }
      Cells[0,0] := 'Pegs';
      Cells[1,0] := 'Colours';
      Cells[2,0] := 'Repeats';
      Cells[3,0] := 'Turns';
      Cells[4,0] := 'Player';

      { Load best score details and extract }
      for i := 0 to slScores.Count - 1 do begin
        sValue := fmMain.IniFile.ReadString
                    (sScores,slScores[i],'');
        try
          Cells[0,k] := Format('%2d',
                          [StrToInt(Copy(slScores[i],2,2))]);
          Cells[1,k] := Format('%2d',
                          [StrToInt(Copy(slScores[i],5,2))]);
          Cells[2,k] := Copy(slScores[i], 8, 1);
          Cells[3,k] := Format('%2d',
                          [StrToInt(Copy(sValue,1,2))]);
          Cells[4,k] := Copy(sValue,3,Length(sValue) - 2);
          Inc(k);
        except
          for j := 0 to 4 do
            Cells[j, k] := '';
        end;
      end;
    finally
      slScores.Free;
    end;
end;
```

Figure 7: Loading the best scores from an .INI file section.

By Danny Thorpe

Smart Linking

Virtual Methods and Polymorphism: Part II

Last month, we explored the magic of polymorphism and its Object Pascal implementation, the virtual method. We discovered that the indicator of which virtual method to invoke on the instance data is stored in the instance data itself.

This month, we conclude our exploration with a discussion of abstract interfaces and how virtual methods can defeat *and* enhance "smart linking."

Abstract Interfaces

An abstract interface is a class type that contains no implementation and no data — only abstract virtual methods. Abstract interfaces allow you to completely separate the user of the interface from the implementation of the interface.

And I do mean completely separate; with abstract interfaces, you can have an object implemented in a DLL and used by routines in an .EXE, just as if the object were implemented in the .EXE itself. Abstract interfaces can bridge:

- conceptual barriers within an application,
- logistical barriers between an application and a DLL,
- language barriers between applications written in different programming languages, and
- address space barriers that separate Win32 processes.

In all cases, the client application uses the interface class just as it would any class it implemented itself.

Let's now take a closer look at how an abstract interface class can bridge the gap between an application and a DLL. (By the way, abstract interfaces are the foundation of OLE programming.)

Importing Objects from DLLs: The Hard Way. If you want an application to use a function in a DLL, you must create a "fake" function declaration that tells the compiler what it needs to know about the parameter list and result type of the function. Instead of a method body, this fake function declaration contains a reference to a DLL and function name. The compiler sees these and knows what code to generate to call the proper address in the DLL at run time.

To have an application use an object that's implemented in a DLL, you could do essentially the same thing, declaring a separate function for each object method in the DLL. As the number of methods in the DLL object increases, however, keeping track of all those functions will become a chore. To make things a little easier to manage, you could set up the DLL to give you (the client application) an array of function pointers that you would use to call any of the DLL functions associated with a particular DLL class type.

You can see where this is headed. A Virtual Method Table (VMT) is precisely an array of function pointers (we discussed the VMT last month). Why do things the hard way when the compiler can do the dirty work for you?

Importing Objects from DLLs: The Smart Way. The client module (the application) requires a class declaration that will make the compiler "visualize" a VMT that matches the desired DLL's array of function pointers.

Enter the abstract interface class. The class contains a hoard of **virtual; abstract;** method declarations in the same order as the functions in the DLL's array of function pointers. Of course, the abstract method declarations need parameter lists that match the DLL's functions exactly.

Now you can fetch the array of function pointers from the DLL and typecast a pointer to that array into your application's abstract interface class type. (Okay; it actually needs to be a pointer to a pointer to an array of function addresses. The first pointer simulates the object instance, the second pointer simulates the VMT pointer embedded in the instance data, but who's counting?)

With this typecast in place, the compiler will think you have an instance of that class type. When the compiler sees a method call on that typecast pointer, it will generate code to push the parameters on the stack, then look up the *n*th virtual method address in the "instance's VMT" (the pointer to the function table provided by the DLL), and call that address. Voilà! Your application is using an "object" that lives in a DLL as easily as one of its own classes.

Exporting Objects from DLLs. Now for the flip side. Where does the DLL get that array of function pointers? From the compiler, of course! On the DLL side, create a class type with **virtual** methods with the same order and parameter lists as defined by the "red-herring" array of function pointers, and implement those methods to perform the tasks of that class. Then implement and export a simple function from the DLL that creates an instance of the DLL's class and returns a pointer to it. Again, Voilà! Your DLL is exporting an object that can be used by any application that can handle pointers to arrays of function addresses. Also known as objects!

Abstract Interfaces Link User and Implementor. Here's the clincher. How do you guarantee that the order and parameter lists of the methods in the application's abstract interface class exactly match the methods implemented in the DLL?

Simple. Declare the DLL class as a *descendant* of the abstract interface class used by the application, and override all the abstract virtual methods. The abstract interface is shared between the application and the DLL; the implementation is contained entirely within the DLL.

Abstract Interfaces Cross Language Boundaries. This can also be done between modules written in different languages. The Microsoft Component Object Model (COM) is a language-independent specification that allows different programming languages to share objects as just described. At its core, COM is simply a specification for how an array of function pointers should be arranged and used. COM is the foundation of OLE.

Since Delphi's native class type implementation conforms to COM specifications, no conversion is required for Delphi applications to use COM objects, nor is any conversion required for Delphi applications to expose COM objects for other modules to use.

Of course, when dealing with multiple languages, you won't have the luxury of sharing the abstract interface class between the modules. You'll have to translate the abstract interface class into each language, but this is a small price to pay for the ability to share the implementation.

The Delphi IDE is built entirely upon abstract interfaces, allowing the IDE main module to communicate with the editor and debugger kernel DLLs (implemented in BC++), and with the multitude of component design-time tools that live in the component library (CMPLIB32.DCL) and installable expert modules.

Virtuals Defeat Smart Linking

When the Delphi compiler/linker produces an .EXE, the procedures, variables, and static methods that are not referenced by "live" code (code that is actually used) will be left out of the .EXE file. This process is called *smart linking,* and is a great improvement over normal linkers that merely copy all code into the .EXE regardless of whether it's actually needed. The result of smart linking is a smaller .EXE on disk that requires less memory to run.

> **Smart Linking Rule for Virtuals.** *If the type information of a class is touched (for example, by constructing an instance) by live code, all the virtual methods of the class and its ancestors will be linked into the .EXE, regardless of whether the program actually uses the virtual methods.*

For the compiler, keeping track of whether an individual procedure is ever used in a program is relatively simple; figuring out whether a virtual method is used requires a great deal more analysis of the descendants and ancestors of the class. It's not impossible to devise a scheme to determine if a particular virtual method is never used in any descendants of a class type, but such a scheme would certainly require a lot more CPU cycles than normal smart linking, and the resulting reduction in code size would rarely be dramatic. For these reasons (lots of work, greatly reduced compile/link speed, and diminishing returns), adding smart linking of virtual methods to the Delphi linker has not been a high priority for Borland.

If your class has a number of utility methods that you don't expect to use all the time, leaving them static will allow the smart linker to omit them from the final .EXE if they are not used by your program.

Note that including virtual methods involves more than just the bytes of code in the method bodies. Anything that a virtual method uses or calls (including static methods) must also be linked into the .EXE, as well as anything those routines use, etc. Through this cascade effect, one method could potentially drag hundreds of other routines into the .EXE, sometimes at a cost of hundreds of thousands of bytes of additional code and data. If most of these support routines are used only by your unused virtual method, you have a lot of deadwood in your .EXE.

```
type
  TBaseGadget = class
    constructor Create;
    procedure Whirr; virtual;      { Linked in: YES }
  end;

  TOfficeGadget = class(TBaseGadget)
    procedure Whirr; override;     { Linked in: NO }
    procedure Buzz;                { Linked in: NO }
    procedure Pop; virtual;        { Linked in: NO }
  end;

  TKitchenGadget = class(TBaseGadget)
    procedure Whirr; override;     { Linked in: YES }
  end;

  TOfficeManager = class
   private
     FOfficeGadget: TOfficeGadget;
   public
     procedure InstantiateGadget;  { Linked in: NO }
     { Linked in: YES }
     procedure Operate(AGadget: TOfficeGadget); virtual;
 end;

{ ... Non-essential code omitted ... }
procedure TOfficeManager.InstantiateGadget;
begin  { Dead code, never called }
  FOfficeGadget := TOfficeGadget.Create;
end;

procedure TOfficeManager.Operate(AGadget: TOfficeGadget);
{ Live code, virtual method of a constructed class }
begin
  AGadget.Whirr
end;

var
  X: TBaseGadget;
  M: TOfficeManager;
begin
  X := TKitchenGadget.Create;
  M := TOfficeManager.Create;

  X.Free;
  M.Free;
end.
```

Figure 1: Inverse **virtual** smart linking: *TOfficeGadget.Whirr* will not be linked into this program, although *Whirr* is touched by the live method *TOfficeManager.OperateGadget*.

The best general strategy to keep unused virtual methods — and their associated deadwood — under control, is to declare virtual methods sparingly. It's easier to promote an existing static method to virtual when a clear need arises, rather than trying to demote virtual methods down to statics at some late stage of your development cycle.

Virtuals Enhance Smart Linking

Smart linking of virtuals is a two-edged sword: What is so often cursed for bloating executables with unused code can also be exploited to greatly reduce the amount of code in an executable in certain circumstances — even beyond what smart linking could normally achieve with ordinary static methods and procedures. The key is to turn the smart linking rule for virtuals inside out:

> **Inverse Smart Linking Rule for Virtuals.** *If the type information of a class is not touched by live code, then none of that class' virtual methods will be linked into the executable. Even if those virtual methods are called polymorphically by live code!*

In a virtual method call, the compiler emits machine code to grab the VMT pointer from the instance data, and to call an address stored at a particular offset in the VMT. The compiler can't know exactly which method body will be called at run time, so the act of calling a virtual method does not cause the smart linker to pull any method bodies corresponding to that virtual method identifier into the final executable.

The same is true for **dynamic** methods. The act of constructing an instance of the class is what cues the linker to pull in the virtual methods of that particular class and its ancestors. This saves the program from the painful death that would surely result from calling virtual methods that were not linked into the program. After all, how could you possibly call a virtual method of an object instance defined and implemented in your program if you did not first construct said instance? The answer is: you can't. If you obtained the object instance from some external source, e.g. a DLL, then the virtual methods of that instance are in the DLL, not your program.

So, if you have code that calls virtual methods of a class that is never constructed by routines used in the current project, none of the code associated with those virtual methods will be linked into the final executable.

The code in Figure 1 will cause the linker to pull in all the virtual methods of *TKitchenGadget* and *TOfficeManager*, because those classes are constructed in live code (the main program block), and all the virtual methods of *TBaseGadget*, because it's the ancestor of *TKitchenGadget*.

Because *TOfficeManager.Operate* is virtual, its method body is all live code (even though *Operate* is never called). Therefore, the call to *AGadget.Whirr* is a live reference to the virtual method *Whirr*. However, *TOfficeGadget* is not constructed in live code in this example —*TOfficeManager.InstantiateGadget* is never used. Nothing of *TOfficeGadget* will be linked into this program, even though a live routine contains a call to *Whirr* through a variable of type *TOfficeGadget*.

Variations on a Theme. Let's see how the scenario changes with a few slight code modifications. The code in Figure 2 adds a call to *AGadget.Buzz* in the *TOfficeManager.Operate* method. Notice that the body of *TOfficeGadget.Buzz* is now linked in, but *TOfficeGadget.Whirr* is still not. *Buzz* is a static method, so any live reference to it will link in the corresponding code, even if the class is never constructed.

The code in Figure 3 adds a call to the static method *TOfficeManager.InstantiateGadget*. This brings the construction of the *TOfficeGadget* class into the live code of the program, which brings in all the virtual methods of *TOfficeGadget*, including *TOfficeGadget.Whirr* (which is called by live code) and *TOfficeGadget.Pop* (which isn't). If you deleted the call to *AGadget.Buzz*, the *TOfficeGadget.Buzz* method would become

```
type
  TBaseGadget = class
    constructor Create;
    procedure Whirr; virtual;       { Linked in: YES }
  end;

  TOfficeGadget = class(TBaseGadget)
    procedure Whirr; override;      { Linked in: NO }
    procedure Buzz;                 { Linked in: YES }
    procedure Pop; virtual;         { Linked in: NO }
  end;

  TKitchenGadget = class(TBaseGadget)
    procedure Whirr; override;      { Linked in: YES }
  end;

  TOfficeManager = class
    private
      FOfficeGadget: TOfficeGadget;
    public
      procedure InstantiateGadget; { Linked in: NO }
      { Linked in: YES }
      procedure Operate(AGadget: TOfficeGadget); virtual;
  end;

{ ... Non-essential code omitted ... }
procedure TOfficeManager.InstantiateGadget;
begin  { Dead code, never called }
  FOfficeGadget := TOfficeGadget.Create;
end;

procedure TOfficeManager.Operate(AGadget: TOfficeGadget);
{ Live code, virtual method of a constructed class }
begin
  AGadget.Whirr;
  AGadget.Buzz;  { This touches the static method body }
end;
var
  X: TBaseGadget;
  M: TOfficeManager;
begin
  X := TKitchenGadget.Create;
  M := TOfficeManager.Create;

  X.Free;
  M.Free;
end.
```

Figure 2: Notice how the addition of a call to the static *Buzz* method affects its linked-in status. *TOfficeGadget.Whirr* is still not included.

dead code again. Static methods are linked in only if they are used in live code, regardless of whether their class type is used.

Life in the Real World. Let's examine a more complex example of this virtual smart linking technique inside the VCL. The Delphi streaming system has two parts: *TReader* and *TWriter*, which descend from a common ancestor, *TFiler:*

- *TReader* contains all the code needed to load components from a stream.
- *TWriter* contains everything needed to write components to a stream.

These classes were split because many Delphi applications never need to write components to a stream — most applications only read forms from resource streams at program start up. If the streaming system was implemented in one class, all your applications would wind up carrying around all the stream output code, although many don't need it.

```
type
  TBaseGadget = class
    constructor Create;
    procedure Whirr; virtual;       { Linked in: YES }
 end;

  TOfficeGadget = class(TBaseGadget)
    procedure Whirr; override;      { Linked in: YES }
    procedure Buzz;                 { Linked in: YES }
    procedure Pop; virtual;         { Linked in: YES }
  end;

  TKitchenGadget = class(TBaseGadget)
    procedure Whirr; override;      { Linked in: YES }
  end;

  TOfficeManager = class
    private
      FOfficeGadget: TOfficeGadget;
    public
      procedure InstantiateGadget;    { Linked in: YES }
      { Linked in: YES }
      procedure Operate(AGadget: TOfficeGadget); virtual;

 end;

{ ... Non-essential code omitted ... }
procedure TOfficeManager.InstantiateGadget;
begin  { Live code }
  FOfficeGadget := TOfficeGadget.Create;
end;

procedure TOfficeManager.Operate(AGadget: TOfficeGadget);
{ Live code, virtual method of a constructed class }
begin
  AGadget.Whirr;
  AGadget.Buzz;  { This touches the static method body }
end;

var
  X: TBaseGadget;
  M: TOfficeManager;
begin
  X := TKitchenGadget.Create;
  M := TOfficeManager.Create;

  M.InstantiateGadget;

  X.Free;
  M.Free;
end.
```

Figure 3: With a call to *InstantiateGadget*, the construction of *TOfficeGadget* becomes live and all of *TOfficeGadget*'s virtual methods are linked.

So, splitting the streaming system into two classes improved smart linking. End of story? Not quite.

In a careful examination of the code linked into a typical Delphi application, the Delphi R&D team noticed that bits of *TWriter* were being linked into the .EXE. This seemed odd, because *TWriter* was definitely never instantiated in the test program. Some of those *TWriter* bits touched a lot of other bits that piled up rather quickly into a lot of unused code. Let's backtrack a little to see what lead to this code getting into the .EXE, and its surprising solution.

Delphi's *TComponent* class defines virtual methods that are responsible for reading and writing the component's state in a stream, using *TReader* and *TWriter* classes. Because *TComponent* is the ancestor of just about everything of importance in Delphi, *TComponent* is almost always linked into your Delphi programs, along with all the virtual methods of *TComponent*.

Some of *TComponent*'s virtual methods use *TWriter* methods to write the component's properties to a stream. Those *TWriter* methods were static methods. Therefore, *TComponent* virtual methods are always included in Delphi form-based applications, and some of those virtual methods (e.g. *TComponent.WriteState*) call static methods of *TWriter* (e.g. *TWriter.WriteData*). Thus, those static method bodies of *TWriter* were being linked into the .EXE. *TWriter.WriteData* is the kingpin method that drives the entire stream output system, so when it is linked in, almost all the rest of *TWriter* tags along (everything, ironically, except *TWriter.Create*).

The solution to this code bloat (caused indirectly by the *TComponent.WriteState* virtual method) may throw you for a loop: To eliminate the unneeded *TWriter* code, make more methods of *TWriter* (e.g. *WriteData*) virtual!

The all-or-none clumping of virtual methods that we curse for working against the smart linker can be used to our advantage, so that *TWriter* methods that must be called by live code are not actually included unless *TWriter* itself is instantiated in the program. Because methods such as *TWriter.WriteData* are always used when you use a *TWriter*, and *TWriter* is a mule class (no descendants), there is no appreciable cost to making *TWriter.WriteData* virtual. The benefits, however, are appreciable: making *TWriter.WriteData* virtual shaved nearly 10KB off the size of a typical Delphi 2 .EXE. Thanks to this and other code trimming tricks, Delphi 2 packs more standard features (e.g. form inheritance and form linking) into smaller .EXEs than Delphi 1.

What's Really in Your Executables? The simplest way to find out if a particular routine is linked into a particular project is to set a breakpoint in the body of that routine and run the program in the debugger. If the routine is not linked into the .EXE, the debugger will complain that you have set an invalid breakpoint.

To get a complete picture of what's in your .EXE or DLL, configure the linker options to emit a detailed map file. From Delphi's main menu, select **Project | Options** to display the Project Options dialog box. Select the Linker tab. In the **Map File** group box, select **Detailed**. Now recompile your project. The map file will contain a list of the names of all the routines (from units compiled with `$D` + debug information) that were linked into the .EXE.

Because the 32-bit Delphi Compiled Unit (.DCU) file has none of the capacity limitations associated with earlier, 16-bit versions of the Borland Pascal product line, there is little reason to ever turn off debug symbol information storage in the .DCU. Leave the `$D`, `$L`, and `$Y` compiler switches enabled at all times so the information is available when you need it in the integrated debugger, map file, or object browser. (If hard disk space is a problem, collect the loose change beneath the cushions of your sofa and buy a new 1GB hard drive.)

Novelty of Inverse Virtual Smart Linking. This technique of using virtual methods to improve smart linking is not unique to Delphi, but because Delphi's smart linker has a much finer granularity than other compiler products, this technique is much more effective in Delphi than in other products.

Most compilers produce intermediate code and limited symbol information in an .OBJ format, and most linkers' atom of granularity for smart linking is the .OBJ file. If you touch something inside a library of routines stored in one .OBJ module, the entire .OBJ module is linked into the .EXE. Thus, C and C++ libraries are often broken into swarms of little .OBJ modules in the hope of minimizing dead code in the .EXE.

Delphi's linker granularity is much finer — down to individual variables, procedures, and classes. If you touch one routine in a Delphi unit that contains lots of routines, only the thing you touch (and whatever it uses) is linked into the .EXE. Thus, there is no penalty for creating large libraries of topically-related routines in one Delphi unit. What you don't use will be left out of the .EXE.

Developing clever techniques to avoid touching individual routines or classes is generally more rewarding in Delphi than in most other compiled languages. In other products, the routines you so carefully avoided will probably be linked into the .EXE anyway because you are still using one of the other routines in the same module. Measuring with a micrometer is futile when your only cutting tool is a chainsaw.

Conclusion

Virtual methods are often maligned for bloating applications with unnecessary code. While it's true that virtuals can drag in code that your application doesn't need, this series has shown that careful and controlled use of virtual methods can achieve greater smart linking efficiency than would be possible with static methods alone. ∆

This article series was adapted from material for Danny Thorpe's book, Delphi Component Design *[Addison-Wesley Publishing Co., 1996].*

Danny Thorpe is a Delphi R&D engineer at Borland. He has also served as technical editor and advisor for dozens of Delphi programming books, and recently completed his book, *Delphi Component Design*, on advanced topics in Delphi programming. When he happens upon some spare time, he rewrites his to-do list manager to ensure that it doesn't happen again.

Object Pascal / Delphi 2

By Peter Dove and Don Peer

The World Is Flat

Delphi Graphics Programming: Part II

Last month, we covered some 3D math fundamentals, created the *TGMP* component, and developed an application to render wireframe objects. This month, we'll add some features to that application, including polygon filling, flat shading, directional light sources, vectors, normals, bit shifting, and backface removal. These features add volume to those wireframes you've generated, and take you a step closer to building a 3D, rendered component.

Let There Be Light

At times, wireframe objects will play tricks on your eyes, making it hard to discern which way the object is turning and which side is closer. This is because, with wireframe objects, your eyes get very little *depth cueing* — only a few elements are available for the eye and brain to create an accurate 3D picture of the object.

Adding shading in relationship to a light source will help make the object appear three dimensional. By doing this, you can see that as the object rotates away from the light source, the faces become progressively darker, while the other faces coming toward the light source become progressively lighter. Figure 1 shows the application rendering an object in shaded mode.

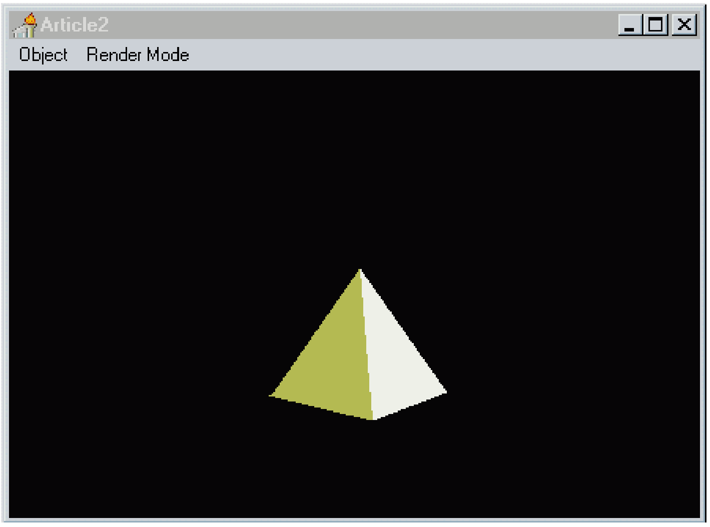


Figure 1: The application rendering an object in shaded mode.

Getting Your Fill

Before we can add a light source to the objects, we must add polygon filling. This will, as the name implies, fill the polygons, creating a solid look. To do this, we need to change *TGMP*'s structure.

Fortunately, these changes aren't as difficult as you may think. First, we must change the way we think about drawing the objects. Last month, we saw that drawing the object was just a matter of drawing many well-organized lines, whereas now we must think in terms of polygons.

A polygon is a figure of three or more sides that is either concave or convex. Testing to see what type of polygon you have is simple. If you draw a line through the polygon and the line only crosses two sides, then you are dealing with a convex polygon. If any line crosses more than two sides, then the polygon is concave (see Figure 2). Our 3D rendering engine will only support convex polygons

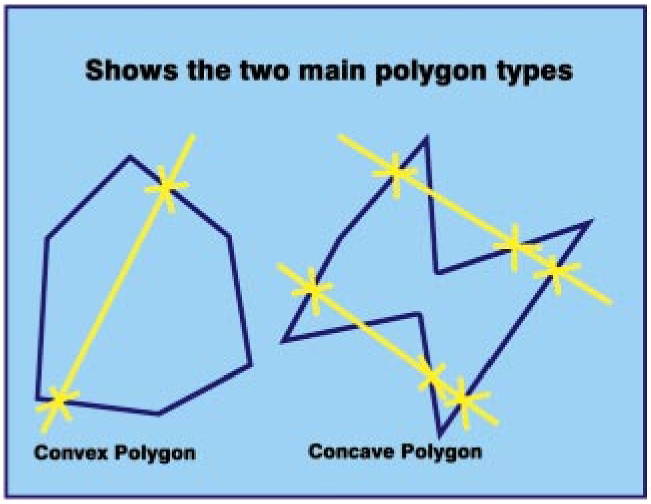


Figure 2: Testing to see if a polygon is convex or concave.

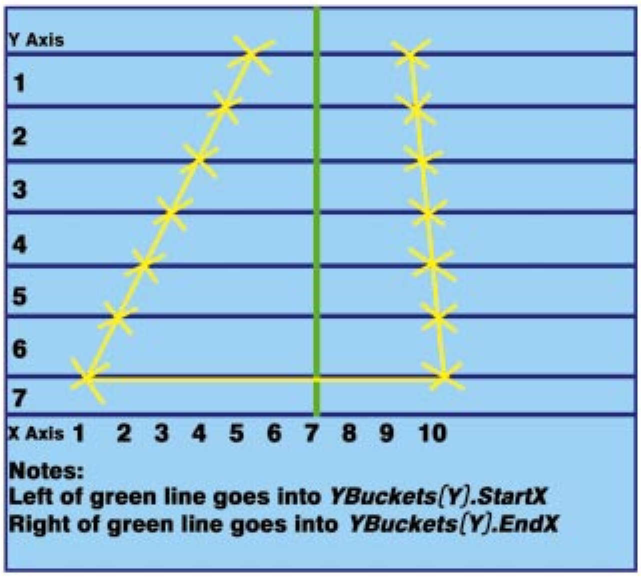

Y Axis
1
2
3
4
5
6
7
X Axis 1 2 3 4 5 6 7 8 9 10
Notes:
Left of green line goes into YBuckets(Y).StartX
Right of green line goes into YBuckets(Y).EndX

Figure 3: An illustration of the YBucket system.

because it's much easier and quicker to *scan convert* (which converts the 3D polygons into 2D horizontal scan lines) a convex polygon.

The YBucket System

Many systems have been developed for drawing filled polygons. We'll use the *YBucket* system, an array of *TYBucket*. Declare *TYBucket* as follows:

```
TYBucket = record
    StartX, EndX : Integer;
  end;
```

The YBucket system simply draws one of the lines that creates the 3D object, and maintains a list of each Y-value change. The X value is then stored in either *StartX* or *EndX*. After we've drawn all the polygon's lines, we'll have a list of horizontal lines for the number of Y points on the screen. These lines are put together to create a convex polygon (see Figure 3).

Now we need to create a new procedure, *DrawSolidLine2D*, that will work with our YBuckets. First,

```
procedure TGMP.DrawSolidLine2D(X1, Y1, X2, Y2 : Integer);
var
  CurrentX, XIncr  : Single;
  Y, Temp, Length  : Integer;
begin
  { No point in drawing horizontal lines! The rest of
    the polygon will define the edges }
  if Y1 = Y2 then
    Exit;

  { Swap if Y1 is less than Y2 so we always draw
    from top to bottom }
  if Y2 < Y1 then
    begin
      Temp  := Y1;
      Y1    := Y2;
      Y2    := Temp;
      Temp  := X1;
      X1    := X2;
      X2    := Temp;
    end;

  Length := (Y2 - Y1) + 1;

  { Xincr is how much the X must increment
    though each Y increment }
  Xincr := ((X2 - X1) + 1) / Length;

CurrentX := X1;

  { Loop through Y Values and fill Y buckets }
  for Y := Y1 to Y2 do
    begin
      { All Ybuckets are initialized to -16000 }
      if YBuckets[Y].StartX = -16000 then
        begin
          YBuckets[Y].StartX  := Round(CurrentX);
          YBuckets[Y].EndX    := Round(CurrentX);
        end
      else
        begin
          { Is Current X less than the Y StartX -
            if so update StartX }
          if CurrentX < YBuckets[Y].StartX then
            YBuckets[Y].StartX := Round(CurrentX);

          { Is Current X greater than the Y EndX -
            if so update EndX }
          if CurrentX > YBuckets[Y].EndX then
            YBuckets[Y].EndX := Round(CurrentX);
        end;
      CurrentX := CurrentX + XIncr;
    end;
end;
```

Figure 4: The *DrawSolidLine2D* procedure.

place the data member and procedure declaration in the **private** section of *TGMP*:

```
YBuckets : array [0..479] of TYBucket;

procedure DrawSolidLine2D(x1, y1, x2, y2 : Integer);
```

Then add the code in Figure 4 to the procedure. As you can see in the figure, *DrawLine2D* doesn't draw anything visible; it merely draws the outline of the polygon into our YBuckets. A few support methods are necessary to draw the polygon on the screen:

```
procedure DrawHorizontalLine(Y, X1, X2 : Integer);
```

will draw a line on the screen from X1 to X2 on the Y scanline, and *RenderYBuckets* iterates through the array of YBuckets and draws all the horizontal lines using *DrawHorizontalLine*.

```
procedure TGMP.ClearYBuckets;
var
  X : Integer;
begin
  for X := 0 to 479 do
    YBuckets[X].StartX := -16000;
end;

procedure TGMP.DrawHorizontalLine (Y, X1, X2 : Integer) ;
begin
  FBackBuffer.Canvas.Penpos := Point(X1, Y);
  FBackBuffer.Canvas.LineTo(X2, Y);
end;

procedure TGMP.RenderYBuckets ;
 var
  Y : Integer;
begin
  for Y := 0 to 479 do
    if YBuckets[Y].StartX <> -16000 then
      DrawHorizontalLine(Y, YBuckets[Y].StartX,
                            YBuckets[Y].EndX);
end;
```

Figure 5: The code for the *ClearYBuckets*, *DrawHorizontalLine*, and *RenderYBuckets procedures*.

The last support method is *ClearYBuckets*; it iterates through the array setting the *StartX* to -16000. Setting the YBucket to an initial value lets us know whether that line holds any X values. This is necessary because we won't draw on all the screen space. Figure 5 shows the *ClearYBuckets*, *DrawHorizontalLine*, and *RenderYBuckets* procedures.

Finally, we must change the *RenderNow* and *Rotate* procedures, modify the *TObject3D* record type, and create a new record type, *TPolygon*, that will allow *TObject3D* to express the idea of polygons rather than lines. Listing Two on page 28 shows the changes required to the procedures *RenderNow* and *Rotate*, and the modified *TObject3D* record that incorporates *TPolygon*.

The Shady Life of *TGMP*

Before we add the shading, we must discuss some 3D math to understand how shading works and how to implement complicated lighting systems (i.e. systems with more than one light).

For the shading algorithm to work correctly, all points on the polygon must be *coplanar*, meaning that the polygon must be flat — it can't have any warps, bends, or twists. From this assumption, we can work out a normal to that polygon. A *normal* is a vector that is perpendicular to the plane of the polygon (see Figure 6).

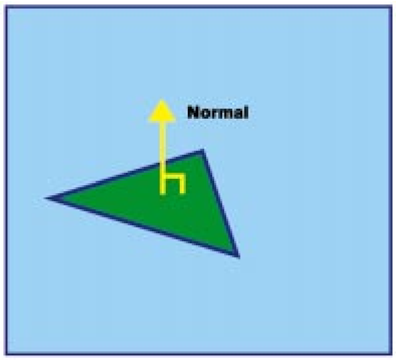


Figure 6: A normal to a polygon.

A *vector* is a directed line that has magnitude. Usually a normal has a magnitude/length of 1. There are two basic uses for vectors and normals:

1) If we have a normal for a polygon, we can make a polygon *sided*; that is, we can give one side a normal. Then we can perform a calculation on the normal and the line-of-sight vector to tell whether the polygon is facing us. (This is useful for *backface removal*.)

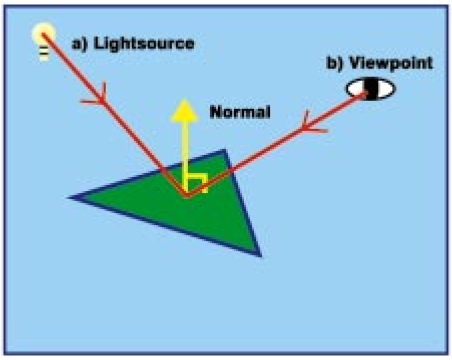


Figure 7: An illustration of the light source and viewpoint of a polygon.

2) When we combine the operation of a normal with a light source vector, we can calculate the angle at which the light strikes the surface of the polygon and therefore calculate the shade (see Figure 7).

Backface removal is an operation used to determine if the side of the polygon with the normal defined is facing toward or away from the line-of-sight vector. If the polygon is facing away from the line-of-sight vector, there is no need to draw it. So effectively, for the moment at least, our polygons can only have one side. This may sound like a restriction, but in all closed objects such as a sphere and a box, you cannot see the inside. Therefore, the polygons only need to be one sided.

Several math procedures will enable us to calculate an object's lighting. The first procedure, *GetVector3D,* takes two points and creates a vector from them, placing the result into the *Vector* parameter of *GetVector3D*. Remember that a vector is defined with one set of x, y, z points because the origin is assumed to be from 0, 0, 0. Place the declaration for this mathematical procedure in the **private** section of the *TGMP* unit:

```
procedure TGMP.GetVector3D(var EndPoint, StartPoint,
  Vector : TPoint3D);
begin
  Vector.X := EndPoint.X - StartPoint.X;
  Vector.y := EndPoint.Y - StartPoint.Y;
  Vector.z := EndPoint.Z - StartPoint.Z;
end;
```

The next procedure, *CrossProduct*, assists in obtaining a normal to a polygon. To find a normal to a polygon, you need two vectors (which comprise three points of the polygon). Remember that a normal is a vector at right angles to the plane of the polygon. The procedure takes the two points in the *U* and *V* parameter and returns the result in the *Normal* parameter.

CrossProduct contains the formulas to calculate the two vectors:

```
procedure TGMP.CrossProduct(var U,V,Normal : TPoint3D);
begin
  Normal.X := (V.Y * U.Z - V.Z * U.Y);
  Normal.Y := -(V.X *(U.Z+ZDistance)-(V.Z+ZDistance)*U.X);
  Normal.Z := (V.X * U.Y - V.Y * U.X);
end;
```

To find the normal of a polygon, we'll combine the *GetVector3D* and *CrossProduct* procedures to produce the *GetNormal* procedure. When three points of the polygon are passed to *GetNormal*, it will create two vectors from the three points, producing the normal from these two vectors. The *GetNormal* procedure creates the two vectors by using *P1* as the start point for both vectors, and *P2* and *P3* as the end points, and returns the result in the *Normal* parameter. The code for *GetNormal* is:

```
procedure TGMP.GetNormal(var P1, P2, P3, Normal : TPoint3D);
var
  U,V : TPoint3D;
begin
  GetVector3D(P2, P1, U);
  GetVector3D(P3, P1, V);
  CrossProduct(U,V,Normal);
end;
```

The next method, *VectorMagnitude*, returns the length of a vector:

```
function TGMP.VectorMagnitude(var Normal: TPoint3D): Single;
var
  X1 : Single;
begin
  X1 := Sqrt((Normal.X * Normal.X) +
             (Normal.Y * Normal.Y) +
             (Normal.Z * Normal.Z));
 { Ensure result is non-zero to
   avoid divide-by-zero errors }
  if X1 = 0 then
    X1 := 0.0000001;
  Result := X1;
end;
```

The last math method, *DotProduct*, multiplies two vectors (parameters *U* and *V*). Remember that a vector is represented by a point whose origin is assumed to be 0,0,0, which is why the *U* and *V* parameters are defined as *TPoint3D*. The resulting number provides a lot of information about the angular relationship of the vectors. If the resulting number is greater than 0, the angle is acute (less than 90 degrees), and if the number is less than 0, the angle is obtuse (greater than 90 degrees). When we combine the light vector with a polygon's normal, we use *DotProduct* to determine the angle at which the light strikes the polygon. We also use the *DotProduct* method to determine whether a polygon is visible or backfacing:

```
function TGMP.DotProduct(var U,V : TPoint3D) : Single;
begin
  Result := U.X * V.X + U.Y * V.Y + U.Z * V.Z;
end;
```

Now, to combine all the mathematical methods, we have one method that iterates through the object and calculates all the lighting and backface removal in one hit.

Now we can render the polygons, each with slightly different shades of the same color. You will see all the previously mentioned functions, plus another method for setting the position of the light source. Also, the GMP object contains two variables that deal with the lighting: *LightStrength,* which is as it sounds; and *AmbientLight*, which is the natural amount of light the scene emits that cannot be directly related to the light source. The ambient light is an attempt to emulate the light reflected off objects. For instance, in a room, places are lit because of the reflection of light off walls; this is what ambient light attempts to simulate.

Additional Functionality

The *RenderMode* property controls how the render methods output your object. *RenderMode* can be set to either *rmWireframe*, *rmSolid*, or *rmSolidShade*.

Three data members, *ViewPoint*, *LightSource*, and *LightStrength*, are included. *ViewPoint* is the position of the camera; *LightSource* is a vector that describes the direction of the light; and *LightStrength* controls the brightness of the light, and is a real number that can be greater than 1, with 1 being full brightness.

The final two new methods are *OrderZ* and *SetLightSource*. *OrderZ* simply orders the polygons by *Z* value so that the renderer draws them in the correct order. *SetLightSource* takes a point indicating the position, and a point indicating the point in space to which the light is directed. Using these two points, *SetLightSource* produces a vector that is used in the lighting calculations.

Lastly, a slight addition to the *Create* constructor initializes some of the extra data members and also sets a default *LightSource* vector.

Our Second Application

In our second application, we will slightly modify the array data for the cube and pyramid objects, because we'll be viewing the objects from a slightly different angle. We will also add two more array types to hold the number of polygon faces for the cube and pyramid objects. The two new arrays are declared as follows in the **type** section of the application code:

```
TPyramidPolys  = array [0..4] of Integer;
TCubePolys     = array [0..5] of Integer;
```

The balance of the code remains basically the same, except for the introduction of the *RenderMode* menu items and a procedure named *ClearMenuCheck* that clears the menu and re-checks the menu options as they are selected.

Conclusion

This has been a fairly intense article because we had a lot of information to cover. Next month, we'll add texture mapping to our 3D rendering component, and develop a cleaner component for windows messaging and event handling. ∆

References

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

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\97\FEB\DI9702DP.

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.

Begin Listing Two — The *RenderNow* and *Rotate* Procedures

```
TPolygon = record
  Point : array [0..3] of TPoint3D;
    { Only allow polygons for now if to 4 points }
  NumberPoints : Integer;
  Visible : Boolean;
  AverageZ : Single;
  PolyColor : TColor;
end;

TObject3D = record
    PolyStore : array [0..49] of Tpolygon;
    { Maximum 50 polys in an object }
    NumberPolys : Integer;
    Color : TColor;
  end;

procedure TGMP.RenderNow(var Object3D : TObject3D);
var
  X, I : Integer;
begin
  { Check to see which rendering mode will be used }
  case RenderMode of
    rmWireframe : // *** Wireframe ***
      begin
        FBackBuffer.Canvas.Pen.Color := Object3D.Color;
        for X := 0 to Object3D.NumberPolys - 1 do
          with Object3D.PolyStore[x] do begin
            DrawLine3D(Point[0].X,Point[0].Y,Point[0].Z,
                       Point[1].X,Point[1].Y,Point[1].Z);
            DrawLine3D(Point[1].X,Point[1].Y,Point[1].Z,
                       Point[2].X,Point[2].Y,Point[2].Z );
            if NumberPoints = 3 then
              DrawLine3D(Point[2].X,Point[2].Y,Point[2].Z,
                         Point[0].X,Point[0].Y,Point[0].Z)
            else
              begin
                DrawLine3D( Point[2].X,Point[2].Y,Point[2].Z,
                           Point[3].X,Point[3].Y,Point[3].Z);
                DrawLine3D( Point[3].X,Point[3].Y,Point[3].Z,
                           Point[0].X,Point[0].Y,Point[0].Z);
              end;
          end;
      end;

    rmSolid :    // *** Solid ***
      begin
        FBackBuffer.Canvas.Pen.Color := Object3D.Color;
        for X := 0 to Object3D.NumberPolys - 1 do
          with Object3D.PolyStore[X] do begin
            ClearYBuckets;
            for I := 0 to NumberPoints -1 do
              if  I <  (NumberPoints - 1) then
               DrawLine3D(Point[I].X,Point[I].Y,Point[I].Z,
                          Point[I+1].X,Point[I+1].Y,
                          Point[I+1].Z)
              else
                DrawLine3D(Point[I].X,Point[I].Y,Point[I].Z,
                           Point[0].X,Point[0].Y,Point[0].Z);
              RenderYBuckets;

          end;
      end;

    rmSolidShade :   // *** Solid Shading ***
      begin
        RemoveBackfacesAndShade(Object3D);
        OrderZ(Object3D);
         for X := 0 to Object3D.NumberPolys - 1 do
          with Object3D.PolyStore[X] do
            if Object3D.PolyStore[X].Visible then
              begin
                FBackBuffer.Canvas.Pen.Color := PolyColor;
                ClearYBuckets;
                for I := 0 to NumberPoints -1 do
                  if I < (NumberPoints - 1) then
                    DrawLine3D(Point[I].X,Point[I].Y,
                               Point[I].Z,Point[I+1].X,
                               Point[I+1].Y,Point[I+1].Z)
                  else
                    DrawLine3D(Point[I].X,Point[I].Y,
                               Point[I].Z,Point[0].X,
                               Point[0].Y,Point[0].Z);
                RenderYBuckets;
              end;
      end;
  end;
end;

procedure TGMP.Rotate(X, Y, Z, Angle : Single;
  var Object3D : TObject3D);
var
  P, I : Integer;
  NewX, NewY, NewZ : Single;
begin
  for P := 0 to Object3D.NumberPolys - 1 do
    with Object3D.PolyStore[P] do begin
      if Z <> 0 then
        for I := 0 to NumberPoints - 1 do begin
          NewX := Point[I].X * cos(Angle) -
                  Point[I].Y * sin(Angle);
          NewY := Point[I].X * sin(Angle) +
                  Point[I].Y * cos(Angle);
          Point[I].X := NewX;
          Point[I].y := NewY;
        end;

      if X <> 0 then
        for I := 0 to NumberPoints - 1 do begin
          NewY := Point[I].Y * cos(Angle) -
                  Point[I].Z * sin(Angle);
          NewZ := Point[I].Y * sin(Angle) +
                  Point[I].Z * cos(Angle);
          Point[I].Y := NewY;
          Point[I].Z := NewZ;
        end;

      if Y <> 0 then
        for I := 0 to NumberPoints - 1 do begin
          NewZ := Point[I].Z * cos(Angle) -
                  Point[I].X * sin(Angle);
          NewX := Point[I].X * cos(Angle) +
                  Point[I].Z * sin(Angle);
          Point[I].z := NewZ;
          Point[I].x := NewX;
        end;
    end;
end;
```

End Listing Two

Delphi / Object Pascal

By Ian Cresswell

Dynamic Arrays?

A Class Wrapper for *TList*

Application programmers sometimes require the ability to store indexed data at run time. On the surface, the **array** data type appears ideal for such problems, but arrays suffer from serious limitations.

Arrays are declared at compile time and are not well suited to problems involving the dynamic use of data. Specifically, if you don't know how much data you want to store:

- too much space can be predeclared for too little data, or
- too little space can be predeclared for too much data.

The traditional means of overcoming such difficulties is to build a dynamic data structure — typically a linked list — and find some way of indexing the structure to make it behave as a "fake" array. In an ideal world, the best means of storing such data is to use a dynamic structure that can have its elements accessed with an index.

In this article we'll examine how to use a built-in, indexed dynamic list provided with Delphi.

Some Explanation

The *TList* class is normally associated with Windows list boxes and was designed to contain the information for an arbitrary list. The "class wrapper" described here allows array-like access to dynamic elements that are predeclared as being of the same type. (Note that producing a generic list — a list containing items of different types in any order — is also possible in a similar manner. However, doing this is beyond the scope of this article.)

During our discussion of the *TList* wrapper class, we must address a number of necessary tricks and tips. Most of these will be introduced in the subtext. First, however, we'll talk about two key issues. First, whether you like it or not, any class you declare is automatically a descendant of the *TObject* class. Therefore, declaring:

```
TMine = class
```

is the same as declaring:

```
TMine = class(TObject)
```

Furthermore, any instance of a class is also dynamic. In other words, whenever you declare a variable of a class type, it's automatically declared as **dynamic** without requiring the traditional Pascal caret (^) pointer notation. We'll revisit this later when we use a trick that relies on knowledge of this particular Delphi feature.

The approach taken to writing the class wrapper is inherently object-oriented; that is, a minimal **public** interface is provided to allow restricted access to **protected** and **private** members. For **protected** and **private** members to work as intended, with respect to class access rights, a second important issue and a quirk of Delphi must be introduced. The **protected** and **private** parts of a class are only truly protected and private if the **class** type declaration is in a separate module (e.g. in a unit) to that in which it's used. This is directly "inherited" from the use of **private** in previous implementations of Borland Pascal and is unusual in comparison to other object-oriented languages. C++, for instance, treats **private** and **protected** members as **private** and **protected** regardless of where they are declared and/or used.

TLists

Delphi's *TList* class can have its elements accessed by the use of an index, in much the same way as a one-dimensional array. The *TList* class is merely a dynamic list of pointers to other items, and allows the creation and management of such lists (see Figure 1). The *TList* class is most commonly used to maintain dynamic lists of strings for Windows list boxes, etc. This provides a much higher level of abstraction than the use of other lower-level, linked list code.

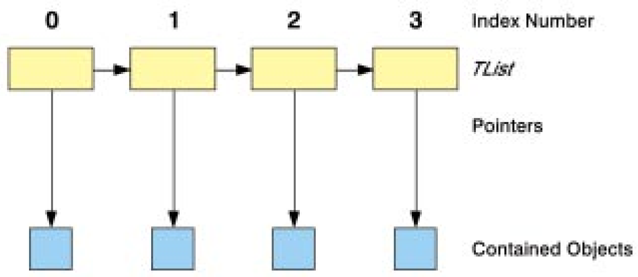


Figure 1: The structure of a *TList*.

*TList*s are one dimensional in nature, and can contain pointers to 16,380 items for each list instantiation. (Note that in Delphi 1, the limit for *TLists* is 16,380 items. This is because *TList*s in Delphi 1 are working with a segmented memory structure. In Delphi 2, the limit is only restricted by available memory.) The code that accompanies this article uses the following methods and properties provided with *TLists*:

- The *Items* property is used to access contained objects by index number.
- The *Add* method adds an item to the end of the list.
- The *Delete* method deletes an item from a specified position and turns the *TList* pointer to **nil**.
- The *Count* property determines how many items are in the *TList*.
- The *Pack* method removes **nil** pointers in the *TList*.

For our purposes, we require some means of managing a *TList* data type and making it behave as an *n*-dimensional array. This will involve building a class that allows us to access a protected *TList* descendant through a publicly declared interface. The *TList* will contain pointers to instances of another class that will contain the data we want to access.

First, we'll look at the class wrappers required for safely implementing a *TList*-based dynamic array. In this example, the class wrapper allows us to instantiate and manage a real number-based *TList* object when required:

```
TListClass = class
  protected
    TheList : TList;
  public
    constructor Create;
    destructor Free;
    procedure SetElement(value: ContainedType;
                         index: Integer);
    function GetElement(index: Integer): ContainedType;
    procedure AddToList(value: ContainedType);
    procedure KillItem(index: Integer);
    function GetNumItems: Integer;
  end;
```

The following code shows the *TContained* class that is instantiated for each *ContainedType* we want to represent:

```
TContained = class
  protected
    ContainedValue: ContainedType;
  public
    constructor Create(CopyIn: ContainedType);
    procedure ChangeItem(CopyIn: ContainedType);
    function GetValue: ContainedType;
  end;
```

Immediately above these type declarations, a single line is included to define the *ContainedType* type:

```
ContainedType = Integer;
```

The constructor is called as part of the instantiation process and accepts an integer that is then copied to the protected *ContainedValue* field during the objects' construction process.

The dynamic list of *TContained* objects is created by using the *Create* constructor associated with *TListClass*, which in turn uses the *Create* method provided for all objects. The integers associated with these objects are tied to the *TListClass* object.

Provided that the relevant integers are inserted into the list in logical order, it's possible to fake the use of an array with any number of dimensions and of any type (with the obvious memory availability considerations). It's a simple matter to calculate a unique position for a data item in a one-dimensional list that is accessed by using *n*-D indices (we'll discuss this consideration in more detail later).

The destruction of the *TContained* objects must occur before the *TList* descendant (holding pointers to them) is itself removed. The consequence of destroying the list first will be to lose any reference to the *TContained* objects, and thereby to lose heap memory in an unrecoverable and "leaky" manner. (The term "leaky" refers to an application that does not restore all the memory it uses. This is normally caused by not disposing memory that was allocated dynamically by the use of new, or perhaps not freeing, objects after they have been created.)

Each element of the list is removed by calling its associated *Free* method. The count field used with *TList* descendant objects is used to step through each of the instantiated *TContained* objects. Once all the *TContained* objects have been removed, it's possible to cleanly remove the *TList* descendant that was used to contain references to them by using the *Free* method on it. Observant readers will notice the use of dynamic memory without the need to resort to the Pascal-style pointer (^) notation.

The *TContained* class has three methods that operate on a single, protected data member of a user-specified type (Real, in this case):

1) The first method to be used is the *Create* constructor that accepts a *ContainedType* by value and copies it to the protected data member:

```
constructor TContained.Create(CopyIn: ContainedType);
begin
  ContainedValue := CopyIn;
end;
```

2) The *ChangeItem* method is provided so the programmer can modify the value of the protected data member at run time. It accepts *ContainedType* by value and copies it to the protected data member:

```
procedure TContained.ChangeItem(CopyIn: ContainedType);
begin
  ContainedValue := CopyIn;
end;
```

3) A third method, the *GetValue* function, will return the current value of the protected data member:

```
function TContained.GetValue: ContainedType;
begin
  result := ContainedValue;
end;
```

The *TListClass* consists of six methods, a constructor, and a destructor. The *Create* constructor is used to instantiate a new *TList* descendant (*TheList*), a protected data member of the class. This instantiation must occur at this point; otherwise a dreaded GPF (General Protection Fault) will occur. Such data member creation and initialization is, after all, one of the main purposes of constructors:

```
constructor TListClass.Create;
begin
  TheList := TList.Create;
end;
```

The destructor frees the memory consumed by *TheList* by calling its *Free* method:

```
destructor TListClass.Free;
var
  LoopCount : Integer;
begin
  for LoopCount := 1 to GetNumItems do
    { Successively removes first item to empty list }
    KillItem(0);
  TheList.Free;
end;
```

Note that this process must occur after each contained *TContained* object has been destroyed. A *TList* is a linked list of pointers to items (in this case *TContained* instantiations), and the consequence of destroying a *TList* before its items have been destroyed is memory leakage. When the list is destroyed, we have no way of referencing what it used to contain and therefore, no way of freeing the heap. So, before calling the *Free* method of *TListClass*, we need to traverse through the list and remove the *TContained* items it holds.

The *AddToList* method takes a *ContainedType* variable by value. A local *TContained* type is declared within the method and the construction process is initiated. This returns a pointer to the new object, which in turn, is added to the end of the list instantiation. In other words, you can view the big picture as being an instance of a *TList* object that contains pointers to dynamically instantiated *TContained* objects:

```
procedure TListClass.AddToList(value: ContainedType);
var
  APtr : TContained;
begin
  APtr := TContained.Create(value);
  try
    TheList.Add(APtr);
  except
    on Exception do
      MessageDlg('Attempt to Add to a Non-Existent List',
                 mtwarning,[mbok],0);
  end;
end;
```

The *SetElement* method is provided so a contained *TContained* instantiation can have its protected data member changed. The *Items* property of a list returns a pointer to an item at a specified position within the list. This generic pointer is then explicitly typecast to be a pointer to a *TContained* object. The *ChangeItem* method associated with the *TContained* object can then be called and the value of the specified element set:

```
procedure TListClass.SetElement(value: ContainedType;
                                index: Integer);
var
  APtr : TContained;
begin
  APtr := TContained(TheList.Items[index]);
  APtr.ChangeItem(value);
end;
```

The *GetElement* method is similar to *SetElement* in all respects. However, *GetElement* gets (rather than sets) the value of the protected data member:

```
function TListClass.GetElement(
  index: Integer): ContainedType;
var
  APtr : TContained;
begin
  APtr    := TContained(TheList.Items[index]);
  Result  := APtr.GetValue;
end;
```

An additional feature is provided as part of the *TList* class type. The *Count* property of a list is used to determine how many elements currently exist in the list. The purpose of the member function *GetNumItems* is to return this value:

```
function TListClass.GetNumItems:Integer;
begin
  Result := TheList.Count;
end;
```

As noted previously, before the list can be destroyed, we must kill all the items that its pointers point to. The *KillItem* procedure accepts an Integer by value that corresponds to the numeric position of the item that will be destroyed. A pointer to the item is retrieved and typecast, and then the *Free* method is called to remove the previously instantiated *TContained* type. The list then has the position that used to point to the killed item deleted; this causes a **nil** pointer to appear at the index position. The list's *Pack* method is then called to remove the **nil** pointer from the list. This is an example of dynamic list sizing in action:

```
procedure TListClass.KillItem(index: Integer);
var
  APtr : TContained;
begin
  try
    APtr := TContained(TheList.Items[index]);
    APtr.Free;
    TheList.Delete(index);
    TheList.Pack;
  except
    on EListError do
      Messagedlg('Cannot remove item from empty list',
                 mtwarning,[mbok],0);
  end;
end;
```

A Sample Program

During the development of this list container, a simple form was designed and tied to a *TListClass* instance to facilitate easy debugging. The form (see Figure 2) has four buttons that allow the creation and destruction of a list and the removal and insertion of items into the list. The entire code listing for this demonstration form is provided in Listing Three, beginning on page 33.

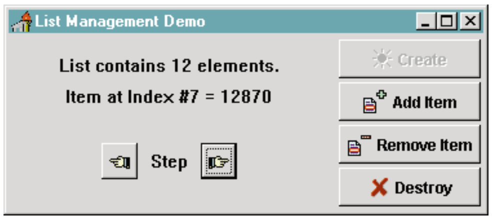


Figure 2: This demonstration form allows you to test *TListClass*.

The **Create** button instantiates a copy of *TListClass* and assigns it to *MyLC*. *MyLC* is declared as being of *TListClass* type in the global **var** section. In addition, to protect the form from destroying a non-existent list, or adding items to a non-existent list, the sample code manages enabling and disabling of the form's buttons.

When a user presses the **Add Item** button, a random integer is added to the current list. Additionally, the current memory available display, along with the number of items, is updated on the form. Also, when the **Add Item** button is pressed, it generates a random integer and uses the *AddToList* method of *TListClass* to add this integer to the list. Similarly, the **Remove Item** button uses the *KillItem* method to delete the item currently being viewed.

Finally, the **Destroy** button causes the list instance to be killed by calling the *Free* method. Note that because the *TListClass* itself manages deletion of the individual items, we don't need to concern ourselves with leaking memory.

An interesting and unusual consequence of Delphi's ability to trap GPFs was noted during the development of this package. "Dropped pointers" (i.e. lost pointer references) caused most of the problems while the software was being written. I found that debugging time was minimized by allowing the Delphi IDE to raise exceptions and monitoring the faulty dynamic objects by using variable watches. This approach also helped me decide where to put the Delphi exception handlers.

Exception Handling and Error Trapping

A powerful feature of Delphi is the ease with which it can handle exceptions. Typically, a dropped pointer will cause a GPF; an exception handler can be written to prevent this from happening.

The Delphi IDE can get in the way of testing your exception handlers because it can also handle exceptions. When configuring your environment (**Options | Environment | Preferences**) you'll see an option named **Break on Exception** is available. While you're testing your exception handlers, you should ensure that this option is disabled; otherwise the IDE will trap the exception that is raised, and your custom handler won't be invoked immediately.

To remove the possibility of an attempt to kill a non-existent item raising an exception, the *KillItem* method of *TListClass* was altered to include a **try..except** construct. Although the design of the demonstration program prevents you from making such mistakes (by dimming some of the buttons), you can comment out these lines of code to see how Delphi manages these exceptions. For example, if you were to create a list and then attempt to remove an item from the empty list, the application will generate an *EListError* exception on the line that attempts to get a pointer to the non-existent item. This will only happen if the **Break on Exception** option is enabled. You should always keep this option turned on unless you're testing your exception handler.

Trapping the raised exception, once it's known, is a simple matter of wrapping the existing code in a **try..except** block and then using an **on..*ExceptionName*..do** statement. If no exception is raised, the code associated with the **try** keyword will execute; otherwise the code associated with the **except** keyword will run. In this case, the exception handler checks for an *EListError* and displays a dialog box if this occurs. More complex error handling should be an immediately obvious possibility. In the case of the *KillItem* method, an exception provides the user with a "Cannot remove item from empty list" error, instead of a GPF.

An exception can also be raised by adding an item to a list that doesn't exist. This will raise the exception because an attempt was made to access data through an uninitialized pointer. The *AddToList* method associated with *TListClass* should be modified to rectify this potential problem:

```
try
  TheList.Add(APtr);
except on EGPFault do
  MessageDlg('Attempt to Add to a Non-Existent List',
             mtwarning,[mbok],0);
end;
```

Other exceptions may be raised, and it's always worthwhile to add custom exception handlers where they are necessary. A *TList* will raise an exception if the program tries to overfill it. Each list may contain up to 16,380 pointer references. Trying to add the 16,381st item would raise an *EListError* exception. (Again, this is a Delphi 1 limitation; in Delphi 2 the number is limited only by available memory.) Trapping other such exceptions is left to the enthusiastic reader.

2D Array-Style Wrappers for *TListClass*

Using a *TList* wrapper to calculate an element's position in a faked, two-dimensional array is relatively straightforward. The index is merely assumed to count normally (array-wise) and the internals deal with the actual position in the dynamic 1-dimension (or *n*=1) list using this simple modification. Any multi-dimensional array (*n*=>2) can be mapped to a one-dimensional array structure by applying a simple mathematical formula.

Assuming that the array is two-dimensional and of size (I, J), the linear list-based position of the element (x, y) is:

```
Linear index := (J*(x-1))+y;
```

A method can then be used in all calls to the *TListClass* methods that require multi-dimensional index parameters. In this case, the *LinearIndex* function is specified to calculate a linear index for a two-dimensional array.

```
function LinearIndex(j,x,y: Integer): Integer;
begin
  result := (j*(x-1))+y;
end;
```

The calculation of linear indices for fake arrays of other dimensionality is similar in nature to that shown above.

Conclusion

The *TList* wrapper class presented in this article was initially developed for use with visual neural network software I'm developing, and has been of tremendous use for this purpose. Many modifications and additions can be made using the other methods and properties associated with lists — this is left to the enthusiast. As with any good object-oriented model, the classes provided are extensible (through inheritance) and the implementation is safe.

One major problem arose during the development of the *TList* wrapper class and indicates a major failing of Delphi in comparison to C++. In the code given earlier, I assume that all elements in the list are of the same type. This is a reasonable assumption because it reflects the nature of normal array structures. It requires the following line of code (or something similar):

```
ContainedType = Integer;
```

In C++, templates could be used to build generic classes that deal with generic types. A linked list class could be built to deal with a generic type *T* and then the decision about which type *T* is used could be deferred until the code is being written. In Delphi, if we required more than one different type of fake array, we would have to needlessly duplicate code (copy-and-paste style) to achieve this. Even with this limitation in mind, the code provided here is powerful and really only amounts to a few actual "do something" statements. Developing this sort of code from scratch in a traditional non-object-oriented way would be a tedious task. ∆

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\97\FEB\DI9702IC.

Dr Ian Cresswell is the Research Coordinator in the School of Computer Science at the University of Central England in Birmingham, England. He welcomes comments and suggestions via e-mail (icressw@leopold.win-uk.net) about this article. He is currently working on a book that incorporates his two main interests — *Machine Intelligence Algorithms in Delphi.*

Begin Listing Three — The Listtry Unit

```
unit Listtry;

interface

uses
  SysUtils, WinTypes, WinProcs, Messages, Classes,
  Graphics, Controls, Forms, StdCtrls, ListUnit, Buttons;

type

 TForm1 = class(TForm)
    lblItem : TLabel;
    lblMessage : TLabel;
    lblStep : TLabel;
    btnCreate : TBitBtn;
    btnDestroy : TBitBtn;
    btnUp : TBitBtn;
    btnDown : TBitBtn;
    btnAddItem : TBitBtn;
    btnRemoveItem : TBitBtn;
    procedure btnCreateClick(Sender: TObject);
    procedure btnDestroyClick(Sender: TObject);
    procedure btnUpClick(Sender: TObject);
    procedure btnDownClick(Sender: TObject);
    procedure btnAddItemClick(Sender: TObject);
    procedure btnRemoveItemClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
  private
    procedure ShowTheMessages;
    procedure RestrictMovement;
  end;

var
  Form1: TForm1;

implementation

var
```

```
  MyLC     : TListClass;
  CurrPos  : integer;

{$R *.DFM}

procedure TForm1.FormCreate(Sender: TObject);
begin
  Randomize;
  ShowTheMessages;
end;

procedure TForm1.FormDestroy(Sender: TObject);
begin
  if MyLC <> nil then
    MyLC.Free;
end;

procedure TForm1.btnCreateClick(Sender: TObject);
begin
  MyLC := TListClass.Create;      // Create list
  Currpos := 0;
  Randomize;
  ShowTheMessages;
  btnCreate.Enabled  := False;    // Prevent another create
  btnDestroy.Enabled := True;     // Allowed to destroy list
  btnAddItem.Enabled := True;     // Allowed to add item
end;

procedure TForm1.btnDestroyClick(Sender: TObject);
begin
  MyLC.Free;                        // Free the list
  MyLC  := nil;
  CurrPos := 0;
  ShowTheMessages;
  btnCreate.Enabled      := True;
  btnDestroy.Enabled     := False;
  btnUp.Enabled          := False;
  btnDown.Enabled        := False;
  btnAddItem.Enabled     := False;
  btnRemoveItem.Enabled  := False;
end;

procedure TForm1.btnAddItemClick(Sender: TObject);
begin
  MyLC.AddToList(Random(32000));
  ShowTheMessages;
  RestrictMovement;
  btnRemoveItem.Enabled := True;
end;

procedure TForm1.btnRemoveItemClick(Sender: TObject);
var
  TotalItems : Integer;
begin
  MyLC.KillItem(CurrPos);    // Delete this item in the list
  TotalItems := MyLC.GetNumItems;    // Get new total items
  if (CurrPos >= TotalItems) and
     (CurrPos > 0) then
    Dec(CurrPos);
  btnRemoveItem.Enabled := (TotalItems > 0);
  ShowTheMessages;
  RestrictMovement;
end;

procedure TForm1.btnUpClick(Sender: TObject);
begin
  Inc(CurrPos);
  ShowTheMessages;
  RestrictMovement;
end;

procedure TForm1.btnDownClick(Sender: TObject);
begin
  Dec(CurrPos);
  ShowTheMessages;
  RestrictMovement;
end;

procedure TForm1.ShowTheMessages;

var
  TotalItems : Integer;
  TmpStr : string[2];
begin
  try
    TotalItems := MyLC.GetNumItems;

    if TotalItems > 1 then
      TmpStr := 's.'
    else
      TmpStr := '.';
    if TotalItems = 0 then
      begin
        lblMessage.Caption  := 'List is empty.';
        lblItem.Caption     := '';
      end
    else
      begin
        lblMessage.Caption  :=
          'List contains '+IntToStr(MyLC.GetNumItems)+
          ' element'+TmpStr;
        lblItem.Caption     :=
          'Item at Index #'+IntToStr(CurrPos)+' =
          '+IntToStr(MyLC.GetElement(CurrPos))
      end;
  except
    lblMessage.Caption := 'List is not created.';
    lblItem.Caption    := '';
  end;
end;

procedure TForm1.RestrictMovement;
begin
  btnDown.Enabled := (CurrPos > 0);
  btnUp.Enabled   := (CurrPos < (MyLC.GetNumItems-1));
end;

end.
```

End Listing Three

By Bill Todd

Directory Assistance

Configuring the BDE for a Paradox Database

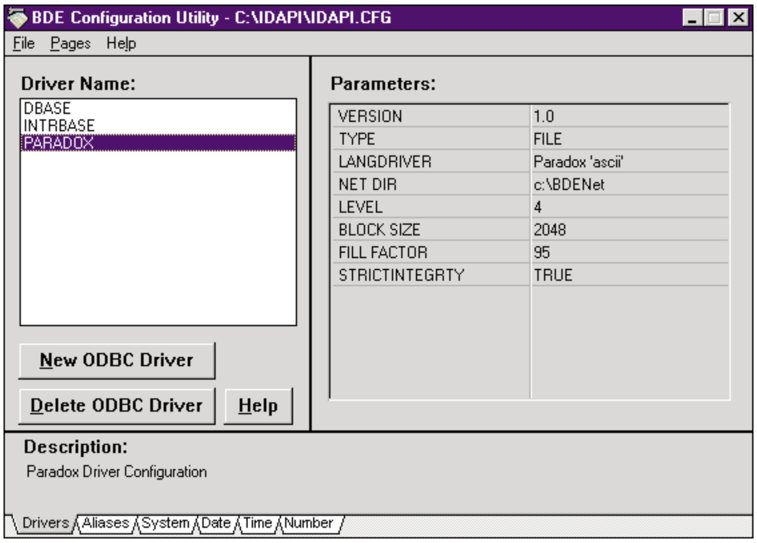
Have you ever written a Delphi program that uses local Paradox tables, and received error messages such as "Not initialized for network access" or "Directory is busy"? Have you set up a multiuser program and discovered that user A doesn't see database changes made by user B? The secret to avoiding these problems is correctly configuring the Borland Database Engine (BDE).

First, let's set the Paradox network directory. Run the BDE Configuration Utility that ships with Delphi, select the Drivers page, and click on the Paradox driver to view the screen shown in Figure 1.

NET DIR

If your program will access files on a shared network drive, you must set the NET DIR parameter. NET DIR is the path to the directory that contains the Paradox network control file named PDOXUSRS.NET. The following rules apply to the NET DIR path:

- It must point to a shared network directory.
- Users who will access Paradox tables must have read, write, create, and delete rights to the directory. PDOXUSRS.NET will be created automatically by the BDE.
- Users who will concurrently access Paradox tables in a directory must use the same network control file directory.
- The NET DIR path must be the same for all users.



VERSION	1.0
TYPE	FILE
LANGDRIVER	Paradox 'ascii'
NET DIR	c:\BDENet
LEVEL	4
BLOCK SIZE	2048
FILL FACTOR	95
STRICTINTEGRTY	TRUE

Paradox Driver Configuration

Figure 1: Setting the BDE Paradox driver parameters.

If you're configuring the BDE for a peer-to-peer network, there's one exception to the last rule. How you handle peer-to-peer networks depends on whether you're using the 16- or 32-bit BDE. Let's look at the easy case first. Because the 32-bit BDE supports UNC file names, it's easy to provide a path to the NET DIR that's the same for all machines. The format of a UNC path is:

```
\\ServerName\ShareName\ShareDir
```

where *ServerName* is the name of the machine on which NET DIR will reside, *ShareName* is the name assigned to a shared directory, and *ShareDir* is the path to a subdirectory underneath the directory shared as *ShareName*. Using UNC, the NET DIR path will be the same on all machines, including the server.

With the 16-bit BDE, you must use the same NET DIR path on every machine, with one exception: the drive letter can be different.

For example, suppose the directory for the network control file is C:\NETDIR on the server. On all other machines, drive C: on the server is mapped to drive G:, so the NET DIR setting for these machines must be G:\NETDIR.

You can programmatically override the NET DIR setting in the BDE configuration file at run time by setting the *Session.NetFileDir* property. For example, the Object Pascal statement:

```
Session.NetFileDir := 'G:\BDENET';
```

will set the network control file directory to the G:\BDENET directory. If you need to change the network control file directory in your program, do it in the project file before any forms are opened, or in the *OnCreate* event handler of the main form before any tables are opened. Whether you set the network control file directory in the main unit or in the project file, remember to include the DB unit in the **uses** clause.

LEVEL

The internal structure of Paradox tables has changed several times over the years; the LEVEL parameter determines the lowest (oldest) table version the BDE will create. The default setting is 4, which is the format first introduced with Paradox 4.0 for DOS (and subsequently used by Paradox 4.5 for DOS, Paradox 1.0 for Windows, and Paradox 4.5 for Windows). Level 5 tables were introduced in Paradox 5.0 for Windows, and level 7 tables are supported by Paradox 7.0 for Windows 95 and Windows NT, and Paradox 7 for Windows 3.1. You can also set the table level to 3 if you need to create tables that can be accessed by Paradox 3.5 for DOS or earlier versions.

The level of any table you create will be determined by the table's features. Let's say the table level in the BDE configuration file is set to 4. If you create a table that uses the long integer field type, or any of the other field types added in Paradox 5.0, the BDE will automatically create a level 5 table. If you use unique or descending secondary indexes, the BDE will automatically create a level 7 table. You may as well leave the default table level set to 4 (the default) for maximum backward compatibility. You won't get better performance — or any other advantage — from using a table level higher than the table and index structures require.

TYPE

The file TYPE parameter is set automatically by the BDE Configuration Utility; you shouldn't change it. The only two valid values are FILE, used for Paradox and dBASE tables, and SERVER, which is used for any database server.

LANGDRIVER

This setting selects the language driver used for your tables and indexes. The language driver determines which language-specific characters can be stored in tables and which sort order is used to build indexes.

The standard Paradox ASCII character set follows the ASCII sort order, which sorts "a" after "Z"; that is, lower-case characters follow upper-case characters. If you want "A" and "a" to be sorted adjacent to one another, you might consider changing to the Paradox International language driver. If you do, however, be warned that — without rebuilding all the indexes — a Paradox user with a different language driver will not be able to use tables that you create.

BLOCK SIZE

The maximum amount of data a Paradox table can hold is 64KB times the block size. Because the default BLOCK SIZE parameter is 2048 bytes, any Paradox table you create is limited to 128MB in size. The allowable block sizes for level 3 and 4 tables are 1024, 2048, and 4096. For level 5 and higher, you can also specify a block size of 8192, 16384, or 32768. The default size, however, is still 2048.

If you change the block size in the BDE configuration file, all the tables you create after making the change will have the new block size. To change the block size of an existing table, first change the block size in the BDE Configuration Utility, then copy or recreate the table. Because reading or writing a single record means reading or writing the entire block that contains the record, you should not make the block size any larger than is necessary to contain the data.

FILL FACTOR

The FILL FACTOR determines how full an index block must be before Paradox will allocate another block when building the index. The default is 95 percent. This means that when the index is initially built, or when it's rebuilt as part of packing a table, each index block will have about five percent of its space empty.

When you add a new record to a table, a record entry must be made in each of the table's indexes. Having some free space in the index blocks means it's less likely that the affected blocks will be full; therefore, adding the new record will be faster.

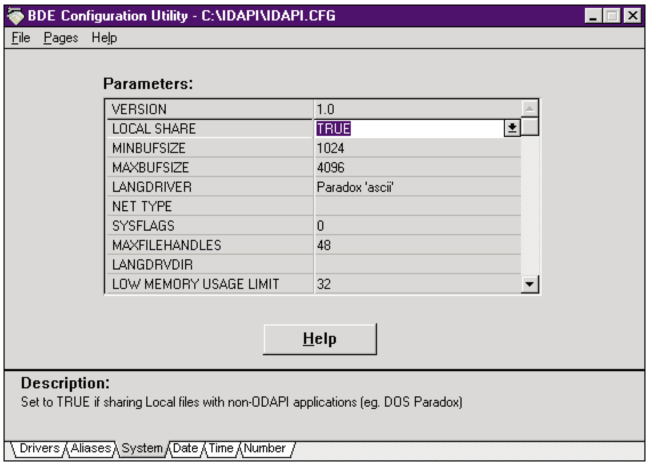
While the FILL FACTOR can be set to any percentage, leaving too much free space in the index blocks makes the index larger. If the index is larger, more index blocks must be read when you search for a record, decreasing performance.

STRICTINTEGRTY

If STRICTINTEGRTY (Strict Integrity) is set to TRUE, programs that do not support referential integrity (such as Paradox for DOS) will not be able to modify any table for which referential integrity is defined. To modify a table in Paradox for DOS, set STRICTINTEGRTY to FALSE. If you do so, and want to protect data integrity, you must write code to enforce referential integrity in your Paradox for DOS application.

LOCAL SHARE

The BDE automatically provides table and record locking for Paradox tables stored on a shared network drive. However, that's


BDE Configuration Utility - C:\IDAPI\IDAPI.CFG
File Pages Help
Help
Drivers Aliases System Date Time Number

Parameters:

VERSION	1.0
LOCAL SHARE	TRUE
MINBUFSIZE	1024
MAXBUFSIZE	4096
LANGDRIVER	Paradox 'ascii'
NET TYPE	
SYSFLAGS	0
MAXFILEHANDLES	48
LANGDRVDIR	
LOW MEMORY USAGE LIMIT	32

Description:
Set to TRUE if sharing Local files with non-ODAPI applications (eg. DOS Paradox)

Figure 2: You may need to change the LOCAL SHARE default setting on the System page.

not the only situation where locking is required to ensure the integrity of your data. If you run multiple programs that simultaneously access the same tables on your local drive, you need locking to prevent one program from changing a record already in use by another.

Figure 2 shows the System page of the BDE Configuration Utility. By default, the LOCAL SHARE parameter is set to FALSE. If you run multiple programs that access the same tables, you must set LOCAL SHARE to TRUE to tell the BDE to provide locking, even though the tables are on your local drive, unless all the programs are using the same version of the BDE. If you're running a multiuser application on a peer-to-peer network, you must also have LOCAL SHARE set to TRUE. This is because the BDE "sees" the drive on the server as a local drive rather than a shared network drive, and each workstation is running its own copy of the BDE.

Private Directory

If you're a Paradox programmer, you'll be surprised to learn that the rules that govern the use of the private directory are much different in Delphi than in Paradox. Paradox creates temporary files with fixed names in your private directory. Examples include ANSWER.DB, CHANGED.DB, DELETED.DB and KEYVIOL.DB. Because these temporary tables show the same name for every user and every instance of Paradox, each user and each session must have its own private directory. When two instances of Paradox are running on the same machine, this ensures that two Paradox sessions will not try to create a table with the same name in the same directory at the same time.

Although the BDE also creates temporary files in the private directory when querying, sorting, and restructuring Paradox and dBASE tables and when performing a query that joins tables on two different servers, the BDE creates unique file names for its temporary files. Therefore, it's safe for multiple Delphi programs running on the same machine at the same time to use the same private directory as long as the program itself does not create files with fixed names in the private directory.

If multiple users are running the same program on a network, and if the BDE will need to create temporary files as the program runs, you'll need to assign a separate private directory for each user. When you start a Delphi database program, it opens the default BDE session and sets the private directory to the startup directory, which will be the one specified in the **Start In** field of the program's icon on the Windows desktop. If none is specified, the startup directory will be the one that contains the .EXE file. However, the BDE does not create lock files in the private directory until either the BDE needs to create a temporary file in the private directory, or you set the path to the private directory from within your program by assigning a path to the *Session.PrivateDir* property.

Because the private directory isn't locked when a program starts, it's possible for two different users to start the same program and share the same private directory. However, as soon as the BDE needs to create a temporary file, it will lock the private directory for that user. If another user's copy of the BDE needs to create a temporary file, that user will receive an exception warning that the "Directory is busy". This is good; client/server programs that don't do heterogeneous joins, or use cached updates, won't have to worry about the private directory, because the BDE won't have to create any temporary files.

However, if you're running a program that uses Paradox tables and many users share a single copy of the program .EXE file on a shared network drive, you have a problem. If every user is running the same copy of the .EXE, then every user will have the same private directory — unless you specify a different startup directory for each user or you change the private directory in your code as soon as the program starts.

The best solution to this problem is to have each Delphi program set the private directory to a unique directory when starting. One way to do this is to create a subdirectory with the same name as the program's .EXE file, then place it on your local hard drive.

The function shown in Figure 3 returns the name of the program's .EXE file with the path and extension removed. Figure 4 shows a procedure that creates a private directory.

```
function GetExeName: string;
{ Returns the name of the .EXE file with no path
  and no extension. }
var
  ExtPos     : Integer;
  Name, Ext  : string;
begin

  Name   := ExtractFileName(Application.ExeName);
  Ext    := ExtractFileExt(Application.ExeName);
  { Find where the extension starts in the name. }
  ExtPos   := Pos(Ext, Name);
  { Extract the name without the extension. }
  if ExtPos > 0 then
    Name   := Copy(Name, 1, ExtPos - 1);
  Result   := Name;

end; { GetExeName. }
```

Figure 3: A function to return the .EXE file name.

```
procedure SetPrivateDir(DirPath: string);
{ Creates the specified directory if it does not
  exist and makes it the private directory. }
begin

  ForceDirectories(DirPath);
  if DirectoryExists(DirPath) then
    Session.PrivateDir := DirPath
  else
    raise Exception.Create(
      'Cannot create private directory ' + DirPath);

end; { SetPrivateDir. }
```

Figure 4: Use the *SetPrivateDir* procedure to create a private directory.

Using these routines, you can set the private directory from within your program:

```
SetPrivateDir('c:\bdepriv\' + GetExeName);
```

This statement creates the private directory as a subdirectory in the C:\BDEPRIV directory, maintaining all your private directories in one location rather than sprinkling them throughout the root directory of your local drive. Alternately, you can create the private directories below the users' home directory on the network. Although performance is better if the private directory is on a local hard drive, the network is a better choice if some users have limited hard disk space.

Because there are always exceptions, it's a good idea to provide a way for the default private directory path to be overridden. You can do this with an .INI file, registry entry, or an optional command-line parameter for your program. Using one of these techniques makes it easy to handle the exceptional case in which one user needs a private directory in a non-standard location.

Conclusion

If your Delphi programs use the BDE, you must configure it properly, setting a unique private directory for each user and each program. This allows safe multiuser and multitasking operation. Configuring the BDE properly will allow multiple programs to safely share data on network or local drives. Δ

Bill Todd is President of The Database Group, Inc., a database consulting and development firm based near Phoenix, AZ. He is co-author of *Delphi 2: A Developer's Guide* [M&T Books, 1996], *Delphi: A Developer's Guide* [M&T Books, 1995], *Creating Paradox for Windows Applications* [New Riders Publishing, 1994], and *Paradox for Windows Power Programming* [QUE, 1995]; and is a member of Team Borland, providing technical support on CompuServe. He is also a contributor to *Delphi Informant*, and has been a speaker at every Borland Developers Conference. He can be reached on CompuServe at 71333,2146, on the Internet at 71333.2146@compuserve.com, or at (602) 802-0178.

By Cary Jensen, Ph.D.

Screening Process

An Introduction to the *TScreen* Class

It's common to think of components as objects that appear on the Component Palette, permitting them to be placed on a form and modified at design time. However, there are many components that don't appear on the palette. In fact, three are automatically created by Delphi at run time: the Application, Screen, and Session components.

These three are instance variables of the *TApplication*, *TScreen*, and *TSession* classes, respectively. You can use these variables to control a number of aspects of your application. This month's column takes a closer look at the one variable you're most likely to overlook in your Delphi work: Screen.

The *TScreen* class, as well as the Screen instance variable, is declared in the Forms unit. Using this variable, you can:

- obtain information concerning the components and forms within your application,
- obtain information about the names of the fonts available on your system,
- control the cursors used by your application, and
- call event handlers when focus moves from control to control, and from form to form within the application.

Properties	Methods	Events
ActiveControl	*Create*	*OnActiveControlChange*
ActiveForm	*Destroy*	*OnActiveFormChange*
ComponentCount	*FindComponent*	
ComponentIndex	*Free*	
Components	*InsertComponent*	
Cursor	*RemoveComponent*	
Cursors		
*DataModuleCount**		
*DataModules**		
Fonts		
FormCount		
Forms		
Height		
Name		
Owner		
PixelsPerInch		
Tag		
Width		

Figure 1: The properties, methods, and events of the *TScreen* class (* available in Delphi 2 only).

Figure 1 contains a list of all of the *TScreen* properties, methods, and events.

Using the Screen component is demonstrated in this article by showing how to determine if a specific form is currently open, get a list of the available fonts, control the screen cursor, and respond to changes in the active form.

Detecting an Open Form

Form control is one of the more interesting and important topics in Delphi, because most Delphi applications require several forms — and many include dozens.

Under most circumstances, form control is fairly easy. Specifically, if a given form is auto-created (meaning it's created from within the project file [.DPR file]), and is not destroyed until the application closes, accessing such a form, even after the user has closed it, isn't difficult. This is because, unless you change the default behavior of the closing form, a form that is closed by the user isn't destroyed; it's merely hidden. Consequently, if you need to make the form visible again, all you need to do is call the form's *Show* or *ShowModal* method, using the form's instance variable as a qualifier.

For example, if you have an auto-created form of the *TForm2* class, and that form's unit defines an instance variable named *Form2*, you can always display the form using one of the following statements:

```
Form2.Show;
```

or

```
Form2.ShowModal;
```

However, accessing a previously created form is far more complex if there is a possibility that the form can be destroyed. Consider the following scenario: You have a form that is not an auto-created form — you create this form at run time by calling its *Create* constructor. Furthermore, it's a form you want to display several times. However, because you don't want it to take up resources when it is no longer used, you ensure it's destroyed each time it is closed.

With a form that is displayed using the *Show* method, you can cause it to be destroyed when it is closed by attaching an event handler to the form's *OnClose* event property, and setting the *OnClose* event handler's *Action* parameter to caFree. For example:

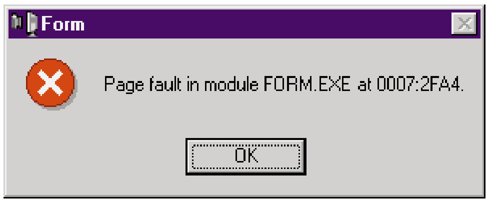
```
procedure TForm2.FormClose(Sender: TObject;
                           var Action: TCloseAction);
begin
  Action := caFree;
end;
```

But what if you want to display this form again? How do you determine if the form is still open (hidden or not), versus its having been destroyed, in which case it needs to be recreated, i.e. have its constructor called again?

When first attempting to perform this task, most Delphi developers test whether the form's instance variable is **nil**. For example:

```
if Form2 <> nil then
  Form2.Show
else
  Form2 := TForm2.Create(Self);
```

While this code is successful for testing whether the *Form2* instance variable has ever been assigned to an instance of a *TForm2* class, it will generate an exception similar to the one shown in Figure 2 when this code is executed after the form pointed to by *Form2* has been destroyed. Specifically, the *Form2* instance variable, after having been assigned a handle to a *TForm2* instance, will not contain **nil**. That is, destroying the form being pointed to by *Form2* does not set *Form2* to **nil**.


Form

Page fault in module FORM.EXE at 0007:2FA4.

OK

Figure 2: If a form has been created and then destroyed, its instance variable will *not* have a **nil** value.

A reliable alternative is to test whether the Screen object's *Forms* property points to a member of the *TForm2* class. *Forms* is a zero-based array of pointers to existing forms. Each time a new form is created, it's automatically added to the *Screen.Forms* array, and each time one is destroyed, it's removed. You can use the *Screen.FormCount* property to identify how many forms are open. The following is a code segment that demonstrates how to test whether an instance of the *TForm2* class is currently in existence (hidden or not), how to make it the active form if it exists, and how to create it if it doesn't:

```
var
  Found, i: Integer;
begin
  Found := -1;
  for i := 0 to Screen.FormCount - 1 do
    if Screen.Forms[i] is TForm2 then
      Found := i;
  if Found >= 0 then
    Screen.Forms[Found].Show
 else
    begin
      ShowMessage('Form2 not found - Creating...');
      Form2 := TForm2.Create(Self);
      Form2.Show;
    end;
end;
```

To use this technique with any other form, substitute the form class of the form you want to set focus to for the *TForm2* class in the preceding code segment.

The use of this technique is demonstrated in the project named FORMS.DPR. The main form of this project is shown in Figure 3. When *Form2* has not yet been created, both the **Use Screen.Forms property** button and the **Use Form2 <> nil** button will correctly report that *Form2* does not exist, then create it. Likewise, if *Form2* has already been created, but not yet destroyed, both buttons can successfully set focus to that form. However, if *Form2* has been created, but subsequently destroyed, only the **Use Screen.Forms property** button will be able to recreate it.

The **Use Form2 <> nil** button will incorrectly assume that *Form2* still exists (because the *Form2* variable still points to the memory that formerly pointed to the now-destroyed form), and will

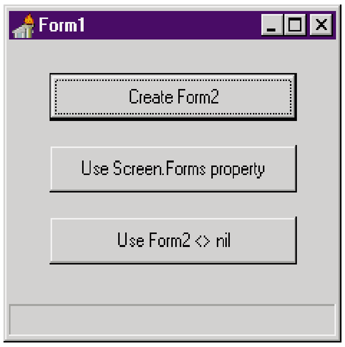


Figure 3: The FORMS.DPR project demonstrates the use of the *Screen.Forms* property to determine if a form of the *TForm2* class is open, and open it if it isn't.

generate an exception when attempting to set focus to the destroyed form.

Listing Available Fonts

The Screen component has a *Fonts* property — a *TStrings* property that contains a list of the fonts available in Windows. If you want to determine if a particular font is available, use the *IndexOf* method of the *TStrings* class. This method returns -1 if the specified string doesn't exist, and the index of the matched item if it does. For example, the following code demonstrates how to test if the current system has the MS LineDraw font:

```
if (Screen.Fonts.IndexOf('MS LineDraw') <> -1) then
  Form1.Font.Name := 'MS LineDraw'
else
  ShowMessage('MS LineDraw not available');
```

The project shown in Figure 4 demonstrates another use of the *Screen.Fonts* property. In this project, the *Screen.Fonts* property is assigned to the *Items* property of a ListBox using the following code:

```
ListBox1.Items := Screen.Fonts;
ListBox1.ItemIndex := 0;
```

Within the *OnChange* event handler of the ListBox, the currently selected font name is assigned to the *Font.Name* property of a Memo control using the following code:

```
Memo1.Font.Name := ListBox1.Items[ListBox1.ItemIndex];
```

Using this simple technique, you can easily inspect the various fonts available on your system.

Setting the Screen Cursor

Using a variety of different cursor shapes within a Delphi application seems easy. By setting the *Cursor* or *DragCursor* property of a given object, you can define which cursor will be displayed whenever the mouse passes over, or drags over, the specified object. However, this technique doesn't affect the cursor shape when the mouse is not over the particular component. So how do you change the shape of the cursor for the entire application? For example, how do you change the shape of the cursor to an hourglass shape for all objects during the execution of a lengthy process?

The answer is found in the *Cursor* property of the Screen component. Specifically, setting this property changes the shape of the default cursor for the application as a whole (regardless of the cursors defined for each individual component). The following code demonstrates how this technique can be used:

```
{ Set the cursor to an hourglass shape }
Screen.Cursor := crHourGlass;
{ Perform a lengthy operation }
SomeLongProcessHere;
{ Restore the original default cursor shape }
Screen.Cursor := crDefault;
```

Responding to Changes in the Active Form

The Screen object has two event properties: *OnActiveControlChange* and *OnActiveFormChange*. By assigning event handlers to these event properties you can respond to changes as focus moves between objects of the application and forms of the application, respectively.

Both these event handlers are *TNotifyEvent* type properties, meaning that they are passed a single argument of type *TObject* when called. To determine which control or form is becoming active, use the *TScreen* properties *ActiveControl* and *ActiveForm*. While these properties define which object is receiving focus, the event handler itself is called immediately before focus is assigned. Consequently, you can use these event handlers to either respond to the fact that focus is being moved, or to prepare the object to which focus is moving. You cannot prevent the focus from changing. However, you *can* use the *SetFocus* method to set focus to another control from within the event handler.

Creating a *TScreen* Event Handler

There is one aspect of using the *TScreen* event properties that makes them more difficult to use than most other component event properties. Specifically, Delphi cannot create *TScreen* event handlers for you. That is, because you cannot work with a Screen component at design time, it isn't possible to select one using the Object Inspector. Consequently, the Object Inspector cannot create a *TScreen* event handler and assign it to the Screen's event properties.

Creating an event handler for a *TScreen* event property requires three steps:
1) Create a forward declaration for the event handler in the form's **type** declaration.
2) Write the implementation of the event handler.
3) Assign the event handler to the appropriate Screen event property.

Adding the forward declaration of the event handler to the form's **type** declaration involves declaring a procedure in the form's **published** section. For example, if you want your

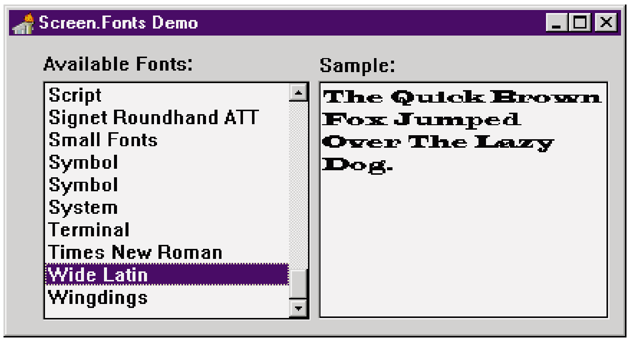


Figure 4: The FONTS.DPR project demonstrates one use of the *Screen.Fonts* property.

Figure 5: The ONFRMCHG.DPR project demonstrates the use of the *Screen.OnActiveFormChange* event handler to update the *Caption* property of the main form in an SDI application.

event handler to be named *FormChange*, you need to add the following statement to your form's **type** declaration:

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

Here's the complete **type** declaration for the form shown in Figure 5:

```
type
  TForm1 = class(TForm)
    MainMenu1: TMainMenu;
    File1: TMenuItem;
    Exit1: TMenuItem;
    OpenDialog1: TOpenDialog;
    SpeedButton1: TSpeedButton;
    Open1: TMenuItem;
    N1: TMenuItem;
    procedure FormChange(Sender: TObject);
    procedure Exit1Click(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure Open1Click(Sender: TObject);
  end;
```

(There are some Delphi developers who recommend you never add your own code to the **published** section of a **type** declaration created by Delphi. I disagree with this advice. By adding event handlers that you must create manually to the **published** section of a **type** declaration, you make them available during design time. This permits you to assign this same event handler to other objects' event properties at design time using the Object Inspector.)

The second step to creating a Screen object's event handler is to create the implementation. Because the event handler is a method of the form's class, the procedure name in the implementation section of the unit must be qualified with the form's class name.

For example, the *OnActiveFormChange* event handler used in the ONFRMCHG.DPR project shown in Figure 5 includes the following *FormChange* implementation:

```
procedure TForm1.FormChange(Sender: TObject);
begin
  if Self = Screen.ActiveForm then
    Self.Caption :=
      'Screen.OnActiveFormChange Event Property Demo'
  else
    Self.Caption :=
      'Detail form: ' + Screen.ActiveForm.Caption;
end;
```

The final step to manually creating an event handler for the Screen object is to assign the name of the event handler method to the Screen's event property. In most cases, this is performed in the *OnCreate* event handler for the form. The following is the *OnCreate* event handler for the form in the ONFRMCHG.DPR project:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Self.Top  := 1;
  Self.Left := 1;
  Screen.OnActiveFormChange := FormChange;
end;
```

This event handler performs two tasks. First, it places the SDI application's main form in the upper-left corner of the screen when the application starts. Second, it assigns the *FormChange* method to the *OnActiveFormChange* event property of the Screen component.

Conclusion

Screen is an instance variable of the *TScreen* class. Using this variable you can get information about, and control of, various elements of your Delphi application that would otherwise be inaccessible. ∆

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\97\FEB\DI9702CJ.

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.

At Your Fingertips

Delphi / Object Pascal

By David Rippy

No problem can stand the assault of sustained thinking. — *Voltaire*

How do I populate a DBListBox with the contents of a table?

As a general rule, it's good practice to populate DBListBoxes and DBComboBoxes dynamically at run time using a Table object — instead of editing the component's string list at design time. The obvious benefit is that when the *Items* list of the DBListBox or DBComboBox must be updated, you simply make the change in the table rather than in the program itself. Because the DBComboBox and DBListBox have a *DataSource* property, you may think they have the capability to populate themselves. However, this is not the case. The DBComboBox and DBListBox are for filling values in a table rather than showing you values from a table.

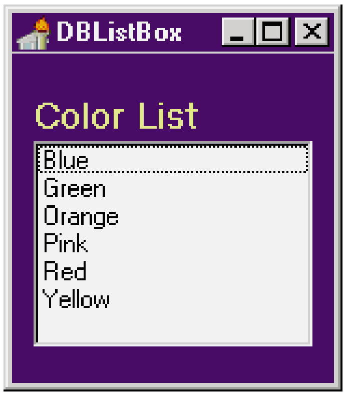
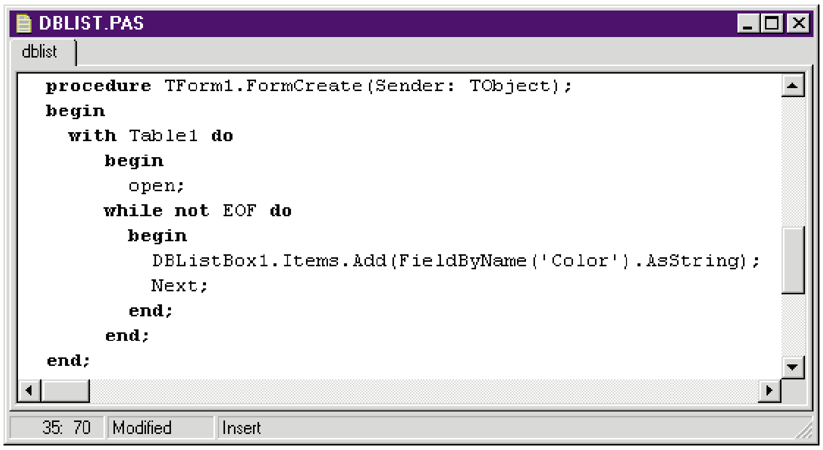


Figure 1: The DBListBox in this form is populated dynamically at run time.

Luckily, we can show values from a table in a DBListBox with just a few lines of Object Pascal. Figure 1 displays a form containing a DBListBox that will be populated at run time. Examine the code snippet in Figure 2. This code is located in *Form1*'s *OnCreate* method, causing *DBListBox1* to be populated with the names of several colors when the form is created.

In a nutshell, *Table1* is scanned using a **while** construct. As *Table1* (COLOR.DB) is iterated, the *Items* property of *DBListBox1* is populated with the values from the Color field in *Table1*.

For extra credit, see if you can figure out how to set the default value of the DBListBox to the first value of its *Items* list. — *D.R.*



```
procedure TForm1.FormCreate(Sender: TObject);
begin
  with Table1 do
     begin
       open;
     while not EOF do
       begin
         DBListBox1.Items.Add(FieldByName('Color').AsString);
         Next;
       end;
     end;
end;
```

Figure 2: The *OnCreate* event handler for *Form1*.

How can I repeatedly play a MIDI file?

With the advent of the "multimedia PC," users are beginning to request sound and animation to enhance the basic functions of their applications. Delphi makes it a trivial task to add such features using the MediaPlayer component.

MediaPlayer can play MIDI files (.MID), wave files (.WAV), animations (.AVI), and other forms of media. One of the most common requests is to have an application continually play a MIDI file in the background. Unfortunately, MediaPlayer doesn't feature a property that allows looping. Thankfully, this is an easy feature to implement.

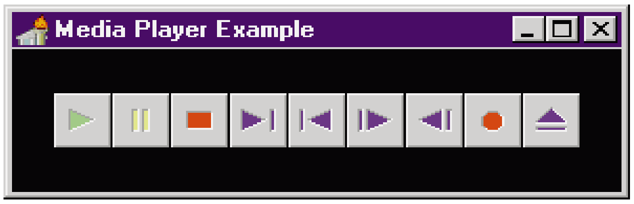


Figure 3: A form containing the MediaPlayer component.

```
procedure TForm1.MediaPlayer1Notify(Sender: TObject);
{ When MIDI song ends, restart }
begin
  with MediaPlayer1 do
    if (NotifyValue = nvSuccessful) and
       (Mode = mpStopped)            then
      MediaPlayer1.Play;
end;
```

Figure 4: *MediaPlayer1*'s *OnNotify* event handler.

The form in Figure 3 contains a MediaPlayer component (*MediaPlayer1*) that plays the MIDI file "Canyon". To cause Canyon to play repeatedly, we need to add some code. Figure 4 shows the *OnNotify* event handler for *MediaPlayer1*. *OnNotify* can trap four notification values:

- *nvSuccessful*
- *nvAborted*
- *nvFailure*
- *nvSuperseded*

The code in Figure 4 specifically looks for *nvSuccessful*, meaning that MediaPlayer successfully completed its most recent command.

We must also ensure that MediaPlayer has finished playing the song by checking its *Mode* property. If the value of *Mode* is *mpStopped*, a *Play* command is issued, causing Canyon to play again. This process will continue until a *Stop* command is issued to MediaPlayer by the user or through code.

Note that because the MediaPlayer works with .AVI files, this tip can be applied to animations as well. — *D.R.*

How can I determine the date and time stamp of a file?

There are many occasions when you may need to access a file's date or time stamp. Installation routines, for example, rely heavily on this information to decide whether an existing file should be replaced with a more recent version.

Delphi makes accessing the date and time stamp easy with the *FileAge* command. Unfortunately, the Help documentation is not much "help" explaining its use, and actually implies *FileAge* does something different.

Figure 5 shows a form containing the standard Delphi controls for selecting a file: DirectoryListBox and FileListBox. Pressing the **Date / Time** button will update *Label1* with the date and time stamp of the file selected in *FileListBox1*. Examine the *OnClick* event handler for the **Date / Time** button (*Button1*) in Figure 6. You will notice that three functions are involved:

1) First, the *FileAge* function is called and passed the filename currently selected in *FileListBox1*.
2) The return result is then converted to a *TDateTime* value by using the *FileDateToDateTime* function.
3) That return result is then converted to a string representation of the date by using the function, *DateTimeToStr*.

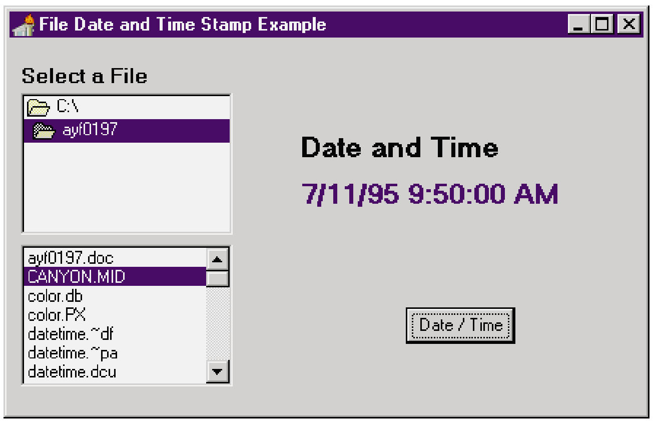


Figure 5: This form displays the date and time stamp for the selected file.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Label1.Caption := DateTimeToStr(FileDateToDateTime(
                      FileAge(FileListBox1.FileName)));
end;
```

Figure 6: The **Date / Time** button's *OnClick* event handler.

This final result is then assigned to the *Caption* value of *Label1*. — *D.R.*

Quick Tip: Working with the Delphi SpeedBar

If you use Delphi's IDE function keys and menu choices more frequently than the SpeedBar buttons, you can hide the SpeedBar. To show or hide the SpeedBar, de-select **SpeedBar** from the **View** menu. Alternatively, you can resize the Component Palette to show more or less of the SpeedBar. — *D.R.* ∆

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\97\FEB\DI9702DR.

David Rippy is a Senior Consultant with Ensemble Corporation, specializing in the design and deployment of client/server database applications. He has contributed to several books published by QUE. David can be reached on CompuServe at 74444,415.

By R. Bryan Ellis

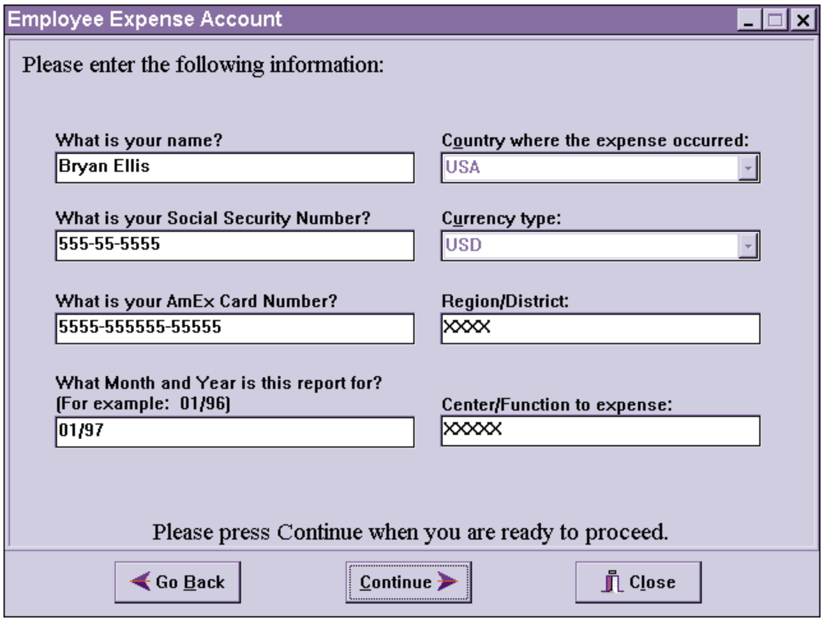
The Employee Expense Account

Monarch Technology Group Automates Corporate Expense Tracking

In the summer of 1996, an Atlanta-based Fortune 500 parcel service hired Monarch Technology Group, Inc. to design and implement an accounting-related application to track management expenses. The Employee Expense Account (EEA) was created to enable over 30,000 managers to automate their corporate credit card and personal expense accounts.

Before the EEA, managers tracked expenses manually and submitted a hard copy report to the accounting department. While this approach was somewhat effective, it left room for error because each person performed calculations and international currency exchanges without the benefit of automation.

Some employees, however, used a spreadsheet application for their calculations, producing reports that did not match the company's standard expense account forms. As such, management agreed that the best solution was to create a custom software product that provided the flexibility of the paper-based approach with the accuracy and automation of a spreadsheet solution.


Employee Expense Account

Please enter the following information:

What is your name? Bryan Ellis	Country where the expense occurred: USA
What is your Social Security Number? 555-55-5555	Currency type: USD
What is your AmEx Card Number? 5555-555555-55555	Region/District: XXXX
What Month and Year is this report for? (For example: 01/96) 01/97	Center/Function to expense: XXXXX

Please press Continue when you are ready to proceed.

Requirements

The requirements were considerable, with the biggest challenge being the users themselves. Although many potential EEA users are computer literate, a substantial percentage would see their first regular use of computers as a result of this product. Consequently, a primary design requirement was to implement an extraordinarily high level of user friendliness. Another primary requirement was to simplify the distribution requirements of the application. The clear preference was to create a solution that would exist as a single .EXE file that could be distributed via a single diskette or a network.

After determining the primary requirements, the first step was to decide which development tool to use. There was significant internal experience with Visual Basic, but the need for a single-file solution prompted the selection of Delphi. Monarch Technology Group chose Delphi 1 because most of the users operate on the Windows 3.x platform.

Having selected a programming tool, the next step was to create a report module that could print a form precisely matching the client's standard expense tracking form. Although reports are usually implemented later in the development process, this task was given primary importance because the report was complex and specialized. The idea was to verify that it would be possible to create the report in a reasonable amount of time before continuing development on other pieces of the application.

Given the fundamental importance of the hard copy report, significant consideration was given to the reporting tool. Because a primary requirement was to fit the entire application on a single diskette, many reporting tools — including ReportSmith and Crystal Reports — were automatically eliminated due to the size of their run-time files. Several other reporting tools were evaluated, but in the end, Monarch Technology Group manually coded the report using Delphi's printer canvas (*TPrinter.Canvas*). Although this made implementation of the report more time consuming, it provided the necessary level of flexibility with a very small disk footprint.

The next step was to create the user interface. To achieve user friendliness, two approaches for the user interface design were evaluated. The first approach emphasized the traditional menu-oriented look of most Windows applications. This approach would maximize the inherent familiarity of the system's interface and users would immediately be comfortable with the application.

However, this didn't consider users who were unfamiliar with computer software and would, therefore, be ill at ease with any application, no matter how standardized it was. With this in mind, Monarch Technology Group prototyped a second style of user interface that involved an interview metaphor similar to Microsoft-style wizards. With this approach, the application could be broken down into a series of prompt screens, each with a specific theme. This provided for a much higher initial level of ease of use, but the trade-off would be a relative loss of flexibility. After evaluating both styles of user interfaces, the wizard style was determined to be most appropriate for the client's user population because of the usability advantages associated with that style of interface.

Application Profile

The Employee Expense Account was developed by R. Bryan Ellis of Monarch Technology Group, Inc. to provide a simple and automated system for tracking and managing corporate expense accounts.

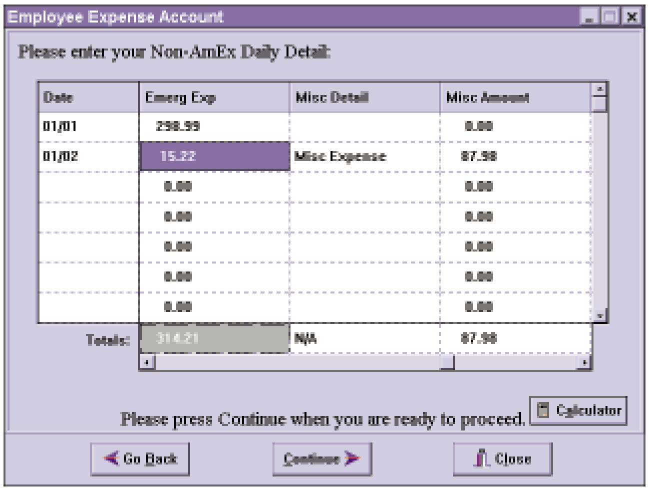
Target Audience: Employees needing to manage expense accounts and corporate credit card usage.

Users: Over 30,000 parcel service managers.

Third-Party Tools: Orpheus by TurboPower Software Company.

Monarch Technology Group, Inc.
P.O. Box 1247
Stockbridge, GA 30281
Phone: (800) 377-0319
Fax: (770) 570-5079
E-Mail: Internet: bryan@monarch-technology.com
Web Site: http://www. monarch-technology.com

TurboPower Software Company
P.O. Box 49009
Colorado Springs, CO 80949
Phone: (800) 333-4160
Fax: (719) 260-7151
E-Mail: Internet: orders@tpower.com
Web Site: http://www.tpower.com



Please enter your Non-AmEx Daily Detail:

Date	Emerg Exp	Misc Detail	Misc Amount
01/01	298.99		0.00
01/02	15.22	Misc Expense	87.98
	0.00		0.00
	0.00		0.00
	0.00		0.00
	0.00		0.00
	0.00		0.00
Totals:	314.21	N/A	87.98

Please press Continue when you are ready to proceed.

Breaking It Down

After some close study, it was determined the application could be partitioned into six categories:
1) User Information,
2) ATM Advances,
3) Daily Expenditures,
4) Corporate Credit Card Expenditures,
5) Mileage Reimbursements, and
6) Reimbursement Allocations

Each category was to become an individual prompt screen which the user would be presented in a logical sequence. The goal of each prompt screen was to narrow the user's focus to a particular category and to acquire one or more specific pieces of information about that category.

In most cases, the information to be gathered was simple enough to use an Edit component for input. But in the case of data to be entered for multiple days, such as Daily Expenditures and Mileage Expenses, a more elaborate input mechanism was required. In such cases, the grid component in TurboPower's Orpheus was used. Although Monarch Technology Group accomplished all the requirements using the Orpheus grid, the product definitely favored flexibility over simplicity, requiring relatively large amounts of code to accomplish simple feats. In the end, however, the product's flexibility was a major boon to the development effort, and it proved to be more than sufficient for Monarch Technology Group's requirements. Of particular value was the control that Orpheus offers over cell focus changes and also the ability to specify a data type and edit mask for grid columns.

Aside from the major categories previously listed, several other screens had to be implemented. These screens allow the user to make simple selections and navigation decisions.

For example, one screen allows the user to indicate whether the expense account was for expenditures made using US dollars or a foreign currency. Another screen offers a print preview for the user to scan the newly created expense report. A final screen enables the user to print the report or save the data to disk.

Success

After experimentation and fine tuning, the Employee Expense Account application was rolled out. It met each of the design requirements, including user friendliness, distribution simplicity, and accurate and automated calculations for expense account data. EEA is now being used by thousands of managers to track all types of foreign and domestic expenditures. ∆

R. Bryan Ellis is principal of Monarch Technology Group, Inc., a consulting firm focusing on process improvement and corporate software development. He can be reached at (800) 377-0319, or visit Monarch's Web site at http://www.monarch-technology.com.

table of CONTENTS

By Douglas Horn

Light Lib Magic Menus

DFL Software's Tool Enhances Menus with Images

Some programming tools offer Delphi developers a radically different approach to programming, allowing them to do things they never dreamed possible. Light Lib Magic Menus from DFL Software, Inc. isn't one of these tools; it offers Delphi developers only cosmetic improvements. Despite this, many developers may be eager to use its features.

Magic Menus gives developers the ability to add images to standard Delphi menus. Although the product has several shortcomings in its documentation and installation routine, the core functionality of the code performs as promised. Adding Magic Menus to an application is simple, and can be surprisingly powerful, *after* the developer gives up on the documentation and plays around with the components.

What to Expect

Simply put, Magic Menus allows a developer to add bitmap images to a Delphi application's menus. The images may be backgrounds or icons, and will work with main, sub, and popup menus. Magic Menus accomplishes this by replacing the standard Delphi Form and Menu unit with its modified versions. For this reason, developers who use customized forms or menus in their applications should be aware that they may need to re-modify their components to incorporate Magic Menus. Magic Menus also uses two custom components to hold properties for menus and menu items.

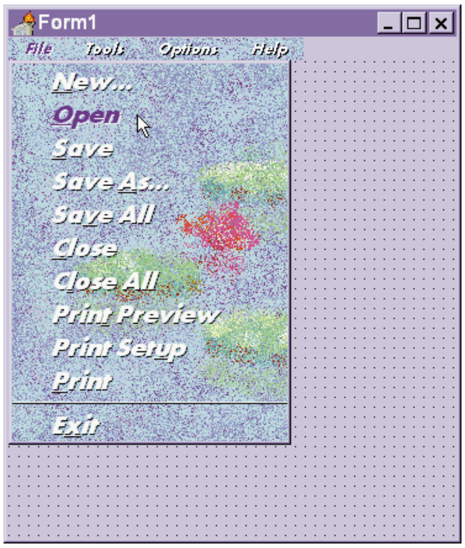


Figure 1: Magic Menus lets Delphi developers add images as backgrounds and specify the menu text's typeface, size, style, and color, improving the menu's appearance.

Because Magic Menus must replace standard components, it contains two sets of components — one for Delphi 1 (16-bit) and another for Delphi 2 (32-bit). Each version has the same properties and, from the programmer's and user's standpoint, behaves identically. The menus attainable from Magic Menus can be very attractive, and are highly customizable. For example, a developer can place a background image on any menu, as well as control the menu text's typeface, size, style, and other attributes — something not possible with standard menus (see Figure 1). It's difficult to deny that menus with images and improved typefaces liven up applications. They may not contribute to a uniform-looking operating environment, or add functionality, but they *look* great. And for some programs, such as kiosk presentations and educational programs, the improved menus contribute to usability.

Types of Menus

Magic Menus offers seven menu styles.

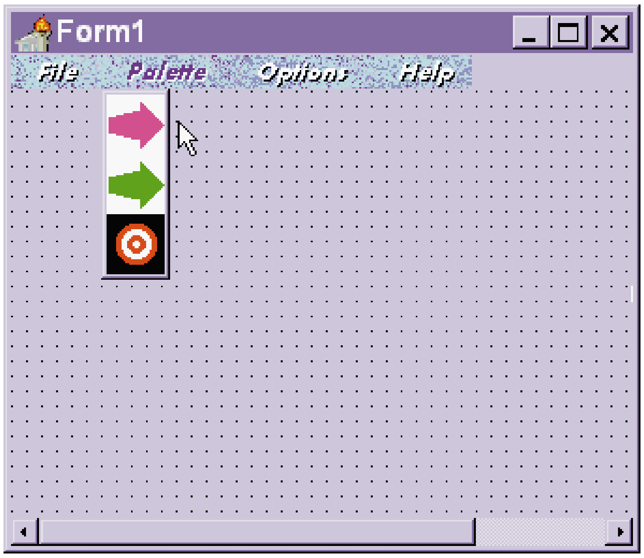


Figure 2: Icons and images can be used instead of text, allowing developers to create menu tool palettes and other effects.

The Default style is a standard Windows menu. The Bitmap style displays a bitmap image in lieu of any text, and the BitmapText style displays a background bitmap and a text label. Similarly, the BitmapUpDown style allows developers to create a two-state "button," with one image used for the up state and another for the down state. This can be used to create button menus for things such as tool palettes (see Figure 2). The IconText style places 16- or 32-bit icons next to text labels, much like those on the Windows 95 start Menu. Finally, the User and Custom styles add 3D and button-like effects that some developers find useful.

Magic Menus also allows custom icons to be used for checked menu items. Additionally, it allows special 3D effects to be enabled or disabled for "grayed" menu items. Effects may be applied throughout menus, or to specific submenus or independent items. By combining various properties, it's possible to create impressive visual effects. It's not difficult to imagine creating user interfaces that operate independently of language for specific multilingual applications.

Using Magic Menus with Delphi Applications

After installation, Magic Menus components are easily added to Delphi applications. Because the customized form and menu components replace Delphi's original components, the developer doesn't need to worry about specifically choosing these when building an application. Developers can simply build menus as they normally would with Delphi's Menu Editor. To add Magic Menus' effects, the developer simply needs to add the appropriate components to the form — MagicMenu for menus, submenus, and popups, and MagicItem to assign specific effects to individual menu items.

After the MagicMenu and MagicItem components have been added, they can be attached to a menu by associating them with an existing menu component (as one would associate a DataSource component with a table). All properties can be modified at the MagicMenu or MagicItem level, and a single MagicMenu can control all menus, if desired.

Applications using Magic Menus compile just as any other Delphi application. Program performance doesn't seem to suffer either, even when large bitmaps and complex menu schemes are used. For slower computers, the component files may be edited to build the components statically rather than dynamically, which may offer some performance benefits. Magic Menus is a Delphi VCL (as opposed to an OCX or other non-Delphi tool), that serves as a wrapper around a dynamic linked library (LLU16.DLL or LLU32.DLL).

Annoying Installation

These days, it's rare for a software reviewer to call any great attention to an installation routine. Windows applications have adopted an increasingly standardized approach, and most developers have come to realize that their customers want to know what is being installed, as well as to be able to control where it goes. Unfortunately, DFL's installation routine flies in the face of this philosophy. The Magic Menus installation routine is one of the more aggravating this reviewer has encountered in some time.

The unsuspecting developer's first clue that something is amiss with the installation is when he notices that Magic Menus ships on six floppy disks — seemingly far too many for such a simple tool. When the installation routine begins, it asks what development environments are on the system (including Delphi 1 and 2, Microsoft Foundation Classes, CA-Visual Objects, OCX, MS Visual Basic, and Visual FoxPro), and informs the user that it will install the tool for all applicable environments. That won't sound too bad to most Delphi developers, especially those who use both the 16- and 32-bit versions. What the program neglects to say is that it will install evaluation versions of *all* Light Lib products to the hard drive. Ostensibly, this offers users a chance to sample these tools, but few will want to. Another annoyance is that, after the first twenty minutes of use, a screen — reminding the user that this is an evaluation copy — pops up every minute that the product is being evaluated. Users who don't want this on their system have no way to avoid installing the evaluation copies. In fact, it's never mentioned in the documentation.

Aggravating Documentation

If the installation routine is annoying, the documentation — or lack thereof — is downright aggravating. This reviewer applauds the trend toward online-only documentation, but users need to be given some sort of "Getting Started" guide to tell them what is required of their system and what to expect from the installation. Magic Menus includes neither a printed sheet nor a README file that can be viewed prior to installation. If either were available, users could at least be informed in advance of the peculiar installation routine.

The documentation consists of a meager Windows Help file and a single sample application. The Help file only briefly describes the MagicMenu and MagicItem classes, as well as their ancestor class, MagicAbstract. The pertinent informa-

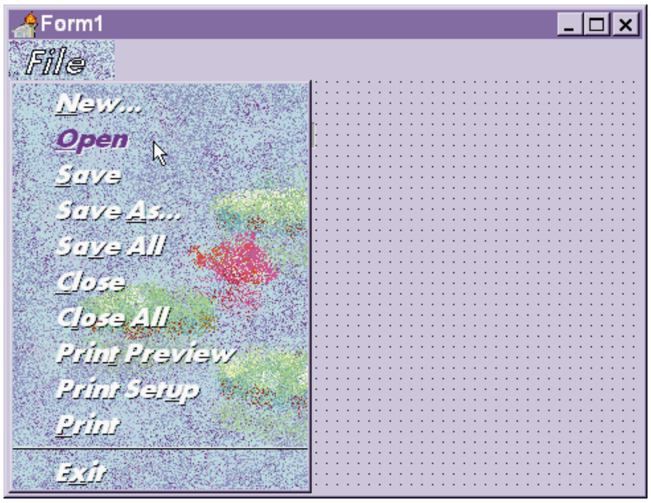


Figure 3: The text of a menu heading cannot exceed standard size. The work-around is to create a bitmap image of the desired text, although this has some programmatic drawbacks.

tion contained in this Help file could easily fit on four printed pages. It serves as a basic reference, but offers no examples of how to implement Magic Menus into an application.

The single sample file allows a developer to get a basic understanding of how to use the tool. That is, it shows how to associate MagicMenu and MagicItem components with Delphi menus and menu items, and how to set the basic properties. These are the fundamentals, and there are few Delphi developers who will find them hard to grasp. But to get the full use of the product — to be able to recreate the menu effects shown in the Magic Menus advertisements — takes some creativity and knowledge of the product. To this end DFL is no help whatsoever. They offer no tutorials beyond the extremely basic sample application. For example, after playing with the product for a short while, developers will discover that while they can set the size of the menu text as shown in the sample, the menu items on the heading cannot be enlarged beyond the standard menu height. This can be frustrating, as the ads and brochure show that large menu headings are possible. The developer is left to his own devices to figure out that the only way to enlarge the menu heading "font" is to create a bitmap with the desired word in the desired font, then display this bitmap instead of text (see Figure 3).

In fact, after developers explore Magic Menus at some length, they may question whether DFL fully understands the capabilities of its tool. For example, the text file in the sample application suggests that to prevent an image from inverting its colors when selected, the developer should set the *BitmapOperation* property to *boCopy* and the *BitmapRelativePos* property to *False*. However, this doesn't do what most developers would expect — more reliable results can be obtained simply by setting the *BitmapDown* and *BitmapUp* properties (controlling which images are shown in the selected and non-selected menu states) to the same image.

Conclusion

DFL's Light Lib Magic Menus is a fine tool for upgrading the appearance of menus in Delphi applications. The only noticeable bug in the tool itself is the inability to control the text size of a menu heading. The work-around to this is acceptable, but still offers the occasional challenge (e.g. changing menus or providing a consistent menu appearance). Aside from this shortcoming, the tool does what no other Delphi tool can. Unfortunately, the clumsy installation routine and substandard documentation mar the tool's usefulness. Delphi developers can likely overcome the poor documentation, but would get a lot more immediate usefulness if the product were explained more thoroughly. ∆

INFORMANT FACT FILE

Light Lib Magic Menus v. 1.41 is a cosmetic improvement for Delphi that allows images to be used in menus. The product works as promised, but its usefulness is marred by substandard documentation — developers must discover the ins and outs of Magic Menus on their own.

DFL Software, Inc.
55 Eglinton Avenue East, Suite 208
Toronto, ON, Canada M4P 1G8

Phone: (416) 487-2660
Fax: (416) 487-3656
E-Mail: sales@dfl.com
CompuServe: GO DFLSW
Web Site: http://www.dfl.com
Price: US$99

Douglas Horn is a freelance writer and computer consultant in Seattle, WA. His Web site (http://www.halcyon.com/horn/default.htm) contains numerous articles and programming samples from *Delphi Informant* and other magazines, but no images of his dog. He can be reached via e-mail at horn@halcyon.com.

Flawed Classic Still Manages to *KickAss*

KickAss Delphi Programming could easily be titled "Advanced Topics in Delphi." Based on its content and approach, this book certainly deserves such a description, but *KickAss Delphi* is unquestionably more attention-grabbing.

Entirely missing (thank goodness) is an introduction/tutorial of Object Pascal. Instead, this multi-author extravaganza contains a wealth of powerful Delphi code and useful new components. Unfortunately, the accompanying CD-ROM contains as many problems: pesky code errors, missing files, missing 32-bit versions of code, and incorrectly named files. While many of these problems are easily correctable (as we'll see), others are not.

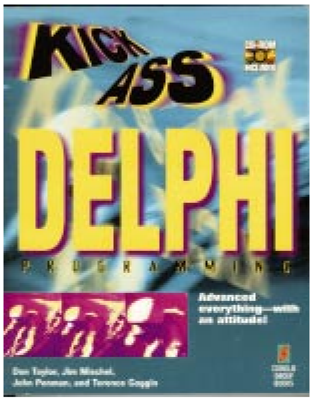
In the first three chapters, Jim Mischel discusses several interesting topics from an advanced perspective. Chapter 1 provides a practical introduction to writing 32-bit console applications (the Windows 95 equivalent of a DOS box). Specific topics include text file filtering, processing command-line statements, reading/writing text files quickly, and adding the filtering program to the Repository to use as a template in future projects.

Chapter 2 discusses important principles of drag-and-drop beyond Delphi's basic implementation. Chapter 3, which covers DLLs, discusses some of the more esoteric aspects of these useful libraries, and provides an example of wrapping a form in a DLL.

The highlight of Chapter 2 is a new Form component that accepts files dragged from a Windows File Manager program. (I found it works equally well with Norton Navigator.) Here, I encountered the first "bug." Trying to install the Form component, I received several compilation errors. After a few e-mail messages, Mischel pointed me in the right direction to fix things. The main drag-and-drop unit exists in two incarnations: FMDD and FMDD1. The latter was designed for use by the File Manager Drag and Drop Component (FMDDFORM.PAS); but the former was mistakenly listed in the **uses** clause and in several statements in the **implementation** section. Here's the fix to make FMDDFORM.PAS work:

- Change FMDD to FMDD1 in the **uses** statement
- Change every instance of FMDD.* to FMDD1.*

With these changes, I could install the new component



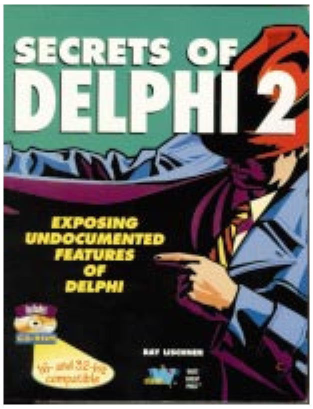
onto the Component Palette. Similar changes must be made in the final project that uses this component.

The next two chapters, by John Penman, discuss the

"Flawed Classic Still Manages to KickAss" continued on page 52

Secrets Revealed

Secrets of Delphi 2 by Ray Lischner is a gold mine of information on a wide variety of topics — many of which are poorly documented, or not documented at all. While many of these topics are advanced in nature, the author's approach in presenting them is accessible to most Delphi programmers. If you have been working with Delphi for several months or have worked your way through one of the introductory books, and are anxious to explore more advanced topics, this book may be the perfect place to begin. Whether your objective is to create custom components, extend the capabilities of Delphi's programming environment, or to simply write powerful Windows applications, you'll find a wealth of information in these 800-plus pages.



Starting with a chapter titled "The Readiness is All," *Secrets* moves through increasingly complex and advanced topics. Each chapter is organized in similar fashion, beginning with an introduction to the topic, a capsule sketch of the main sub-topics, a full discussion of the topic accompanied by numerous code examples, and a brief summary.

Lischner devotes the second chapter to an introduction of components and properties, discussing the differences between design time and run time, explaining the *ComponentState* property and default property types, and introducing RTTI, which is further explained later in its own chapter. There Lischner discusses how to retrieve RTTI and explains the different kinds of RTTI and their use in published properties.

Other chapters enhance our understanding of components and their properties. For example, three chapters deal solely with property editors: the first provides the fundamentals of writing property editors, the second explains some of their undocumented features, and the third lists and explains all the property editors Delphi doesn't document. This latter chapter also explains how to sub-class any of Delphi's property editors, and provides an example of

"Secrets Revealed" continued on page 53

Flawed Classic Still Manages to *KickAss* (cont.)

Windows Winsock API, and create a component that encapsulates the entire interface. Again however, I had problems compiling the code on the CD-ROM. I thought the code was 32-bit; it's actually intended for Delphi 1. Interestingly, some techniques that were legal in Delphi 1's implementation of Object Pascal are no longer legal in Delphi 2. For example, in one method in the support unit, WSAPIMP.PAS, there was an attempt to modify a **for** loop variable within the **for** loop; this simply won't fly in Delphi 2. You will need to assign another variable, *j*, to the current value of the **for** loop variable, *i*, and modify the new variable.

Now, at least, the code will compile without error in either version. However, since the code on the CD-ROM for this chapter was 16-bit, it didn't work under Delphi 2. Penman sent me the complete 32-bit code, which is very different from the code that appears on the CD-ROM. It worked fine.

For me, these two chapters were in many ways the high point of the book. I learned several new things about working with online communications in Delphi. Furthermore, Penman provides an excellent model for wrapping any previously unsupported interface into a Delphi component — a model I'll use in the future.

Based on an article from Coriolis' *Visual Developer* magazine, Ray Konopka presents his Def Leppard screen saver in Chapter 8. As a bonus, you get two new components: a trackbar and color combo box that gives you the selectable names and actual colors. There are also useful discussions of animation techniques and Windows callback functions. Again however, the code on the CD-ROM did not include the 32-bit version, but Konopka did send the version for Delphi 2. I encountered one problem that was easy to solve. The name of the bitmap used for the trackbar component was the same as that for the color combo box component, generating a Duplicate ID error when I tried to install the components on the Component Palette. Using the Image Editor, I changed the name of the Bitmap Resource for the trackbar component to TRZTRACKBAR (it must be upper case) and everything worked fine.

Terence Goggin contributed two interesting and useful chapters, one documenting the new floating point Math unit in Delphi 2, the other providing techniques and components to create dynamic user interfaces. This time, the code on the CD-ROM was 32-bit, but all the long filenames were in short DOS form, causing numerous problems for the compiler. And in Chapter 10, several required files weren't even present! (When requested, Goggin sent these to me.) You must make the filename changes in the table in Figure 1 for the code to compile properly.

Towards the middle of the book is an interesting chapter, "Problems with Persistents and Other Advice," that includes valuable tips, tricks, and techniques. You'll learn about a "seeming" bug with the *TPersistent* class, disabling RadioGroup buttons, a new drag-and-drop cursor to add a Borland Pascal 7-style *ForEach* iterator method to the *TStringList* class, and more.

Chap.	Short File Names	Required File Names
9	Polypr~1.*	PolyProject.*
9	Statsp~1.*	StatsProject.*
9	Statsu~1.*	StatsUnit.*
10	Dynami~1.*	DynamicForm.*
10	DynamicForm.DPR	DynamicForms.DPR
10	DynamicForm.RES	DynamicForms.RES
10	OIComp~1.*	OICompDemo.*
13	Splash~1.*	SplashDem.*

Figure 1: Changes for the filenames.

In Chapter 11, Richard Haven presents some interesting approaches to working with data in a table in a tree-like, hierarchical fashion. The discussion includes using queries and SQL to accomplish these tasks, as well as getting around the conditional branching limitation in SQL. Again, there was a problem running the code on the CD-ROM, and an internal Borland compiler error was generated!

The last four chapters by Don Taylor re-introduce the saga of "Ace Breakpoint," a character familiar to readers of the *Delphi Programming EXplorer* series. These chapters revisit a few of the topics introduced earlier, but in different ways, including drag-and-drop techniques and dynamic user interfaces (resizing forms). At the same time, many new techniques are discussed. The floating toolbar presented in Chapter 14 is fascinating; however, it cries out to be made into a component. The message broadcasting project in Chapter 15, demonstrating how to set up communication between two independently running applications (a sender and receiver), is brilliant. And the Win95 Walking program that provides information about active modules should find a place in many readers' suites of utilities.

Despite the considerable frustrations I experienced with the CD-ROM, I still recommend this book. The text itself is excellent, and on that basis, *KickAss Delphi Programming* deserves to become a classic. At the same time, however, I cannot over emphasize the seriousness of the problems with the code on the CD-ROM. I trust and expect that Coriolis will find an adequate solution both for the readers who have already purchased the book and for the copies yet to be distributed.

— *Alan C. Moore, Ph.D.*

KickAss Delphi Programming by Don Taylor, Jim Mischel, John Penman, Terence Goggin, et. al. The Coriolis Group, 7339 E. Acoma Dr., Ste. 7, Scottsdale, AZ 85260, (602) 483-0192.

ISBN: 1-57610-044-8
Price: US$39.99 (504 pages, CD-ROM)

Secrets Revealed (cont.)

creating a dialog box to edit a property.

There is a separate chapter on Component Editors. After an introduction, Lischner discusses the two basic classes, *TComponentEditor* and *TDefaultEditor*, and their functions. It provides an example of deriving a new component editor, and concludes with an overview of Delphi's built-in component editors. A chapter entitled "Metacomponents" introduces this interesting component-type — a component whose sole function is to control the behavior of other components.

In chapter 15, Lischner starts by introducing the *TFieldDataLink* class, discusses its methods and properties along with those of the *TDataLink* class, and concludes by building a simple data-aware control (a data-aware button) and a much more complex one (a data-aware rich text editor). Finally, there is a superb chapter on writing online component Help files. This chapter includes a nifty utility called HdxDump which reads a binary Help index (HDX) file and extracts useful information to a text file, including keywords, titles, contexts, and the source *.HLP files.

Other chapters present information which will be of particular use to application developers. Chapters on the *TStream* class and the *TList* class go beyond simply explaining the uses of these useful constructs. After explaining some of the limitations in these classes, the author derives new classes which address these limitations. Other useful chapters discuss some of the more esoteric aspects of Delphi forms, the Windows registry, and the Windows heap. One of the final chapters discusses Delphi experts and how they can be used to extend the programming environment. This chapter alone — which surpasses any discussion I have seen on this topic — will ensure that *Secrets* continues to spend more time off my bookshelf than on it.

The accompanying CD-ROM includes all the code in the book (and some that had to be left out), utilities, experts, and seventeen new custom components. You can install all the components in either Delphi 1 or 2 from a single registration unit.

I recommend this book highly, particularly for intermediate or advanced Delphi programmers. Unlike some of the advanced books I have read, this book assumes only the most basic prior knowledge of Delphi. This approach will be particularly welcome to intermediate programmers who may have heard about RTTI or the Virtual Method Table, but know little more than their names. Advanced developers may find the first few pages of each chapter a bit boring; however, the unexpected gem encountered a few pages later makes the journey worthwhile. Nowhere have I found so much interesting and obscure information in one book. I think Ray Lischner's *Secrets of Delphi 2* will be a very popular book.

— Alan C. Moore, Ph.D.

Secrets of Delphi 2 by Ray Lischner. Waite Group Press, 200 Tamal Plaza, Corte Madera, CA 94925, (415) 924-2575.

ISBN: 1-57619-026-3
Price: US$49.99
(831 pages, CD-ROM)

Delphi in the World of Shrink-Wrap

While RAD tools like Visual Basic have proven popular for custom software and shareware, the world of Windows shrink-wrapped software has traditionally been dominated by the C and C++ programming language family. Within this closed camp, however, Delphi is quietly emerging as an attractive alternative to C++. Why are some well-known independent software vendors (ISVs) using Delphi for product development? And — perhaps as important — why aren't more looking Delphi's way?

Enabling Rapid Development. Not surprisingly, the single most important reason software houses are considering Delphi is that development is much faster than with Visual C++ or Borland C++. The importance of Delphi's visual IDE rang home recently when I was using Adobe PageMaker 6.0. Spoiled by applications like Delphi, Word, and Excel, I became frustrated by the lack of common UI features available in PageMaker, such as dockable toolbars, extensible toolboxes, and so on. The only logical reason I can think of why these standard features are not present in their 32-bit version was the additional time that would have been added to the product's development schedule. While such UI features may add considerable development time to a C++ application, if PageMaker were written in Delphi, I could add such features in a heart beat.

Another factor in Delphi's favor is that a Delphi-based application has none of the shortcomings of other RAD products. While you can spot a Visual Basic application a mile away, you cannot tell a Delphi application from one developed with C++.

Third, Delphi is proving attractive because of its ability to integrate tightly with C++. You can already use Delphi to create or work with .OBJ files. More important, Borland's new "Delphi for C++" product — named CBuilder — destroys the barriers between C++ and Object Pascal. Now, you can easily mix and match two code bases, and even keep the Visual Component Library standard you are familiar with. Even if you have an existing C++ code base, you can use Delphi or CBuilder going forward.

Against the Grain. Given these compelling advantages, why aren't more ISVs jumping on the Delphi bandwagon? Perhaps the biggest reason is that Object Pascal and RAD environments are simply counter-culture. The commercial software world is dominated by C++, and, on the Windows platform, MFC (the Microsoft Foundation Class Library). Delphi, therefore, goes against the grain of most software shops.

And there's more than a psychological barrier; there are practical ramifications to this issue. It's far easier to find quality C++ expertise than strong Delphi skills, especially in commercial software circles. For example, one ISV I know of wants to use Delphi in a future product, but is having trouble finding experienced engineers.

Another hindrance is the lingering issue of the viability of Borland, especially with the recent exodus of Anders Hejlsberg, Delphi's chief architect. Pondering the use of Delphi, an engineer from a well-known vendor recently asked me for my thoughts about the product's future. Fortunately, Borland has persevered through many trials since the early 1990s, and will likely do so again. Even if the company itself does not survive, Delphi surely will.

A final shortcoming of Delphi is that it supports only the Microsoft Windows platform. Although the Windows platform predominates, Delphi would make a poor choice as a development environment if you need to develop cross-platform products. With the rise of Java, this limitation, which seemed so inconsequential in 1995, is seen by some as a major limitation in 1997.

Wrap It Up. If you produce off-the-shelf Windows software, you owe it to yourself to check out Delphi. It will not be the best choice for all situations; for some, however, it will be a tool that increases chances for success in a highly competitive marketplace. ∆

— Richard Wagner

Richard Wagner is the Chief Technology Officer of Acadia Software in the Boston, MA area. He welcomes your comments at rwagner@acadians.com or on the File | New *home page at http://www.acadians.com/filenew.*

table of CONTENTS