Beating the System: Animated Icons And Cursors, 2

by Dave Jewell

In last month's *Beating the System*, I presented the code for `TAniIcon`, a class which descends from `TGraphic` and implements the low-level code needed to prise apart an animated icon file (extension of .ANI), extract and 'play' the various icon images contained therein. This month, as promised, we're going to use `TAniIcon` as a basis for creating some useful Delphi components which make use of animated cursors.

Firstly, one small word of warning. Last month's `TAniIcon` class has changed just a little: the `Clear` method is now public and includes more re-initialisation code than it did last month, and I've also tweaked the `LoadFromStream` routine. Therefore, please be sure to use the code that's included on this month's disk or else things won't compile properly. All the code has been developed and tested under Delphi 3.

A Beginner's Guide To Property Editors...

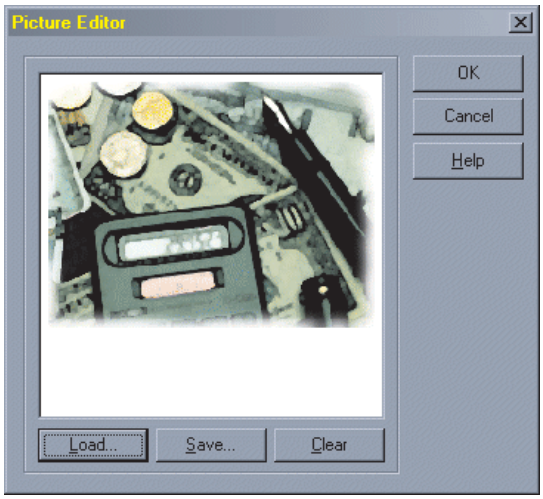
Later in this article we'll develop a couple of new components, both of which make use of the `TAniIcon` class, publishing an object of this class as a property. However, this implies that we need some way of assigning to an object of `TAniIcon` at design-time from the form designer. You've probably realised that I'm talking about the need for a custom property editor capable of manipulating `TAniIcon` properties. Just in case you've never created a property editor before, I'm going to take this opportunity to explain the basics along the way. If you develop many Delphi components of your own and have a need to regularly create custom property editors, I'd strongly suggest that you get hold of one of the excellent in-depth books that cover this topic, such as Ray Lischner's *Hidden Paths of Delphi 3*. The Inprise documentation is woefully inadequate when we get into this sort of area.

The simplest way of implementing a property editor for `TAniIcon` is to derive a new editor from the existing `TClassProperty` class, which you'll find defined inside the DSGNINTF.PAS file, assuming you've got the VCL source code. The code for my new property editor can be found in Listing 1. As you can see, the class name is `TAniIcon PropertyEditor`. As is usual with the Tools API, deriving a new property editor is simply a matter of 'filling in the blanks' by overriding the methods which need to implement behaviour that differs from that of the ancestor class. In this case, four different methods are involved, and I'll be dealing with each of them in turn.

Firstly, there's the `GetAttributes` function. This tells Delphi (or more specifically, Delphi's Object Inspector) what sort of property editor it's dealing with. In our case, we want to emulate the behaviour of the built-in picture editor, creating a modal dialog which allows the user to select a new .ANI file from those present on disk. For this reason, we return a function result comprising only the `paDialog` attribute. This tells the Object Inspector that editing this property will invoke a modal dialog, and the Object Inspector accordingly displays a small button with an ellipsis (three dots to thee and me...) when this particular property is selected in the Object Inspector.

The next method we need to override is called `GetValue`. Again, this method is invoked by the Object Inspector. It's called in order to obtain a value string which is displayed in the right hand column of the Object Inspector to show the value which is currently assigned to the property. Again, we want to emulate the behaviour of existing `TGraphic` descendants such as `TIcon`, so we display the string `(None)` if the underlying `TAniIcon` property isn't currently assigned to an animated icon. If it is, then we display the class name in parentheses: again, this is for consistency with existing property editors.

In order to access the actual property itself, we make use of another method called `GetOrdValue`. This returns the current value of the property as a 32-bit quantity. If this were a simple scalar property such as an integer, the 32-bit value would correspond directly to the property value. However, in the case of `TAniIcon`, the property type is a distinct



➤ *Figure 1: Here's the standard Delphi Picture Editor. This Property Editor dialog is implemented in the file PICEDIT.PAS, one of the 'secret' files whose source code Inprise choose not to include along with the rest of the VCL code.*

Delphi class, and in such situations, the value returned by `GetOrdValue` is an actual object instance of the given type which we have to cast to `TAniIcon` before use.

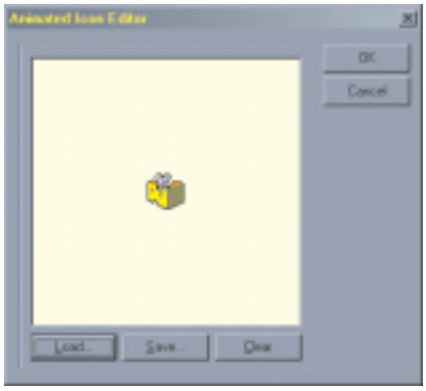
The third method we need to override is `SetValue`. Again, I've done this for consistency with existing property editors. The `SetValue` method performs the reverse job to `GetValue`: given a user-supplied string, it converts this string into a property value that 'makes sense' for this type of property. To see how this works, fire up Delphi, create a new form, and set the form's `Icon` property to any old icon file that you have lying around. If you then click inside the value column of the `Icon` property (within the Object Inspector) and manually set the value of `Icon` to an empty string, you'll see that this is interpreted as a clear operation, resetting the property to `(None)`. The reason this works is because Delphi's built-in picture property overrides the `SetValue` method in exactly the same way that I've done here.

Last, but definitely not least, is the `Edit` method. This is where the rubber hits the road, because the `Edit` method is what gets called when the aforementioned ellipsis button is clicked to invoke the property editor itself. As you can see from the code listing, my `Edit` method works by making use of the `TAniIconEditorDialog` class, also defined within the same unit, the end result being shown in Figure 2. If this property editor looks strangely familiar to you, it's probably because I... err... *recycled* the form resource that's used by Delphi's built-in picture editor. If you wanted to get fancy, you could (for example) make this form a little larger and add a couple of `TLabel` components to display the animated icon's `Author` and `Title` properties as described last month.

The `TAniIconEditDialog` form is pretty standard stuff: the form contains a panel component, within which is a `TPaintBox`. The `Canvas` property of the paintbox is used to render the currently selected animation much like the example code that I showed you at the end of last month's article. The form

➤ *Listing 1*

```
unit UCAniIconEdit;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, ExtDlgs, ExtCtrls, StdCtrls, UCAniIcon,
  DsgnIntf;
type
  TAniIconEditorDialog = class(TForm)
    OKButton: TButton;
    CancelButton: TButton;
    GroupBox1: TGroupBox;
    ImagePanel: TPanel;
    ImagePaintBox: TPaintBox;
    Load: TButton;
    Save: TButton;
    Clear: TButton;
    OpenDialog: TOpenPictureDialog;
    SaveDialog: TSavePictureDialog;
    Timer1: TTimer;
    procedure ClearClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure LoadClick(Sender: TObject);
    procedure SaveClick(Sender: TObject);
    procedure Timer1Timer(Sender: TObject);
  private
    Icon: TAniIcon;
  public
    procedure Reset;
  end;
  TAniIconPropertyEditor = class (TClassProperty)
  public
    function GetValue: String; override;
    function GetAttributes: TPropertyAttributes; override;
    procedure SetValue (const Value: string); override;
    procedure Edit; override;
  end;
implementation
{$R *.DFM}
procedure TAniIconEditorDialog.FormCreate(Sender: TObject);
begin
  Icon := TAniIcon.Create;
  Save.Enabled := False;
end;
procedure TAniIconEditorDialog.FormDestroy(Sender: TObject);
begin
  Icon.Free;
end;
procedure TAniIconEditorDialog.ClearClick (Sender: TObject);
begin
  Timer1.Enabled := False;
  Save.Enabled := False;
  Clear.Enabled := False;
  Icon.Clear;
  ImagePaintBox.Invalidate;
end;
procedure TAniIconEditorDialog.LoadClick (Sender: TObject);
begin
  if OpenDialog.Execute then begin
    Icon.LoadFromFile (OpenDialog.Filename);
    Reset;
  end;
end;
procedure TAniIconEditorDialog.Reset;
begin
  Icon.BackgroundColor := ImagePaintBox.Color;
  ImagePaintBox.Invalidate;
  Save.Enabled := not Icon.Empty;
  Clear.Enabled := not Icon.Empty;
  Timer1.Enabled := not Icon.Empty;
end;
procedure TAniIconEditorDialog.SaveClick(Sender: TObject);
begin
  if (Icon.Empty = False) and SaveDialog.Execute then
    Icon.SaveToFile (SaveDialog.Filename);
end;
procedure TAniIconEditorDialog.Timer1Timer(Sender: TObject);
var r: TRect;
begin
  if not Icon.Empty then begin
    Icon.Animate;
    r := Rect(0, 0, Icon.Width, Icon.Height);
    OffsetRect(r, (ImagePaintBox.Width - Icon.Width) div 2,
      (ImagePaintBox.Height - Icon.Height) div 2);
    Icon.Draw (ImagePaintBox.Canvas, r);
  end;
end;
function TAniIconPropertyEditor.GetValue: String;
var Icon: TAniIcon;
begin
  Icon := TAniIcon (GetOrdValue);
  if Icon.Empty then
    Result := '(None)'
  else
    Result := '(' + Icon.ClassName + ')';
end;
function TAniIconPropertyEditor.GetAttributes:
  TPropertyAttributes;
begin
  Result := [paDialog];
end;
procedure TAniIconPropertyEditor.SetValue(
  const Value: string);
begin
  if Value = '' then SetOrdValue (0);
end;
procedure TAniIconPropertyEditor.Edit;
begin
  with TAniIconEditorDialog.Create (Nil) do
    try
      Icon.Assign (TAniIcon (GetOrdValue));  Reset;
      if ShowModal = mrOK then
        TAniIcon (GetOrdValue).Assign(Icon);
    finally
      Free;
    end;
end;
initialization
  RegisterPropertyEditor (TypeInfo (TAniIcon), Nil, '',
    TAniIconPropertyEditor);
end.
```



➤ *Figure 2: And here's my replacement for the standard Picture Editor. This version, of course, is specific to TAniIcon property types and the currently selected .ANI file can be seen animating in the centre of the preview area.*

includes a `TTimer` component whose `Interval` property is set to 50 (again, as per last month's discussion on playback speeds) and this is used to trigger the `Timer1Timer` method which calls the `TAniIcon` object's `Animate` method to advance the animation, and then draws the current 'frame' at the centre of the paintbox control.

The private `TAniIcon` object is created in the form's `OnCreate` handler, and destroyed in the form's `OnDestroy` handler. The event handlers for the three buttons (`Load`, `Save` and `Clear`) is likewise very straightforward and pretty much self-explanatory, using the `Load FromFile`, `SaveToFile` and `Clear` methods of the `TAniIcon` object to do all the real work. If you want to, you can use the `Load` and `Save` buttons to create copies of existing .ANI files.

Returning to the `TAniIcon-PropertyEditor.Edit` method, you can see that it works by first creating the editor form, then assigning to the private `Icon` object according to the object instance data returned from `GetOrdValue`. The reverse process takes place when the form is dismissed, assuming that the `OK` button was pressed. The only other thing in Listing 1 which is worthy of note is the call to `Register PropertyEditor` in the unit's `initialization` clause. This causes the Delphi IDE to associate our new property editor with the `TAniIcon` class, causing it to be invoked whenever the Object Inspector is used to perform design-time modification of the class. Magic!

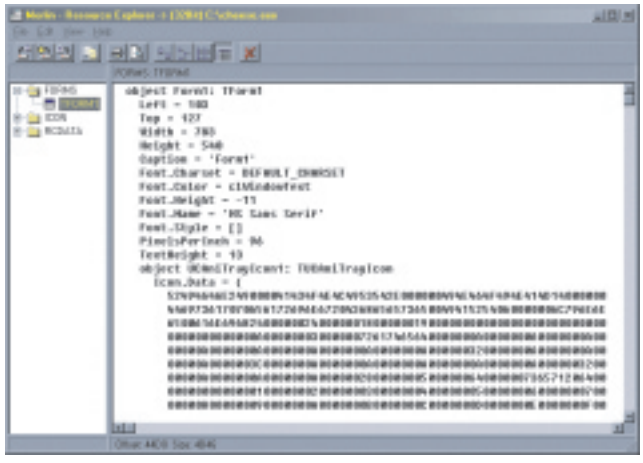
At this point, the rest is relatively plain-sailing. We've now got a class which encapsulates the animated icon file format, and we've got a cute property editor with which to massage properties of that class. With the code modules I've discussed so far installed into the Delphi IDE, you'll find that it's easy to create a new component, add a published property of type `TAniIcon` and then edit it to your heart's content. The critically important behaviour, of course, is that when such a component is streamed out to a .DFM file, the currently loaded icon data gets streamed into the .DFM file as well. This means that you can distribute applications that make use of animated icons without having to deploy a set of associated .ANI files and without having to muck around with resource data. To put it more succinctly, you can do things 'the Delphi way'. Figure 3 shows Merlin's Resource Explorer peeking at the `Icon` property of just such a component (of type `TUCAniTrayIcon`).

A Window, A Window, My Kingdom For A Window...

`TUCAniTrayIcon`? What's that then? Well, one of the rash promises I made last month was to develop a Delphi component that could display animated icons on the 'tray' area of the Explorer taskbar. You've probably seen several Delphi components that can put static icons onto the tray area, but I thought I'd try and improve on that.

The complete source code for `TUCAniTrayIcon` is given in Listing 2. As you can see, this is a non-visual Delphi component which derives from `TComponent`, because none of the drawing stuff takes place on the form itself, only in the Windows tray area. The control implements six properties, the most pertinent of which is `Icon`, enabling us to set up an icon animation using our shiny new property editor. The `Hint` property (not to be confused with the familiar `Hint` property used by visual Delphi components) is the hint string that appears when the mouse cursor is held over the tray icon for any length of time. The `Animate` property controls whether or not the icon is animating and the `Visible` property determines whether the icon is visible on the tray. Finally, the `OnLeftClick` and `OnRightClick` events provide a couple of simple event handlers through which the application can be notified when the left or right mouse button has been clicked on the tray icon. Please bear in mind that this isn't meant to be a presentation of the quintessential tray icon component: many tray icon components implement up and down handlers for each of the three mouse buttons, plus a mouse event handler plus... well, you get the idea. The emphasis here is simply on how to integrate our animated icon class into a tray icon component, not how to provide

➤ *Figure 3: This screenshot illustrates how the animated icon data contained within a TAniIcon property is streamed out as part of the .DFM form file. Look carefully, you'll see the RIFF' file signature which I mentioned last month.*



more functionality than you're ever likely to need.

That said, an interesting problem in the design of any tray icon component concerns the window handle that we're going to use to receive messages. As you may appreciate from reading the SDK documentation on Shell_NotifyIcon, it's necessary to provide the shell with the handle of a window that will receive messages when the mouse is clicked and moved over the tray icon. Since this is a non-visual, non-windowed component (and rightly so) it does not have an API-level window handle of its own. So what window handle can we use? If we elected to use the window handle of the parent form, then we'd have to sub-class the form in order to receive the messages posted to it. This would lead to all sorts of unpleasant code and would necessitate additional care in cleaning up when a component instance is removed, ensuring that there's only one instance of TUCAniTrayIcon on a form, and so forth. This is definitely not the way to go. Another solution would be to derive TUCAniTrayIcon from a windowed ancestor such as TWinControl, but again this is messy and leads to all sorts of complications.

A good tip here is to look at the way in which the TTimer control has been implemented. The

➤ *Listing 2*

```
unit UCAniTrayIcon;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, UCAniIcon;
const
  UCTrayCallBack = $89AB;
type
  TUCAniTrayIcon = class (TComponent)
  private
    fHint: String;
    fWindowHandle: hWnd;
    fVisible: Boolean;
    fAniIcon: TAniIcon;
    fOnLeftClick: TNotifyEvent;
    fOnRightClick: TNotifyEvent;
    fExclusionLock: Boolean;
    fAnimate: Boolean;
    procedure SetHint (const Value: String);
    procedure SetVisible (Value: Boolean);
    procedure SetAniIcon (Value: TAniIcon);
    procedure UpdateTray (Visible: Boolean);
    procedure SetAnimate (Value: Boolean);
    procedure WndProc (var Message: TMessage);
    procedure TrayMessage (var Message: TMessage);
  protected
    procedure Loaded; override;
  public
    constructor Create (AOwner: TComponent); override;
    destructor Destroy; override;
  published
    property Icon: TAniIcon read fAniIcon write SetAniIcon;
    property Hint: String read fHint write SetHint;
    property Animate: Boolean read fAnimate
      write SetAnimate default False;
    property Visible: Boolean read fVisible
      write SetVisible default False;
    property OnLeftClick: TNotifyEvent read fOnLeftClick
      write fOnLeftClick;
    property OnRightClick: TNotifyEvent read fOnRightClick
      write fOnRightClick;
  end;
procedure Register;
implementation
uses ShellAPI;
constructor TUCAniTrayIcon.Create (AOwner: TComponent);
begin
  Inherited Create (AOwner);
  fWindowHandle := AllocateHWnd (WndProc);
  fAniIcon := TAniIcon.Create;
end;
procedure TUCAniTrayIcon.Loaded;
begin
  Inherited Loaded;
  UpdateTray (fVisible);
end;
destructor TUCAniTrayIcon.Destroy;
begin
  SetAnimate (False);
  SetVisible (False);
  DeallocateHWnd (fWindowHandle);
  fAniIcon.Free;
  Inherited Destroy;
end;
procedure TUCAniTrayIcon.SetHint (const Value: String);
begin
  if fHint <> Value then begin
    fHint := Value;
    UpdateTray (fVisible);
  end;
end;
procedure TUCAniTrayIcon.TrayMessage(var Message: TMessage);
begin
  if not fExclusionLock then begin
    fExclusionLock := True;
    case Message.lParam of
      wm_LButtonDown : if Assigned(fOnLeftClick) then
        fOnLeftClick (Self);
      wm_RButtonDown : if Assigned(fOnRightClick) then
        fOnRightClick (Self);
    end;
    fExclusionLock := False;
  end;
end;
procedure TUCAniTrayIcon.WndProc (var Message: TMessage);
begin
  with Message do if Msg = UCTrayCallBack then
    try
      TrayMessage (Message);
    except
      Application.HandleException (Self);
    end
  else if (Msg = wm_Timer) and (fAniIcon.Empty = False)
    and fVisible then begin
    fAniIcon.Animate;
    UpdateTray (fVisible);
  end else
    Result :=
      DefWindowProc(fWindowHandle,Msg,wParam,lParam);
end;
procedure TUCAniTrayIcon.SetAniIcon (Value: TAniIcon);
begin
  fAniIcon.Assign (Value);
  UpdateTray (fVisible);
end;
procedure TUCAniTrayIcon.SetVisible (Value: Boolean);
begin
  if fVisible <> Value then
    UpdateTray (Value);
  fVisible := Value;
end;
procedure TUCAniTrayIcon.SetAnimate (Value: Boolean);
begin
  if (fAnimate <> Value) then begin
    fAnimate := Value;
    if fAnimate then
      SetTimer (fWindowHandle, 1, 50, Nil)
    else
      KillTimer (fWindowHandle, 1);
  end;
end;
procedure TUCAniTrayIcon.UpdateTray (Visible: Boolean);
var
  Msg: DWord;
  tid: TNotifyIconData;
begin
  if not (csDesigning in ComponentState) then begin
    tid.cbSize := sizeof (tid);
    tid.Wnd := fWindowHandle;
    tid.uID := 0;
    tid.uFlags := nif_Message + nif_Icon + nif_Tip;
    tid.uCallbackMessage := UCTrayCallBack;
    if fAniIcon.Empty then
      tid.hIcon := Application.Icon.Handle
    else
      tid.hIcon := fAniIcon.Icon;
    StrPCopy (tid.szTip, fHint);
    if Visible <> fVisible then begin
      if Visible then Msg := nim_Add
      else Msg := nim_Delete;
    end else
      Msg := nim_Modify;
    Shell_NotifyIcon (Msg, @tid);
  end;
end;
procedure Register;
begin
  RegisterComponents ('UnCommon', [TUCAniTrayIcon]);
end;
end.
```

humble timer control requires a window in order to receive WM_TIMER messages, but it has the same problem, which window do we send the messages to? The timer component gets around this by using the deeply cunning AllocateHWnd routine which is actually defined inside the Forms unit. This routine takes a single argument, the address of a method, and it creates a new, hidden, window, setting up things such that the window procedure of the window equates to the specified method. As you can see, I call AllocateHWnd in the constructor of TUCAniTrayIcon, and this gives me a window handle, fWindowHandle, to which messages can be sent from the shell. To reverse the process, be sure to call DeallocateHWnd in the component's destructor.

OK, we've got a window handle; what next? The core of the component is the UpdateTray method. This is called in two circumstances: firstly, when we want to make the tray icon either visible or invisible, and secondly whenever we want to change some attribute of the icon such as the hint string, or the actual icon image that's displayed. From this, you'll appreciate that the UpdateTray code will be called repeatedly in order to display successive frames of the animation in the tray area.

UpdateTray works by setting up a small data structure of type TNotifyIconData, passing it to the Shell via the call to Shell_NotifyIcon. Part of the passed information includes the handle of the window we created to receive shell notifications, and UCTrayCallBack, a custom notification message we've defined. I've written the UpdateTray code in such a way that it'll do nothing at design-time (the csDesigning bit set in ComponentState). In theory, you could add icons to the tray at design-time, and even have a tray icon animating while working in Delphi's form designer, but I considered this to be rather confusing because, when you execute the program from the IDE, you actually end up with two icons on the tray, one belonging to the running program and one belonging to the TUCAniTrayIcon instance sat on the form designer! Perhaps there's a simple solution to this problem, but it's just past Christmas, there's still far too much brandy butter in my bloodstream, and I can't think of one!

The WndProc method is particularly important, it's the address of this method that was passed to the AllocateHWnd routine I discussed earlier. Whenever a UCTrayCallBack message is received from the shell, the TrayMessage method is called to handle it. If you look at the TrayMessage code, you'll see that I've implemented a simple exclusion lock within this routine. Essentially, it's just a simple 'busy flag' which ensures that if the shell is currently calling one of the two assigned event handlers, any subsequent UCTrayCallBack messages will be politely ignored. Why is this important? Well, suppose you arranged for a configuration dialog box to appear when the user clicked the right hand mouse button on the tray icon. Without the exclusion lock, clicking the tray icon five times would bring up five copies of the same dialog, which probably isn't what you wanted to happen.

This leaves the question of how to 'drive' the icon animation. One solution would be to create a TTimer component within the constructor of TUCAniTrayIcon and use this to keep the animation running. However, as I've already explained, the TTimer component creates a hidden window of its own in order to receive WM_TIMER messages. Wouldn't it be much more efficient if we could simply make use of the existing window that we've already got? It turns out to be quite easy to do this through an API-level routine called SetTimer. Just pass it the handle of an existing window, a timer ID (it's possible to associate multiple timers with a single window) an interval count and off you go. Similarly, you use the KillTimer routine to destroy the existing timer when the Animate property is set to False.

Thus, you'll appreciate that I'm really using our hidden window to kill two birds with one stone: it's not only receiving tray notification messages from the Shell, but it's also receiving WM_TIMER messages which keep the animation going. As you'll see (back in the WndProc method), whenever a timer message is received, I check if an animated icon is assigned to fAniIcon, check if the icon is visible on the tray and, if so, execute last month's Animate method before calling UpdateTray to write a (possibly) new frame to the tray area.

There's really not much more to say about TUCAniTrayIcon, the remaining methods are relatively trivial and self-explanatory. One thing that I *wouldn't* advise you to do is repeat my mistake. Out of curiosity I added this line of code immediately after the call to fAniIcon.Animate:

```
Application.Icon.Handle :=
  fAniIcon.Icon;
```

At design-time, this will work for a while, animating the Delphi IDE's own icon, and causing even the icon displayed in the top-left corner of Delphi's form designer window to start animating! However, a GPF results as soon as you try to run the program, and I'm not too sure why. If you want to animate the application icon of your program, you'd be better advised to experiment with doing it inside the UpdateTray method, thus ensuring that it only happens at runtime. This is an avenue that I haven't explored.

Cursor Craziness!

Listing 3 contains the source code for TUCAnimatedIcon, the final piece in our animated icon jigsaw puzzle. As before, this is a Delphi control which makes use of TAniIcon. This time, however, it's a visual component which writes directly to its own canvas. You'd typically use this control to draw an animated icon on a form. Because it doesn't have to allocate a hidden window or communicate with the Windows shell, it's a very much simpler affair than TUCAniTrayIcon. This component isn't without its rough edges, and you'll have your own

ideas about what ancestor class you want to inherit from (fancy a TBitBtn with an animated graphic?) but it should serve as a starting point for your own explorations.

Finally, before leaving the subject of animated icons, let me mention one more bit of programming magic that can be performed with them.

If you examine the source for TAniIcon, you will see that I've added a new method, SetAnimatedCursor, which wasn't present in last month's code. The source code to this method is given in Listing 4. This method takes the raw data from a .ANI file, writes it to a temporary file, and then uses the API LoadImage routine to reload the data as an animated cursor, writing the animated cursor handle to the TScreen object's Cursors handle in the usual way. Perhaps you think this is a little counter-intuitive but remember that (as far as Windows Explorer is concerned) .ANI files are really 'Animated *Cursor* Files'. I've taken pains to refer to .ANI files as 'animated icons' throughout these two articles simply to highlight the fact that my code has picked apart the .ANI format in order to get at the individual icon frames, but strictly speaking, Microsoft designed the .ANI file format in order to implement animated cursors. Consequently, no discussion of animated icons and cursors would be complete without me showing you how to do this under Delphi.

```
procedure TAniIcon.SetAnimatedCursor (Index: Integer);
var TempFileName: String;
begin
  if not Empty then begin
    TempFileName := FormatDateTime ('__$$hhnnss$$__', Now);
    SaveToFile(TempFileName);
    try
      Screen.Cursors[Index] := LoadImage (0, PChar(TempFileName),
        Image_Cursor, 0, 0, lr_LoadFromFile);
    finally
      DeleteFile (TempFileName);
    end;
  end;
end;
```

➤ *Above: Listing 4*

➤ *Below: Listing 5*

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Screen.Cursors [1000] := LoadImage(0, 'c:\windows\cursors\m_busy.ani',
    Image_Cursor, 0, 0, lr_LoadFromFile);
  Cursor := 1000;
end;
```

➤ *Table 1*

UCAniIcon	TGraphic derived class to implement core functionality
UCAniIconEdit	Property editor for UCAniIcon
UCAniTrayIcon	Non-visible component for tray icon animations
UCAnimatedIcon	Visible component to add icon animations to forms

➤ *Listing 3*

```
unit UCAnimatedIcon;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, UCAniIcon;
type
  TUCAnimatedIcon = class(TCustomControl)
  private
    fIcon: TAniIcon;
    fAnimate: Boolean;
    procedure SetIcon (Value: TAniIcon);
    procedure SetAnimate (Value: Boolean);
    procedure TimerTick(var Msg:TMessage); message wm_Timer;
  protected
    procedure Paint; override;
  public
    constructor Create (AOwner: TComponent); override;
    destructor Destroy; override;
  published
    property OnEnter;
    property OnExit;
    property OnKeyDown;
    property OnKeyPress;
    property OnKeyUp;
    property TabOrder;
    property TabStop;
    property Visible;
    property Color;
    property ParentColor;
    property Hint;
    property Icon: TAniIcon read fIcon write SetIcon;
    property Animate: Boolean read fAnimate
      write SetAnimate default False;
  end;
procedure Register;
implementation
constructor TUCAnimatedIcon.Create (AOwner: TComponent);
begin
  Inherited Create (AOwner);
  fIcon := TAniIcon.Create;
  Width := GetSystemMetrics (sm_cxIcon);
  Height := GetSystemMetrics (sm_cyIcon);
end;
destructor TUCAnimatedIcon.Destroy;
begin
  fIcon.Free;
  Inherited Destroy;
end;
procedure TUCAnimatedIcon.SetIcon (Value: TAniIcon);
begin
  fIcon.Assign (Value);
  fIcon.BackgroundColor := Color;
end;
procedure TUCAnimatedIcon.SetAnimate (Value: Boolean);
begin
  fAnimate := Value;
  if fAnimate then
    SetTimer (Handle, 1, 50, Nil)
  else
    KillTimer (Handle, 1);
end;
procedure TUCAnimatedIcon.Paint;
var R: TRect;
begin
  if (not fAnimate) and (not fIcon.Empty) then begin
    R := Rect (0, 0, fIcon.Width, fIcon.Height);
    OffsetRect(R, (Width-fIcon.Width) div 2,
      (Height-fIcon.Height) div 2);
    fIcon.Draw (Canvas, R);
  end;
end;
procedure TUCAnimatedIcon.TimerTick (var Msg: TMessage);
var R: TRect;
begin
  if not fIcon.Empty then begin
    fIcon.Animate;
    R := Rect (0, 0, fIcon.Width, fIcon.Height);
    OffsetRect (R, (Width-fIcon.Width) div 2,
      (Height-fIcon.Height) div 2);
    fIcon.Draw (Canvas, R);
  end;
end;
procedure Register;
begin
  RegisterComponents('UnCommon', [TUCAnimatedIcon]);
end;
end.
```

The above method is really just a convenience for those who are already using my `TAniIcon` class within their program, either to perform some visual animation on a form, or perhaps to animate a tray icon. However, if you simply wish to have an animated cursor and you're not interested in any of the `TAniIcon` functionality, then you can get away with something as simple as Listing 5.

This code uses the 'mouse meets cheese' .ANI file to create an animated cursor that's used to replace the regular form's cursor. More typically, you'd set up specialised animated cursors for one or more components on the form, rather than the form itself.

Included on this month's disk is a Delphi 3 package called ANIICON.DPL. This package contains the four files which are shown in Table 1. The finished .DPL package is around 36Kb, although if you were building a commercial application with this code, you'd be better advised to link everything into your EXE file.

Well, that's it, and I hope you've enjoyed this foray into guts of the .ANI file format!

Dave Jewell is a freelance consultant, programmer and technical journalist specialising in system-level Windows and DOS work. He is the Technical Editor of *Developers Review* which is also published by iTec. You can contact Dave as Dave@HexManiac.com