Tooltips Under Your Control

by Brian Long

This article is about tooltips, or hint windows. But don't worry, this is not a 'try the Hint and ShowHint properties' article. Nor is it a 'you might find the Application object's OnHint and OnShowHint events interesting' article. And it certainly is not one of those 'here is how to have funny shaped tooltips' articles (but if you wanted to read that last article, you could turn to Issue 16, page 47, and read *Hints With Attitude*).

No, none of the above are to be addressed here. Instead, to get an idea of what we are to be looking at, let's consider the Delphi 4 environment and how it uses some tooltips (note that my code works in all versions).

Delphi Editor

If you type some code in the editor, and then pause your mouse over one of the Pascal symbols, the *Tooltip Symbol Insight* will kick in, assuming it has not been disabled. Once the minor amount of background compilation has finished, a tooltip appears roughly where the mouse is, in much the normal way. The tooltip contains information on where the symbol is defined. So this is a fairly normal tooltip (of type THintWindow), whose text content is based on what the mouse is over.

Now let's think of another IDE tooltip. When you type an open parenthesis after a subroutine or method name, or press Shift+Ctrl+Space when the cursor is in a parameter list, the *Code Parameters* hint window appears. This functions a little differently to normal ones. It remains visible indefinitely (ie, it doesn't disappear after a short interval), or at least until you type the closing parenthesis, move the cursor, or switch away to another window. Also, scrolling the editor window with the scrollbar causes the tooltip to move, so it stays in the same position relative to the code. Finally, if you shrink the editor when the tooltip is displayed, it will automatically disappear when the editor becomes too small for it.

The *Code Parameters* window is of type TTokenWindow, presumably having extra code to draw the current parameter in bold text and also to deal with overloaded subroutine parameters. This hint window is controlled by custom code over and above the normal VCL tooltip code.

Object Inspector

Now let's try another Delphi 4 tooltip. Take a look at the Object Inspector. If you move the mouse over any property or any property value that you cannot see in its entirety, another tooltip appears over that Object Inspector cell displaying the whole value. This tooltip is a window of type TPropertyHintWindow. Again, it exhibits non-standard behaviour in that it always sits exactly over one of the cells on the Object Inspector's grid, rather than being at a specific offset from the mouse cursor. It also does not disappear until you move the mouse off that cell, or take focus away from the Object Inspector.

The *Code Parameters* window and the Object Inspector property hint window are prime examples of non-standard tooltips that we will try to emulate. A tooltip that goes where we want it to, and stays there until we want it to go, and a tooltip that reveals obscured text in a control, say a string grid cell, or a data-aware grid cell. So with the introduction out of the way, we will start with the first issue.

Persistent Hint Windows

Let's take an example requirement where we have some edit controls on a form, amongst other things. The application, for reasons beyond our control, needs to have a hint window displayed when any

➤ *Listing 1*

```
{ When control gains focus, display the hint }
procedure TForm1.DBEditEnter(Sender: TObject);
var HintRect: TRect;
begin
  { Create instance of currently registered
    hint window class }
  if not Assigned(HintWnd) then
    HintWnd := HintWindowClass.Create(Self);
  { Use current VCL hint colour }
  HintWnd.Color := Application.HintColor;
  Control := TControl(Sender);
  { How big should it be? }
  HintRect := CalcHintRect(Screen.Width, Control.Hint);
  with CalcHintTopLeft(Control) do
    OffsetRect(HintRect, X, Y);
  { Show it }
  HintWnd.ActivateHint(HintRect, Control.Hint)
end;
{ When control loses focus, remove the hint }
procedure TForm1.DBEditExit(Sender: TObject);
begin
  Control := nil;
  { Keep object, but destroy underlying window }
  HintWnd.ReleaseHandle;
end;
function TForm1.CalcHintRect(MaxWidth: Integer;
  const AHint: string; HintWnd: THintWindow): TRect;
{$ifdef DelphiLessThan3}
var
  Buf: array[0..511] of Char;
begin
  Result := Rect(0, 0, MaxWidth, 0);
  { Ask Windows to do the hard calculation work }
  DrawText(HintWnd.Canvas.Handle, StrPCopy(Buf, AHint),
    -1, Result, DT_CALCRECT or DT_LEFT or DT_WORDBREAK or
    DT_NOPREFIX);
  { Add some breathing room }
  Inc(Result.Right, 6);
  Inc(Result.Bottom, 2);
{$else}
begin
  { Delphi 3+ makes this method available }
  Result := HintWnd.CalcHintRect(Screen.Width, AHint, nil)
{$endif}
end;
function TForm1.CalcHintTopLeft(Control: TControl): TPoint;
const
  HintOffset = 4;
begin
  { Where should it go? }
  Result := Point(Control.Left + HintOffset,
    Control.Top + Control.Height);
  Result := ClientToScreen(Result);
end;
```

of the edit controls gains focus. The hint window must stay visible for as long as the edit control has focus. This means that if we switch to another application, or another control in the same application, the hint window must disappear. Also, if the form is moved, the hint window must notice this and move along with it.

The sample project HINTS.DPR on the disk attempts to meet this specification. It has four data aware edit controls connected through a datasource to some table. There is also a navigator and an exit button.

The idea will be to ignore the normal VCL-controlled hint window and not use it at all. Instead, we will manufacture and control one of our own. The normal VCL hint window class is `THintWindow`, so the form has a data field of this type called `HintWindow`. Also, to keep track of which control we are displaying a hint for, we have another field called `Control` of type `TControl`.

The four edits share their `OnEnter` event handler, and also their `OnExit` event handler (see Listing 1). The first time one of them gains focus, the `OnEnter` event handler creates an object for `HintWindow` to refer to. Rather than hard-coding the VCL's `THintWindow` type in the `constructor` call, it uses the `FORMS` unit's `HintWindowClass` class reference variable. `HintWindowClass` starts its life initialised to `THintWindow`, but this can be replaced by anything inherited from `THintWindow`. This means if anyone has installed a more interesting hint window class, this code will unwittingly benefit from it.

Apart from this, the rest of the code in this event handler executes the same for every invocation. The hint colour is set to match the VCL hint window's colour, and the form's `Control` field is set to point at the relevant edit control. The next step is to work out the size of the hint window, based upon the text to display and font to be used.

This calculation is delegated to the `CalcHintRect` method, which in Delphi 3 and later simply involves a call to a method of the hint window. Delphi 1 and 2 don't support this method, so the code must be written out in full.

When we have an appropriately proportioned `TRect`, we position it on screen at the bottom of, and just to the right of, the control. When the control loses focus, the `OnExit` event handler resets the `Control` field to `nil` and destroys the hint object's underlying window handle, making the hint disappear.

Final Touches

So that covers the basic requirements. We also need to deal with the other issues, moving the hint in tandem with the form, and making sure the hint disappears when the application is switched away from, reappearing when it is switched back to.

First of all, the form movement problem. This is quite straightforward, in that it involves a simple message handler for any `wm_Move` messages that the form receives. It recalculates the position of the hint window and moves it to the new location (Listing 2).

To deal with the other remaining issue, an event handler is shared between both the `Application` object's `OnActivate` and `OnDeactivate` events (Listing 3). When the application is activated or deactivated, assuming we are in a position to be potentially displaying a hint, some code executes to see if our application has focus. If it does, then we make sure the hint is visible, otherwise we hide it. `SetWindowPos` is used for this, as opposed to `ShowWindow`, so that we can use the `SWP_NOACTIVATE` flag to ensure the hint window does not receive focus itself (the edit control must retain focus).

To see if our application has focus, a routine called `ForegroundTask` is called. Delphi 3 (and later) has this routine available from the `Forms` unit, but it was not surfaced in Delphi 1 or 2. When compiling with those versions, a simplistically equivalent routine that hopefully will do just as well is supplied.

```
TForm1 = class(TForm)
...
private
...
  procedure WMMove(var Msg: TWMMove); message wm_Move;
end;
...
procedure TForm1.WMMove(var Msg: TWMMove);
begin
  inherited;
  { If we have a control's tooltip showing }
  if Assigned(Control) then
    with CalcHintTopLeft(Control) do
      { We'll move it }
      MoveWindow(HintWnd.Handle, X, Y, HintWnd.Width, HintWnd.Height, True);
end;
```

➤ *Above: Listing 2* ➤ *Below: Listing 3*

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  { These two ensure manufactured hints disappear and reappear as appropriate }
  Application.OnActivate := ActivateOnOff;
  Application.OnDeActivate := ActivateOnOff;
end;
{ Move hint, and hide/show it as specified }
procedure TForm1.MoveControl(Control: TWinControl; ShowControl: Boolean);
const
  Visibility: array[Boolean] of Cardinal = (SWP_HIDEWINDOW, SWP_SHOWWINDOW);
begin
  with HintWnd do
    SetWindowPos(Handle, HWND_TOPMOST, Left, Top, Width, Height,
      Visibility[ShowControl] or SWP_NOACTIVATE)
end;
procedure TForm1.ActivateOnOff(Sender: TObject);
  {$ifdef DelphiLessThan3}
  function ForegroundTask: Boolean;
  begin
    { Does the active window map onto some object in this app? }
    Result := FindControl(GetActiveWindow) <> nil
  end;
  {$endif}
begin
  if Assigned(Control) then
    { If we lost focus, hide the tooltip. If we gain focus, show it }
    MoveControl(HintWnd, ForegroundTask)
end;
```

Having dealt with these issues, the program almost works fine. One last remaining issue is that if the form is resized smaller, so that the edit with the hint is no longer visible, the hint remains visible on the desktop. This is no good. An `OnResize` event handler is required, and Listing 4 has one that does the job. If a control has a hint displayed, the `OnResize` event handler works out whether the control has disappeared (by checking whether the top left of the control is within the form's client area). If it *has* disappeared, the hint is hidden, otherwise it is displayed.

So there we have the first application, with its own customised and reasonably sensible hints, as shown in Figure 1.

Text Completion Tooltips

Now on to the next phase. When a string grid is used to display information, sometimes the text in the cells can be partially obscured. To make the grid more helpful, as the mouse is moved over a cell with partially obscured text it will pop up a hint window over the cell containing the full cell value. This will not be done if the user is currently editing the cell.

In short, we want an `OnMouseMove` event handler to execute, containing all the relevant logic. When the `OnMouseMove` event triggers, we need code to translate from the X and Y coordinates passed as parameters into cell coordinates, so we can then extract that cell's text. A `TStringGrid` has a method called `MouseToCell` that fits the bill. The logic must then establish whether the text in the cell, when drawn on the grid's canvas, is wider than the cell it is drawn in. If this is the case, and the cell does not have its in-place editor sitting in it, and the application is the foreground application, a hint window is required.

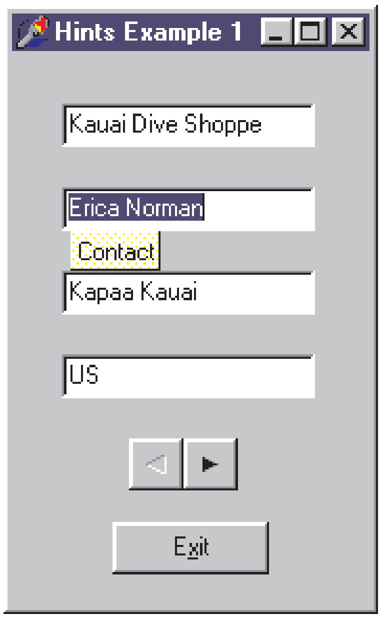
```
procedure TForm1.FormResize(Sender: TObject);
begin
  { If we have a control's tooltip showing }
  if Assigned(Control) then
    { Hide it if the control is no longer visible, else show it }
    MoveControl(HintWnd, PtInRect(Rect(0, 0, ClientWidth, ClientHeight),
      Point(Control.Left, Control.Top)))
end;
```

➤ *Listing 4*

As before, a `THintWindow` (or some user-installed derivative) object is created if none exists, and then initialised with sensible attributes. The grid's hint is set to contain the text of the cell in question and that leaves the messy business of working out where it should go on the screen.

`CalcHintRect` is the same as before and calculates the rectangle needed to contain the tooltip text. The tooltip needs to be drawn pretty much at the top left of the cell in question, so the grid's `CellRect` method is employed. Assuming the tooltip is not already on screen in the same place, it is activated at the relevant location. Listing 5 shows all the gory details just described. You can see the whole thing in HINTS2.DPR.

Testing this code shows one downfall. If you move the mouse onto a cell at the edge of the grid, to produce the hint window, and then sharply move the mouse off the grid, the hint window will remain. So a timer is employed to periodically validate the hint situation, and get rid of it if the user has moved the mouse away (Listing 6).



Kauai Dive Shoppe
Erica Norman
Kapaa Kauai
US

➤ *Figure 1*

New TStringGrid Component

I like this new grid functionality so much that it seems a shame to have to set it up manually each time I use it. So to help myself out I have turned the code used so far into an approximately equivalent component, `THintStringGrid` (Listing 7 has some code from this).

➤ *Listing 5*

```
procedure TForm1.SGMouseMove(Sender: TObject;
  Shift: TShiftState; X, Y: Integer);
const
  TextOffset = 2;
var
  Col, Row: Longint;
  R, OldR: TRect;
  Pt: TPoint;
begin
  SG.MouseToCell(X, Y, Col, Row); { Check cell under mouse }
  { If it is a cell, and text is bigger than screen space
    and in-place editor not present }
  Canvas.Font := Font;
  if (Col <> -1) and (Row <> -1) and
    (SG.Canvas.TextWidth(SG.Cells[Col, Row]) + TextOffset >
    SG.ColWidths[Col]) and not SG.EditorMode and
    ForegroundTask then
  begin
    { Make sure hint window exists }
    if not Assigned(GridHintWnd) then begin
      GridHintWnd := HintWindowClass.Create(Self);
      GridHintWnd.Color := Application.HintColor;
    end;
    { Set hint text }
    SG.Hint := SG.Cells[Col, Row];
    { Calculate rect size }
    R := CalcHintRect(Screen.Width, SG.Hint, GridHintWnd);
    { Find target location }
    Pt := SG.ClientToScreen(SG.CellRect(Col, Row).TopLeft);
    { Tweak position so it is the same as the grid cell
      (hopefully) }
   {$ifdef DelphiLessThan3}
    Inc(Pt.Y);
   {$else}
    Dec(Pt.X);
    Dec(Pt.Y);
   {$endif}
    OffsetRect(R, Pt.X, Pt.Y);
    { Only draw it if it has moved - compare top-left (could
      compare whole rect but hint sometimes grows itself) }
    GetWindowRect(GridHintWnd.Handle, OldR);
    if not IsWindowVisible(GridHintWnd.Handle) or
      not ((R.Left = OldR.Left) and (R.Top = OldR.Top)) then
      GridHintWnd.ActivateHint(R, SG.Hint)
  end else if Assigned(GridHintWnd) then
    GridHintWnd.ReleaseHandle
end;
```

The OnMouseMove event handler code has been transferred to a method called DoHint which is called from a wm_MouseMove message handler. One advantage of writing a self-contained grid-with-hints component is that the component will be sent a cm_MouseEnter message when the mouse moves into the control, and a cm_MouseLeave message when it is moved out. So when the mouse leaves the grid, the cm_MouseLeave message handler can remove the hint, precluding the need for a timer.

That said, these messages are not without problems. TSpeedButton objects use them as well when their Flat property is True. When the mouse enters the speed button draws itself as a 3D button, and when it leaves it redraws itself 2D. Unfortunately, if the mouse is snatched from the speed button, the cm_MouseLeave message will not be generated and the button stays 3D. You can test this on the speed bars in Delphi 3 or later.

New TDBGrid Component

Having proven the principle with a TStringGrid, let's now try to do the same thing with a TDBGrid. The component will contain exactly the same extra code as THintStringGrid, but the implementation of DoHint will be a little different in this case.

Basically we need to replace the code that gets the text from the cell. In the case of a TDBGrid, we need to read the value of the field displayed in a cell, taking any display formatting attributes into account. Listing 8 (taken from DBHNTGRD.PAS) shows the result. You can see that a TDBGrid does not have a MouseToCell method, so MouseToCoord is used instead, along with some further manipulation code. Care must be taken when obtaining the field, as the Options property of the grid allows the title row and indicator column to be optionally removed.

New TListBox Component

Whilst we are on a roll, we may as well continue. A TListBox gets a vertical scrollbar when required, but not a horizontal one. You can programmatically give a horizontal scrollbar to a listbox, but maybe it would be nicer to have the listbox proactively inform the user of the full text of an item with a hint window, as with the grids.

Again, the code in the new component is practically the same as before, with different code to extract the text from the item under the mouse cursor. HINTLIST.PAS contains a working component, with the different code shown in Listing 9.

The HINTS3.DPR project uses one of each of these new components: see Figure 2. The screenshot has been doctored so you can see all three components displaying their tooltips simultaneously. In practice, you only see a tooltip when the mouse is over part of the component which contains partially obscured text.

Final Notes

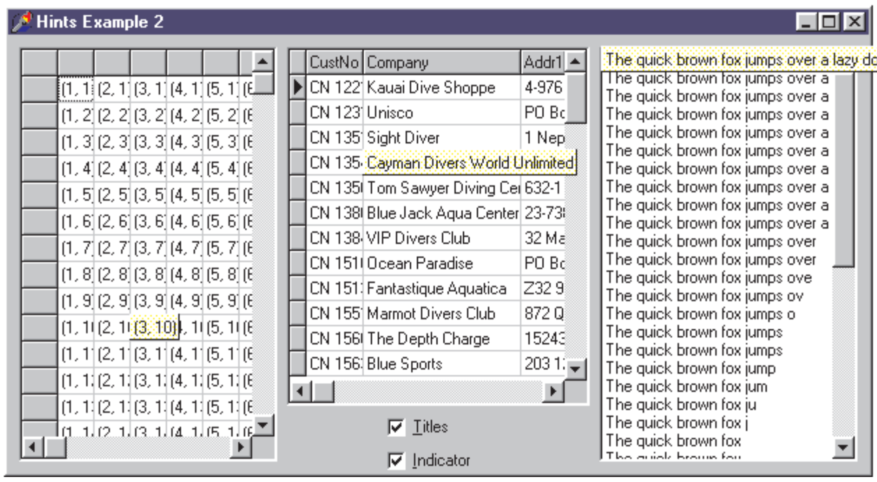
When compiling this custom hint functionality into Delphi 1 and 2, a problem arises. Generally speaking, hints are displayed just off the client area of the control they are related to. In the applications presented here, this is not necessarily the case: the hints are displayed within the component, immediately under the mouse.

The implementation of THintWindow in Delphi 1 and 2 is slightly lacking in that if you click on the

➤ *Listing 6*

```
procedure TForm1.Timer1Timer(
  Sender: TObject);
var Pt: TPoint;
begin
  GetCursorPos(Pt);
  Pt := ScreenToClient(Pt);
  if not PtInRect(
    SG.BoundsRect, Pt) and
    Assigned(GridHintWnd) then
    GridHintWnd.ReleaseHandle
end;
```

➤ *Figure 2*



Hints Example 2

(3, 10)

CustNo	Company	Addr1
CN 122	Kauai Dive Shoppe	4-976
CN 123	Unisco	PO Bc
CN 135	Sight Diver	1 Nep
CN 135	Cayman Divers World Unlimited	
CN 135	Tom Sawyer Diving Cei	632-1
CN 138	Blue Jack Aqua Center	23-73
CN 138	VIP Divers Club	32 Ma
CN 151	Ocean Paradise	PO Bc
CN 151	Fantastique Aquatica	Z32 9
CN 155	Marmot Divers Club	872 Q
CN 156	The Depth Charge	15243
CN 156	Blue Sports	203 1

[✓] Titles
[✓] Indicator

The quick brown fox jumps over a lazy dog

➤ *Listing 7*

```
THintStringGrid = class(TStringGrid)
private
  FHintWnd: THintWindow;
protected
  function CalcHintRect(MaxWidth: Integer;
    const AHint: string; HintWnd: THintWindow): TRect;
  procedure DoHint(X, Y: Integer);
public
  procedure CMMouseEnter(var Msg: TMessage);
    message cm_MouseEnter;
  procedure CMMouseLeave(var Msg: TMessage);
    message cm_MouseLeave;
  procedure WMMouseMove(var Msg: TWMMouseMove);
   message wm_MouseMove;
end;
procedure THintStringGrid.CMMouseEnter(var Msg: TMessage);
var Pt: TPoint;
begin
  GetCursorPos(Pt);
  Pt := ScreenToClient(Pt);
  DoHint(Pt.X, Pt.Y)
end;
procedure THintStringGrid.CMMouseLeave(var Msg: TMessage);
begin
  inherited;
  if Assigned(FHintWnd) then
    FHintWnd.ReleaseHandle; { quicker than Destroy }
end;
procedure THintStringGrid.WMMouseMove(
  var Msg: TWMMouseMove);
begin
  inherited;
  DoHint(Msg.XPos, Msg.YPos)
end;
```

```
procedure THintDBGrid.DoHint(X, Y: Integer);
const TextOffset = 2;
var
  Col, Row, LogCol, LogRow: Longint;
  R, OldR: TRect;
  Pt: TPoint;
  GPt: TGridCoord;
  OldActive: Integer;
  Text: String;
begin
  GPt := MouseCoord(X, Y);  { Check cell under mouse }
  Col := GPt.X;
  Row := GPt.Y;
  LogCol := Col;
  LogRow := Row;
  { Title row needs to be taken account of }
  if dgTitles in Options then Dec(LogRow);
  { Indicator column needs to be taken account of }
  if dgIndicator in Options then Dec(LogCol);
  Text := '';
  if (LogCol >= 0) and (LogRow >= 0) then begin
    OldActive := DataLink.ActiveRecord;
    try
      Datalink.ActiveRecord := LogRow;
     {$ifdef Win32}
      Text := Columns[LogCol].Field.DisplayText
     {$else}
      Text := Fields[LogCol].DisplayText
     {$endif}
    finally
      Datalink.ActiveRecord := OldActive
    end
  end;
  { If it is a cell, in-place editor not present, and text
    bigger than screen space, and not at design-time }
  Canvas.Font := Font;
  if (Text <> '') and not EditorMode and ForegroundTask
    and (Canvas.TextWidth(Text) + TextOffset >
    ColWidths[Col]) and not
    (csDesigning in ComponentState) then
    { code much the same as before }
end;
```

➤ *Above: Listing 8*

➤ *Below: Listing 9*

```
procedure THintListBox.DoHint(X, Y: Integer);
const
  TextOffset = 2;
var
  Item: Longint;
  R, TmpR, OldR: TRect;
  Pt: TPoint;
begin
  { Check item under mouse }
  Item := ItemAtPos(Point(X, Y), True);
  { If it is an item cell, and text is bigger than screen
    space, and not at design-time }
  Canvas.Font := Font;
  if (Item >= 0) and (Canvas.TextWidth(Items[Item]) +
    TextOffset > ClientWidth) and ForegroundTask
    and not (csDesigning in ComponentState) then
    { code much the same as before }
end;
```

hint window it beeps. This is quite likely to happen, as the user will wish to select grid cells and listbox items, so we need to fix this problem. Code exists in the appropriate projects to install a customised hint class (TCustomHint) that avoids the problem by being enabled and claiming to be transparent. Listing 10 shows the code. Again, to read up on the subject of customising normal VCL hint behaviour, check the references at the start of this article.

So that's the end, and hopefully this article may have prompted some ideas about how to make your applications a little more helpful. Remember, if you want tooltips to do something different to the norm, just ignore the normal VCL support and control them manually with your own code.

Brian Long is an independent consultant and trainer. You can reach him at brian@blong.com

```
{$ifdef DelphiLessThan3}
{ Hint window in Delphi 1 & 2 beeps if you click it.
  These modifications fix that }
TCustomHint = class(THintWindow)
private
  procedure WMNCHitTest(var Msg: TWMNCHitTest);
    message wm_NCHitTest;
protected
  procedure CreateParams(var Params: TCreateParams);
    override;
end;
procedure TCustomHint.CreateParams(
  var Params: TCreateParams);
begin
  inherited CreateParams(Params);
  Params.Style := Params.Style and not ws_Disabled;
end;
procedure TCustomHint.WMNCHitTest(var Msg: TWMNCHitTest);
begin
  Msg.Result := HTTRANSPARENT;
end;
initialization
  Application.ShowHint := not Application.ShowHint;
  HintWindowClass := TCustomHint;
  Application.ShowHint := not Application.ShowHint;
{$endif}
end.
```

➤ *Listing 10*