Using the lazarus grids

Michaël Van Canneyt

August 4, 2008

Abstract

The Grid control in Lazarus is to a very large degree compatible to its nephew in Delphi, so the experienced Delphi programmer will not have too much trouble working with it. Nevertheless, there are some nice additions and some marked differences with the Delphi grid. This article explores these, and shows how the grid can be extended.

1 Introduction

When an application needs to present an overview of data, the programmer can choose between a `TListView`, a `TTreeView`, or a TCustomGrid, and its data-aware counterpart `TDBGrid`. In this article, the implementation of the grid in Lazarus is examined: The reader is assumed to be roughly familiar with the Delphi `TStringGrid` control implementation, so only the differences will be explained.

Just as in Delphi, there are 3 flavours of grids in Lazarus:

TDrawGrid Here all cells of the grid must be painted by the programmer. The grid component has no storage for data associated with the cells.

TStringGrid This grid has the ability to associate a string to each cell in the grid, and it knows how to present the strings.

TDBGrid is a special data-aware grid. It knows how to display the data in a `TDataset` descendent.

All three descend from `TCustomGrid`, and most of the functionality is implemented in this common ancestor.

To demonstrate the capabilities, a `TStringGrid` will be used, but most (if not all) capabilities are present in the other grids as well.

2 Extra possibilities

The stringgrid implementation of Lazarus has some extra possibilities which are exposed in extra properties:

AlternateColor If this color property is set to a color different from the color property, then the grid will be shown with alternating row colors: one row is painted in the color set in the 'Color' property, and the next row in the color set in the `AlternateColor` property.

AutoAdvance When the contents of a cell is edited, and the user hits the 'Enter' or 'Tab' keys, this property determines which cell is selected next. By default (`aaRight`) this is the next cell in the current row, till the end of the row is reached. Setting this to `aaNone` disables the automatic advance.

AutoEdit Clicking on a cell will put it in edit mode. The main difference with the `goAlwaysShowEditor` option is that with the latter, the cells are always in edit mode. `AutoEdit` puts a cell in edit mode if it is clicked. In case neither `AutoEdit` nor `goAlwaysShowEditor` are specified, 2 clicks are needed to put a cell in edit mode: one to select the cell, and one more to put it in edit mode.

AutoFillColumns If set to `True` then the columns are automatically resized so they always fill the complete width of the grid. All columns are resized evenly, but this can be arranged using some properties - see below for more about this.

ExtendedSelect Allows to select multiple cells even if the grid is editable. In Delphi, range selections are not possible for editable grids.

Flat if set to `True`, the fixed cells will be drawn using a flat look. (This property may dissapear to be replaced with an extra TitleStyle option).

HeaderHotZones Determines which fixed cells respond to hottracking (see below).

HeaderPushZones Determines which fixed cells respond to pushing (see below).

TitleStyle Determines how the fixed cells are drawn. Can be one of `tsLazarus`, `tsNative` or `tsDefault`. For `tsLazarus`, a flat look is used, and the `Flat` property is ignored.

UseXORFeatures If this property is set to `True`, then the focus rectangle will be drawn using a XOR pen, so it will be visible no matter what the background color is.

The `Options` property is a set property, and has the elements that Delphi has, but in addition to those, it has some additional properties:

goColSpanning This option allows text to span multiple cells, as in Excel: when a cell text must be painted, then the text rectangle is extended to span as much columns as needed to show the cell.

goDblClickAutoSize This feature enables a feature found e.g. in Excel: double-clicking the border between two column header cells will auto-size the column to the left: the column will be widened or made smaller so it's width matches the widest text entry in the column. The `goColsizing` option must be specified for this option to take effect.

goFixedRowNumbering If enabled, the row number is shown in the first fixed column.

goHeaderHotTracking If enabled, the header cell over which the mouse pointer is currently positioned will be shown in a different color. The cells for which the hottracking is enabled can be specified in the `HeaderHotZones` property.

goHeaderPushedLook If enabled, clicking a header cell will look as a pressed button when clicked. The fixed cells for which this is enabled can be specified in the `HeaderPushZones` property.

goRelaxedRowSelect If this property is specified, then the currently selected cell is drawn inverted to the rest of the selected row.

goScrollKeepVisible If this property is specified, the selected cell remains at the same position relative to the top of the grid when scrolling with the scrollbar. By default the selected cell will scroll along with the other cells.

goSmoothScroll If this option is not set, then the grid is scrolled cell-by-cell, i.e. the top row always shows a complete cell. If the option is set, then the grid is scrolled on a pixel-by-pixel basis. Obviously, this option only has effect if `goThumbTracking` is specified.

All of these properties can be examined in the griddemo program provided on the CD-Rom acompagnying this issue. It features a stringgrid, and a property inspector grid. There is a button to fill the cells of the grid with a text: the coordinates of the cell.

The `goDblClickAutoSize` option only works for columns which are not fixed columns. Luckily, the grid has a method which allows to automatically resize a column. The form contains a button which, when clicked, resizes the fixed columns:

```
procedure TMainForm.BAutoSizeFixedClick(Sender: TObject);

Var
  I : integer;

begin
  For I:=0 to SGDemo.FixedCols-1 do
    SGDemo.AutoSizeColumn(i);
end;
```

The `AutoSizeColumns` method of the grid auto-sizes all columns in the grid.

On top of the extra properties, the stringgrid can be edited in design time: it is possible to set the strings in the grid using the component editor (available from the designer popup menu when right-clicking the grid component).

3 Columns

Perhaps the biggest difference with Delphi's stringgrid is that the Lazarus implementation has also a `Columns` property: a collection of `TGridColumn` items which describe the properties of a column in a grid, similar to the columns property in a `TDBGrid`. The property is mutually exclusive with the `ColCount` property: as soon as columns are used, the `ColCount` property can no longer be set. The `Columns` property describes only the non-fixed columns. That is, if there are 2 fixed columns (FixedColCount=2) and 3 Columns, then the grid will display 5 columns.

The `Columns.Enabled` property determines whether columns are in use. To demonstrate this, the griddemo program has been enhanced with a button 'Columns', which, when clicked, transforms the grid to a grid using columns:

```
procedure TMainForm.BColumnsClick(Sender: TObject);

Var
  I,ACount : Integer;

begin
  ACount:=SGDemo.ColCount-SGDemo.FixedColCount;
  SGDemo.Columns.Clear;
```

```
  For I:=1 to ACount do
    With TGridColumn(SGDemo.Columns.Add) do
      Title.Caption:='Column '+IntToStr(i);
end;
```

The properties of a column can be examined and set by double-clicking a column:

```
procedure TMainForm.SGDemoDblClick(Sender: TObject);

Var
  C : TGridColumn;
  F : TColumnPropertiesForm;

begin
  If (SGDemo.Col>=SGDemo.FixedCols) and
     SGDemo.Columns.Enabled then
    begin
    If (FColProps=Nil) then
      begin
      F:=TColumnPropertiesForm.Create(Self);
      F.OnDestroy:=@DoFree;
      FColProps:=F
      end
    else
      F:=TColumnPropertiesForm(FColProps);
    F.Column:=SGDemo.Columns[SGDemo.Col-SGDemo.FixedCols];
    F.Show;
    end;
end;
```

The code first checks whether columns are in use, and if so then a `TColumnPropertiesform` instance is created if needed, and shown. The clicked column is passed to this form. Just like the main form, the form `TColumnPropertiesform` contains a `TIPropertyGrid`, which will be used to show the properties of the clicked column. If a previously visible instance is present, then it is re-used. For this, the main form must be notified when the form is destroyed: in the `DoFree`, the `FColProps` reference to the `TColumnPropertiesform` is set to `Nil`.

The form is shown in action in figure 1 on page 5.

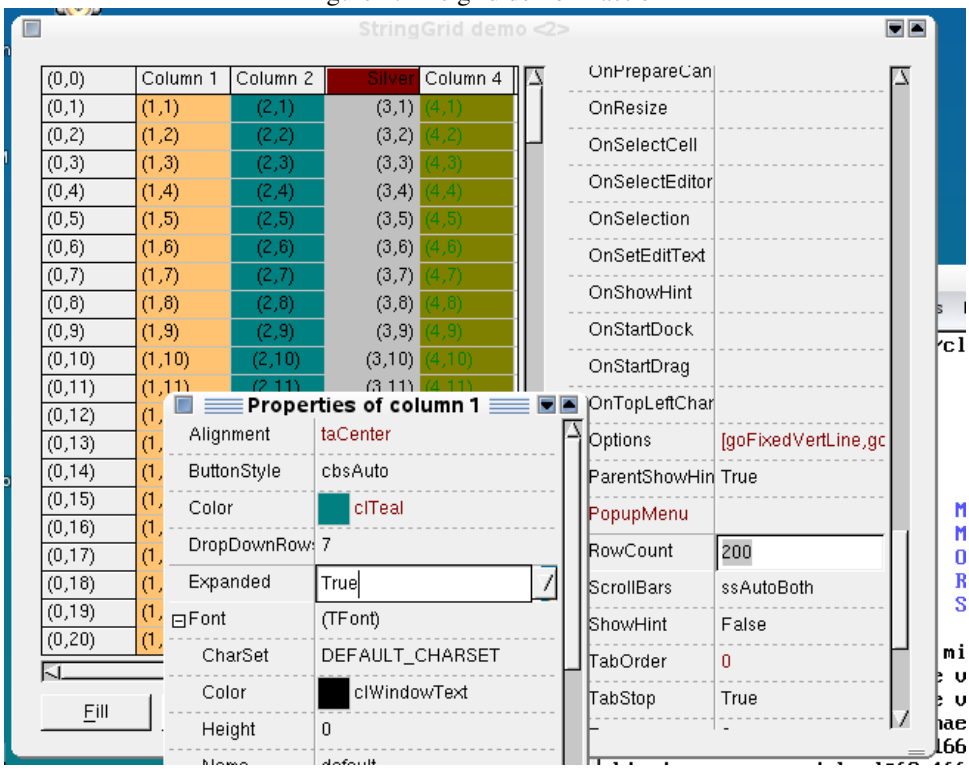
4 Editing

The Grid in Lazarus doesn't use the `TInPlaceEditor` class as the Delphi grid does. Instead, it uses directly a `TButton`, `TMaskEdit` or `TComboBox` controls. Which of these controls is used in a cell is controlled by the `ButtonStyle` property of`TGridColumn`: the control to be used can be set per column in the grid.

The appearance of the edit controls cannot be set by properties of the grid. Instead, the `OnSelectEditor` event allows to customize the editor when it is about to be displayed. The following is an example of an `OnSelectEditor` event handler:

```
procedure TMainForm.DoSelectEditor(Sender: TObject;
                                   aCol, aRow: Integer;
                                   var Editor: TWinControl);
```

Figure 1: The grid demo in action




```
begin
  If Editor is TStringCellEditor then
    CustomizeEdit(aCol,ARow,Editor as TStringCellEditor)
  else if Editor is TButtonCellEditor then
    CustomizeButton(aCol,ARow,Editor as TButtonCellEditor)
  else if Editor is TPickListCellEditor then
    CustomizePickList(aCol,ARow,Editor as TPickListCellEditor);
end;
```

Since the actual editor can be one of several different classes, the `Editor` argument is of type `TWinControl`, which makes it necessary to typecast it to its actual type. The `TStringCellEditor`, `TButtonCellEditor` and `TPickListCellEditor` are the descendents of `TCustomMaskedEdit`, `TButton` and `TComboBox`, used by the grid.

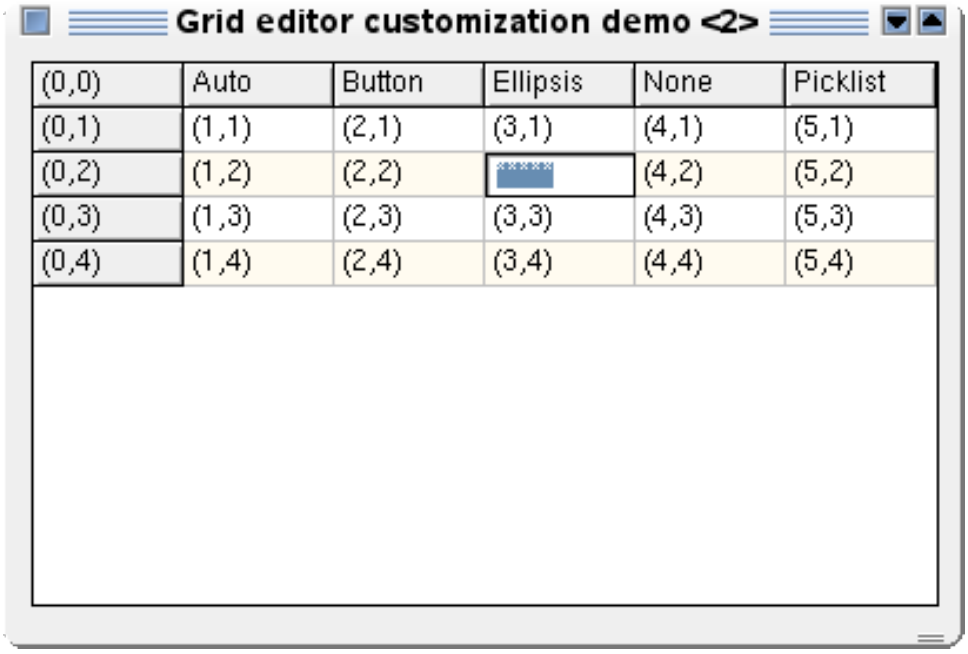
An example of customization would be the following:

```
procedure TMainForm.CustomizeEdit(Acol,ARow : Integer;
                                  Editor : TStringCellEditor);

begin
  Editor.PasswordChar:='*';
  Editor.MaxLength:=5;
end;
```

Which would have an effect as shown in figure 2 on page 6. The full source of the customize program is available and contains some customizations for the other 2 possible

Figure 2: A customized editor



editors as well.

5 Alternative editors

The `OnSelectEditor` event handler passes the `Editor` parameter by reference: this can be used to set the control to any `TWinControl` descendent. To demonstrate this, a small demonstration program is made which fills the grid with dates, and replaces the editor with a `TDateEdit` control in the `OnSelectEditor` handler:

```
procedure TMainForm.DoSelectEditor(Sender: TObject;
                                   aCol, aRow: Integer;
                                   var Editor: TWinControl);
begin
  if (ACol>0) and (ARow>0) then
    begin
    If (DE=Nil) then
      begin
      DE:=TStringDateEditor.Create(Self);
      De.Parent:=StringGrid1;
      DE.Visible:=False;
      end;
    Editor:=DE;
    end;
end;
```

The `TStringDateEditor` control is a descendent from the `TDateEdit` control. The `TDateEdit` control cannot be used directly: the grid uses some messages to communicate with the edit control:

GM_SETVALUE is sent when the grid wants to set the text to be edited.

GM_GETVALUE is sent when the grid wants to retrieve the edited text.

GM_SETGRID is sent with the grid instance to set the grid instance.

GM_SETBOUNDS is sent to set the position of the editor control.

GM_SELECTALL is sent when the control should perform a 'select all'.

GM_SETMASK is sent when the edit mask should be set.

GM_SETPOS is set to set the coordinates of the cell that should be edited.

And the editor control should handle these messages, and take appropriate action. Not all these messages should be responded to. The following class declaration of the `TStringDateEditor` control, shows which messages it responds to:

```
TStringDateEditor=class(TDateEdit)
private
  FGrid: TCustomGrid;
  FCol,FRow:Integer;
protected
  procedure msgSetValue(var Msg: TGridMessage);
    message GM_SETVALUE;
  procedure msgGetValue(var Msg: TGridMessage);
    message GM_GETVALUE;
  procedure msgSetGrid(var Msg: TGridMessage);
    message GM_SETGRID;
  procedure msgSetBounds(var Msg: TGridMessage);
    message GM_SETBOUNDS;
  procedure msgSelectAll(var Msg: TGridMessage);
    message GM_SELECTALL;
end;
```

The `TGridMessage` record is defined as follows:

```
TGridMessage=record
  LclMsg: TLMessage;
  Grid: TCustomGrid;
  Col,Row: Integer;
  Value: string;
  CellRect: TRect;
  Options: Integer;
end;
```

The meaning of the fields should be clear from their names. The `Options` field can be used to communicate back to the grid when the `GM_SETGRID` message is received:

```
procedure TStringDateEditor.msg_SetGrid(var Msg: TGridMessage);
begin
  FGrid:=Msg.Grid;
  Msg.Options:=EO_SELECTALL or EO_HOOKKEYPRESS or EO_HOOKKEYUP;
end;
```

The `Options` field can be a OR-ed combination of the following constants:

EO_AUTOSIZE Tells the grid that it can set the size and position of the editor directly. If it is not specified, the grid will send a `GM_SETBOUNDS` call.

EO_HOOKKEYDOWN, EO_HOOKKEYPRESS and EO_HOOKKEYUP Tells the grid to set the editor control's `OnKeyDown`, `OnKeyPress` and `OnKeyUp` event handlers to handle some common keys.

EO_SELECTALL Tells the grid that it can send a `GM_SELECTALL` message to the editor control.

The `GM_SETVALUE` and `GM_GETVALUE` message handlers are quite simple:

```
procedure TStringDateEditor.msgSetValue(var Msg: TGridMessage);
begin
  Self.Date:=StrToDate(Msg.Value);
end;

procedure TStringDateEditor.msgGetValue(var Msg: TGridMessage);
begin
  Msg.Value:=DateToStr(Self.Date);
end;
```

The `GM_SETBOUNDS` message is used for some special handling:

```
procedure TStringDateEditor.msgSetBounds(var Msg: TGridMessage);

Var
  R : TRect;

begin
  R:=Msg.CellRect;
  R.Right:=R.Right-ButtonWidth;
  BoundsRect:=R;
end;
```

The width of the `TDateEdit` control does not include the width of the pushbutton. Therefor, the width of the button must be substracted of the bounding rectangle prior to setting the `BoundsRect` property.

All that remains to do is implement the `GM_SELECTALL` handler:

```
procedure TStringDateEditor.msgSelectAll(var Msg: TGridMessage);
begin
  SelectAll;
end;
```

And the control is all done and ready to be used. The resulting can be seen in figure 3 on page 9.

6 Conclusion

The grid control in Lazarus has some definite advantages over the implementation of the standard grid in Delphi: it offers more properties to customize the behaviour and look of the grid. As can be seen in the code in this article, it is also quite easy to enhance the editing experience of the grid. In fact, it is quite easy to create a descendent which allows to use more advanced editing controls. It is left for a future contribution to show this.

Figure 3: The date stringgrid editor