KGrid user manual

Revision 3, December 1th 2009 by TK

Contents

1 Introduction

KGrid's primary purpose is to replace the feature poor TStringGrid/TDrawGrid controls from standard VCL with nearly zero programming effort and substantially extend their behavior.

In this document several important enhancements to the Borland's TStringGrid and TDrawGrid are described. I'm going to describe only those features that might be not obvious for all users, not even after studying the demo and reading the help file.

In Lazarus standard TStringGrid/TDrawGrid have more features than the Borland's components but KGrid supersedes this substantially, although the not full .

Most important here is all features are supported both in Delphi/C++ Builder and in Lazarus.

2 Compatibility issues

KGrid has been designed to retain the highest possible compatibility with TStringGrid and TDrawGrid (further referred as *standard grids*), programmers just need to replace TStringGrid/TDrawGrid with TKGrid. The compatibility is retained on public level only. The existing third party descendants of the Borland's TCustomGrid will most likely not work when rewritten in the following style:

```
type TxxxxGrid = class(TKCustomGrid) // K is inserted
```

All properties available in standard grids have been implemented in KGrid, version 1.1. However, there are several incompatibility issues:

- The class TKCustomGrid stands for TCustomGrid and TCustomDrawGrid. The class TKGrid stands for TDrawGrid and TStringGrid. Classes TInplaceEdit and TInplaceEditList and all supporting types that are in conflict with the basic KGrid principles (like TGetEditEvent) are not implemented and replaced by KGrid specific classes/types. All types have the TKGrid... prefix (except TKCustomGrid and TKGrid classes). Some members of similar structures got more meaningful names (for example in TKGridCoord and TKGridRect).
- KGrid is an Unicode control and uses only WideString data type up to Delphi 2007. In later RAD Studio versions and in Lazarus common string type is used (as it is UTF16 for RAD Studio and UTF8 in Lazarus).
- Properties Cols and Rows in TStringGrid are of type TStrings whereas in KGrid these are of type TKGridAxisItem. The TKGridAxisItem class supports a limited subset of TStrings methods and properties I‘ve considered to be relevant for a grid control. The TKGridAxisItem.Assign method is supported either for TStrings (ANSI strings) or optionally as TWideStrings (a class available in JCL – when JCL is configured in KControls.inc).
- The TKGrid.Canvas property is deprecated to use. Programmers should use the CellPainter.Canvas property instead for custom painting in new programs. Painting to TKGrid.Canvas will not work if goDoubleBufferedCells is True.
- The Objects property is deprecated in KGrid. Instead of using Objects and TKGridObjectCell, programmers should override TKGridCell and add custom properties.
- The DefaultDrawing property is obsolete and no longer supported. It is provided for backward compatibility and does nothing.

3 Editing capabilities

One of the most important features of KGrid is the possibility to assign a different in-place editor to each selectable and thus editable cell.

TKCustomGrid uses the following sequence while handling cell editing requests:

3.1 Creating a custom in-place editor and applying the cell value to it

Everytime the cell is about to be edited (mouse click, keyboard selection), the following sequence takes place:

1) **OnEditorCreate** is called. You must create an in-place editor here if you want to edit the cell or assign an existing one. You should just create the editor here and not assign any properties that require a Parent control to be assigned to the editor:

```
procedure TForm1.KGrid1EditorCreate(Sender: TObject; ACol, ARow: Integer;
  var AEditor: TWinControl);
begin
```

```
  AEditor := TEdit.Create(nil);
end;
```

In the above example, a simple TEdit control will be used to edit the cell content. Because only a simple text can be edited with TEdit, it will in most cases be used to edit the text of a textual cell instance like TKGridTextCell.

2) **OnEditorDataFromGrid** is called. Properties requiring a parent control to be assigned to the editor can be set here. The editor is still invisible but has a parent control already (this is always the TKCustomGrid instance passed as Sender to OnEditorDataFromGrid). At this point you should assign the actual cell data to the cell editor, as this processing usually requires in-place editor's Parent to be set:

```
procedure TForm1.KGrid1EditorDataFromGrid(Sender: TObject;
  AEditor: TWinControl; ACol, ARow: Integer; var AssignText: Boolean);
begin
  if AEditor is TEdit then
  begin
    TEdit(AEditor).Text := KGrid1.Cells[ACol, ARow];
    AssignText := False;
  end;
end;
```

In the above example, the cell text will be assigned to the TEdit in-place editor. Setting AssignedText to False prevents TKGrid's default behavior, which tries to assign the cell text (if any) into the in-place editor.

Note: TEdit is not a Unicode control up to Delphi 2007. Characters from another ANSI code pages than the default system code page will not be converted into TEdit.Text. You must use third party edit controls as in-place editor to implement true unicode behavior. In later RAD Studio versions or in Lazarus this control supports Unicode fully.

3.2 Assigning the value entered into the in-place editor to a cell value and destroying the editor

Everytime the editor should disappear (selection moves or Esc is pressed), the following sequence takes place:

1) **OnEditorDataToGrid** is called. The in-place editor has still Parent control assigned here. At this point you should update the cell data, as this processing still might require in-place editor's Parent to be set.

```
procedure TForm1.KGrid1EditorDataToGrid(Sender: TObject; AEditor:
TWinControl; ACol, ARow: Integer; var AssignText: Boolean);
begin
  if AEditor is TEdit then
  begin
```

```
    KGrid1.Cells[ACol, ARow] := TEdit(AEditor).Text;
    AssignText := False;
  end;
end;
```

In the above example, the text in TEdit will be assigned to the cell text, i.e. the Text property of the corresponding TKGridTextCell or TKGridAttrTextCell (or your custom descendant) cell instance becomes TEdit.Text.

2) **OnEditorDestroy** is called. The in-place editor has no Parent control here anymore. You should perform some editor-specific actions here just before the editor is destroyed. Although you can destroy the editor yourself by using FreeAndNil(AEditor), this is not necessary because the editor is destroyed automatically just after OnEditorDestroy terminates. In most cases you do not need to implement OnEditorDestroy at all:

```
procedure TForm1.KGrid1EditorDestroy(Sender: TObject; var AEditor:
TWinControl;
  ACol, ARow: Integer);
begin
  FreeAndNil(AEditor); // not necessary, we do not need to implement this
event handler in this case!
end;
```

The good reason when you must implement OnEditorDestroy is if you wish to reuse a previously created control as in-place editor again. In this case, you can prevent its destruction by setting AEditor := nil. At following OnEditorCreate call you can assign this control instead of creating a new one.

3.3 Another options for customizing the behavior of in-place editors

In addition to the four basic events, another three events concerning the in-place editor are triggered:

- **OnEditorKeyPreview** is called each time when a grid-specific key is pressed. The grid cannot automatically determine the correct behavior for these keys for all in-place editors. Thus, OnEditorKeyPreview is here to decide which key is about to be handled by in-place editor and which by TKCustomGrid itself.
- **OnEditorSelect** is called just after OnEditorDataToGrid to decide where the caret should be placed. The information about this comes from the grid. This event is intended for custom in-place editors that use carets. There should be no need to implement custom behavior for in-place editors that subclass the standard edit control from Win32-API.
- **OnEditorResize** is called every time the in-place editor needs to be positioned within the cell rectangle. You can modify the position of the editor within the cell here. Please note that the in-place editor’s window cannot extend the cell bounding rectangle and is clipped accordingly (Note for Lazarus: it is possible only under Windows OS).

There is always one in-place editor active within the grid. However, you can pre-cache several in-place editors (create them e.g. somewhere in FormCreate), and use them on demand in the OnEditorCreate - OnEditorDataFromGrid - OnEditorDataToGrid - OnEditorDestroy sequence. All you need in this case is to assign the corresponding editor into AEditor in OnEditorCreate and prevent destroying of the editor in OnEditorDestroy by setting AEditor to nil, as already mentioned before.

3.4 Appearance of an in-place editor

The in-place editor in standard grids was always a modified TEdit control covering the entire cell bounding rectangle. It has been displayed without a border.

In TKGrid, in-place editors can be displayed in any appearance including true transparency, i.e. the cell background is painted to the device context of the in-place editor – default behavior for this is performed by TKGridCellPainter.DrawEmptyCell.

Because it is impossible to determine which control should appear truly transparent in standard VCL/LCL, a new property EditorTransparency has been introduced. Through this property you can tell TKGrid whether the current in-place editor should be handled as transparent control, thus the cell background should be painted to its device context first.

The best point where to assign EditorTransparency is the OnEditorCreate event. You cannot design all editors transparent as this would cause incorrect painting of some controls or flickering if no double buffering is used for the grid. If Editortransparency is etDefault, TKGrid decides which control should be painted with transparency and which not. This default behavior should work with most controls commonly used as in-place editors.

4 Sorting interface

It is possible to sort rows and/or columns in TKGrid.

Rows can be sorted by arbitrary column. If the grid has fixed rows, an arrow shape is used to indicate that column in the first fixed row or as specified by TKGridAxisItem.SortArrowIndex property. The same behavior applies to columns, which can be sorted by arbitrary row.

During sorting, TKGrid compares cell contents. You can customize this behavior via the OnCompareCells event handler. A simple example for comparing two cell texts could be:

```
function TForm1.KGrid1CompareCells(Sender: TObject; Col1, Row1, Col2,
  Row2: Integer): Integer;
begin
  Result := CompareWideStrings(KGrid1.Cells[Col1, Row1], KGrid1.Cells[Col2,
Row2]);
end;
```

In above example for Delphi 2007 and older Delphi versions, the built in function CompareWideStrings is called to perform correct comparison of the cell texts.

Although this is a very quite simple notation, it is not very fast and thus usable only with limited number of sorted rows. The reason lies in the current implementation of WideString which doesn't use fast reference counting mechanism as ANSI string up to Delphi 2007. Another reason is the index checks performed while accessing the Cells property. To boost the speed of the above comparison, use this notation instead:

```
function TForm1.KGrid1CompareCells(Sender: TObject; Col1, Row1, Col2,
  Row2: Integer): Integer;
begin
  Result := CompareWideChars(
    TKGridTextCell(KGrid1.ArrayOfCells[Row1, Col1]).TextPtr,
    TKGridTextCell(KGrid1.ArrayOfCells[Row2, Col2]).TextPtr);
end;
```

With this notation PWideChars are compared instead of WideStrings. Furthermore, direct read only access to cells in TKGrid is used by means of ArrayOfCells. This approach is much more faster.

For later RAD Studio versions and Lazarus the common string type is used which supports the fast reference counting mechanism so you don't need to use the TextPtr properties.

Sorting is performed by an optimized quick sort algorithm. This algorithm is not recursive, so you don't need to worry about stack overflows in OnCompareCells.

If you just need to sort persisting rows or columns, you obviously click the first fixed column or row by mouse or programmatically call TKGrid.**SortRows** or TKGrid.**SortCols**.

However, it is possible to insert new rows or columns into previously sorted grid. This can be done via the TKGrid.**InsertSortedRow** and TKGrid.**InsertSortedCol** methods. In this case, you need to adapt your OnCompareCells event handler to be able to compare any existing cell value with a new value.

This modification is simple. During InsertSortedRow, OnCompareCells is called with the Row1 parameter equal to cInvalidIndex. Similarly, it is called with the Col1 parameter equal to cInvalidIndex during InsertSortedCol. Following example shows the modified OnCompareCells event:

```
function TForm1.KGrid1CompareCells(Sender: TObject; Col1, Row1, Col2,
  Row2: Integer): Integer;
var
  W: PWideChar;
begin
  if (Col1 = cInvalidIndex) or (Row1 = cInvalidIndex) then
    W := PWideChar(FTextToInsert)
  else
    W := TKGridTextCell(KGrid1.ArrayOfCells[Row1, Col1]).TextPtr;
  Result := CompareWideChars(W,
    TKGridTextCell(KGrid1.ArrayOfCells[Row2, Col2]).TextPtr);
end;
```

With that modification of OnCompareCells, you can call InsertSortedRow in the following manner:

```
procedure TForm1.ACInsertSortedRowExecute(Sender: TObject);
var
  ARow, ByCol: Integer;
begin
  FTextToInsert := 'new text'; // FTextToInsert is of type WideString
  if KGrid1.InsertSortedRow(ByCol, ARow) then
  begin
    { Now it is time to actually update the inserted empty row:
      - ARow is the index of the newly inserted row now
      - ByCol is the index of the column by that rows are actually sorted }
    KGrid1.LockSortMode; // lock the sorting status
    try
      KGrid1.Cells[ByCol, ARow] := FTextToInsert;
      // and fill another columns in that row...
    finally
      KGrid1.UnlockSortMode; // unlock the sorting status
    end;
  end;
end;
```

5 Cell merging

KGrid has powerfull cell merging and splitting features which, however, are easy to use. To merge the first selectable cell across 2 columns and 2 rows just call:

```
KGrid1.CellSpan[KGrid1.FixedCols, KGrid1.FixedRows] := MakeCellSpan(2, 2);
```

Or:

```
KGrid1.Cell[KGrid1.FixedCols, KGrid1.FixedRows].Span := MakeCellSpan(2, 2);
```

Or:

```
KGrid1.Cell[KGrid1.FixedCols, KGrid1.FixedRows].ColSpan := 2;
KGrid1.Cell[KGrid1.FixedCols, KGrid1.FixedRows].RowSpan := 2;
```

As you can see, the implementation is similar to HTML. To split the merged cell just call:

```
KGrid1.Cell[KGrid1.FixedCols, KGrid1.FixedRows].CellSpan := MakeCellSpan(1,
1);
```

Note: Unlike tables in e.g. Microsoft Word, cells cannot be split to more or less columns or rows than the number of columns/rows they are merged from.

Note: Cells cannot be merged across fixed area boundaries. It means, for example, if you have just one fixed row, those fixed cells cannot be merged across rows.

Note: On each grid layout change, merged cells are checked for range violations. It is the user's responsibility to disallow respective layout changes to prevent existing cell merging layout to be lost.

6 Printing and previewing

The tables created in KGrid can be (instantly) previewed and printed in a very easy manner. KGrid fully supports the built in KControls printing and previewing engine. To print the grid to a default printer with default settings, just call:

```
KGrid1.PrintOut;
```

To preview the grid, just drop a TKPrintPreview component to a form and assign your KGrid instance to its Control property.

To preview the grid changes instantly, use e.g. TActionlist and TAction.OnUpdate event. In this event, just read the KGrid1.PageSetup property. In the following example, a TAction instance with name ACPreview is defined:

```
procedure TForm1.ACPreviewUpdate(Sender: TObject);
begin
  //...
  KGrid1.PageSetup; // validates page setup
end;
```

Because TAction.OnUpdate is called at idle time and (almost) every grid change invalidates the grid's PageSetup, requesting this instance validates the PageSetup and invalidates all associated TKPrintPreview controls (only if the grid invalidated the page setup). There can be more TKPrintPreviews associated to a single KGrid.

This implementation doesn't affect the grid performance (well, except when there is few idle time and TKPrintPreview repainting takes even more time).

7 Column, row and grid autosizing

KGrid has built in cell autosizing features. Cell autosizing is implemented by a versatile cell measurement system. Autosizing must be invoked explicitly by calling functions AutoSizeCol, AutoSizeRow and AutoSizeGrid.

Default cell measurement is implemented in TKGridCellPainter.DefaultMeasure. It can be adjusted by overriding TKGridCellPainter or by using the OnMeasureCell event:

```
procedure TForm1.KGrid1MeasureCell(Sender: TObject; ACol, ARow: Integer;
  R: TRect; State: TKGridDrawState; var Extent: TPoint);
Begin
```

```
  { Here you should implement your specific cell measurement code or
    just adjust the cellpainter settings exactly like in OnDrawCell and
    call KGrid1.CellPainter.DefaultMeasure: }
  //... adjust cellpainter like in OnDrawCell
  Extent := KGrid1.CellPainter.DefaultMeasure;
end;
```

8 Cell painting, double buffering and clipping

There is possible to:

- Set the DoubleBuffered property to True as usual in all VCL controls. In KGrid, using DoubleBuffered is (temporary) memory intensive but fast for huge grids.
- Set the goDoubleBufferedCells option to True in TKGrid.Options. This method is less memory intensive because it makes only the cells double buffered. However, it is not so fast for huge grids as DoubleBuffered.
- Set the goClippedCells option to True in TKGrid.Options This makes the cell contents to be clipped within the cell bounding rectangle. You can use this option when painting any custom shapes or images within the cell that cannot be easily clipped within the cell.
- Set the goEraseBackground option to True in TKGrid.Options to allow the WM_ERASEBKGND to be called in all painting modes except DoubleBuffered mode. Although the control fills the entire client area .

9 Cell aware methods vs. event handlers

In standard grids, you could customize the grid's behavior only by means of event handlers. In KGrid, another way is possible: **cell aware methods**.

Cell aware methods are implemented within the base cell class TKGridCell. All other classes can only override these cell aware methods and cannot add new ones.

Difference between event handlers and cell aware methods:

Cell aware methods are intended for custom overrides of TKGridCell. You can override these methods in your TKGridCell descendant. Within the overriden methods you can implement the same behavior as with event handlers. Event handlers, when assigned, have always priority against cell aware methods, so if you want your cell aware method to be called, do not assign the corresponding event handler.

Example:

```
type
  TMyCell = class(TKGridCell) // this is my new cell class
  public
    // this is the overridden cell aware method
    procedure EditorCreate(ACol, ARow: Integer; var AEditor: TWinControl);
override;
    // etc ...
  end;

procedure TKGridCell.EditorCreate(ACol, ARow: Integer; var AEditor:
TWinControl);
begin
  AEditor := TComboBox.Create(nil)
end;

// etc...
```

The above example works exactly like:

```
procedure TForm1.KGrid1EditorCreate(Sender: TObject; ACol, ARow: Integer;
  var AEditor: TWinControl);
begin
  AEditor := TComboBox.Create(nil);
end;
```

Note: Do not call cell aware methods directly. They are called automatically by TKGrid.

Note: Only cell related events are available as cell aware method counterparts.