Last edited on May 5th, 2008. Pre-release software. Specifications subject to change.

Syncfusion

Syncfusion Essential Grid – WPF Edition

Contents

Syncfusion Essential Grid WPF Edition

Beta 1 (Publically available on May 12, 2008)

This document contains a short overview of features implemented for the first public beta version of Essential Grid WPF Edition. Essential Grid WPF Edition is a powerful cell oriented grid control that offers excellent performance characteristics.

The grid at its core functions as a very efficient display engine for tabular data that can be customized down to the cell level. It does not make any assumptions on the structure of the data (many grid controls implemented as straight data bound controls make such explicit assumptions). This leads to a very flexible design that can be easily adapted to a variety of tasks including the display of completely unstructured data and the display of structured data from say, a database.

The display system also hosts a powerful and complete styles architecture. Settings can be specified at the cell level or at higher levels using parent styles that are referred to as base styles. Base styles can affect groups of cells. Cell level settings override any higher level settings and enable easy customization right down to that level.

With this version our core focus has been on the underlying architecture for displaying cells with virtualized cell editors in a manner that enables good performance characteristics. The core display system also supports several building block features such as nested grids, virtual mode and support for a virtually unlimited number of rows and columns. Higher level features such as clean XAML integration, design-time support and routed events have not been worked on for this release. These features have however, been taken into account and implementing these is just a matter of adding this additional code once we are comfortable that our work on the core display framework is complete.

Users of Essential Grid Windows Forms edition will be pleased to observe the same architectural underpinnings in the WPF Edition.

Please note that this version is an early preview release. Some higher level features remain unimplemented. These will be added in later builds.

We will walk through common usage scenarios first and then discuss some of the architectural and implementation underpinnings that will help you better understand the Essential Grid WPF Edition platform.

Common Usage

Adding the grid to a window control

It is simple to host the grid inside a ScrollViewer element. Scrollbar visibility can be set with HorizontalScrollBarVisibility and VerticalScrollBarVisibility.

When hosted thus, the grid integrates nicely with the ScrollViewer control supporting both CanContentScroll modes. If CanContentScroll is set to True the grid will be nested inside the scrollviewer and its MeasureOverride method will have no effect. The grid will be responsible for scrolling the rows and columns in the given viewable area and will synchronize as required with the scrollbars of the ScrollViewer. Rows and columns cells will be virtualized (loaded on demand).

```
<ScrollViewer CanContentScroll="True"
HorizontalScrollBarVisibility="Auto" VerticalScrollBarVisibility="Auto">

     <sf:GridControl x:Name="grid"/>

</ScrollViewer>
```

If CanContentScroll is set to False the grid will return the total height of all rows and total width of all columns in its MeasureOverride method. It will not handle any scrolling delegating it to the parent control. Rows and column cells will not be virtualized since the grid handles this mode as if all rows and columns are visible. In this mode setting frozen rows or columns will have no effect.

```
        <ScrollViewer CanContentScroll="False"
HorizontalScrollBarVisibility="Auto" VerticalScrollBarVisibility="Auto"
Grid.Row="0">

            <StackPanel >

                <TextBlock>Grid 1 embedded inside ScrollViewer with
CanContentScroll set to False</TextBlock>

                <Border BorderBrush="LightGray" BorderThickness="1pt">

                    <sf:GridControl x:Name="grid1"/>

                </Border>

            </StackPanel>

        </ScrollViewer>
```

You can also place the grid directly into a parent panel. This scenario is very similar to the ScrollViewer case when CanContentScroll is set to False. The grid will return the total height of all rows and total width of all columns in its MeasureOverride method, and leave any scrolling up to the parent element.

```
        <StackPanel Grid.Row="1">

            <TextBlock>Grid 2 as child of StackPanel</TextBlock>

            <Line Fill="Gray"/>

            <Border BorderBrush="LightGray" BorderThickness="1pt">

                <sf:GridControl x:Name="grid2"/>

            </Border>

        </StackPanel>
```

Setting the row and column count

Each grid control instance is tied to a model which contains the data represented by the grid control. The GridModel has RowCount and ColumnCount properties. These can be set to change the number of rows and columns in the grid control.

```
            grid1.Model.RowCount = 8;

            grid1.Model.ColumnCount = 5;
```

Setting row heights and column widths

The grid model also stores information on row heights and column widths. Its ColumnWidths and RowHeights properties can be changed using indexers as shown below.

```
            grid1.Model.ColumnWidths[0] = 30;

            grid1.Model.ColumnWidths[1] = 80;
            grid1.Model.ColumnWidths[2] = 100;

            grid1.Model.ColumnWidths[3] = 50;

            grid1.Model.ColumnWidths[4] = 250;

            grid1.Model.RowHeights[5] = 40;

            grid1.Model.RowHeights[3] = 40;
```

You can also specify the DefaultLineSize setting on ColumnWidths and RowHeights in order to set the default width or height.

```
            grid1.Model.RowHeights.DefaultLineSize = 20;

            grid1.Model.ColumnWidths.DefaultLineSize = 100;
```

Hiding rows and columns

Essential Grid supports efficiently hiding rows and columns from display. You can hide and unhide ranges of rows and columns as shown below.

```
        private void checkBoxRows_Checked(object sender, RoutedEventArgs
e)

        {

            bool hide = checkBoxRows.IsChecked == true;

            grid.Model.RowHeights.SetHidden(2, 100, hide);

            grid.Model.RowHeights.SetHidden(110, 1000, hide);

            grid.Model.RowHeights.SetHidden(1010, 10000, hide);

            grid.Invalidate(true);

        }
        private void checkBoxColumns_Checked(object sender, 
RoutedEventArgs e)

        {

            bool hide = checkBoxColumns.IsChecked == true;

            grid.Model.ColumnWidths.SetHidden(2, 100, hide);

            grid.Model.ColumnWidths.SetHidden(110, 1000, hide);

            grid.Model.ColumnWidths.SetHidden(1010, 10000, hide);

            grid.Invalidate(true);

        }
```

Header, Frozen and Footer Rows and Columns

Header rows and columns adorn the top and left segments of a grid control. From an implementation perspective, Header rows and columns are cells that cannot be navigated to with arrow keys. Other than this they are no different from regular cells. The default appearance for these cells is defined by the Model.HeaderStyle base style. You can change the appearance of the header cells by manipulating the HeaderStyle. For instance you can set their background color to red using the code below.

```
grid.Model.HeaderStyle.Background = Brushes.Red;
```

The base style for regular cells is the Model.TableStyle base style.

Typically you will want to freeze header rows and columns. Frozen rows and columns do not scroll along with the rest of the rows and columns. They stay in place when you scroll. Frozen rows and columns will always be visible at the top and left of the grid. When you scroll through the grid these rows and columns always stay in the viewable area.

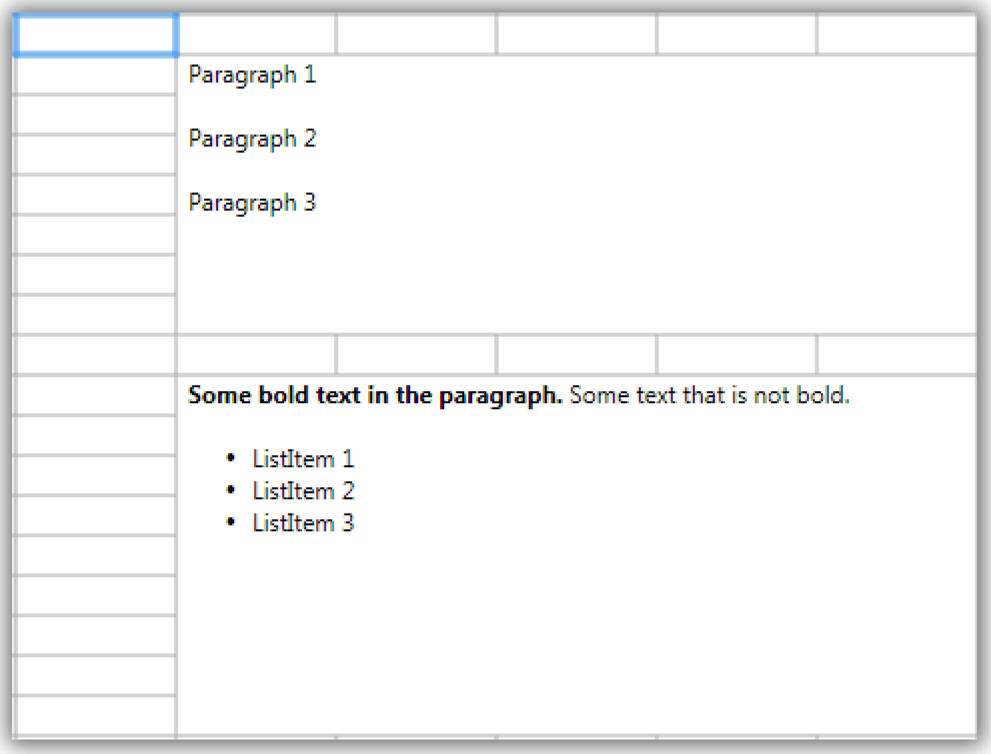
You can do this by setting both HeaderRows and FrozenRows properties to the same value. Similar settings are available for columns. The default value for HeaderRows, FrozenRows, HeaderColumns and FrozenColumns is 1 but these can be adjusted individually. There can certainly be more Frozen Rows than Header Rows for instance.

It is also possible to specify Footer rows and Columns. Footer rows and columns are specified with the FooterRows and FooterColumns properties. These cells will always remain visible at the bottom and right side of the grid when you scroll through the grid.

```
            grid1.Model.HeaderRows = 1;

            grid1.Model.FrozenRows = 2;

            grid1.Model.FooterRows = 1;

            grid1.Model.FrozenColumns = 1;

            grid1.Model.FooterColumns = 1;
```

Covered Cells

Covered Cells are cells that span over neighboring cells. The combined cells will act as if they are one single cell visually and programmatically.


Paragraph 1

Paragraph 2

Paragraph 3

Some bold text in the paragraph. Some text that is not bold.

- ListItem 1
- ListItem 2
- ListItem 3

```
            grid1.CoveredCells.Add(new CoveredCellInfo(4, 0, 6, 1));
            grid1.CoveredCells.Add(new CoveredCellInfo(4, 18, 6, 19));
```

CellSpanBackgrounds

CellSpanBackgrounds combine the background of neighboring cells. The cells will still be independent cells unlike with covered cells. Only the background for these cells is managed by CellSpanBackground and will override any individual GridStyleInfo.Background set for individual cells in the range.

```
            Pen bgPen = new Pen(Brushes.Gold, 6);

            bgPen.EndLineCap = PenLineCap.Triangle;

            bgPen.StartLineCap = PenLineCap.Triangle;

            bgPen.Freeze();



            SolidColorBrush bg1 = new
SolidColorBrush(Color.FromArgb(128, 128, 0, 0));



            GradientBrush bg = new
LinearGradientBrush(Color.FromArgb(128, 255, 255, 0),
Color.FromArgb(128, 0, 255, 255), 45.0);

            CellSpanBackgrounds.Add(new CellSpanBackgroundInfo(9, 2, 22,
8, false, false, bg1, bgPen));

            CellSpanBackgrounds.Add(new CellSpanBackgroundInfo(0, 0, 12,
1, false, false, bg, null));
```

Virtual Grid

Essential grid supports virtual mode of operation. The grid lets you provide cell styles on demand using the QueryCellInfo event handler. It also supports saving these back into a custom data source CommitCellInfo event handler.

```
            // fill cell contents on demand.

            grid.Model.QueryCellInfo += new
GridQueryCellInfoEventHandler(Model_QueryCellInfo);



            // save back cell value into dictionary

            grid.Model.CommitCellInfo += new
GridCommitCellInfoEventHandler(Model_CommitCellInfo);





        Dictionary<RowColumnIndex, object> committedValues = new
Dictionary<RowColumnIndex, object>();



        void Model_CommitCellInfo(object sender,
GridCommitCellInfoEventArgs e)

        {

            if (e.Style.HasCellValue)

            {

                committedValues[e.Cell] = e.Style.CellValue;

                e.Handled = true;

            }

        }



        void Model_QueryCellInfo(object sender,
GridQueryCellInfoEventArgs e)
```

```
        {

            if (e.Cell.RowIndex == 0)

            {

                if (e.Cell.ColumnIndex > 0)

                    e.Style.CellValue = e.Cell.ColumnIndex;

            }

            else if (e.Cell.RowIndex > 0)

            {

                if (e.Cell.ColumnIndex == 0)

                    e.Style.CellValue = e.Cell.RowIndex;

                else if (e.Cell.ColumnIndex > 0)

                {

                    if (committedValues.ContainsKey(e.Cell))

                        e.Style.CellValue = committedValues[e.Cell];

                    else

                        e.Style.CellValue = String.Format("{0}/{1}",
e.Cell.RowIndex, e.Cell.ColumnIndex);

                }

            }
```

The usage model for virtual mode is fairly simple as shown in the code snippet above. There is however a lot of work going on behind the scenes to make this work in an efficient manner.

Volatile Cell Styles

QueryCellInfo will be raised the first time you access the contents of a cell with a call to Grid.Model[rowIndex, columnIndex] or when the grid calls this indexer internally before painting cells. The indexer returns an object of type GridStyleInfo. After querying the cell contents they remain cached in a volatile cache that holds weak references to the cell styles. This ensures that this data is available for reuse when needed. At the same time it does not stand in the way of the garbage collector if memory needs to be freed. Once a style gets garbage collected it will be removed from the volatile cache.

You can manually force QueryCellInfo to be called again when you call GridControl.InvalidateCell(cell) or GridModel.VolatileCellStyles.Clear(cell).

Render Cell Styles

Prior to display of a cell, the PrepareRenderCell event is raised. This event is not raised from the Model. Instead it is raised from the GridControlBase directly. This has the advantage that if you want multiple grids to display the same Model, individual grids can override the cell contents individually.

```
           // view specific cell color

             grid.PrepareRenderCell += new
GridPrepareRenderCellEventHandler(grid_PrepareRenderCell);



        void grid_PrepareRenderCell(object sender,
GridPrepareRenderCellEventArgs e)

        {

            if (e.Cell.RowIndex > 0 && e.Cell.ColumnIndex > 0)

            {

                if (e.Cell.RowIndex % 2 == 0)

                    e.Style.Background = Brushes.LightSkyBlue;

            }

        }
```

PrepareRenderCell is used to initialize the so called RenderStyles. RenderStyles are of type GridRenderStyleInfo and derive from GridStyleInfo. GridRenderStyleInfo are tied to a GridControlBase instance. The render style provides additional properties to obtain access to the associated GridControl , CellRenderer and the underlying ModelStyle instance from the volatile cells cache described earlier.

To access the render style for a cell you can call GridControlBase.GetRenderStyleInfo.

Render Styles are created only for the cells that are visible and will be discarded the moment a cell is scrolled out of view (with the exception being if the current cell is scrolled out of view; they are retained in such case alone).

In contrast, Volatile Cell Styles from the GridModel are often also created for cells outside the viewable area and can stay in the cache even when a cell is scrolled out of view.

Cell Types and Cell Renderers

You can change the CellType for a cell with the GridStyleInfo.CellType property.

```
             grid.Model[1,1].CellType = "TextBox";
```

To change the default cell type you can also change it in a base style.

```
             grid.Model.TableStyle.CellType = "TextBox";

             grid.Model.HeaderStyle.CellType = "Static";
```

You can add custom cell types by deriving classes from GridCellRendererBase and GridCellModelBase. To register new cell types with a grid add the cell models to the GridModel.CellModels collection.

```
              grid.Model.CellModels.Add("FlowDocumentReader", new
FlowDocumentReaderCellModel());
```

To assign this cell type to a cell set the cells CellType to the name you used for registering the cell.

```
                FlowDocument flowDocument3 =
(FlowDocument)TryFindResource("FlowDocument3");

                if (flowDocument3 != null)

                {

                    grid.Model[50, 2].CellType = "FlowDocumentReader";
                    grid.Model[50, 2].CellValue = flowDocument3;

                    grid.Model.CoveredCells.Add(new CoveredCellInfo(50,
2, 68, 9));

                }
```

If you wish to create a cell renderer that hosts a WPF control you should derive it from GridVirtualizingCellRenderer. The most important method to override is the OnInitializeContent method. It will be called for every UIElement that is created for cells that show this renderer.

```
using System.Windows;

using System.Windows.Controls;

using System.Windows.Documents;

using System.Windows.Input;

using Syncfusion.Windows.Controls.Cells;

using Syncfusion.Windows.Controls.Grid;

using Syncfusion.Windows.Controls.Scroll;



namespace Syncfusion.Samples.RichTextBoxCell

{

    public class FlowDocumentReaderCellModel :
GridCellModel<FlowDocumentReaderCellRenderer>

    {

    }



    public class FlowDocumentReaderCellRenderer :
GridVirtualizingCellRenderer<FlowDocumentReader>
```

```
    {

        public FlowDocumentReaderCellRenderer()

        {

            IsControlTextShown = false; // CellValue must be
FlowDocument

            IsFocusable = true;

            AllowRecycle = false;

        }



        /// <summary>

        /// Called to initialize the content of the cell

        /// using the information from the cell style (value, text,

        /// behavior etc.). You must override this method in your

        /// derived class.

        /// </summary>

        /// <param name="flowDocumentReader">The text box.</param>

        /// <param name="style">The cell style info.</param>

        public override void OnInitializeContent(FlowDocumentReader
flowDocumentReader, GridRenderStyleInfo style)

        {

            flowDocumentReader.Padding = new Thickness(0);

            flowDocumentReader.Document = style.CellValue as
FlowDocument;
VirtualizingCellsControl.SetWantsMouseInput(flowDocumentReader, true);

        }



        protected override void OnUnwireUIElement(FlowDocumentReader
uiElement)

        {

            // need to reset Document when scrolled out of view.
Otherwise an exception is

            // thrown next time the assigned document is assigned to
another new FlowDocumentReaderCell

            uiElement.Document = null;

            base.OnUnwireUIElement(uiElement);

        }



        protected override bool
ShouldGridTryToHandlePreviewKeyDown(KeyEventArgs e)

        {

            if (CurrentCellEditor.IsFocused && e.Key != Key.Escape)

                return false;



            return true;

        }
```

```
    }



}
```

Virtualized Cell Editors

The grid supports virtualization of live WPF controls inside cells. When a cell is scrolled out of view the control inside that cell is unloaded or “recycled”. This usage model is designed for optimal resource utilization and is based on the FlyWeight pattern. For additional details on this pattern please refer to http://en.wikipedia.org/wiki/Flyweight_pattern .

GridVirtualizingCellRenderer is the base class for cell renderers that need live UIElement visuals displayed in a cell. You can derive from this class and provide the type of the UIElement you want to show inside cells as type parameter. The class provides strong typed virtual methods for initializing content of the cell and arranging the visuals.

The class manages the creation of cells UIElement objects when the cell is scrolled into view and also unloading of the elements when the cell scrolls out of view. The class offers an optimization in which elements can be recycled when the "AllowRecycle" setting is set. In this case when a cell is scrolled out of view it is moved into a “recycle bin” and the next time a new element is scrolled into view the element is recovered from the recycle bin and re-initialized with the new content of the cell.

Another optimization is support for cells rendering themselves directly to the drawing context. When "SupportsRenderOptimization" is true the UIElement will only be created when the user moves the mouse over the cell or if the UIElement is needed for other reasons.

After a UIElement was created the virtual methods "WireUIElement" and "UnwireUIElement" are called to wire any event listeners.

When you implement custom controls please note that updates to the appearance and content of child elements, creation and unloading of elements will not trigger ArrangeOverride or Render calls in the parent canvas.

The following sample demonstrates the SupportsRenderOptimization optimization.

```
            grid.Model.CellModels.Add("VirtualizedCell", new VirtualizedCellModel());

            grid.Model.TableStyle.CellType = "VirtualizedCell";

            grid.Model.TableStyle.CellValue = "Edit Me!";
```

```
    public class VirtualizedCellModel :
GridCellModel<VirtualizedCellRenderer>

    {

    }



    public class VirtualizedCellRenderer :
GridVirtualizingCellRenderer<TextBox>

    {



        // The Virtualized Cell will display "Edit Me!" when

        // it is a live UIElement editor. If text is just

        // rendered in OnRender it will display "Render".

        //

        // When you hover the mouse over a cell it will become

        // a live UIElement editor and not render the cell anymore.

        // You can click inside it and edit its contents.

        //

        // When you scroll a cell outside view it will be switched

        // back to a rendered cell.

        //

        // This approach improves scrolling speed since rendering
        // text directly is faster then placing a UIElement as soon

        // a cell becomes visible.

        //

        // You can disable this mechanism by setting
"SupportsRenderOptimization = false" in the ctor.



        TextBoxPaint renderText;



        public VirtualizedCellRenderer()

        {

            SupportsRenderOptimization = true;

            AllowRecycle = true;

            IsControlTextShown = true;

            IsFocusable = true;



            renderText = new TextBoxPaint(new Typeface("Arial"), 10,
Brushes.Black);

            renderText.Trimming = TextTrimming.None;

            renderText.HorizontalAlignment = TextAlignment.Left;

            renderText.VerticalAlignment = VerticalAlignment.Top;

            renderText.WrapText = true;

        }
        protected override void OnRender(DrawingContext dc,
RenderCellArgs rca, GridRenderStyleInfo cellInfo)

        {

            if (rca.CellVisuals != null)

                return;



            // Will only get hit if SupportsRenderOptimization is true,
otherwise rca.CellVisuals is never null.

            string s = String.Format("Render{0}/{1}", rca.RowIndex,
rca.ColumnIndex);

            renderText.DrawText(dc, rca.CellRect, s);

        }



        /// <summary>

        /// Called to initialize the content of the cell

        /// using the information from the cell style (value, text,

        /// behavior etc.). You must override this method in your

        /// derived class.

        /// </summary>

        /// <param name="textBox">The text box.</param>

        /// <param name="style">The cell style info.</param>

        public override void OnInitializeContent(TextBox textBox,
GridRenderStyleInfo style)
```

```
        {
            textBox.Padding = new Thickness(0);
            textBox.Text = GetControlText(style);
            VirtualizingCellsControl.SetWantsMouseInput(textBox, true);
        }


        public override string GetControlTextFromEditor()
        {
            return CurrentCellEditor.Text;
        }


        protected override void OnInitialize()
        {
            base.OnInitialize();
            ControlText = GetControlText(CurrentStyle);
        }


        protected override void OnWireUIElement(TextBox textBox)
        {
            base.OnWireUIElement(textBox);
            textBox.TextChanged += new
TextChangedEventHandler(textBox_TextChanged);
```

```
        }



        /// <summary>

        /// Unwire previously wired events from textBox.

        /// </summary>

        /// <param name="textBox"></param>

        protected override void OnUnwireUIElement(TextBox textBox)

        {

            base.OnUnwireUIElement(textBox);

            textBox.TextChanged -= new
TextChangedEventHandler(textBox_TextChanged);

        }



        void textBox_TextChanged(object sender, TextChangedEventArgs e)

        {

            TextBox textBox = (TextBox)sender;

            if (!this.IsInArrange && IsCurrentCell(textBox))

            {

                TraceUtil.TraceCurrentMethodInfo(textBox.Text);

                if (!SetControlText(textBox.Text))

                    RefreshContent(); // reverses change.

            }
        }



        protected override void
OnGridPreviewTextInput(TextCompositionEventArgs e)

        {

            CurrentCell.ScrollInView();

            CurrentCell.BeginEdit(true);

        }



        protected override bool
ShouldGridTryToHandlePreviewKeyDown(KeyEventArgs e)

        {

            if (CurrentCellEditor.IsFocused && e.Key != Key.Escape)

                return false;



            return true;

        }





    }
```

DataTemplates in Cells

Essential grid has a DataTemplate cell you can use to assign data templates to individual cells. This allows for a very flexible display system.

```
        void Model_QueryCellInfo(object sender, 
Syncfusion.Windows.Controls.Grid.GridQueryCellInfoEventArgs e)

        {

            if (e.Cell.RowIndex > 1 && e.Cell.ColumnIndex == 2)

            {

                e.Style.CellType = "DataTemplate";

                e.Style.CellTemplateKey = "editableEmployee";

                e.Style.CellValue = 
employeesSource.Employees[e.Cell.RowIndex % 
employeesSource.Employees.Count];

                e.Style.Background = Brushes.Linen;



            }

        }



Window1.xaml

<Window x:Class="DataTemplateCell.Window1"

    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:sfScroll="clr-
namespace:Syncfusion.Windows.Controls.Scroll;assembly=Syncfusion.GridCom
mon.Wpf"

    xmlns:sf="http://schemas.syncfusion.com/wpf"

  Title="Window1" Height="300" Width="300">

    <Window.Resources>

        <DataTemplate x:Key="editableEmployee">

            <StackPanel Margin="8,0"  Orientation="Horizontal">

                <TextBlock FontWeight="Bold"
sfScroll:VisualContainer.WantsMouseInput="False" Text="{Binding
Path=Name}" Width="70" />

                <TextBox Text="{Binding Path=Title}" BorderThickness="0"
Padding="0" Margin="0" Width="130" x:Name="tb"/>

            </StackPanel>

        </DataTemplate>

        <DataTemplate x:Key="nonEditableEmployee">

            <StackPanel Orientation="Horizontal"
sfScroll:VisualContainer.WantsMouseInput="False" >

                <TextBlock FontWeight="Bold" Text="{Binding Path=Name}"
Width="70"/>

                <TextBlock Text="{Binding Path=Title}"  />

            </StackPanel>

        </DataTemplate>

    </Window.Resources>

    <Grid>
```

```
        <ScrollViewer CanContentScroll="True"
HorizontalScrollBarVisibility="Auto" VerticalScrollBarVisibility="Auto">

            <sf:GridControl x:Name="grid"/>

        </ScrollViewer>

    </Grid>

</Window>



Employees.cs

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Collections.ObjectModel;

using System.ComponentModel;



namespace DataTemplateCell

{

    public class EmployeesSource

    {

        private ObservableCollection<Employee> employees;



        public ObservableCollection<Employee> Employees
        {
            get { return employees; }
        }


        public EmployeesSource()
        {
            employees = new ObservableCollection<Employee>();
            for (int n = 0; n < 50; n++)
            {
                employees.Add(new Employee("Matt", "Program Manager"));
                employees.Add(new Employee("Joan", "Developer"));
                employees.Add(new Employee("Mark", "Programming 
Writer"));
                employees.Add(new Employee("Mary", "Test Lead"));
                employees.Add(new Employee("Karen", "Developer"));
                employees.Add(new Employee("George", "Programming 
Writer"));
                employees.Add(new Employee("Peter", "Program Manager"));
            }
        }
    }
    public class Employee : INotifyPropertyChanged

    {

        private string name;



        public string Name

        {

            get { return name; }

            set

            {

                name = value;

                OnPropertyChanged("Name");

            }

        }



        private string title;



        public string Title

        {

            get { return title; }

            set

            {
                title = value;

                OnPropertyChanged("Title");

            }

        }




        public Employee(string name, string title)

        {

            this.name = name;

            this.title = title;

        }



        public event PropertyChangedEventHandler PropertyChanged;



        private void OnPropertyChanged(string propertyName)

        {

            if (PropertyChanged != null)

            {

                PropertyChanged(this, new
PropertyChangedEventArgs(propertyName));

            }

        }
        }


}
```

CellValue, CellValueType and Format

GridStyleInfo has CellValueType and Format properties that let you format the cells value to display text with String.Format, format descriptors. If you are an Essential Grid for Windows Forms customer please note that this is the same as with Essential Grid for Windows Forms.

There are also virtual GetFormattedText and ApplyFormattedText methods in GridCellModelBase which let you adapt the behavior to the needs of your control.

CellRenderer ControlText and ControlValue

CellRenderer has a ControlText which is the text displayed in the text cell and a ControlValue which is the underlying value. The ControlValue is commited with a CommitCellInfo when the current cell is deactivated.

CurrentCell

Current Cell is the current active cell in the grid. You can navigate this cell with the arrow keys and page up / page down keys. To edit the cell simply start typing or click the mouse inside the cell.

Mouse Controllers

The grid forwards all mouse events to the MouseControllerDispatcher which has a collection of mouse controllers.

IMouseController defines the interface for mouse controllers to be used with MouseControllerDispatcher. Any mouse controller needs to implement the IMouseController interface.

In its implementation of MouseController.HitTest, the mouse controller should determine whether your controller wishes to handle mouse events based on the current context.

Selecting Cells

A simple mouse controller has been implemented to select multiple cells with the mouse.

An interesting experiment -Try checking to see how the grid can switch modes from selecting text inside a cell to selecting cells in the grid and vice versa.

Start with clicking inside a text cell, then select text in the cell, then drag the mouse outside the cell and see how multiple cells are now selected. When you drag the mouse back into the cell (while all the time holding the mouse button pressed …) you see the grid switches back to selecting text inside the cell. This is accomplished using the coordinated action of the mouse controller system.

Resizing Rows and Columns

Mouse controllers are implemented for resizing rows and columns. Implementation of similar features is straight forward and can be accomplished by implementing a mouse controller. For example, the Resize Rows mouse controller looks as follows:

```
using System;

using System.Windows;

using System.Windows.Input;

using Syncfusion.Windows.Controls.Cells;

using Syncfusion.Windows.Controls.Scroll;

using Syncfusion.Windows.Diagnostics;

using Syncfusion.Windows.GridCommon;



namespace Syncfusion.Windows.Controls.Grid

{



    class GridResizeRowsMouseController : IMouseController

    {

        ScrollAxisBase scrollRows { get { return host.ScrollRows; } }

        ScrollAxisBase scrollColumns { get { return host.ScrollColumns;
} }
```

```
        double hitTestPrecision = 4;

        GridControlBase host;

        VisibleLineInfo dragLine = null;



        public GridResizeRowsMouseController(GridControlBase grid)

        {

            this.host = grid;

        }



        private VisibleLineInfo HitTest(Point point)

        {

            return scrollRows.GetLineNearCorner(point.Y,
hitTestPrecision);

        }



        #region IMouseController Members



        public string Name

        {

            get { return "ResizeRowsMouseController"; }

        }
        public Cursor Cursor

        {

            get { return CellCursors.ResizeHeightCursor; }

        }



        public void MouseHoverEnter(MouseEventArgs e)

        {

        }



        public void MouseHover(MouseControllerEventArgs e)

        {

        }



        public void MouseHoverLeave(MouseEventArgs e)

        {

        }



        public void MouseDown(MouseControllerEventArgs e)

        {

            Point point = e.Location;

            dragLine = HitTest(point);
        }



        public void MouseMove(MouseControllerEventArgs e)

        {

            Point point = e.Location;

            VirtualizingCellsControl cellsControl = host;

            if (dragLine != null)

            {

                double delta = point.Y - dragLine.Corner;

                //Console.WriteLine(String.Format("{0} {1}", delta,
dragLine));

                host.ScrollRows.SetLineResize(dragLine.LineIndex,
Math.Max(0, dragLine.Size + delta));

            }

        }



        public void MouseUp(MouseControllerEventArgs e)

        {

            Point point = e.Location;

            double delta = point.Y - dragLine.Corner;

            host.SetRowHeight(dragLine.LineIndex, Math.Max(0,
dragLine.Size + delta));

            host.ScrollRows.ResetLineResize();
            dragLine = null;

        }



        public void CancelMode()

        {

            if (dragLine != null)

            {

                host.ScrollRows.ResetLineResize();

            }

            // Lazy way: Don't reset dragLine - then RestoreMode will
work just fine.

            //dragLine = null;

        }



        public int HitTest(MouseControllerEventArgs mouseEventArgs,
IMouseController controller)

        {

            Point point = mouseEventArgs.Location;



            RowColumnIndex pos = host.PointToCellRowColumnIndex(point,
true);

            CoveredCellInfo cc = host.GetCoveredCell(pos);

            VisibleLineInfo hit = HitTest(point);
            if (hit != null)

            {

                if (cc == null || cc.Top - 1 == hit.LineIndex ||
cc.Bottom == hit.LineIndex)

                {

                    VisibleLineInfo column =
scrollColumns.GetVisibleLineAtPoint(point.X);

                    if (column != null && column.IsHeader)

                        return 1;



                    return 0;

                }

            }

            return 0;

        }



        public bool SupportsCancelMouseCapture

        {

            get { return false; }

        }



        public bool SupportsMouseTracking
        {

            get { return false; }

        }



        public void RestoreMode()

        {

        }



        #endregion

    }



}
```

Nested Grids

Nested grids are an important component of the basic architecture of Essential Grid. They provide for the easy display of complex user interfaces using a flat grid. They also form the underpinnings for the display of hierarchical and grouped data. You can nest grids inside a row, column or covered range.

When you nest a grid inside a covered range you can specify whether the rows or columns derive their state from the parent control. You have multiple independent options for both rows and columns.

Nest grid inside a row of parent grid

In this case the grid will maintain its own row heights. When you resize rows the grid will also notify the parent grid that its total height is changed. When scrolling you can scroll row by row through the nested grid. The nested grid will have no scrollbars. They are shared with the parent grid.

The same way you can nest a grid inside a complete row you can also nest a grid inside a whole column.

Nest grid inside a covered range and tie its rows to the rows of the parent grid.

In this case the grid will have its own unique column widths but the row heights are shared with the parent grid. When scrolling through rows in the nested grid you also scroll the rows in the parent grid to keep them in sync. The nested grid will have no scrollbars. They are shared with the parent grid. When you resize rows they will also be resized in the parent grid and vice versa.

Nest grid inside a covered range and tie its columns to the columns of the parent grid.

In this case the grid will have its own unique row height but the column widths are shared with the parent grid. When scrolling through columns in the nested grid you also scroll the columns in the parent grid to keep them in sync. The nested grid will have no scrollbars. They are shared with the parent grid. When you resize columns they will also be resized in parent grid and vice versa.

Nest grid inside a covered range with its rows and columns independent of parent grid.

In this case the nested grid maintains its own row heights and column widths. You can scroll through this grid without scrolling the parent grid. Resizing rows and columns will also not affect the parent grid.

Please refer to the NestedGrids example for additional information.

Performance optimization notes - TraderGridTest sample.

This sample shows a grid bound to a flat DataView. It has a header row with field names and a footer rows with summaries.

Inside the sample code(SampleGridControl.cs) you can define various test scenarios.

```
        // adjust settings below to modify scenario

        bool sortByThirdColumn = false;

        bool insertAndRemoveRecords = false;

        bool insertAndRemoveColumns = false;

        TimeSpan timerInterval = new TimeSpan(0, 0, 0, 0, 50);

        int numberOfChangesEachTimer = 200;

        int blinkTime = 700;
```

The basic test will load a DataTable with values and then modify the records inside a timer. This will raise ListChanged events that the FlatDataViewGridControl listens to. The control the updates the display and will highlight cells that were changed and also update the resulting summaries.

The FlatDataViewGridControl is implemented using a Virtual Grid approach wiring the grid to the DataView with a QueryCellInfo and CommitCellInfo event handler.

Blinking behavior is implemented by handling the PrepareRenderCellInfo event.

The grid handles all ListChanged notifications including ItemAdded, ItemMoved, ItemDeleted, Reset, PropertyDescriptorAdded, PropertyDescriptorDeleted and PropertyDescriptorChanged.

When the current cell is active, you can keep typing while rows are re-arranged and modify the contents of the currently active cell. This is often a requirement in applications that wish to maintain editing compatibility even when processing a large number of updates.