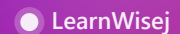


DataGridView Cell Painting

Lesson Guide · Lab / Exam Guide



Lesson Guide

Painting inside the grid: UserPaint at cell or column scope, DataGridViewCellPaintEventArgs with RowIndex and ColumnIndex, progress and sparkline cells that read rows cheaply, and the honest comparison against AllowHtml cell content on a grid with a thousand rows. What this means for an experienced .NET developer who already builds Wisej.NET screens and now needs custom graphics — and how to read this module.



Production focus

Painting inside the grid: UserPaint at cell or column scope, DataGridViewCellPaintEventArgs with RowIndex and ColumnIndex, progress and sparkline cells that read rows cheaply, and the honest comparison against AllowHtml cell content on a grid with a thousand rows.

1. The production problem

Module 2 gave VisualOperationsStudio a `TelemetryGauge`: one control, one machine, a few hundred repaints an hour. The operations grid is a different animal. A shift supervisor opens a list of a thousand machines and wants to see at a glance which ones are near their limit and which have been drifting all morning. Numbers in a column will not do that, and one gauge per row would be a thousand server-side controls in a single session. The grid itself has to draw, once per visible cell, and stop the moment the cell scrolls away.

2. Key concepts

Concept	Why it matters
<code>UserPaint</code>	The explicit switch: <code>DataGridViewCell.UserPaint = true</code> for one cell, <code>DataGridViewColumn.UserPaint = true</code> for a whole column. Without it the grid never asks you to paint.
<code>CellPaint</code>	The grid-level event raised for user-painted cells when they need to be drawn; you subscribe once on the grid, not once per row, and route inside the handler.
<code>DataGridViewCellPaintEventArgs</code>	Derives from <code>PaintEventArgs</code> , so it carries <code>Graphics</code> and <code>ClipRectangle</code> , and adds <code>RowIndex</code> and <code>ColumnIndex</code> so one handler can serve several columns.
no WinForms helpers	The documented Wisej.NET event arguments do not expose the WinForms convenience methods <code>PaintBackground()</code> and <code>PaintContent()</code> . Your handler owns the whole visual result of that cell.
<code>e.ClipRectangle</code> as the surface	The cell's local destination rectangle. Columns can be resized and the grid can be laid out responsively, so every coordinate you draw has to be derived from it rather than from a constant.
pure render helpers	Methods shaped like <code>DrawProgressCell(Graphics g, Rectangle rect, double value)</code> , so the handler becomes routing code and the same helper can later render into an off-screen bitmap.
<code>AllowHtml</code> cell content	The browser-native alternative: icons, spans, labels and CSS rendered by the client, and the reason the documentation reserves <code>CellPaint</code> for visuals too complex for HTML.

3. Lesson narrative

Turning a cell into a painting surface

Nothing about a Wisej.NET `DataGridView` is user-painted by default. You opt in per cell or per column by setting `UserPaint` to `true`, and only then does the grid raise `CellPaint` for that surface. Forgetting the switch is the commonest reason a correct-looking handler never runs, so keep the enabling line and the subscription line together in your column-configuration method.

Prefer column scope. In the lab you add explicit columns to `gridOperations` and set `UserPaint = true` on the Health and Trend columns only, then subscribe to `CellPaint` once on the grid. That gives you one subscription for the whole screen instead of one per row, and it keeps every other column — identifier, machine name, site, last reading — as ordinary grid content that the browser can select, search and scale. Enabling user painting for the entire grid is the opposite decision: it hands you cells you never meant to draw and asks the handler to filter them out on every repaint.

What the event hands you, and what it does not

Because `DataGridViewCellPaintEventArgs` derives from `PaintEventArgs`, the drawing part is what Module 2 taught: `e.Graphics` is the surface, `e.ClipRectangle` is your area, and you dispose every pen, brush and path you create while leaving `e.Graphics` alone. What is new is routing. `e.RowIndex` and `e.ColumnIndex` both need a guard before you touch anything: a negative `RowIndex` is not a data row, and a column you did not opt into should return immediately.

The compact part of the API is the part that catches people out. Wisej's event arguments do not expose `PaintBackground()` or `PaintContent()`, so a WinForms sample that calls them has to be rewritten rather than compiled, and you decide for yourself what the cell's background and content look like. Treat that as a feature: when the handler produces the entire cell, there is no half-drawn state to reason about.

```
private void gridOperations_CellPaint(object sender, DataGridViewCellPaintEventArgs e)
{
    if (e.RowIndex < 0)
        return;

    var rect = Rectangle.Inflate(e.ClipRectangle, -4, -5);

    if (e.ColumnIndex == healthColumn.Index)
        DrawHealthBar(e.Graphics, rect, Convert.ToInt32(
            gridOperations.GetValue(e.ColumnIndex, e.RowIndex) ?? 0));
    else if (e.ColumnIndex == trendColumn.Index)
        DrawSparkline(e.Graphics, rect, TrendOf(e.RowIndex));
}
```

Progress, status and sparkline cells

Three shapes cover most operational grids. A progress cell fills an inactive track with an active rectangle whose width is proportional to a normalised value, and draws a number only when the column is wide enough. A status cell is a single filled ellipse, sometimes with a short label; it is the cheapest useful visual in the grid and often clearer than an icon. A sparkline normalises a small series to the cell rectangle and connects the points with `DrawLines`, as the Trend column does in the lab: its job is to summarise, never to reproduce a ten-thousand-point series inside forty pixels.

Keep the maths out of the handler. Give each visual a helper that takes a `Graphics`, a `Rectangle` and the values it needs, and the handler shrinks to a guard and two calls. The Module 6 off-screen renderer can then call the same method to put the identical sparkline into an exported PNG. Do not forget the states the grid owns: a user-painted cell still has to be readable when its row is selected, so either pick selection-aware colours or keep the visual restrained and let the row highlight speak. Never let colour be the only carrier of a critical status; the same value should also exist as text or a tooltip.

When painting is the wrong tool

Wisej.NET also renders HTML inside cells through `AllowHtml`, and the documentation positions `CellPaint` as the answer for content too complex for HTML or for existing .NET drawing code you want to reuse. Choose HTML when the visual can be expressed with text, icons, spans and CSS, when search, selection, accessibility or browser scaling benefit from real DOM content, or when there are many

thousands of cells. Choose painting when you need pixel-level geometry, when the visual is compact and changes rarely, or when the HTML would end up more complicated than the drawing code.

A second alternative has nothing to do with HTML: the grid's own cell types and its ability to host controls. A button, a checkbox, a date picker or a combo box should stay a real interactive cell. Painting is a rendering model, not an interaction model, and a drawn button forces you to rebuild pointer feedback, keyboard access, focus and commands by hand. That is why the lab builds the status indicator twice and asks you to write down which one you would ship.

Scrolling a thousand rows

A `DataGridView` handles large data sets well with binding and virtual modes, and a careless `CellPaint` handler undoes that advantage in one line. Scrolling calls the handler again for every newly visible cell, so anything expensive is multiplied by how fast the user drags the scrollbar.

- Never perform I/O, database or service calls inside the handler.
- Do not load images or create fonts per cell; build them once and reuse them.
- Paint the smallest possible set of columns, and reduce sparkline point counts.
- Keep state retrieval $O(1)$: index the list you bound, or read the cell value through the grid, but never search a collection per cell.
- Measure with realistic row counts — a hundred rows proves nothing about a thousand.

The lab's thousand generated rows exist for that last point. Bind them, drag the scrollbar from top to bottom, and compare what the two painted columns cost against the HTML column beside them. That measurement, not a preference, belongs in your notes.



Surface-first drawing habit

Choose the rendering surface before you write a single drawing call. Ask where the pixels are produced and what crosses the network: a control Paint handler runs on the server and ships an image, `CellPaint` does the same inside one grid cell, Canvas sends drawing commands the browser executes and keeps nothing across a resize, and `System.Drawing.Managed` produces image bytes with no control at all. Keep the semantic model in C# objects, make every renderer a deterministic function of that model, invalidate only when a visual dependency changed, and always leave a textual equivalent of anything important that exists only as pixels.



Learning objectives

By the end of this module you'll be able to:

- ✓ Enable painting at cell or column scope with `UserPaint`.
- ✓ Use `DataGridViewCellPaintEventArgs` safely.
- ✓ Build progress, status and sparkline cells.
- ✓ Compare user painting with Cell HTML, built-in cell types, cell controls and images.
- ✓ Design cell painting for large data sets without turning scrolling into a rendering bottleneck.

Lab / Exam Guide

What you'll build in the hands-on lab, the suggested approach, and the required deliverables.



Hands-on lab

Give VisualOperationsStudio an operations grid with two user-painted columns. Add explicit columns to `gridOperations`, set `UserPaint = true` on the Health and Trend columns only, and handle `CellPaint` once: return immediately when `e.RowIndex` is negative or `e.ColumnIndex` is not one of the two targets, then draw a threshold-coloured progress bar for Health and a polyline sparkline for Trend inside `e.ClipRectangle`. Read each row's values from the bound `MachineStatus` object rather than searching a list inside the handler, respect the selected row's colours, and keep the handler free of side effects. Bind 1,000 generated rows, scroll, and then build the same status indicator a second time as an `AllowHtml` cell so you can state in writing which one you would ship.

Starting point

You open VisualOperationsStudio as Module 2 left it: the `VisualOperationsPage` carries the `TelemetryGauge` control and its `GaugeGeometry` helper, and the page has room for a grid but no data yet. Add a `MachineStatus` class with an identifier, a name, a site, a `Health` percentage and a short list of recent readings for the trend, plus a generator that can produce a thousand of them.

Suggested approach

- ✓ Add `gridOperations` with explicit columns for identifier, machine, site, Health and Trend, bind ten `MachineStatus` rows first, and confirm the plain columns render before any painting exists.
- ✓ Set `UserPaint = true` on the Health and Trend columns only, subscribe to `gridOperations.CellPaint` once, and start the handler with a guard that returns when `e.RowIndex` is negative or `e.ColumnIndex` is neither target column.
- ✓ Write `DrawHealthBar(Graphics, Rectangle, int)` and `DrawSparkline(Graphics, Rectangle, IReadOnlyList<double>)` as pure helpers that derive every coordinate from the rectangle they are given, choose the fill colour from the warning and critical thresholds, and dispose each pen and brush they create.
- ✓ Make the handler pure routing: shrink `e.ClipRectangle` with `Rectangle.Inflate`, fetch the row's values in one $O(1)$ step, pass them to the helper, and keep the selected row readable by choosing colours that still contrast with the selection background.
- ✓ Bind 1,000 generated rows, drag the scrollbar from top to bottom, then add a third status column that uses `AllowHtml` to render the same warning or critical state, scroll again and record both results.

Required deliverables

- ✓ `gridOperations` with explicit columns and `UserPaint = true` on only the Health and Trend columns
- ✓ One `CellPaint` handler that guards `e.RowIndex` and `e.ColumnIndex` before drawing anything
- ✓ Threshold-coloured progress cell and a polyline sparkline cell drawn inside `e.ClipRectangle`

- ✓ Row values read from the bound MachineStatus object with no per-cell lookup, allocation or control creation
- ✓ An AllowHtml status column built alongside, plus a note comparing both on a 1,000-row scroll

Student review questions

- ✓ A colleague pastes a WinForms cell-painting sample that starts by calling `e.PaintBackground( ... )`. Explain what the Wisej.NET event arguments give you instead, and what the handler now has to decide for itself.
- ✓ Your sparkline helper hard-codes a 100-pixel plot width and looks fine on your screen. Describe two user actions that break it, and what the helper should read instead.
- ✓ The HTML status column and the painted Health column both show a critical machine. Argue which one you would ship for a grid that support staff search and read with a screen reader, and what you would lose either way.



Surface-first drawing habit

Choose the rendering surface before you write a single drawing call. Ask where the pixels are produced and what crosses the network: a control Paint handler runs on the server and ships an image, CellPaint does the same inside one grid cell, Canvas sends drawing commands the browser executes and keeps nothing across a resize, and System.Drawing.Managed produces image bytes with no control at all. Keep the semantic model in C# objects, make every renderer a deterministic function of that model, invalidate only when a visual dependency changed, and always leave a textual equivalent of anything important that exists only as pixels.