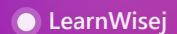


The Wisej.NET Drawing Architecture

Lesson Guide · Lab / Exam Guide



Lesson Guide

Four rendering pipelines and how to choose between them: control Paint that ships a server-rendered image, DataGridView CellPaint inside one cell, Wisej.Web.Canvas that sends browser drawing commands, and off-screen System.Drawing.Managed that produces image bytes — plus the decision ladder that keeps standard controls and CSS as the default. 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

Four rendering pipelines and how to choose between them: control Paint that ships a server-rendered image, DataGridView CellPaint inside one cell, Wisej.Web.Canvas that sends browser drawing commands, and off-screen System.Drawing.Managed that produces image bytes — plus the decision ladder that keeps standard controls and CSS as the default.

1. The production problem

VisualOperationsStudio has to show the same machine reading in four places: a dial on the operations page, a health bar inside a grid row, a node on a topology map, and a PNG attached to the nightly shift report. Each one is "draw a number", yet reaching for whichever drawing API you remember from the desktop produces four pieces of code with wildly different CPU, bandwidth and resize behaviour. This module puts the choice of surface before the drawing code, because the surface decides where the pixels are produced and what travels to the browser.

2. Key concepts

Concept	Why it matters
<code>Paint</code> / <code>PaintEventArgs</code>	The control-level painting event: your C# runs on the server against the <code>Graphics</code> it supplies, inside the <code>ClipRectangle</code> it reports, and Wisej.NET turns the result into an image for the browser.
<code>DataGridView.CellPaint</code>	The same painting mechanism scoped to one cell; its <code>DataGridViewCellPaintEventArgs</code> derives from <code>PaintEventArgs</code> and adds <code>RowIndex</code> and <code>ColumnIndex</code> .
<code>UserPaint</code>	The switch a cell or a whole column needs before the grid raises <code>CellPaint</code> for it, which keeps painting opt-in per column instead of grid-wide.
<code>Wisej.Web.Canvas</code>	An immediate-mode 2D browser surface mirroring the HTML5 <code>CanvasRenderingContext2D</code> model; the drawing calls are executed by the client, not rendered into a picture on the server.
<code>Redraw</code>	The Canvas event in which the whole scene is reconstructed from your model, because the canvas keeps pixels and loses them when it is recreated or resized.
<code>System.Drawing.Managed</code>	Off-screen rendering with no visible control: create a <code>Bitmap</code> , get its <code>Graphics</code> from <code>Graphics.FromImage()</code> , and keep the image bytes for a report, an export or a download.
<code>Invalidate()</code>	The call that tells a painted control its visual is stale; it updates the client widget and the Visual Studio designer surface, so it belongs where state changed, not inside drawing helpers.

3. Lesson narrative

Four pipelines that all look like ordinary C#

The first pipeline is the control `Paint` event, which every Wisej.NET control supports. The handler receives a `Wisej.Web.PaintEventArgs` whose `Graphics` property is a `System.Drawing.Graphics` surface and whose `ClipRectangle` identifies the rectangle being painted. Wisej.NET renders the painted PNG background and sends it to the browser; the custom-painting documentation describes the result as a PNG that is transferred and then removed from memory. That is what makes an old GDI+-style routine unusually reusable in a web application.

The second pipeline is `DataGridView.CellPaint`, specialised for cells. A cell or an entire column must have `UserPaint` set to `true`; the grid then raises `CellPaint` with `DataGridViewCellPaintEventArgs`, which carries the familiar `Graphics` and `ClipRectangle` plus `RowIndex` and `ColumnIndex`. It suits compact visuals — status gauges, tiny bar charts, sparklines, badges — and routines that would be awkward to reproduce as markup.

The third pipeline is `Wisej.Web.Canvas`, an immediate-mode browser canvas whose API deliberately resembles the HTML5 `CanvasRenderingContext2D` model. The fourth has no control in it at all: create a

`Bitmap`, obtain a `Graphics` through `Graphics.FromImage()`, draw, and save or send the image. Wisej.NET 4 ships `System.Drawing.Managed`, which implements the familiar `System.Drawing` namespace in managed code on modern .NET targets. In this module's lab all four render the single `TelemetrySample` the page holds.

Where the code runs and what crosses the wire

A server-painted control does not send your `DrawLine()` call to the browser. The server executes the drawing against a graphics surface and Wisej.NET transfers the finished image — convenient, because the browser never has to understand your algorithm, and expensive when the surface is large and changes often, since server CPU and bytes are paid on every repaint. Measure that before letting a painted control repaint on a timer.

Canvas inverts this. The Wisej `Canvas` API represents browser 2D canvas operations, and those operations are executed at the client. `LiveUpdate` is `false` by default, so commands issued during a request travel with the normal Wisej.NET update; setting it to `true` pushes them immediately, which suits incremental demonstrations but is no substitute for client-side animation — a server round trip per frame is the wrong architecture. Off-screen rendering crosses nothing until you hand the bytes over, and `CellPaint` uses the same image mechanism as `Paint`, bounded by one cell.

Retained model, immediate pixels

"Immediate mode" is the part that catches people out. A Canvas remembers pixels, not your business objects, and Wisej.NET does not replay every line and rectangle after a refresh. Keep the model — shapes, points, the selected item, zoom, pan, colours — in ordinary C# objects, and make `Redraw` deterministic: given that model and the current canvas size, it reconstructs the complete visual. In the lab you resize the browser and write down which surfaces came back on their own.

Paint handlers deserve the same discipline. Treat them as render functions: observe state, draw it, change nothing. If the reading changes, update the model and invalidate the control; never adjust the reading because a paint was requested. Rendering code that mutates state is why a visual ends up depending on how many times it was repainted.

Invalidate() at the boundary of a state change

`Control.Invalidate()` makes the control update its client widget, and it also updates the full surface in the Visual Studio designer. In a user-painted control it is the natural call when a property affecting the visual changes, and its natural home is the property setter, guarded so an unchanged value costs nothing:

```
private double _reading;

public double Reading
{
    get => _reading;
    set
    {
        value = Math.Clamp(value, 0, 100);
        if (_reading == value)
            return;

        _reading = value;
        this.Invalidate();
    }
}
```

The drawing code can now assume the value is valid and simply render the current state. Keep invalidation at those boundaries rather than sprinkling it through low-level helpers, where it turns into repaint storms nobody can trace. One design constraint to plan around: the custom-painting guide is explicit that `BackgroundImage` and custom painting cannot be combined on the same control, because painting takes the background over. For an image behind your drawing, draw it yourself with `Graphics.DrawImage()`.

The ladder that keeps custom painting last

Custom graphics should be the answer to a requirement, never a reflex carried over from desktop work. Walk the ladder in order:

- Can a standard control, theme, icon, image, progress bar, chart extension or CSS already do it? Prefer that.
- Is the content naturally HTML and readable as DOM content? Consider markup before pixels.
- Is it a small visual belonging to a control's background that changes at human speed? `Paint`.
- Is it a compact visual inside a grid cell? `CellPaint`, after comparing it with cell HTML.
- Is it an interactive or browser-native scene? `Canvas`. Is the output a file rather than a screen? Off-screen `System.Drawing.Managed`. Is it frame-rate sensitive? Keep the loop on the client.

Two checks close the decision. Pixels carry no meaning for assistive technology, so anything important that exists only as a drawing needs a parallel textual channel — a label, a tooltip, `AccessibleName` or `AccessibleDescription`. And network transfer plus server CPU belong in the drawing budget, exactly as allocations belong in a hot loop's budget. A painted gauge that wins on looks and loses on both is a decision you should be able to defend in review.



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:

- ✓ Distinguish control Paint, DataGridView.CellPaint, Wisej.Web.Canvas and off-screen System.Drawing.Managed rendering.
- ✓ Explain the server/browser boundary for each drawing technique.
- ✓ Understand when a repaint sends an image and when Canvas sends browser drawing commands.
- ✓ Use Invalidate() deliberately when visual state changes.
- ✓ Identify cases where HTML, CSS, a standard Wisej.NET control or an extension is a better solution than custom painting.

Lab / Exam Guide

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



Hands-on lab

Build the VisualOperationsStudio shell and render one telemetry reading on four surfaces at once. Create a VisualOperationsPage holding a single TelemetrySample model instance, then present its value as (1) a plain Label plus ProgressBar, (2) a Panel with a Paint handler that draws a bar from PaintEventArgs.Graphics and e.ClipRectangle, (3) a Wisej.Web.Canvas whose Redraw handler rebuilds the same bar from the model, and (4) an off-screen Bitmap rendered with Graphics.FromImage and shown in a PictureBox. Wire a button that changes the model value and refreshes all four surfaces through Invalidate() and a Canvas redraw, then resize the browser and record which surfaces survive and which have to be reconstructed.

Starting point

Create a new Wisej.NET 4.x project called VisualOperationsStudio, or open the empty shell if your instructor supplied one: a start page, a default theme, and nothing drawn by hand anywhere. Everything in this lab is added by you, and the four surfaces you build here are the ones every later module extends into a real gauge, a painted grid, a topology editor and a PNG export.

Suggested approach

- ✓ Add a `VisualOperationsPage` and a small `TelemetrySample` class holding the machine name, a reading between 0 and 100 and a timestamp. Create exactly one instance as a field on the page; every surface must read that instance and none may keep its own copy.
- ✓ Lay out four labelled areas on the page. Fill the first with a `Label` and a `ProgressBar` bound to the reading, so you have a no-pixels baseline to compare the other three against.
- ✓ Drop a `Panel` in the second area and handle its `Paint` event. Draw the bar from `e.Graphics`, size it inside `e.ClipRectangle`, and dispose the pens and brushes you create. Read the reading from the model and change nothing in the handler.
- ✓ Put a `Wisej.Web.Canvas` in the third area and move all of its drawing into the `Redraw` handler, so the scene is rebuilt from the model rather than accumulated. In the fourth area render an off-screen `Bitmap` with `Graphics.FromImage` and show it in a `PictureBox`.
- ✓ Add a button that assigns a new value to the model and then refreshes every surface: update the label and progress bar, call `Invalidate()` on the painted panel, redraw the canvas, and rebuild the off-screen image inside `try / catch`. Then resize the browser window and write your notes.

Required deliverables

- ✓ VisualOperationsPage holding one TelemetrySample model that every surface reads from
- ✓ Paint handler drawing from PaintEventArgs.Graphics inside e.ClipRectangle
- ✓ Canvas Redraw handler that rebuilds the whole scene from the model after a resize
- ✓ Off-screen Bitmap rendered with Graphics.FromImage and displayed in the page

- ✓ SurfaceNotes.md recording what crosses the network per surface and which visuals survive a resize

Student review questions

- ✓ You set the model's reading from the button but forget the `Invalidate()` call on the painted panel. The label and the canvas update. Explain exactly why the panel does not, and what the user is looking at until something else forces it to repaint.
- ✓ Your first Canvas version draws the bar once when the page loads instead of from `Redraw`. Describe what the user sees after resizing the browser, and why the off-screen `Bitmap` in the fourth area behaves differently.
- ✓ Your reviewer asks why the first area — the plain `Label` and `ProgressBar` — is in the lab at all. Answer with the costs the other three surfaces pay that it does not, and name one requirement that would justify paying them.



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.