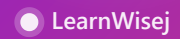


Interactive Canvas: State, Input, Hit Testing and Performance

Lesson Guide · Lab / Exam Guide



Lesson Guide

Turning a Canvas into an editor: the Model, Input, State, Render loop, pointer and touch events from Control, screen-to-world conversion so hit testing survives pan and zoom, per-session scene state instead of statics, viewport culling, and the point where a client-side loop replaces server round trips. 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

Turning a Canvas into an editor: the Model, Input, State, Render loop, pointer and touch events from Control, screen-to-world conversion so hit testing survives pan and zoom, per-session scene state instead of statics, viewport culling, and the point where a client-side loop replaces server round trips.

1. The production problem

Module 4 left VisualOperationsStudio with a Canvas playground: a `DrawPlayground` method that the `Redraw` event calls, rebuilding every path, gradient and transform from scratch. It is correct, and it is completely inert. Operations staff want to click a pump, drag it somewhere sensible, pan across the plant and zoom into a cluster — and a drawing method cannot answer any of those requests, because the browser canvas holds pixels and pixels do not know which shape is a pump. The moment a surface becomes interactive it stops being a renderer and becomes a small state machine that happens to render.

2. Key concepts

Concept	Why it matters
Model, Input, State, Render	Four layers with one rule between them: input changes state, and only then does a single render method turn the current state into drawing calls.
Scene objects	C# classes such as <code>NodeModel</code> and <code>EdgeModel</code> that carry the stable ID, world-space bounds, status and <code>Selected</code> flag the pixels can only suggest.
Per-session scene state	The scene belongs to one user's session; a mutable <code>static</code> list would give every visitor the same nodes and the same selection.
Control input events	<code>Canvas</code> derives from <code>Control</code> , so mouse down/move/up, click, double click and the touch and gesture events such as <code>TouchStart</code> , <code>Tap</code> , <code>Pinch</code> , <code>Rotate</code> , <code>Swipe</code> and <code>Track</code> are all available.
Model-based hit testing	The documented Wisej Canvas API exposes no <code>isPointInPath</code> or <code>isPointInStroke</code> , so "what did the user click?" is answered by comparing a point against geometry you own.
Screen-to-world conversion	If rendering applies $screen = world \times zoom + pan$, then every incoming pointer coordinate must be run back through the inverse before it is compared with anything in the model.
Viewport culling	Skipping scene objects whose bounds do not intersect the visible world rectangle, so a large topology costs only what is actually on screen.

3. Lesson narrative

Four layers and a single render method

Split the editor into a model layer (domain and scene objects), an input layer (mouse, pointer, touch, keyboard and commands), a state-transition layer (selection, drag, pan, zoom, edits) and a render layer. The render layer is deliberately dull: it is a deterministic function of the current scene, and it decides nothing. It does not work out which pump is selected; it reads `node.Selected` and picks a fill accordingly. That constraint is what makes the surface survive a resize, because `Redraw` can call the same method at any moment and get the same picture back.

The opposite temptation is strong and always loses. A `MouseMove` handler that draws a highlight rectangle directly is one resize, one scroll or one theme change away from a scene that no longer matches the application's own idea of the world. In VisualOperationsStudio the scene lives in a `TopologyScene` object held by the page instance, so each session edits its own topology, and the lab's `RenderScene` is the only method in the file that is allowed to touch the canvas.

Input from a Control, not from a canvas

Because `Wisej.Web.Canvas` inherits from `Control`, input arrives the way it does anywhere else in Wisej.NET. `MouseDown` selects and starts a gesture, `MouseMove` moves the selected node or the viewport, and `MouseUp` ends the gesture by clearing the drag state. Every one of those handlers follows the same two-line shape: change state, then call the render method. Nothing draws inside an event handler.

Be honest about what that costs. These are server-side events, so each one is a request, and a drag at human speed produces a stream of them. That is acceptable for selecting and repositioning a few dozen nodes; it is not acceptable for freehand ink or game-like motion. Plan for touch as well: the gesture events exist, but hover-only affordances do not, so give handles enough size to hit with a thumb and never hide an essential action behind a hover state.

Hit testing against geometry you own

Hit testing is a model question. Walk the node list backwards — later items were drawn on top, so reverse iteration matches z-order — and return the first node whose bounds contain the point. Rectangles use `Contains`; circles compare squared distance against the squared radius, which avoids a square root per node; line-shaped edges need point-to-segment distance with a tolerance, because nobody clicks exactly on a one-pixel line. For genuinely complex paths, keep a simplified hit region in the model rather than trying to reason backwards from pixels.

Keep the same discipline for keyboard users. Canvas nodes are not native DOM controls, so nothing in the browser will focus them for you. Offer commands that select the next node and nudge the selected one, and keep a textual list of the same nodes beside the surface. Both views read and write the one scene, so selection stays consistent whichever route the user took.

Pan, zoom and the bug that looks like a selection bug

Store pan and zoom as scene state and apply them once, at the top of the render, bracketed by `Save()` and `Restore()` so nothing leaks into the next frame. Input then arrives in the wrong coordinate system and has to be converted back before it meets the model.

```
private PointF ToWorld(Point screen)
{
    return new PointF((screen.X - _scene.PanX) / _scene.Zoom,
                      (screen.Y - _scene.PanY) / _scene.Zoom);
}

private NodeModel HitTest(Point screen)
{
    PointF p = ToWorld(screen);
    for (int i = _scene.Nodes.Count - 1; i ≥ 0; i--)
        if (_scene.Nodes[i].Bounds.Contains(p))
            return _scene.Nodes[i];
    return null;
}
```

Skip that conversion and the editor behaves perfectly at zoom 1.0 with no pan, then starts missing nodes the moment either changes — a defect reported as "selection is broken after zooming" that is really a coordinate-system mistake. Convert in one place, use it for hit testing and for the drag offset, and add `ToScreen` for placing a world position against a fixed overlay.

A redraw budget, and the point where the client takes over

For moderate scenes, prefer a full deterministic redraw. Incremental drawing looks cheaper and brings dirty regions, z-order repair, erase artefacts and state drift with it. When the full redraw does become

expensive, work down a fixed ladder before changing the architecture:

- Reduce the number of scene primitives, then cull anything outside the visible world rectangle.
- Simplify geometry at low zoom, and cache static layers as images.
- Separate static and dynamic surfaces if the layout allows it.
- Only then move truly high-frequency behaviour to the client.

Leave `LiveUpdate` off for the main render. It pushes drawing calls immediately, which is useful for showing progress through a long build, but each push has a cost and a single render performed as part of an input request already updates the scene in one response. The same reasoning kills the server timer that redraws thirty or sixty times a second: it burns server capacity per session and every frame crosses the network. When a visual is an animation rather than a data change, send the new target value from C# and let a client script animate towards it, keeping the server authoritative for the semantic number. Write down, from the lab's own measurements, where your cutoff sits.



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:

- ✓ Implement the Model, Input, State, Render pattern.
- ✓ Use Wisej control pointer/mouse/touch events with a Canvas surface.
- ✓ Implement server-side hit testing for moderate-frequency interaction.
- ✓ Decide when to move interaction or animation logic to client JavaScript.
- ✓ Use LiveUpdate, invalidation and redraw without excessive traffic.

Lab / Exam Guide

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



Hands-on lab

Turn the Canvas playground into a topology editor for VisualOperationsStudio. Create `NodeModel` and `EdgeModel` classes with stable IDs and world-space geometry, hold the scene in a `TopologyScene` object stored per session rather than in a static field, and keep `RenderScene` as the only method that draws. Add `ToWorld` and `ToScreen` conversions for the current pan offset and zoom factor, then handle `MouseDown` to hit-test node bounds in world coordinates and set `Selected`, `MouseMove` to drag the selected node or pan the viewport, and `MouseUp` to end the gesture — each of them changing state and then calling `RenderScene`. Add wheel or button zoom about the pointer, skip nodes outside the visible rectangle when rendering, give keyboard users a way to select and nudge a node, and record the request traffic a drag produces plus the point at which you would move the loop to client JavaScript.

Starting point

VisualOperationsStudio after Module 4: the Canvas playground page draws its whole scene from a `DrawPlayground` method called by the `Redraw` event, with `Save()` / `Restore()` around each transformed section, but every shape is hard-coded in screen coordinates and nothing on the surface can be selected, moved or panned.

Suggested approach

- ✓ Write `NodeModel` and `EdgeModel` with a stable `Id`, world-space `Bounds`, a label and a `Selected` flag, gather them in a `TopologyScene` that also carries `PanX`, `PanY` and `Zoom`, and hold that scene on the page instance so one session's edits never reach another.
- ✓ Replace `DrawPlayground` with a `RenderScene` method that clears the surface, calls `Save()`, applies `Translate` and `Scale` from the scene, draws edges then nodes, and calls `Restore()`; point the `Redraw` handler at it and confirm a browser resize brings the same picture back.
- ✓ Add `ToWorld` and `ToScreen`, then a `HitTest` that converts the pointer position once and walks the node list backwards so the node drawn last wins, returning `null` for empty space.
- ✓ Wire `MouseDown`, `MouseMove` and `MouseUp` so each one only changes scene state — selection, the drag offset in world units, the pan origin — and then calls `RenderScene`; add zoom about the pointer and check selection still lands on the right node afterwards.
- ✓ Compute the visible world rectangle once per render and skip anything that does not intersect it, add keyboard commands that select the next node and nudge the selected one, then count the requests one two-second drag produces and write the numbers down.

Required deliverables

- ✓ `NodeModel`, `EdgeModel` and a per-session `TopologyScene` with stable IDs and no mutable static state
- ✓ `ToWorld` and `ToScreen` conversions used for hit testing so selection stays correct after pan and zoom

- ✓ MouseDown, MouseMove and MouseUp handlers that change state and then call the single RenderScene method
- ✓ Viewport culling that skips nodes outside the visible rectangle, plus a keyboard path for selecting and nudging
- ✓ InteractionNotes.md with the request traffic measured during a drag and the client-side cutoff you would apply

Student review questions

- ✓ A colleague reports that selection is fine until they zoom in, after which clicks land on the wrong pump or on nothing. Nothing about the drawing code changed. Which single conversion is missing, and why does the defect stay invisible at zoom 1.0 with no pan?
- ✓ Two operators open the topology at the same time and each sees the other's node move under their cursor. What did the scene get stored in, and what does that tell you about where interactive state belongs in a server-side web framework?
- ✓ Your drag produces a request for every pointer move. At what measured point would you stop optimising the server loop and move the interaction into client JavaScript instead, and what would still have to stay authoritative on the server?



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.