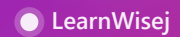


# Interaktives Canvas: Zustand, Eingaben, Trefferprüfung und Performance

Lektionsleitfaden · Lab- und Prüfungsleitfaden



## Lektionsleitfaden

Eine Canvas in einen Editor verwandeln: die Schleife aus Modell, Eingabe, Zustand und Rendering, Pointer- und Touch-Events aus Control, die Umrechnung von Bildschirm- in Weltkoordinaten, damit die Trefferprüfung Verschieben und Zoomen übersteht, Szenenzustand pro Sitzung statt in statischen Feldern, Viewport-Culling und der Punkt, an dem eine Schleife im Client die Server-Roundtrips ablöst. Was das für erfahrene .NET-Entwickler bedeutet, die bereits Wisej.NET-Oberflächen bauen und nun eigene Grafiken brauchen – und wie Sie dieses Modul lesen.



### Fokus: Produktivbetrieb

Eine Canvas in einen Editor verwandeln: die Schleife aus Modell, Eingabe, Zustand und Rendering, Pointer- und Touch-Events aus Control, die Umrechnung von Bildschirm- in Weltkoordinaten, damit die Trefferprüfung Verschieben und Zoomen übersteht, Szenenzustand pro Sitzung statt in statischen Feldern, Viewport-Culling und der Punkt, an dem eine Schleife im Client die Server-Roundtrips ablöst.

## 1. Das Problem im Produktivbetrieb

Modul 4 hat VisualOperationsStudio mit einer Canvas-Spielwiese zurückgelassen: einer Methode `DrawPlayground`, die das `Redraw`-Event aufruft und jeden Pfad, jeden Farbverlauf und jede Transformation von Grund auf neu aufbaut. Sie ist korrekt – und völlig regungslos. Die Betriebsmannschaft möchte auf eine Pumpe klicken, sie an eine sinnvolle Stelle ziehen, über die Anlage schwenken und in eine Gruppe hineinzoomen — und eine Zeichenmethode kann keine dieser Anfragen beantworten, denn die Browser-Canvas hält Pixel, und Pixel wissen nicht, welche Form eine Pumpe ist. In dem Moment, in dem eine Oberfläche interaktiv wird, hört sie auf, ein Renderer zu sein, und wird zu einer kleinen Zustandsmaschine, die nebenbei rendert.

## 2. Zentrale Konzepte

| Konzept                             | Warum es wichtig ist                                                                                                                                                                                                                                                                                        |
|-------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Modell, Eingabe, Zustand, Rendering | Vier Schichten mit einer Regel dazwischen: Eingaben ändern den Zustand, und erst danach macht eine einzige Render-Methode aus dem aktuellen Zustand Zeichenaufrufe.                                                                                                                                         |
| Szenenobjekte                       | C#-Klassen wie <code>NodeModel</code> und <code>EdgeModel</code> , die stabile ID, Grenzen in Weltkoordinaten, Status und das <code>Selected</code> -Kennzeichen tragen – all das, was Pixel nur andeuten können.                                                                                           |
| Szenenzustand pro Sitzung           | Die Szene gehört der Sitzung einer Benutzerin; eine veränderliche <code>static</code> -Liste würde jedem Besucher dieselben Knoten und dieselbe Auswahl geben.                                                                                                                                              |
| Eingabe-Events von Control          | <code>Canvas</code> leitet sich von <code>Control</code> ab, also stehen Mouse-Down/Move/Up, Click, Double Click sowie die Touch- und Gestenereignisse wie <code>TouchStart</code> , <code>Tap</code> , <code>Pinch</code> , <code>Rotate</code> , <code>Swipe</code> und <code>Track</code> zur Verfügung. |
| Modellbasierte Trefferprüfung       | Die dokumentierte Canvas-API von Wisej bietet weder <code>isPointInPath</code> noch <code>isPointInStroke</code> , also wird „worauf hat die Benutzerin geklickt?“ beantwortet, indem ein Punkt mit Geometrie verglichen wird, die Ihnen gehört.                                                            |
| Umrechnung Bildschirm nach Welt     | Wendet das Rendering <code>Bildschirm = Welt × Zoom + Verschiebung</code> an, dann muss jede eingehende Zeigerkoordinate durch die Umkehrung laufen, bevor sie mit irgendetwas im Modell verglichen wird.                                                                                                   |
| Viewport-Culling                    | Szenenobjekte überspringen, deren Grenzen das sichtbare Weltrechteck nicht schneiden, sodass eine große Topologie nur das kostet, was tatsächlich auf dem Bildschirm ist.                                                                                                                                   |

## 3. Der rote Faden der Lektion

### Vier Schichten und eine einzige Render-Methode

Teilen Sie den Editor in eine Modellschicht (Domänen- und Szenenobjekte), eine Eingabeschicht (Maus, Pointer, Touch, Tastatur und Befehle), eine Schicht für Zustandsübergänge (Auswahl, Ziehen, Verschieben, Zoomen, Änderungen) und eine Rendschicht. Die Rendschicht ist bewusst langweilig: Sie ist eine deterministische Funktion der aktuellen Szene und entscheidet nichts. Sie ermittelt nicht, welche Pumpe ausgewählt ist; sie liest `node.Selected` und wählt entsprechend eine Füllung. Genau diese Beschränkung lässt die Oberfläche eine Größenänderung überstehen, denn `Redraw` kann dieselbe Methode jederzeit aufrufen und bekommt dasselbe Bild zurück.

Die gegenteilige Versuchung ist stark und verliert immer. Ein `MouseMove`-Handler, der direkt ein Hervorhebungsrechteck zeichnet, ist eine Größenänderung, ein Scrollvorgang oder ein Theme-Wechsel von einer Szene entfernt, die nicht mehr zu dem passt, was die Anwendung selbst für die Welt hält. In VisualOperationsStudio lebt die Szene in einem `TopologyScene`-Objekt, das die Page-Instanz hält, sodass jede Sitzung ihre eigene Topologie bearbeitet – und `RenderScene` aus dem Lab ist die einzige Methode in der Datei, die die Canvas anfassen darf.

## Eingaben von einem Control, nicht von einer Canvas

Weil `Wisej.Web.Canvas` von `Control` erbt, kommen Eingaben genauso an wie überall sonst in Wisej.NET. `MouseDown` wählt aus und startet eine Geste, `MouseMove` bewegt den ausgewählten Knoten oder den Viewport, und `MouseUp` beendet die Geste, indem es den Ziehzustand löscht. Jeder dieser Handler folgt derselben zweizeiligen Form: Zustand ändern, dann die Render-Methode aufrufen. In einem Eventhandler wird nichts gezeichnet.

Seien Sie ehrlich, was das kostet. Es sind serverseitige Events, jedes davon ist also eine Anfrage, und ein Ziehvorgang in menschlichem Tempo erzeugt einen ganzen Strom davon. Für das Auswählen und Umsetzen einiger Dutzend Knoten ist das vertretbar; für Freihandzeichnen oder spielähnliche Bewegung nicht. Planen Sie auch Touch ein: Die Gestenereignisse gibt es, aber Bedienelemente, die nur beim Überfahren erscheinen, gibt es dort nicht – geben Sie Anfassern also genug Größe, um sie mit dem Daumen zu treffen, und verstecken Sie nie eine wesentliche Aktion hinter einem Hover-Zustand.

## Trefferprüfung gegen Geometrie, die Ihnen gehört

Die Trefferprüfung ist eine Modellfrage. Durchlaufen Sie die Knotenliste rückwärts — spätere Elemente wurden obenauf gezeichnet, die umgekehrte Iteration entspricht also der Z-Reihenfolge — und geben Sie den ersten Knoten zurück, dessen Grenzen den Punkt enthalten. Rechtecke nutzen `Contains`; Kreise vergleichen die quadrierte Entfernung mit dem quadrierten Radius, was eine Wurzel pro Knoten spart; linienförmige Kanten brauchen den Abstand Punkt-zu-Strecke mit einer Toleranz, denn niemand klickt exakt auf eine ein Pixel breite Linie. Für wirklich komplexe Pfade halten Sie einen vereinfachten Trefferbereich im Modell, statt rückwärts aus Pixeln schließen zu wollen.

Halten Sie dieselbe Disziplin für Tastaturnutzer ein. Canvas-Knoten sind keine nativen DOM-Controls, also fokussiert sie im Browser nichts für Sie. Bieten Sie Befehle an, die den nächsten Knoten auswählen und den ausgewählten verschieben, und halten Sie neben der Fläche eine Textliste derselben Knoten. Beide Ansichten lesen und schreiben dieselbe Szene, sodass die Auswahl konsistent bleibt, welchen Weg die Benutzerin auch genommen hat.

## Verschieben, Zoomen und der Fehler, der wie ein Auswahlfehler aussieht

Speichern Sie Verschiebung und Zoom als Szenenzustand und wenden Sie sie einmal an, zu Beginn des Renderings, eingeklammert von `Save()` und `Restore()`, damit nichts in das nächste Bild überläuft. Eingaben treffen dann im falschen Koordinatensystem ein und müssen zurückgerechnet werden, bevor sie auf das Modell treffen.

```

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;
}

```

Lassen Sie diese Umrechnung weg, und der Editor verhält sich bei Zoom 1,0 ohne Verschiebung tadellos und beginnt Knoten zu verfehlen, sobald sich eines von beidem ändert — ein Defekt, der als „Auswahl ist nach dem Zoomen kaputt“ gemeldet wird und in Wahrheit ein Fehler im Koordinatensystem ist. Rechnen Sie an einer Stelle um, verwenden Sie das für die Trefferprüfung und für den Ziehversatz, und ergänzen Sie `ToScreen`, um eine Weltposition gegen ein festes Overlay zu platzieren.

### Ein Budget fürs Neuzeichnen und der Punkt, an dem der Client übernimmt

Bevorzugen Sie bei mittelgroßen Szenen ein vollständiges deterministisches Neuzeichnen. Inkrementelles Zeichnen wirkt günstiger und bringt Dirty Regions, Reparaturen der Z-Reihenfolge, Löschartefakte und driftenden Zustand mit sich. Wird das vollständige Neuzeichnen doch teuer, arbeiten Sie eine feste Leiter ab, bevor Sie die Architektur ändern:

- Verringern Sie die Anzahl der Szenenprimitiven und lassen Sie dann alles außerhalb des sichtbaren Weltrechtecks weg.
- Vereinfachen Sie die Geometrie bei geringem Zoom und speichern Sie statische Ebenen als Bilder zwischen.
- Trennen Sie statische und dynamische Flächen, wenn das Layout es zulässt.
- Erst danach verlagern Sie wirklich hochfrequentes Verhalten in den Client.

Lassen Sie `LiveUpdate` für das Hauptrendering aus. Es sendet Zeichenaufrufe sofort, was nützlich ist, um den Fortschritt eines langen Aufbaus zu zeigen, aber jedes Senden hat seinen Preis, und ein einzelnes Rendering im Rahmen einer Eingabeabfrage aktualisiert die Szene bereits in einer einzigen Antwort. Dieselbe Überlegung erledigt den Server-Timer, der dreißig- oder sechzigmal pro Sekunde neu zeichnet: Er verbrennt Serverkapazität pro Sitzung, und jedes Bild geht über das Netzwerk. Wenn eine Darstellung eine Animation und keine Datenänderung ist, senden Sie den neuen Zielwert aus C# und lassen ein Client-Skript darauf zuanimieren, während der Server für die semantische Zahl maßgeblich bleibt. Halten Sie anhand der Messungen aus dem Lab fest, wo Ihre Grenze liegt.



### **Gewohnheit: erst die Oberfläche, dann das Zeichnen**

Wählen Sie die Rendering-Oberfläche, bevor Sie einen einzigen Zeichenaufruf schreiben. Fragen Sie, wo die Pixel entstehen und was über das Netzwerk geht: Der Paint-Handler eines Controls läuft auf dem Server und liefert ein Bild aus, CellPaint tut dasselbe innerhalb einer Grid-Zelle, Canvas schickt Zeichenbefehle, die der Browser ausführt, und behält über eine Größenänderung hinweg nichts, und System.Drawing.Managed erzeugt Bild-Bytes ganz ohne Control. Halten Sie das semantische Modell in C#-Objekten, machen Sie jeden Renderer zu einer deterministischen Funktion dieses Modells, invalidieren Sie nur, wenn sich eine visuelle Abhängigkeit geändert hat, und lassen Sie immer ein textliches Äquivalent für alles Wichtige zurück, das nur als Pixel existiert.



### **Lernziele**

Am Ende dieses Moduls können Sie:

- ✓ Das Muster aus Modell, Eingabe, Zustand und Rendering umsetzen.
- ✓ Pointer-, Maus- und Touch-Events von Wisej-Controls mit einer Canvas-Oberfläche verwenden.
- ✓ Serverseitige Trefferprüfung für Interaktionen mittlerer Frequenz umsetzen.
- ✓ Entscheiden, wann Interaktions- oder Animationslogik nach JavaScript im Client gehört.
- ✓ LiveUpdate, Invalidierung und Neuzeichnen ohne übermäßigen Datenverkehr einsetzen.

## Lab- und Prüfungsleitfaden

Was Sie im praktischen Lab bauen, das vorgeschlagene Vorgehen und die geforderten Ergebnisse.



### Praktisches Lab

Machen Sie aus der Canvas-Spielwiese einen Topologie-Editor für VisualOperationsStudio. Erstellen Sie die Klassen `NodeModel` und `EdgeModel` mit stabilen IDs und Geometrie in Weltkoordinaten, halten Sie die Szene in einem `TopologyScene`-Objekt pro Sitzung statt in einem statischen Feld und lassen Sie `RenderScene` die einzige zeichnende Methode bleiben. Ergänzen Sie `ToWorld`- und `ToScreen`-Umrechnungen für den aktuellen Verschiebe-Offset und den Zoomfaktor, behandeln Sie dann `MouseDown`, um die Knotengrenzen in Weltkoordinaten per Trefferprüfung zu ermitteln und `Selected` zu setzen, `MouseMove`, um den ausgewählten Knoten zu ziehen oder den Viewport zu verschieben, und `MouseUp`, um die Geste zu beenden — jeweils so, dass sie den Zustand ändern und anschließend `RenderScene` aufrufen. Ergänzen Sie Zoomen per Mausrad oder Button um die Zeigerposition herum, überspringen Sie beim Rendern Knoten außerhalb des sichtbaren Rechtecks, geben Sie Tastaturnutzern einen Weg, einen Knoten auszuwählen und zu verschieben, und halten Sie fest, wie viele Anfragen ein Ziehvorgang erzeugt und ab welchem Punkt Sie die Schleife nach JavaScript im Client verlagern würden.

## Ausgangspunkt

VisualOperationsStudio nach Modul 4: Die Seite mit der Canvas-Spielwiese zeichnet ihre gesamte Szene aus einer Methode `DrawPlayground`, die das `Redraw`-Event aufruft, mit `Save()` / `Restore()` um jeden transformierten Abschnitt – aber jede Form ist in Bildschirmkoordinaten fest verdrahtet, und nichts auf der Fläche lässt sich auswählen, bewegen oder verschieben.

## Vorgeschlagenes Vorgehen

- ✓ Schreiben Sie `NodeModel` und `EdgeModel` mit einer stabilen `Id`, `Bounds` in Weltkoordinaten, einer Beschriftung und einem `Selected`-Kennzeichen, fassen Sie sie in einer `TopologyScene` zusammen, die außerdem `PanX`, `PanY` und `Zoom` trägt, und halten Sie diese Szene an der Page-Instanz, damit die Änderungen einer Sitzung nie eine andere erreichen.
- ✓ Ersetzen Sie `DrawPlayground` durch eine Methode `RenderScene`, die die Fläche löscht, `Save()` aufruft, `Translate` und `Scale` aus der Szene anwendet, erst Kanten und dann Knoten zeichnet und `Restore()` aufruft; richten Sie den `Redraw`-Handler darauf aus und prüfen Sie, dass eine Größenänderung des Browsers dasselbe Bild zurückbringt.
- ✓ Ergänzen Sie `ToWorld` und `ToScreen` sowie ein `HitTest`, das die Zeigerposition einmal umrechnet und die Knotenliste rückwärts durchläuft, damit der zuletzt gezeichnete Knoten gewinnt, und für leere Fläche `null` zurückgibt.
- ✓ Verdrahten Sie `MouseDown`, `MouseMove` und `MouseUp` so, dass jeder nur den Szenenzustand ändert — Auswahl, Ziehversatz in Welteinheiten, Verschiebe-Ursprung — und danach `RenderScene` aufruft; ergänzen Sie Zoomen um die Zeigerposition herum und prüfen Sie, dass die Auswahl danach weiterhin den richtigen Knoten trifft.
- ✓ Berechnen Sie das sichtbare Weltrechteck einmal pro Rendering und überspringen Sie alles, was es nicht schneidet, ergänzen Sie Tastaturbefehle, die den nächsten Knoten auswählen und den

ausgewählten verschieben, zählen Sie dann die Anfragen, die ein zweisekündiger Ziehvorgang erzeugt, und schreiben Sie die Zahlen auf.

## Geforderte Ergebnisse

- ✓ NodeModel, EdgeModel und eine TopologyScene pro Sitzung mit stabilen IDs und ohne veränderlichen statischen Zustand
- ✓ ToWorld- und ToScreen-Umrechnungen für die Trefferprüfung, damit die Auswahl nach Verschieben und Zoomen korrekt bleibt
- ✓ MouseDown-, MouseMove- und MouseUp-Handler, die den Zustand ändern und dann die eine Methode RenderScene aufrufen
- ✓ Viewport-Culling, das Knoten außerhalb des sichtbaren Rechtecks überspringt, dazu ein Tastaturweg zum Auswählen und Verschieben
- ✓ InteractionNotes.md mit den während eines Ziehvorgangs gemessenen Anfragen und der Grenze, ab der Sie auf den Client umstellen würden

## Wiederholungsfragen

- ✓ Eine Kollegin meldet, dass die Auswahl funktioniert, bis sie hineinzoomt – danach landen Klicks auf der falschen Pumpe oder auf nichts. Am Zeichencode hat sich nichts geändert. Welche einzelne Umrechnung fehlt, und warum bleibt der Defekt bei Zoom 1,0 ohne Verschiebung unsichtbar?
- ✓ Zwei Bedienende öffnen die Topologie gleichzeitig, und jede sieht den Knoten der anderen unter dem eigenen Cursor wandern. Worin wurde die Szene gespeichert, und was sagt Ihnen das darüber, wo interaktiver Zustand in einem serverseitigen Webframework hingehört?
- ✓ Ihr Ziehvorgang erzeugt für jede Zeigerbewegung eine Anfrage. Ab welchem gemessenen Punkt würden Sie aufhören, die Serverschleife zu optimieren, und die Interaktion stattdessen nach JavaScript im Client verlagern – und was müsste trotzdem auf dem Server maßgeblich bleiben?



### **Gewohnheit: erst die Oberfläche, dann das Zeichnen**

Wählen Sie die Rendering-Oberfläche, bevor Sie einen einzigen Zeichenaufruf schreiben. Fragen Sie, wo die Pixel entstehen und was über das Netzwerk geht: Der Paint-Handler eines Controls läuft auf dem Server und liefert ein Bild aus, CellPaint tut dasselbe innerhalb einer Grid-Zelle, Canvas schickt Zeichenbefehle, die der Browser ausführt, und behält über eine Größenänderung hinweg nichts, und System.Drawing.Managed erzeugt Bild-Bytes ganz ohne Control. Halten Sie das semantische Modell in C#-Objekten, machen Sie jeden Renderer zu einer deterministischen Funktion dieses Modells, invalidieren Sie nur, wenn sich eine visuelle Abhängigkeit geändert hat, und lassen Sie immer ein textliches Äquivalent für alles Wichtige zurück, das nur als Pixel existiert.