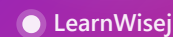


Canvas interattivo: stato, input, hit testing e prestazioni

Guida della lezione · Guida al lab / esame



Guida della lezione

Trasformare un Canvas in un editor: il ciclo Model, Input, State, Render, gli eventi pointer e touch del Control, la conversione da coordinate schermo a coordinate mondo perché l'hit testing sopravviva a pan e zoom, lo stato della scena per sessione invece che statico, il culling del viewport e il punto in cui un loop lato client sostituisce i round trip al server. Che cosa significa per uno sviluppatore .NET esperto che già costruisce schermate Wisej.NET e ora ha bisogno di grafica personalizzata — e come affrontare questo modulo.



Focus di produzione

Trasformare un Canvas in un editor: il ciclo Model, Input, State, Render, gli eventi pointer e touch ereditati dal Control, la conversione da coordinate schermo a coordinate mondo perché l'hit testing sopravviva a pan e zoom, lo stato della scena per sessione invece che statico, il culling del viewport e il punto in cui un loop lato client sostituisce i round trip al server.

1. Il problema in produzione

Il Modulo 4 ha lasciato VisualOperationsStudio con un playground Canvas: un metodo `DrawPlayground` che l'evento `Redraw` richiama, ricostruendo da zero ogni path, gradiente e trasformazione. È corretto, ed è completamente inerte. Il personale di operations vuole fare clic su una pompa, trascinarla dove ha senso, spostarsi sull'impianto con il pan e zoomare su un gruppo — e un metodo di disegno non può rispondere a nessuna di queste richieste, perché il canvas del browser contiene pixel e i pixel non sanno quale forma sia una pompa. Nel momento in cui una superficie diventa interattiva smette di essere un renderer e diventa una piccola macchina a stati che, per inciso, rende.

2. Concetti chiave

Concetto	Perché è importante
Model, Input, State, Render	Quattro livelli con una sola regola fra loro: l'input cambia lo stato, e solo allora un unico metodo di render trasforma lo stato corrente in chiamate di disegno.
Oggetti della scena	Classi C# come <code>NodeModel</code> ed <code>EdgeModel</code> che portano l'ID stabile, i limiti in coordinate mondo, lo stato e il flag <code>Selected</code> che i pixel possono soltanto suggerire.
Stato della scena per sessione	La scena appartiene alla sessione di un singolo utente; una lista <code>static</code> mutabile darebbe a ogni visitatore gli stessi nodi e la stessa selezione.
Eventi di input del Control	<code>Canvas</code> deriva da <code>Control</code> , quindi mouse down/move/up, click, doppio click e gli eventi touch e gesture come <code>TouchStart</code> , <code>Tap</code> , <code>Pinch</code> , <code>Rotate</code> , <code>Swipe</code> e <code>Track</code> sono tutti disponibili.
Hit testing basato sul modello	L'API Canvas documentata di Wisej non espone <code>isPointInPath</code> né <code>isPointInStroke</code> , quindi alla domanda «su che cosa ha fatto clic l'utente?» si risponde confrontando un punto con la geometria che possiedi tu.
Conversione da schermo a mondo	Se il rendering applica $\text{schermo} = \text{mondo} \times \text{zoom} + \text{pan}$, allora ogni coordinata del puntatore in arrivo va ripassata attraverso l'inversa prima di essere confrontata con qualsiasi cosa nel modello.
Culling del viewport	Saltare gli oggetti della scena i cui limiti non intersecano il rettangolo di mondo visibile, così che una topologia grande costi solo ciò che è davvero sullo schermo.

3. Il filo della lezione

Quattro livelli e un unico metodo di render

Dividi l'editor in un livello di modello (oggetti di dominio e di scena), un livello di input (mouse, pointer, touch, tastiera e comandi), un livello di transizione di stato (selezione, trascinamento, pan, zoom, modifiche) e un livello di render. Il livello di render è volutamente noioso: è una funzione deterministica della scena corrente e non decide nulla. Non calcola quale pompa sia selezionata; legge `node.Selected` e sceglie di conseguenza un riempimento. È proprio quel vincolo a far sopravvivere la superficie a un ridimensionamento, perché `Redraw` può chiamare lo stesso metodo in qualsiasi momento e riottenere la stessa immagine.

La tentazione opposta è forte e perde sempre. Un handler di `MouseMove` che disegna direttamente un rettangolo di evidenziazione dista un ridimensionamento, uno scorrimento o un cambio di tema da una scena che non corrisponde più all'idea di mondo dell'applicazione stessa. In VisualOperationsStudio la scena vive in un oggetto `TopologyScene` tenuto dall'istanza della pagina, così ogni sessione modifica la propria topologia, e `RenderScene` del lab è l'unico metodo del file autorizzato a toccare il canvas.

L'input arriva da un Control, non da un canvas

Poiché `Wisej.Web.Canvas` eredita da `Control`, l'input arriva come ovunque altro in Wisej.NET.

`MouseDown` seleziona e avvia un gesto, `MouseMove` sposta il nodo selezionato o il viewport, e `MouseUp`

chiude il gesto azzerando lo stato di trascinamento. Ognuno di questi handler segue la stessa forma in due righe: cambia lo stato, poi chiama il metodo di render. Dentro un gestore di evento non si disegna nulla.

Sii onesto su quanto costa. Sono eventi lato server, quindi ciascuno è una richiesta, e un trascinamento a velocità umana ne produce un flusso. È accettabile per selezionare e riposizionare qualche decina di nodi; non lo è per il disegno a mano libera o per movimenti da videogioco. Pianifica anche per il touch: gli eventi gesture esistono, ma le affordance basate solo sull'hover no, quindi dai alle maniglie una dimensione sufficiente per essere colpite con il pollice e non nascondere mai un'azione essenziale dietro uno stato di hover.

Hit testing contro la geometria che possiedi tu

L'hit testing è una questione di modello. Percorri la lista dei nodi all'indietro — gli elementi successivi sono stati disegnati sopra, quindi l'iterazione inversa rispetta l'ordine z — e restituisci il primo nodo i cui limiti contengono il punto. I rettangoli usano `Contains`; i cerchi confrontano la distanza al quadrato con il raggio al quadrato, il che evita una radice quadrata per nodo; gli archi a forma di linea richiedono la distanza punto-segmento con una tolleranza, perché nessuno fa clic esattamente su una linea da un pixel. Per path davvero complessi, tieni nel modello una regione di hit semplificata invece di cercare di ragionare a ritroso dai pixel.

Mantieni la stessa disciplina per chi usa la tastiera. I nodi del canvas non sono controlli DOM nativi, quindi nel browser nulla darà loro il focus al posto tuo. Offri comandi che selezionano il nodo successivo e spostano quello selezionato, e tieni accanto alla superficie una lista testuale degli stessi nodi. Entrambe le viste leggono e scrivono l'unica scena, così la selezione resta coerente qualunque strada abbia preso l'utente.

Pan, zoom e il bug che sembra un bug di selezione

Memorizza pan e zoom come stato della scena e applicali una volta sola, all'inizio del render, racchiusi fra `Save()` e `Restore()` così che nulla sconfini nel frame successivo. L'input arriva allora nel sistema di coordinate sbagliato e va riconvertito prima di incontrare il modello.

```
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;
}
```

Salta quella conversione e l'editor si comporterà perfettamente a zoom 1.0 senza pan, per poi cominciare a mancare i nodi nel momento in cui uno dei due cambia — un difetto segnalato come «la selezione si

rompe dopo lo zoom» che in realtà è un errore di sistema di coordinate. Converti in un unico punto, usalo per l'hit testing e per l'offset del trascinamento, e aggiungi **ToScreen** per collocare una posizione del mondo rispetto a un overlay fisso.

Un budget di ridisegno, e il punto in cui il client prende il sopravvento

Per scene di dimensioni moderate, preferisci un ridisegno completo e deterministico. Il disegno incrementale sembra più economico e si porta dietro regioni sporche, riparazione dell'ordine z, artefatti di cancellazione e deriva dello stato. Quando il ridisegno completo diventa davvero costoso, scendi lungo una scala fissa prima di cambiare architettura:

- Riduci il numero di primitive della scena, poi scarta tutto ciò che sta fuori dal rettangolo di mondo visibile.
- Semplifica la geometria a zoom basso e metti in cache i livelli statici come immagini.
- Separa le superfici statiche da quelle dinamiche, se il layout lo consente.
- Solo a quel punto sposta sul client i comportamenti davvero ad alta frequenza.

Lascia **LiveUpdate** disattivato per il rendering principale. Spinge subito le chiamate di disegno, il che è utile per mostrare l'avanzamento di una costruzione lunga, ma ogni push ha un costo e un singolo render eseguito come parte di una richiesta di input aggiorna già la scena in un'unica risposta. Lo stesso ragionamento liquida il timer lato server che ridisegna trenta o sessanta volte al secondo: consuma capacità del server per ogni sessione e ogni frame attraversa la rete. Quando un elemento visivo è un'animazione e non un cambiamento di dati, invia da C# il nuovo valore obiettivo e lascia che uno script client animi verso di esso, mantenendo il server autorevole sul numero semantico. Metti per iscritto, sulla base delle misure del lab, dove sta la tua soglia.



La regola di scegliere prima la superficie

Scegli la superficie di rendering prima di scrivere una sola chiamata di disegno. Chiediti dove vengono prodotti i pixel e che cosa attraversa la rete: l'handler di Paint di un controllo gira sul server e spedisce un'immagine, CellPaint fa lo stesso dentro una singola cella di griglia, il Canvas invia comandi di disegno che il browser esegue e non conserva nulla attraverso un resize, e System.Drawing.Managed produce byte di immagine senza alcun controllo. Tieni il modello semantico in oggetti C#, rendi ogni renderer una funzione deterministica di quel modello, invalida solo quando è cambiata una dipendenza visuale e lascia sempre un equivalente testuale di tutto ciò che è importante ed esiste solo come pixel.



Obiettivi di apprendimento

Alla fine di questo modulo sarai in grado di:

- ✓ Implementare il pattern Model, Input, State, Render.
- ✓ Usare gli eventi pointer/mouse/touch dei controlli Wisej con una superficie Canvas.
- ✓ Implementare l'hit testing lato server per interazioni a frequenza moderata.
- ✓ Decidere quando spostare la logica di interazione o di animazione su JavaScript lato client.
- ✓ Usare LiveUpdate, invalidazione e ridisegno senza traffico eccessivo.

Guida al lab / esame

Che cosa costruirai nel lab pratico, l'approccio suggerito e i deliverable richiesti.



Lab pratico

Trasforma il playground Canvas in un editor di topologia per VisualOperationsStudio. Crea le classi `NodeModel` ed `EdgeModel` con ID stabili e geometria in coordinate mondo, tieni la scena in un oggetto `TopologyScene` memorizzato per sessione invece che in un campo statico, e fai di `RenderScene` l'unico metodo che disegna. Aggiungi le conversioni `ToWorld` e `ToScreen` per l'offset di pan e il fattore di zoom correnti, poi gestisci `MouseDown` per fare hit testing sui limiti dei nodi in coordinate mondo e impostare `Selected`, `MouseMove` per trascinare il nodo selezionato o spostare il viewport, e `MouseUp` per chiudere il gesto — ciascuno cambia lo stato e poi chiama `RenderScene`. Aggiungi lo zoom con la rotella o con pulsanti attorno al puntatore, salta i nodi fuori dal rettangolo visibile durante il rendering, dai a chi usa la tastiera un modo per selezionare e spostare un nodo, e registra il traffico di richieste prodotto da un trascinamento più il punto a partire dal quale sposteresti il loop su JavaScript lato client.

Punto di partenza

VisualOperationsStudio dopo il Modulo 4: la pagina del playground Canvas disegna l'intera scena da un metodo `DrawPlayground` chiamato dall'evento `Redraw`, con `Save()` / `Restore()` attorno a ogni sezione trasformata, ma ogni forma è scritta a mano in coordinate schermo e nulla sulla superficie può essere selezionato, spostato o pannato.

Approccio suggerito

- ✓ Scrivi `NodeModel` ed `EdgeModel` con un `Id` stabile, `Bounds` in coordinate mondo, un'etichetta e un flag `Selected`, raccoglili in un `TopologyScene` che porta anche `PanX`, `PanY` e `Zoom`, e tieni quella scena sull'istanza della pagina così che le modifiche di una sessione non raggiungano mai un'altra.
- ✓ Sostituisci `DrawPlayground` con un metodo `RenderScene` che pulisce la superficie, chiama `Save()`, applica `Translate` e `Scale` dalla scena, disegna prima gli archi e poi i nodi, e chiama `Restore()`; punta l'handler di `Redraw` su di esso e verifica che un ridimensionamento del browser riporti la stessa immagine.
- ✓ Aggiungi `ToWorld` e `ToScreen`, poi un `HitTest` che converte una sola volta la posizione del puntatore e percorre all'indietro la lista dei nodi così che vinca il nodo disegnato per ultimo, restituendo `null` per lo spazio vuoto.
- ✓ Collega `MouseDown`, `MouseMove` e `MouseUp` così che ciascuno cambi soltanto lo stato della scena — selezione, offset del trascinamento in unità mondo, origine del pan — e poi chiami `RenderScene`; aggiungi lo zoom attorno al puntatore e verifica che dopo la selezione cada ancora sul nodo giusto.
- ✓ Calcola una volta per render il rettangolo di mondo visibile e salta tutto ciò che non lo interseca, aggiungi comandi da tastiera che selezionano il nodo successivo e spostano quello selezionato, poi conta le richieste prodotte da un trascinamento di due secondi e annota i numeri.

Deliverable richiesti

- ✓ NodeModel, EdgeModel e un TopologyScene per sessione con ID stabili e nessuno stato statico mutabile
- ✓ Conversioni ToWorld e ToScreen usate per l'hit testing, così che la selezione resti corretta dopo pan e zoom
- ✓ Handler di MouseDown, MouseMove e MouseUp che cambiano lo stato e poi chiamano l'unico metodo RenderScene
- ✓ Culling del viewport che salta i nodi fuori dal rettangolo visibile, più un percorso da tastiera per selezionare e spostare
- ✓ InteractionNotes.md con il traffico di richieste misurato durante un trascinamento e la soglia oltre la quale passeresti al lato client

Domande di autovalutazione

- ✓ Un collega segnala che la selezione funziona finché non zooma, dopodiché i clic cadono sulla pompa sbagliata o nel vuoto. Nel codice di disegno non è cambiato nulla. Quale singola conversione manca, e perché il difetto resta invisibile a zoom 1.0 senza pan?
- ✓ Due operatori aprono la topologia nello stesso momento e ciascuno vede muoversi sotto il proprio cursore il nodo dell'altro. In che cosa è stata memorizzata la scena, e che cosa ti dice questo su dove debba stare lo stato interattivo in un framework web lato server?
- ✓ Il tuo trascinamento produce una richiesta per ogni movimento del puntatore. A quale punto misurato smetteresti di ottimizzare il loop lato server per spostare invece l'interazione su JavaScript lato client, e che cosa dovrebbe comunque restare autorevole sul server?



La regola di scegliere prima la superficie

Scegli la superficie di rendering prima di scrivere una sola chiamata di disegno. Chiediti dove vengono prodotti i pixel e che cosa attraversa la rete: l'handler di Paint di un controllo gira sul server e spedisce un'immagine, CellPaint fa lo stesso dentro una singola cella di griglia, il Canvas invia comandi di disegno che il browser esegue e non conserva nulla attraverso un resize, e System.Drawing.Managed produce byte di immagine senza alcun controllo. Tieni il modello semantico in oggetti C#, rendi ogni renderer una funzione deterministica di quel modello, invalida solo quando è cambiata una dipendenza visuale e lascia sempre un equivalente testuale di tutto ciò che è importante ed esiste solo come pixel.