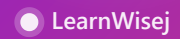


Canvas interactivo: estado, entrada, prueba de impacto y rendimiento

Guía de la lección · Guía del laboratorio / examen



Guía de la lección

Convertir un Canvas en un editor: el bucle Modelo, Entrada, Estado y Render; los eventos de puntero y táctiles que llegan desde Control; la conversión de pantalla a mundo para que la prueba de impacto sobreviva al desplazamiento y al zoom; estado de escena por sesión en lugar de campos estáticos; descarte por viewport; y el punto en el que un bucle en el cliente sustituye a los viajes de ida y vuelta al servidor. Qué significa esto para un desarrollador .NET con experiencia que ya construye pantallas Wisej.NET y ahora necesita gráficos personalizados, y cómo leer este módulo.



Enfoque de producción

Convertir un Canvas en un editor: el bucle Modelo, Entrada, Estado y Render; los eventos de puntero y táctiles que llegan desde Control; la conversión de pantalla a mundo para que la prueba de impacto sobreviva al desplazamiento y al zoom; estado de escena por sesión en lugar de campos estáticos; descarte por viewport; y el punto en el que un bucle en el cliente sustituye a los viajes de ida y vuelta al servidor.

1. El problema en producción

El Módulo 4 dejó a VisualOperationsStudio con un área de pruebas de Canvas: un método `DrawPlayground` al que llama el evento `Redraw` y que reconstruye desde cero todos los trazados, degradados y transformaciones. Es correcto, y es completamente inerte. El personal de operaciones quiere hacer clic en una bomba, arrastrarla a un sitio razonable, desplazarse por la planta y ampliar un grupo de elementos, y un método de dibujo no puede responder a ninguna de esas peticiones, porque el canvas del navegador guarda píxeles y los píxeles no saben qué forma es una bomba. En cuanto una superficie se vuelve interactiva deja de ser un renderizador y pasa a ser una pequeña máquina de estados que, además, dibuja.

2. Conceptos clave

Concepto	Por qué importa
Modelo, Entrada, Estado, Render	Cuatro capas con una regla entre ellas: la entrada cambia el estado y solo entonces un único método de render convierte el estado actual en llamadas de dibujo.
Objetos de escena	Clases C# como <code>NodeModel</code> y <code>EdgeModel</code> que llevan el identificador estable, los límites en espacio de mundo, el estado y la marca <code>Selected</code> que los píxeles solo pueden insinuar.
Estado de escena por sesión	La escena pertenece a la sesión de un usuario; una lista <code>static</code> mutable daría a todos los visitantes los mismos nodos y la misma selección.
Eventos de entrada de Control	<code>Canvas</code> deriva de <code>Control</code> , así que están disponibles mouse down/move/up, clic, doble clic y los eventos táctiles y de gestos como <code>TouchStart</code> , <code>Tap</code> , <code>Pinch</code> , <code>Rotate</code> , <code>Swipe</code> y <code>Track</code> .
Prueba de impacto basada en el modelo	La API de Canvas de Wisej documentada no expone <code>isPointInPath</code> ni <code>isPointInStroke</code> , así que «¿en qué ha hecho clic el usuario?» se responde comparando un punto con una geometría que te pertenece.
Conversión de pantalla a mundo	Si el renderizado aplica <code>pantalla = mundo × zoom + desplazamiento</code> , entonces cada coordenada de puntero entrante hay que pasarla por la inversa antes de compararla con nada del modelo.
Descarte por viewport	Omitir los objetos de la escena cuyos límites no intersecan el rectángulo de mundo visible, para que una topología grande cueste solo lo que hay realmente en pantalla.

3. Desarrollo de la lección

Cuatro capas y un único método de render

Divide el editor en una capa de modelo (objetos de dominio y de escena), una capa de entrada (ratón, puntero, táctil, teclado y comandos), una capa de transición de estado (selección, arrastre, desplazamiento, zoom, ediciones) y una capa de render. La capa de render es deliberadamente aburrida: es una función determinista de la escena actual y no decide nada. No averigua qué bomba está seleccionada; lee `node.Selected` y elige un relleno en consecuencia. Esa restricción es lo que hace que la superficie sobreviva a un redimensionado, porque `Redraw` puede llamar a ese mismo método en cualquier momento y recuperar la misma imagen.

La tentación contraria es fuerte y siempre pierde. Un controlador de `MouseMove` que dibuja directamente un rectángulo de resaltado está a un redimensionado, un desplazamiento o un cambio de tema de tener una escena que ya no coincide con la idea que la propia aplicación tiene del mundo. En VisualOperationsStudio la escena vive en un objeto `TopologyScene` que mantiene la instancia de la página, así que cada sesión edita su propia topología, y el `RenderScene` del laboratorio es el único método del archivo con permiso para tocar el canvas.

La entrada llega desde un Control, no desde un canvas

Como `Wisej.Web.Canvas` hereda de `Control`, la entrada llega igual que en cualquier otro punto de Wisej.NET. `MouseDown` selecciona e inicia un gesto, `MouseMove` mueve el nodo seleccionado o el viewport, y `MouseUp` termina el gesto borrando el estado de arrastre. Todos esos controladores siguen la misma forma de dos líneas: cambiar el estado y después llamar al método de render. Nada se dibuja dentro de un controlador de eventos.

Sé honesto sobre lo que cuesta. Son eventos del lado del servidor, así que cada uno es una petición, y un arrastre a velocidad humana produce un chorro de ellas. Eso es aceptable para seleccionar y recolocar unas decenas de nodos; no lo es para tinta a mano alzada o movimiento tipo videojuego. Planifica también para el uso táctil: los eventos de gestos existen, pero las ayudas que solo aparecen al pasar el ratón no, así que da a los controladores tamaño suficiente para acertarles con el pulgar y no escondas nunca una acción esencial tras un estado de hover.

Prueba de impacto contra una geometría que te pertenece

La prueba de impacto es una cuestión del modelo. Recorre la lista de nodos hacia atrás —los elementos posteriores se dibujaron encima, así que el recorrido inverso se corresponde con el orden z— y devuelve el primer nodo cuyos límites contengan el punto. Los rectángulos usan `Contains`; las circunferencias comparan la distancia al cuadrado con el radio al cuadrado, lo que evita una raíz cuadrada por nodo; las aristas con forma de línea necesitan la distancia de punto a segmento con una tolerancia, porque nadie hace clic exactamente sobre una línea de un píxel. Para trazados realmente complejos, mantén en el modelo una región de impacto simplificada en lugar de intentar razonar hacia atrás a partir de los píxeles.

Mantén la misma disciplina para quienes usan el teclado. Los nodos del Canvas no son controles nativos del DOM, así que nada en el navegador les dará el foco por ti. Ofrece comandos que seleccionen el siguiente nodo y desplacen el seleccionado, y mantén una lista textual de esos mismos nodos junto a la superficie. Ambas vistas leen y escriben en la única escena, así que la selección se mantiene coherente sea cual sea la vía que haya usado el usuario.

Desplazamiento, zoom y el error que parece un error de selección

Guarda el desplazamiento y el zoom como estado de la escena y aplícalos una sola vez, al principio del render, delimitados por `Save()` y `Restore()` para que nada se filtre al siguiente fotograma. La entrada llega entonces en el sistema de coordenadas equivocado y hay que reconvertirla antes de que se encuentre con el modelo.

```
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;
}
```

Sáltate esa conversión y el editor se comportará perfectamente con zoom 1,0 y sin desplazamiento, y empezará a fallar nodos en cuanto cambie cualquiera de los dos: un defecto que se reporta como «la selección se rompe al hacer zoom» y que en realidad es un error de sistema de coordenadas. Convierte en un solo sitio, úsalo para la prueba de impacto y para el desplazamiento del arrastre, y añade `ToScreen` para situar una posición de mundo frente a una superposición fija.

Un presupuesto de redibujado y el punto en el que el cliente toma el relevo

Para escenas moderadas, prefiere un redibujado completo y determinista. El dibujo incremental parece más barato y trae consigo regiones sucias, reparación del orden z, artefactos de borrado y desviaciones de estado. Cuando el redibujado completo sí se vuelva caro, recorre una escala fija antes de cambiar la arquitectura:

- Reduce el número de primitivas de la escena y después descarta todo lo que quede fuera del rectángulo de mundo visible.
- Simplifica la geometría con zoom bajo y guarda en caché las capas estáticas como imágenes.
- Separa las superficies estáticas de las dinámicas si el diseño lo permite.
- Solo entonces, lleva al cliente el comportamiento realmente de alta frecuencia.

Deja `LiveUpdate` desactivado para el render principal. Envía las llamadas de dibujo de inmediato, lo que resulta útil para mostrar el progreso de una construcción larga, pero cada envío tiene un coste y un único render hecho como parte de una petición de entrada ya actualiza la escena en una sola respuesta. El mismo razonamiento acaba con el temporizador de servidor que redibuja treinta o sesenta veces por segundo: quema capacidad de servidor por sesión y cada fotograma cruza la red. Cuando un elemento visual es una animación y no un cambio de datos, envía el nuevo valor de destino desde C# y deja que un script del cliente anime hacia él, manteniendo el servidor como autoridad del número semántico. Anota, a partir de las mediciones del propio laboratorio, dónde está tu límite.



Regla de dibujo: primero la superficie

Elige la superficie de renderizado antes de escribir una sola llamada de dibujo. Pregúntate dónde se producen los píxeles y qué cruza la red: un controlador de Paint de un control se ejecuta en el servidor y envía una imagen, CellPaint hace lo mismo dentro de una celda de cuadrícula, Canvas envía comandos de dibujo que ejecuta el navegador y no conserva nada tras un redimensionado, y System.Drawing.Managed produce bytes de imagen sin ningún control de por medio. Mantén el modelo semántico en objetos C#, haz que cada renderizador sea una función determinista de ese modelo, invalida solo cuando haya cambiado una dependencia visual, y deja siempre un equivalente textual de todo lo importante que exista únicamente como píxeles.



Objetivos de aprendizaje

Al final de este módulo podrás:

- ✓ Implementar el patrón Modelo, Entrada, Estado y Render.
- ✓ Usar los eventos de puntero, ratón y táctiles de los controles de Wisej con una superficie Canvas.
- ✓ Implementar la prueba de impacto en el servidor para interacciones de frecuencia moderada.
- ✓ Decidir cuándo llevar la lógica de interacción o de animación a JavaScript en el cliente.
- ✓ Usar LiveUpdate, la invalidación y el redibujado sin generar tráfico excesivo.

Guía del laboratorio / examen

Lo que construirás en el laboratorio práctico, el enfoque sugerido y los entregables requeridos.



Laboratorio práctico

Convierte el área de pruebas de Canvas en un editor de topología para VisualOperationsStudio. Crea las clases `NodeModel` y `EdgeModel` con identificadores estables y geometría en coordenadas de mundo, guarda la escena en un objeto `TopologyScene` almacenado por sesión en lugar de en un campo estático, y deja `RenderScene` como el único método que dibuja. Añade conversiones `ToWorld` y `ToScreen` para el desplazamiento y el factor de zoom actuales; después gestiona `MouseDown` para hacer la prueba de impacto contra los límites de los nodos en coordenadas de mundo y marcar `Selected`, `MouseMove` para arrastrar el nodo seleccionado o desplazar el viewport, y `MouseUp` para terminar el gesto, cambiando cada uno el estado y llamando después a `RenderScene`. Añade zoom con la rueda o con botones centrado en el puntero, omite al dibujar los nodos que quedan fuera del rectángulo visible, da a quien usa el teclado una forma de seleccionar y desplazar un nodo, y registra el tráfico de peticiones que genera un arrastre junto con el punto a partir del cual llevarías el bucle a JavaScript en el cliente.

Punto de partida

VisualOperationsStudio después del Módulo 4: la página del área de pruebas de Canvas dibuja toda su escena desde un método `DrawPlayground` al que llama el evento `Redraw`, con `Save()` / `Restore()` alrededor de cada sección transformada, pero todas las formas están fijadas en el código en coordenadas de pantalla y nada de la superficie se puede seleccionar, mover ni desplazar.

Enfoque sugerido

- ✓ Escribe `NodeModel` y `EdgeModel` con un `Id` estable, `Bounds` en espacio de mundo, una etiqueta y una marca `Selected`; reúnelos en un `TopologyScene` que lleve además `PanX`, `PanY` y `Zoom`, y mantén esa escena en la instancia de la página para que las ediciones de una sesión nunca lleguen a otra.
- ✓ Sustituye `DrawPlayground` por un método `RenderScene` que limpie la superficie, llame a `Save()`, aplique `Translate` y `Scale` a partir de la escena, dibuje primero las aristas y después los nodos, y llame a `Restore()`; apunta el controlador de `Redraw` hacia él y confirma que un redimensionado del navegador devuelve la misma imagen.
- ✓ Añade `ToWorld` y `ToScreen`, y después un `HitTest` que convierta la posición del puntero una sola vez y recorra la lista de nodos hacia atrás, para que gane el último nodo dibujado, devolviendo `null` en el espacio vacío.
- ✓ Conecta `MouseDown`, `MouseMove` y `MouseUp` de modo que cada uno solo cambie el estado de la escena —la selección, el desplazamiento del arrastre en unidades de mundo, el origen del desplazamiento— y después llame a `RenderScene`; añade zoom centrado en el puntero y comprueba que la selección sigue cayendo después sobre el nodo correcto.
- ✓ Calcula el rectángulo de mundo visible una vez por render y omite todo lo que no lo interseque, añade comandos de teclado que seleccionen el siguiente nodo y desplacen el seleccionado, y después cuenta las peticiones que produce un arrastre de dos segundos y anota las cifras.

Entregables requeridos

- ✓ NodeModel, EdgeModel y un TopologyScene por sesión con identificadores estables y sin estado estático mutable
- ✓ Conversiones ToWorld y ToScreen usadas en la prueba de impacto para que la selección siga siendo correcta tras desplazar y hacer zoom
- ✓ Controladores de MouseDown, MouseMove y MouseUp que cambian el estado y después llaman al único método RenderScene
- ✓ Descarte de los nodos que quedan fuera del rectángulo visible, más una vía de teclado para seleccionar y desplazar
- ✓ InteractionNotes.md con el tráfico de peticiones medido durante un arrastre y el límite a partir del cual pasarías al cliente

Preguntas de repaso

- ✓ Un compañero informa de que la selección va bien hasta que amplía, y a partir de ahí los clics caen sobre la bomba equivocada o sobre nada. El código de dibujo no ha cambiado. ¿Qué única conversión falta y por qué el defecto se mantiene invisible con zoom 1,0 y sin desplazamiento?
- ✓ Dos operarios abren la topología a la vez y cada uno ve moverse bajo su cursor el nodo del otro. ¿Dónde se guardó la escena y qué te dice eso sobre dónde corresponde el estado interactivo en un framework web del lado del servidor?
- ✓ Tu arrastre produce una petición por cada movimiento del puntero. ¿En qué punto medido dejarías de optimizar el bucle del servidor y llevarías la interacción a JavaScript en el cliente, y qué tendría que seguir siendo autoritativo en el servidor?



Regla de dibujo: primero la superficie

Elige la superficie de renderizado antes de escribir una sola llamada de dibujo. Pregúntate dónde se producen los píxeles y qué cruza la red: un controlador de Paint de un control se ejecuta en el servidor y envía una imagen, CellPaint hace lo mismo dentro de una celda de cuadrícula, Canvas envía comandos de dibujo que ejecuta el navegador y no conserva nada tras un redimensionado, y System.Drawing.Managed produce bytes de imagen sin ningún control de por medio. Mantén el modelo semántico en objetos C#, haz que cada renderizador sea una función determinista de ese modelo, invalida solo cuando haya cambiado una dependencia visual, y deja siempre un equivalente textual de todo lo importante que exista únicamente como píxeles.