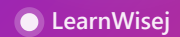


Arquitectura de dibujo en producción, rendimiento y proyecto final

Guía de la lección · Guía del laboratorio / examen



Guía de la lección

Llevar el código de dibujo a producción: un único modelo visual con una capa de geometría pura y renderizadores separados por superficie, instrumentación del tiempo de renderizado y del tamaño de la carga útil, perfilado de una ruta dibujada frente a una ruta fuera de pantalla, comportamiento ante redimensionados, temas, datos vacíos y fuentes ausentes, un equivalente textual accesible, y el proyecto final Visual Operations Studio. 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

Llevar el código de dibujo a producción: un único modelo visual con una capa de geometría pura y renderizadores separados por superficie, instrumentación del tiempo de renderizado y del tamaño de la carga útil, perfilado de una ruta dibujada frente a una ruta fuera de pantalla, comportamiento ante redimensionados, temas, datos vacíos y fuentes ausentes, un equivalente textual accesible, y el proyecto final Visual Operations Studio.

1. El problema en producción

VisualOperationsStudio es ya dueño de cuatro superficies de renderizado, y cada una se construyó en su propio módulo con su propia idea privada de lo que significa la salud de una máquina. Así es como se pudre un subsistema de dibujo: alguien mueve un umbral de advertencia, tres superficies coinciden y el PNG exportado contradice en silencio la pantalla de la que salió. Este módulo funde esas cuatro interpretaciones en un único modelo tras una única capa de geometría y, después, demuestra con mediciones, rutas de fallo deliberadas y un equivalente semántico que el resultado sobrevive a datos reales, plataformas reales y usuarios reales.

2. Conceptos clave

Concepto	Por qué importa
Modelo visual	Solo estado semántico: lecturas, rótulos, nodos y aristas, series de tendencia, la severidad como decisión de dominio y la selección compartida, para que ninguna superficie necesite una copia.
Capa de geometría	Cálculo casi puro que convierte el estado del modelo más el tamaño actual de la superficie en rectángulos, ángulos de barrido, posiciones de etiqueta y regiones de impacto, comprobable sin ningún control presente.
Adaptador de superficie	Un renderizador por superficie en lugar de una interfaz común: <code>System.Drawing.Graphics</code> y <code>Wisej.Web.Canvas</code> conservan sus propias primitivas sobre una geometría compartida.
Presupuesto de rendimiento	Una cifra acordada antes de optimizar, por ejemplo un control de estado dibujado por debajo de 5 ms y un redibujado completo de Canvas dentro de unos 16 a 50 ms.
Instrumentación del renderizado	El tiempo transcurrido registrado, las dimensiones del mapa de bits, el número de objetos dibujados y el crecimiento de la carga útil por superficie, para que una optimización se argumente a partir de un antes y un después.
Prueba de imagen de referencia	Un renderizado fuera de pantalla con entrada fija comparado con una referencia guardada usando una tolerancia, porque el suavizado de bordes y los codificadores difieren entre plataformas.
Equivalente semántico	La etiqueta, el <code>AccessibleDescription</code> , el tooltip o la lista acompañante que llevan todos los valores esenciales que muestran los gráficos, para que nada importante exista solo como píxeles.

3. Desarrollo de la lección

Un modelo, una capa de geometría, un renderizador por superficie

El diseño reutilizable tiene tres capas. El modelo visual guarda estado semántico y nada más. La capa de geometría convierte ese estado y las dimensiones actuales del viewport en rectángulos, puntos, trazados, posiciones de etiqueta y regiones de impacto. El renderizador convierte la geometría en una superficie, y solo ahí sabe algo si está hablando con un objeto `Graphics`, con un `Wisej.Web.Canvas` o con un fragmento de HTML. Un único modelo de topología se puede mostrar entonces de forma interactiva en Canvas, exportar a PNG con `System.Drawing.Managed` y enumerar para la accesibilidad sin que las reglas de dominio existan tres veces.

Resiste la tentación de unificar los renderizadores en sí: su semántica difiere de verdad, y una interfaz limitada a lo que ambos pueden hacer cuesta más de lo que ahorra salvo que el elemento visual sea trivial. Comparte en su lugar el modelo y la geometría. Para `VisualOperationsStudio` la reestructuración es sobre todo reubicación: los cálculos del indicador del Módulo 2 y las conversiones a coordenadas de mundo del Módulo 5 pasan a `Geometry`, mientras que el controlador de `Paint`, el de `CellPaint`, el método `Redraw` del Canvas y el renderizador fuera de pantalla se convierten en adaptadores finos. La selección

también pasa al modelo, que es lo que permite que un clic en una fila de la cuadrícula reoriente el indicador y seleccione ese mismo nodo del Canvas.

Primero los presupuestos, después la instrumentación

Decide las cifras antes de optimizar. Presupuestos de partida razonables: un control de estado dibujado por debajo de 5 ms de renderizado en el servidor a tamaño normal, una columna de cuadrícula dibujada por el usuario sin tirones visibles al desplazarse con un número de filas representativo, un redibujado completo de Canvas dentro de unos 16 a 50 ms, y una exportación que puede tardar más porque es un comando explícito, pero que no debe bloquear a sesiones ajenas. Lo que registras cambia según la superficie:

- `Paint` y `CellPaint` en el servidor: tiempo transcurrido del controlador, tasa de reservas de memoria, tamaño de la imagen temporal, crecimiento de la carga útil de la respuesta y redibujados por flujo de trabajo.
- Canvas: comandos de dibujo por redibujado, tamaño de la carga útil de la escena, frecuencia de redibujado, latencia de ida y vuelta del puntero y el coste propio del navegador con una escena grande.
- `System.Drawing.Managed` fuera de pantalla: dimensiones del mapa de bits y memoria de píxeles, tiempo de codificación, comportamiento de carga de fuentes, tamaño de la salida y diferencias visuales por plataforma.

```
var watch = System.Diagnostics.Stopwatch.StartNew();
using (var bitmap = new Bitmap(width, height))
using (var g = Graphics.FromImage(bitmap))
{
    _topologyRenderer.Render(g, scene, layout);
    watch.Stop();
    _metrics.Record("topology.export", watch.Elapsed,
        bitmap.Width, bitmap.Height, layout.VisibleNodes.Count);
}
```

Con el registro en su sitio, perfila una ruta dibujada en el servidor frente a una ruta fuera de pantalla usando las herramientas de uso de CPU y de reservas de memoria, más microbenchmarks alrededor de la geometría pura. Un controlador que es rápido para una celda no lo es automáticamente para mil, los grafos de objetos por celda son el culpable habitual, y las sesiones concurrentes forman parte de la medición y no de lo que viene después.

Dos matrices de pruebas: plataformas y navegadores

El renderizado fuera de pantalla hay que verificarlo en todas las plataformas que prometas: un servidor Windows con .NET moderno, un contenedor Linux, macOS para desarrollo y pruebas donde esté soportado, e iOS o Android Hybrid si los distribuyes. Automatiza las métricas de texto, las fuentes obligatorias, las imágenes transparentes, los degradados, el suavizado de bordes y el guardado de imágenes, respaldado por salidas de referencia comparadas con una tolerancia en lugar de por igualdad de bytes, porque un codificador distinto o medio píxel de suavizado es una diferencia de plataforma y no una regresión.

Canvas necesita una segunda matriz, porque el renderizado lo hace el navegador: verifica Chromium, Firefox y Safari actuales, vigilando fuentes, composición, sombras y gestos táctiles. Redimensiona todo mientras pruebas, ya que `Redraw` debe reconstruir la escena entera a partir del modelo y un controlador de `Paint` debe recalcular la geometría a partir del ancho y el alto actuales. Estructura la batería de pruebas por capas, como recomienda el pack: pruebas unitarias de la normalización, la conversión de valor a ángulo, la prueba de impacto, las transformaciones de zoom y el descarte; pruebas de imagen de referencia para el renderizador fuera de pantalla; pruebas de integración del renderizado de la página, el redibujado tras un redimensionado, el desplazamiento por la cuadrícula y los cambios de tema; y pasadas manuales para uso táctil, teclado, lectores de pantalla, zoom del navegador y alta densidad de píxeles.

Degrada en lugar de lanzar excepciones, y no distribuyas nunca solo píxeles

Dibuja los estados malos a propósito. Una imagen ausente pasa a ser un glifo de marcador de posición más una entrada de registro; una fuente ausente se resuelve mediante una cadena de alternativas definida; unos datos vacíos pasan a ser un elemento visual neutro de «sin datos»; un número no válido se acota o se rechaza antes de dibujar; y un fallo de exportación aparece como un mensaje sobre el que el usuario puede actuar en lugar de como una imagen en blanco. Los temas merecen el mismo cuidado: evita fijar en el código un fondo blanco y texto negro, acepta los colores como propiedades públicas o resuélvelos desde el tema de la aplicación, y prueba tanto con un tema oscuro como con uno claro.

Todo valor dibujado esencial necesita además un gemelo semántico. Un indicador recibe una etiqueta visible como «CPU 72%» junto con un `AccessibleDescription`. Una sparkline recibe un texto accesible del estilo de «Tendencia: +12% en 7 días». Una topología en Canvas recibe un `TreeView` o un `ListView` acompañante con selección por teclado. La severidad expresada como color necesita una forma o una palabra al lado, porque el rojo y el verde por sí solos no llegan a todo el mundo.

El proyecto final Visual Operations Studio

El proyecto final es una página con cuatro vistas coordinadas: el `TelemetryGauge` dibujado para el activo seleccionado, la cuadrícula de operaciones con sus columnas Health y Trend dibujadas por el usuario, la topología en Canvas con selección, arrastre, desplazamiento y zoom, y un comando Export que produce un PNG mediante `System.Drawing.Managed`. La selección es compartida, y la exportación la genera un renderizador de `System.Drawing` a partir del modelo visual, nunca intentando hacer una captura del navegador.

Junto a la aplicación en funcionamiento entregas las evidencias: dónde se ejecuta cada renderizador y qué cruza la red, la prueba de que ningún estado de escena por sesión reside en un estático de ámbito de proceso, tu política de liberación de recursos y de fuentes, mediciones de dos rutas de renderizado con una optimización que puedas defender, un plan de verificación en Windows y en contenedor Linux con las implicaciones para iOS, Android y macOS, y la alternativa accesible. Después recorre la lista de comprobación del diseño final y responde que sí a cada línea, incluidas la de si el elemento visual se reconstruye a partir del estado del modelo, la de si la superficie se ajusta a la frecuencia de actualización y la de si la animación de alta frecuencia se queda en el cliente.



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:

- ✓ Construir un modelo visual compartido con renderizadores separados de System.Drawing y de Canvas.
- ✓ Perfilar la CPU de dibujo, las reservas de memoria, los tamaños de imagen, el desplazamiento de la cuadrícula y el comportamiento de las sesiones.
- ✓ Probar el renderizado multiplataforma y el comportamiento de redibujado en el navegador.
- ✓ Diseñar la accesibilidad, la exportación, los temas, el diseño adaptable y el tratamiento de errores dentro de los elementos visuales personalizados.
- ✓ Completar el proyecto final Visual Operations Studio.

Guía del laboratorio / examen

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



Laboratorio práctico

Completa el proyecto final de VisualOperationsStudio poniendo un único modelo detrás de las cuatro superficies. Reestructura el proyecto en carpetas Models, Geometry, Renderers y de adaptadores de Wisej, de modo que el indicador dibujado, las celdas de cuadrícula dibujadas por el usuario, la topología en Canvas y la exportación a PNG lean todos el mismo OperationsModel y compartan la capa de geometría pura, sin duplicar estado de negocio en cada renderizador. Añade un ayudante RenderMetrics que registre la duración del renderizado, el tamaño del mapa de bits de salida y el número de objetos de la escena dibujados, y úsalo para comparar el indicador dibujado en el servidor con el renderizador fuera de pantalla, antes y después de una optimización que elijas y justifiques. Ejercita las rutas de fallo —datos vacíos, una superficie de tamaño cero, un rango no válido, una fuente ausente— para que cada una degrade a un marcador de posición legible en lugar de provocar una excepción, añade una tabla accesible que enumere todos los valores que muestran los gráficos y redacta las notas del proyecto final cubriendo la arquitectura, la frontera entre navegador y servidor y la matriz de pruebas.

Punto de partida

VisualOperationsStudio tal como lo dejaste tras el Módulo 6: un control `TelemetryGauge` con sus propios cálculos de indicador, una cuadrícula de operaciones con columnas Health y Trend con `UserPaint`, un editor de topología en Canvas que mantiene una escena por sesión con prueba de impacto, desplazamiento y zoom, y un `TopologyImageRenderer` que escribe un PNG mediante `Graphics.FromImage`. Todas las superficies funcionan, pero cada una guarda su propia copia de umbrales, colores y posiciones de nodo, no se mide nada, y una fuente ausente o un conjunto de datos vacío siguen llegando al código de dibujo como una excepción.

Enfoque sugerido

- ✓ Crea primero las carpetas `Models`, `Geometry`, `Renderers` y la de adaptadores, y después lleva una superficie cada vez a un único `OperationsModel` que sea dueño de los activos, sus lecturas, los nodos y aristas de topología, los umbrales de severidad y la selección compartida. Elimina sobre la marcha la copia privada de cada renderizador y mantén el estado de la escena en la sesión en lugar de en un campo estático.
- ✓ Saca los cálculos de coordenadas de los cuatro renderizadores y llévalos a la capa de geometría, de modo que reciba el estado del modelo más un tamaño y devuelva los límites del anillo, los ángulos de barrido, los rectángulos de celda, los rectángulos de nodo y las posiciones de las etiquetas. Prueba unitariamente allí la conversión de valor a ángulo, la normalización de un rango invertido, la prueba de impacto y el descarte de la región visible, donde no hace falta ningún control de Wisej.
- ✓ Escribe `RenderMetrics` con un método `Record` que reciba el nombre de la superficie, el tiempo transcurrido del `Stopwatch`, el ancho y el alto de salida y el número de objetos dibujados, y después llámalo desde el adaptador de `Paint` del indicador y desde el

renderizador fuera de pantalla. Toma una medición de referencia, aplica una optimización que puedas defender y vuelve a registrar las mismas cifras.

- ✓ Añade las rutas degradadas: devuelve un elemento visual neutro de «sin datos» cuando el modelo esté vacío, sal de inmediato ante una superficie de tamaño cero, acota o rechaza un rango no válido antes de dibujar, y resuelve las fuentes mediante una cadena de alternativas para que una familia ausente produzca un marcador de posición legible en lugar de una excepción. Envuelve la exportación en `try / catch` para que un fallo llegue al usuario como un mensaje y no como una imagen en blanco.
- ✓ Construye la tabla accesible que enumera todos los valores que muestran los gráficos, dale al indicador una lectura visible más un `AccessibleDescription`, y después escribe `CapstoneNotes.md` con el diagrama de capas, dónde se ejecuta cada renderizador y qué cruza la red, la matriz de pruebas para Windows, contenedores Linux y los destinos Hybrid, y las mediciones de antes y después.

Antes de dar por terminado el proyecto final, ejecuta el guion de la demostración de principio a fin: cambia un valor de telemetría y observa cómo coinciden las cuatro superficies, redimensiona o recrea el Canvas y muestra que `Redraw` lo reconstruye, desplázate por la cuadrícula y explica el compromiso entre dibujo por el usuario y HTML, arrastra y amplía un nodo de topología y explica la prueba de impacto con coordenadas inversas, exporta un PNG y di en voz alta dónde se está ejecutando `System.Drawing.Managed` y, por último, presenta las mediciones y el límite de optimización que elegiste. La nota de arquitectura debe además una alternativa descartada por superficie, así que ten preparado por qué el indicador se dibuja en lugar de aplicarle un tema, por qué la columna de la cuadrícula la dibuja el usuario en lugar de usar HTML, por qué la topología es Canvas y no una imagen dibujada en el servidor, y por qué la exportación es fuera de pantalla y no una captura del navegador.

Entregables requeridos

- ✓ Capas Models, Geometry, Renderers y de adaptadores con un único `OperationsModel` que alimenta las cuatro superficies
- ✓ Ayudante `RenderMetrics` que registra la duración del renderizado, el tamaño del mapa de bits y el número de objetos dibujados por superficie
- ✓ Mediciones antes y después de una optimización, con el razonamiento de por qué la elegiste
- ✓ Rutas de fallo para datos vacíos, tamaño cero, rango no válido y fuente ausente que degradan en lugar de lanzar excepciones
- ✓ `CapstoneNotes.md` con la arquitectura, la frontera servidor/navegador, la matriz de pruebas y una tabla de datos accesible

Preguntas de repaso

- ✓ Tu capa de geometría recibe el modelo y un tamaño y devuelve formas, pero un compañero de equipo quiere que devuelva un `GraphicsPath` para que el renderizador del indicador quede más corto. ¿Qué les cuesta eso al renderizador de Canvas y a las pruebas unitarias, y dónde pondrías tú el límite?

- ✓ El PNG exportado y el Canvas en pantalla difieren en unos pocos píxeles en el agente de compilación Linux, pero coinciden en Windows. ¿Qué mediciones y qué tipo de prueba te dirían si eso es un defecto o una diferencia de plataforma esperada?
- ✓ Quien revisa tu trabajo dice que la tabla accesible es redundante porque todos los indicadores ya tienen un tooltip. Explica qué le da la tabla a quien usa solo el teclado o un lector de pantalla y el tooltip no, y qué implica eso para los chips de severidad que en la cuadrícula solo usan color.



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.