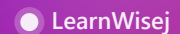


Die Zeichenarchitektur von Wisej.NET

Lektionsleitfaden · Lab- und Prüfungsleitfaden



Lektionsleitfaden

Vier Rendering-Pipelines und wie Sie zwischen ihnen wählen: das Paint-Event eines Controls, das ein auf dem Server gerendertes Bild ausliefert, DataGridView CellPaint innerhalb einer einzelnen Zelle, Wisej.Web.Canvas, das Zeichenbefehle an den Browser schickt, und das Off-Screen-Rendering mit System.Drawing.Managed, das Bild-Bytes erzeugt – dazu die Entscheidungsleiter, die Standard-Controls und CSS als Normalfall beibehält. 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

Vier Rendering-Pipelines und wie Sie zwischen ihnen wählen: das Paint-Event eines Controls, das ein auf dem Server gerendertes Bild ausliefert, DataGridView CellPaint innerhalb einer einzelnen Zelle, Wisej.Web.Canvas, das Zeichenbefehle an den Browser schickt, und das Off-Screen-Rendering mit System.Drawing.Managed, das Bild-Bytes erzeugt — dazu die Entscheidungsleiter, die Standard-Controls und CSS als Normalfall beibehält.

1. Das Problem im Produktivbetrieb

VisualOperationsStudio muss denselben Maschinenmesswert an vier Stellen anzeigen: als Rundinstrument auf der Betriebsseite, als Gesundheitsbalken in einer Grid-Zeile, als Knoten auf einer Topologiekarte und als PNG im Anhang des nächtlichen Schichtberichts. Jedes davon heißt „eine Zahl zeichnen“, und doch führt der Griff zu irgendeiner Zeichen-API, die man noch vom Desktop kennt, zu vier Stücken Code mit völlig unterschiedlichem Verhalten bei CPU, Bandbreite und Größenänderung. Dieses Modul stellt die Wahl der Oberfläche vor den Zeichencode, denn die Oberfläche entscheidet, wo die Pixel entstehen und was zum Browser wandert.

2. Zentrale Konzepte

Konzept	Warum es wichtig ist
<code>Paint</code> / <code>PaintEventArgs</code>	Das Zeichenereignis auf Control-Ebene: Ihr C# läuft auf dem Server gegen das bereitgestellte <code>Graphics</code> , innerhalb des gemeldeten <code>ClipRectangle</code> , und Wisej.NET macht daraus ein Bild für den Browser.
<code>DataGridView.CellPaint</code>	Derselbe Zeichenmechanismus, auf eine Zelle begrenzt; die <code>DataGridViewCellPaintEventArgs</code> leiten sich von <code>PaintEventArgs</code> ab und ergänzen <code>RowIndex</code> und <code>ColumnIndex</code> .
<code>UserPaint</code>	Der Schalter, den eine Zelle oder eine ganze Spalte braucht, bevor das Grid dafür <code>CellPaint</code> auslöst – so bleibt das Zeichnen pro Spalte ausdrücklich gewählt statt gridweit aktiv.
<code>Wisej.Web.Canvas</code>	Eine 2D-Browseroberfläche im Immediate Mode, die das HTML5-Modell <code>CanvasRenderingContext2D</code> spiegelt; die Zeichenaufrufe führt der Client aus, sie werden nicht auf dem Server in ein Bild gerendert.
<code>Redraw</code>	Das Canvas-Event, in dem die gesamte Szene aus Ihrem Modell neu aufgebaut wird, denn die Canvas hält Pixel und verliert sie, sobald sie neu erstellt oder in der Größe geändert wird.
<code>System.Drawing.Managed</code>	Off-Screen-Rendering ganz ohne sichtbares Control: eine <code>Bitmap</code> erzeugen, ihr <code>Graphics</code> über <code>Graphics.FromImage()</code> holen und die Bild-Bytes für einen Bericht, einen Export oder einen Download behalten.
<code>Invalidate()</code>	Der Aufruf, der einem gezeichneten Control sagt, dass seine Darstellung veraltet ist; er aktualisiert das Client-Widget und die Oberfläche im Visual Studio Designer und gehört deshalb dorthin, wo sich der Zustand geändert hat, und nicht in Zeichenhelfer.

3. Der rote Faden der Lektion

Vier Pipelines, die alle wie ganz normales C# aussehen

Die erste Pipeline ist das `Paint`-Event eines Controls, das jedes Wisej.NET-Control unterstützt. Der Handler erhält ein `Wisej.Web.PaintEventArgs`, dessen `Graphics`-Property eine `System.Drawing.Graphics`-Oberfläche ist und dessen `ClipRectangle` das gerade gezeichnete Rechteck benennt. Wisej.NET rendert den gezeichneten Hintergrund und schickt ihn an den Browser; die Dokumentation zum eigenen Zeichnen beschreibt das Ergebnis als PNG, das übertragen und anschließend aus dem Speicher entfernt wird. Genau das macht eine alte Routine im GDI+-Stil in einer Webanwendung ungewöhnlich gut wiederverwendbar.

Die zweite Pipeline ist `DataGridView.CellPaint`, spezialisiert auf Zellen. Eine Zelle oder eine ganze Spalte muss `UserPaint` auf `true` gesetzt haben; das Grid löst dann `CellPaint` mit `DataGridViewCellPaintEventArgs` aus, die das vertraute `Graphics` und `ClipRectangle` sowie `RowIndex` und `ColumnIndex` tragen. Sie eignet sich für kompakte Darstellungen — Status-Gauges,

winzige Balkendiagramme, Sparklines, Badges — und für Routinen, die sich als Markup nur umständlich nachbauen ließen.

Die dritte Pipeline ist `Wisej.Web.Canvas`, eine Browser-Canvas im Immediate Mode, deren API bewusst dem HTML5-Modell `CanvasRenderingContext2D` ähnelt. In der vierten steckt überhaupt kein Control: eine `Bitmap` erzeugen, über `Graphics.FromImage()` ein `Graphics` beziehen, zeichnen und das Bild speichern oder versenden. Wisej.NET 4 liefert `System.Drawing.Managed` mit, das den vertrauten Namespace `System.Drawing` auf modernen .NET-Zielen in verwaltetem Code implementiert. Im Lab dieses Moduls stellen alle vier dasselbe `TelemetrySample` dar, das die Seite hält.

Wo der Code läuft und was über die Leitung geht

Ein serverseitig gezeichnetes Control schickt Ihren `DrawLine()`-Aufruf nicht an den Browser. Der Server führt das Zeichnen gegen eine Grafikoberfläche aus, und Wisej.NET überträgt das fertige Bild — bequem, weil der Browser Ihren Algorithmus nie verstehen muss, und teuer, wenn die Fläche groß ist und sich häufig ändert, denn Server-CPU und Bytes fallen bei jedem Neuzeichnen an. Messen Sie das, bevor Sie ein gezeichnetes Control von einem Timer neu zeichnen lassen.

Canvas dreht das um. Die `Canvas`-API von Wisej bildet 2D-Canvas-Operationen des Browsers ab, und diese Operationen werden im Client ausgeführt. `LiveUpdate` steht standardmäßig auf `false`, sodass während einer Anfrage abgesetzte Befehle mit der normalen Wisej.NET-Aktualisierung reisen; auf `true` gesetzt, werden sie sofort gesendet, was zu schrittweisen Demonstrationen passt, aber keinen Ersatz für clientseitige Animation darstellt — ein Server-Roundtrip pro Bild ist die falsche Architektur. Beim Off-Screen-Rendering geht gar nichts über die Leitung, bis Sie die Bytes übergeben, und `CellPaint` nutzt denselben Bildmechanismus wie `Paint`, begrenzt auf eine Zelle.

Aufbewahrtes Modell, flüchtige Pixel

„Immediate Mode“ ist der Teil, über den viele stolpern. Eine Canvas merkt sich Pixel, nicht Ihre Geschäftsobjekte, und Wisej.NET spielt nach einer Aktualisierung nicht jede Linie und jedes Rechteck erneut ab. Halten Sie das Modell — Formen, Punkte, das ausgewählte Element, Zoom, Verschiebung, Farben — in ganz normalen C#-Objekten und machen Sie `Redraw` deterministisch: Aus diesem Modell und der aktuellen Canvas-Größe baut es die vollständige Darstellung neu auf. Im Lab ändern Sie die Browsergröße und halten fest, welche Oberflächen von selbst zurückkamen.

Paint-Handler verdienen dieselbe Disziplin. Behandeln Sie sie als Renderfunktionen: Zustand beobachten, ihn zeichnen, nichts verändern. Ändert sich der Messwert, aktualisieren Sie das Modell und invalidieren Sie das Control; verändern Sie den Messwert nie, nur weil ein Zeichenvorgang angefordert wurde. Zeichencode, der Zustand verändert, ist der Grund dafür, dass eine Darstellung am Ende davon abhängt, wie oft sie neu gezeichnet wurde.

Invalidate() an der Grenze einer Zustandsänderung

`Control.Invalidate()` lässt das Control sein Client-Widget aktualisieren und aktualisiert außerdem die gesamte Oberfläche im Visual Studio Designer. In einem selbst gezeichneten Control ist es der natürliche Aufruf, wenn sich eine Property ändert, die die Darstellung beeinflusst, und sein natürlicher Ort ist der Property-Setter, abgesichert, sodass ein unveränderter Wert nichts kostet:

```
private double _reading;

public double Reading
{
    get => _reading;
    set
    {
        value = Math.Clamp(value, 0, 100);
        if (_reading == value)
            return;

        _reading = value;
        this.Invalidate();
    }
}
```

Der Zeichencode darf nun davon ausgehen, dass der Wert gültig ist, und stellt einfach den aktuellen Zustand dar. Halten Sie die Invalidierung an diesen Grenzen, statt sie über tief liegende Helfer zu verstreuen, wo sie sich in Neuzeichnen-Stürme verwandelt, die niemand mehr nachverfolgen kann. Eine Entwurfsbeschränkung, die Sie einplanen sollten: Der Leitfaden zum eigenen Zeichnen sagt ausdrücklich, dass `BackgroundImage` und eigenes Zeichnen auf demselben Control nicht kombiniert werden können, weil das Zeichnen den Hintergrund übernimmt. Für ein Bild hinter Ihrer Zeichnung zeichnen Sie es selbst mit `Graphics.DrawImage()`.

Die Leiter, die eigenes Zeichnen zuletzt einordnet

Eigene Grafiken sollten die Antwort auf eine Anforderung sein, nie ein aus der Desktop-Arbeit mitgebrachter Reflex. Gehen Sie die Leiter der Reihe nach durch:

- Kann ein Standard-Control, ein Theme, ein Icon, ein Bild, ein Fortschrittsbalken, eine Diagramm-Erweiterung oder CSS das schon? Dann nehmen Sie das.
- Ist der Inhalt von Natur aus HTML und als DOM-Inhalt lesbar? Erwägen Sie Markup vor Pixeln.
- Ist es eine kleine Darstellung, die zum Hintergrund eines Controls gehört und sich in menschlichem Tempo ändert? `Paint`.
- Ist es eine kompakte Darstellung in einer Grid-Zelle? `CellPaint`, nachdem Sie es mit HTML in der Zelle verglichen haben.
- Ist es eine interaktive oder browsereigene Szene? `Canvas`. Ist die Ausgabe eine Datei statt eines Bildschirms? Off-Screen mit `System.Drawing.Managed`. Ist sie empfindlich für die Bildwiederholrate? Dann bleibt die Schleife im Client.

Zwei Prüfungen schließen die Entscheidung ab. Pixel tragen für assistive Technologien keine Bedeutung, also braucht alles Wichtige, das nur als Zeichnung existiert, einen parallelen Textkanal — ein Label, einen Tooltip, `AccessibleName` oder `AccessibleDescription`. Und Netzwerkübertragung plus Server-CPU gehören ins Zeichenbudget, genau wie Allokationen in das Budget einer heißen Schleife gehören. Ein gezeichnetes Gauge, das bei der Optik gewinnt und bei beidem verliert, ist eine Entscheidung, die Sie im Review verteidigen können sollten.



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:

- ✓ Control-Paint, DataGridView.CellPaint, Wisej.Web.Canvas und das Off-Screen-Rendering mit System.Drawing.Managed unterscheiden.
- ✓ Die Grenze zwischen Server und Browser für jede Zeichentechnik erklären.
- ✓ Verstehen, wann ein Neuzeichnen ein Bild sendet und wann Canvas Zeichenbefehle an den Browser schickt.
- ✓ Invalidate() gezielt einsetzen, wenn sich der visuelle Zustand ändert.
- ✓ Fälle erkennen, in denen HTML, CSS, ein Standard-Control von Wisej.NET oder eine Erweiterung die bessere Lösung ist als eigenes Zeichnen.

Lab- und Prüfungsleitfaden

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



Praktisches Lab

Bauen Sie die Shell von VisualOperationsStudio und stellen Sie einen einzigen Telemetriewert gleichzeitig auf vier Oberflächen dar. Erstellen Sie eine VisualOperationsPage, die genau eine TelemetrySample-Modellinstanz hält, und zeigen Sie deren Wert dann (1) als schlichtes Label mit ProgressBar, (2) in einem Panel mit Paint-Handler, der einen Balken aus PaintEventArgs.Graphics und e.ClipRectangle zeichnet, (3) in einer Wisej.Web.Canvas, deren Redraw-Handler denselben Balken aus dem Modell neu aufbaut, und (4) als Off-Screen-Bitmap, die mit Graphics.FromImage gerendert und in einer PictureBox angezeigt wird. Verdrahten Sie einen Button, der den Modellwert ändert und alle vier Oberflächen über Invalidate() und ein Canvas-Redraw aktualisiert, ändern Sie dann die Browsergröße und halten Sie fest, welche Oberflächen das überstehen und welche neu aufgebaut werden müssen.

Ausgangspunkt

Erstellen Sie ein neues Wisej.NET-4.x-Projekt namens VisualOperationsStudio oder öffnen Sie die leere Shell, falls Ihr Dozent eine bereitgestellt hat: eine Startseite, ein Standard-Theme und nirgends etwas von Hand Gezeichnetes. Alles in diesem Lab fügen Sie selbst hinzu, und die vier Oberflächen, die Sie hier bauen, sind genau die, die jedes spätere Modul zu einem echten Gauge, einem gezeichneten Grid, einem Topologie-Editor und einem PNG-Export ausbaut.

Vorgeschlagenes Vorgehen

- ✓ Fügen Sie eine `VisualOperationsPage` und eine kleine Klasse `TelemetrySample` hinzu, die den Maschinennamen, einen Messwert zwischen 0 und 100 und einen Zeitstempel hält. Erstellen Sie genau eine Instanz als Feld der Seite; jede Oberfläche muss aus dieser Instanz lesen, und keine darf eine eigene Kopie halten.
- ✓ Ordnen Sie vier beschriftete Bereiche auf der Seite an. Füllen Sie den ersten mit einem `Label` und einer `ProgressBar`, die an den Messwert gebunden sind, damit Sie eine Basislinie ohne Pixel haben, gegen die Sie die anderen drei vergleichen können.
- ✓ Setzen Sie ein `Panel` in den zweiten Bereich und behandeln Sie dessen `Paint`-Event. Zeichnen Sie den Balken aus `e.Graphics`, bemessen Sie ihn innerhalb von `e.ClipRectangle` und geben Sie die erzeugten Pens und Brushes frei. Lesen Sie den Messwert aus dem Modell und ändern Sie im Handler nichts.
- ✓ Platzieren Sie eine `Wisej.Web.Canvas` im dritten Bereich und verlagern Sie ihr gesamtes Zeichnen in den `Redraw`-Handler, damit die Szene aus dem Modell neu aufgebaut und nicht angesammelt wird. Rendern Sie im vierten Bereich eine Off-Screen-`Bitmap` mit `Graphics.FromImage` und zeigen Sie sie in einer `PictureBox` an.
- ✓ Fügen Sie einen Button hinzu, der dem Modell einen neuen Wert zuweist und danach jede Oberfläche aktualisiert: Label und Fortschrittsbalken aktualisieren, `Invalidate()` auf dem gezeichneten Panel aufrufen, die Canvas neu zeichnen und das Off-Screen-Bild innerhalb von

`try` / `catch` neu aufbauen. Ändern Sie dann die Größe des Browserfensters und schreiben Sie Ihre Notizen.

Geforderte Ergebnisse

- ✓ VisualOperationsPage mit einem TelemetrySample-Modell, aus dem jede Oberfläche liest
- ✓ Paint-Handler, der aus PaintEventArgs.Graphics innerhalb von e.ClipRectangle zeichnet
- ✓ Canvas-Redraw-Handler, der die gesamte Szene nach einer Größenänderung aus dem Modell neu aufbaut
- ✓ Off-Screen-Bitmap, die mit Graphics.FromImage gerendert und auf der Seite angezeigt wird
- ✓ SurfaceNotes.md mit der Notiz, was pro Oberfläche über das Netzwerk geht und welche Darstellungen eine Größenänderung überleben

Wiederholungsfragen

- ✓ Sie setzen den Messwert des Modells über den Button, vergessen aber den `Invalidate()` - Aufruf auf dem gezeichneten Panel. Label und Canvas aktualisieren sich. Erklären Sie genau, warum das Panel es nicht tut und was die Benutzerin so lange sieht, bis etwas anderes ein Neuzeichnen erzwingt.
- ✓ Ihre erste Canvas-Fassung zeichnet den Balken einmal beim Laden der Seite statt aus `Redraw`. Beschreiben Sie, was die Benutzerin nach einer Größenänderung des Browsers sieht, und warum sich die Off-Screen-`Bitmap` im vierten Bereich anders verhält.
- ✓ Ihr Reviewer fragt, warum der erste Bereich — das schlichte `Label` und die `ProgressBar` — überhaupt im Lab vorkommt. Antworten Sie mit den Kosten, die die anderen drei Oberflächen zahlen und er nicht, und nennen Sie eine Anforderung, die diese Kosten rechtfertigen würde.



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.