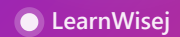


Architecture de dessin en production, performances et projet de synthèse

Guide de la leçon · Guide du lab / de l'examen



Guide de la leçon

Rendre le code de dessin livrable : un modèle visuel unique avec une couche de géométrie pure et des moteurs de rendu distincts par surface, l'instrumentation du temps de rendu et de la taille de la charge utile, le profilage d'un chemin dessiné face à un chemin hors écran, le comportement au redimensionnement, au changement de thème, sur données vides et sur police manquante, un équivalent textuel accessible, et le projet de synthèse Visual Operations Studio. Ce que cela signifie pour un développeur .NET expérimenté qui construit déjà des écrans Wisej.NET et a désormais besoin de graphiques personnalisés — et comment aborder ce module.



Objectif en production

Rendre le code de dessin livrable : un modèle visuel unique avec une couche de géométrie pure et des moteurs de rendu distincts par surface, l'instrumentation du temps de rendu et de la taille de la charge utile, le profilage d'un chemin dessiné face à un chemin hors écran, le comportement au redimensionnement, au changement de thème, sur données vides et sur police manquante, un équivalent textuel accessible, et le projet de synthèse Visual Operations Studio.

1. Le problème en production

VisualOperationsStudio possède désormais quatre surfaces de rendu, et chacune a été construite dans son propre module avec sa propre idée privée de ce que signifie la santé d'une machine. C'est ainsi qu'un sous-système de dessin pourrit : quelqu'un déplace un seuil d'avertissement, trois surfaces s'accordent, et le PNG exporté contredit discrètement l'écran dont il provient. Ce module ramène ces quatre interprétations à un seul modèle derrière une seule couche de géométrie, puis prouve, par des mesures, des chemins d'échec délibérés et un équivalent sémantique, que le résultat survit à de vraies données, à de vraies plateformes et à de vrais utilisateurs.

2. Concepts clés

Concept	Pourquoi c'est important
Modèle visuel	De l'état sémantique uniquement : mesures, légendes, nœuds et arêtes, séries de tendance, gravité comme choix métier et sélection partagée, pour qu'aucune surface n'ait besoin d'une copie.
Couche de géométrie	Des calculs presque purs qui transforment l'état du modèle et la taille courante de la surface en rectangles, angles de balayage, positions de libellé et régions de détection, testables sans aucun contrôle.
Adaptateur de surface	Un moteur de rendu par surface plutôt qu'une interface commune : <code>System.Drawing.Graphics</code> et <code>Wisej.Web.Canvas</code> gardent leurs primitives propres au-dessus d'une géométrie partagée.
Budget de performance	Un chiffre convenu avant tout réglage, par exemple un contrôle d'état dessiné sous 5 ms et un redessin complet du Canvas dans une fourchette d'environ 16 à 50 ms.
Instrumentation du rendu	Temps écoulé relevé, dimensions du bitmap, nombre d'objets dessinés et croissance de la charge utile par surface, pour qu'une optimisation se défende par un avant et un après.
Test d'image de référence	Un rendu hors écran à entrées fixes comparé à une référence stockée avec une tolérance, parce que l'anticrénelage et les encodeurs diffèrent d'une plateforme à l'autre.
Équivalent sémantique	Le libellé, l' <code>AccessibleDescription</code> , l'info-bulle ou la liste d'accompagnement qui portent chaque valeur essentielle montrée par les graphiques, pour que rien d'important n'existe uniquement en pixels.

3. Le fil de la leçon

Un modèle, une couche de géométrie, un moteur de rendu par surface

La conception réutilisable comporte trois couches. Le modèle visuel détient de l'état sémantique et rien d'autre. La couche de géométrie convertit cet état et les dimensions courantes de la vue en rectangles, points, tracés, positions de libellé et régions de détection. Le moteur de rendu convertit la géométrie en une surface, et c'est seulement là que quelque chose sait s'il s'adresse à un objet `Graphics` , à un `Wisej.Web.Canvas` ou à un fragment HTML. Un même modèle de topologie peut alors être affiché de façon interactive sur le Canvas, exporté en PNG via `System.Drawing.Managed` et listé pour l'accessibilité, sans que les règles métier existent en trois exemplaires.

Résistez à la tentation d'unifier les moteurs de rendu eux-mêmes : leurs sémantiques diffèrent réellement, et une interface limitée à ce que les deux savent faire coûte plus qu'elle ne rapporte, sauf si le visuel est trivial. Partagez plutôt le modèle et la géométrie. Pour `VisualOperationsStudio`, la restructuration est surtout un déménagement : les calculs de la jauge du Module 2 et les conversions en coordonnées monde du Module 5 rejoignent `Geometry` , tandis que le gestionnaire `Paint` , le gestionnaire `CellPaint` , la méthode `Redraw` du Canvas et le moteur de rendu hors écran deviennent de minces adaptateurs. La sélection rejoint elle aussi le modèle, et c'est ce qui permet à un clic sur une ligne de grille de recibler la jauge et de sélectionner le même nœud du Canvas.

D'abord les budgets, ensuite l'instrumentation

Fixez les chiffres avant de régler quoi que ce soit. Budgets de départ raisonnables : un contrôle d'état dessiné sous 5 ms de rendu serveur à taille normale, une colonne de grille dessinée par votre code sans à-coup visible au défilement pour un nombre de lignes représentatif, un redessin complet du Canvas dans une fourchette d'environ 16 à 50 ms, et un export qui peut durer plus longtemps parce que c'est une commande explicite, mais qui ne doit pas bloquer des sessions sans rapport. Ce que vous relevez diffère selon la surface :

- `Paint` et `CellPaint` serveur : temps écoulé du gestionnaire, taux d'allocation, taille de l'image temporaire, croissance de la charge utile de réponse et nombre de redessins par parcours.
- Canvas : commandes de dessin par redessin, taille de la charge utile de la scène, fréquence des redessins, latence aller-retour du pointeur et coût propre du navigateur sur une grande scène.
- `System.Drawing.Managed` hors écran : dimensions du bitmap et mémoire de pixels, temps d'encodage, comportement de chargement des polices, taille de sortie et différences visuelles par plateforme.

```
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);
}
```

Une fois l'enregistrement en place, profilez un chemin dessiné par le serveur face à un chemin hors écran avec les outils d'utilisation du CPU et d'allocation, plus des microbanques d'essai autour de la géométrie pure. Un gestionnaire rapide pour une cellule ne l'est pas automatiquement pour mille, les graphes d'objets par cellule sont le coupable habituel, et les sessions concurrentes appartiennent à la mesure, pas à ce qui vient après elle.

Deux matrices de tests : plateformes et navigateurs

Le rendu hors écran doit être vérifié sur chaque plateforme que vous promettez : un serveur Windows en .NET moderne, un conteneur Linux, macOS pour le développement et les tests là où c'est pris en charge, et iOS ou Android Hybrid si vous les livrez. Automatisez les métriques de texte, les polices exigées, les images transparentes, les dégradés, l'anticrénelage et l'enregistrement d'images, adossés à des sorties de référence comparées avec une tolérance plutôt qu'à une égalité d'octets, car un encodeur différent ou un demi-pixel d'anticrénelage est une différence de plateforme, pas une régression.

Le Canvas demande une seconde matrice, puisque c'est le navigateur qui effectue le rendu : vérifiez les versions courantes de Chromium, Firefox et Safari, en surveillant les polices, la composition, les ombres et les gestes tactiles. Redimensionnez tout pendant les tests, puisque `Redraw` doit reconstruire toute la scène depuis le modèle et qu'un gestionnaire `Paint` doit recalculer la géométrie depuis la largeur et la hauteur du moment. Étapez la suite comme le recommande le pack : des tests unitaires pour la normalisation, la conversion valeur vers angle, le hit testing, les transformations de zoom et l'élimination

hors champ ; des tests d'image de référence pour le moteur de rendu hors écran ; des tests d'intégration pour le rendu de la page, le redessin après redimensionnement, le défilement de la grille et les changements de thème ; des passages manuels pour le tactile, le clavier, les lecteurs d'écran, le zoom du navigateur et le haute densité.

Se dégrader au lieu de lever une exception, et ne jamais livrer du pixel seul

Dessinez volontairement les mauvais états. Une image manquante devient un glyphe de remplacement plus une entrée de journal, une police manquante se résout par une chaîne de repli définie, des données vides deviennent un visuel neutre « No data », un nombre invalide est borné ou rejeté avant le rendu, et un échec d'export remonte sous forme de message sur lequel l'utilisateur peut agir plutôt que d'une image blanche. Le thème mérite le même soin : évitez de coder en dur un fond blanc et du texte noir, acceptez les couleurs comme propriétés publiques ou résolvez-les depuis le thème de l'application, et testez un thème sombre autant qu'un thème clair.

Chaque valeur dessinée essentielle a aussi besoin d'un jumeau sémantique. Une jauge reçoit un libellé visible tel que « CPU 72% » accompagné d'une `AccessibleDescription`. Une sparkline reçoit un texte accessible du genre « Trend: up 12% over 7 days ». Une topologie Canvas reçoit un `TreeView` ou un `ListView` d'accompagnement avec sélection au clavier. Une gravité exprimée par la couleur a besoin d'une forme ou d'un mot à côté, parce que le rouge et le vert seuls n'atteignent pas tout le monde.

Le projet de synthèse Visual Operations Studio

Le projet de synthèse est une page unique avec quatre vues coordonnées : la `TelemetryGauge` dessinée pour l'équipement sélectionné, la grille d'exploitation avec ses colonnes Health et Trend dessinées par votre code, la topologie Canvas avec sélection, glisser, panoramique et zoom, et une commande Export qui produit un PNG via `System.Drawing.Managed`. La sélection est partagée, et l'export est généré depuis le modèle visuel par un moteur de rendu `System.Drawing`, jamais en tentant une capture d'écran du navigateur.

À côté de l'application en fonctionnement, vous remettez les preuves : où chaque moteur de rendu s'exécute et ce qui transite sur le réseau, la démonstration qu'aucun état de scène propre à une session ne réside dans un statique partagé par tout le processus, votre politique de libération des ressources et de polices, des mesures pour deux chemins de rendu avec une optimisation que vous pouvez défendre, un plan de vérification Windows plus conteneur Linux avec les implications pour iOS, Android et macOS, et l'alternative accessible. Parcourez ensuite la liste de contrôle finale de conception et répondez oui à chaque ligne, notamment sur le fait que le visuel se reconstruit depuis l'état du modèle, que la surface convient à la fréquence de mise à jour, et que l'animation à haute fréquence reste côté client.



Réflexe dessin : la surface d'abord

Choisissez la surface de rendu avant d'écrire le moindre appel de dessin. Demandez-vous où les pixels sont produits et ce qui transite sur le réseau : un gestionnaire Paint de contrôle s'exécute sur le serveur et expédie une image, CellPaint fait de même à l'intérieur d'une cellule de grille, Canvas envoie des commandes de dessin que le navigateur exécute et ne conserve rien après un redimensionnement, et System.Drawing.Managed produit des octets d'image sans aucun contrôle. Gardez le modèle sémantique dans des objets C#, faites de chaque moteur de rendu une fonction déterministe de ce modèle, n'invalidiez que lorsqu'une dépendance visuelle a changé, et laissez toujours un équivalent textuel de tout ce qui compte et n'existe que sous forme de pixels.



Objectifs d'apprentissage

À la fin de ce module, vous saurez :

- ✓ Construire un modèle visuel partagé avec des moteurs de rendu System.Drawing et Canvas distincts.
- ✓ Profiler le CPU de dessin, les allocations, les tailles d'images, le défilement de la grille et le comportement des sessions.
- ✓ Tester le rendu multiplateforme et le comportement de redessin du navigateur.
- ✓ Intégrer dès la conception l'accessibilité, l'export, le thème, la mise en page responsive et la gestion des erreurs à vos visuels personnalisés.
- ✓ Mener à bien le projet de synthèse Visual Operations Studio.

Guide du lab / de l'examen

Ce que vous allez construire dans le lab pratique, l'approche suggérée et les livrables attendus.

🎯 Lab pratique

Terminez le projet de synthèse VisualOperationsStudio en plaçant un seul modèle derrière les quatre surfaces. Restructurez le projet en dossiers Models, Geometry, Renderers et adaptateur Wisej pour que la jauge dessinée, les cellules de grille dessinées par votre code, la topologie Canvas et l'export PNG lisent tous le même OperationsModel et partagent la couche de géométrie pure, sans aucun état métier dupliqué par moteur de rendu. Ajoutez un helper RenderMetrics qui enregistre la durée de rendu, la taille du bitmap produit et le nombre d'objets de scène dessinés, et servez-vous-en pour comparer la jauge dessinée par le serveur au moteur de rendu hors écran, avant et après une optimisation que vous choisirez et justifiez. Éprouvez les chemins d'échec — données vides, surface de taille nulle, plage invalide, police manquante — afin que chacun se dégrade en un contenu de remplacement lisible plutôt qu'en exception, ajoutez un tableau accessible qui liste toutes les valeurs montrées par les graphiques, et rédigez les notes de synthèse couvrant l'architecture, la frontière navigateur/serveur et la matrice de tests.

Point de départ

VisualOperationsStudio tel que vous l'avez laissé après le Module 6 : un contrôle `TelemetryGauge` avec ses propres calculs de jauge, une grille d'exploitation dotée de colonnes Health et Trend en `UserPaint`, un éditeur de topologie Canvas détenant une scène par session avec hit testing, panoramique et zoom, et un `TopologyImageRenderer` qui écrit un PNG via `Graphics.FromImage`. Chaque surface fonctionne, mais chacune garde sa copie des seuils, des couleurs et des positions de nœuds, rien n'est mesuré, et une police manquante ou un jeu de données vide atteint encore le code de dessin sous forme d'exception.

Approche suggérée

- ✓ Créez d'abord les dossiers `Models`, `Geometry`, `Renderers` et adaptateur, puis faites migrer une surface à la fois vers un unique `OperationsModel` qui possède les équipements, leurs mesures, les nœuds et arêtes de topologie, les seuils de gravité et la sélection partagée. Supprimez au fur et à mesure la copie privée de chaque moteur de rendu, et conservez l'état de la scène sur la session plutôt que dans un champ statique.
- ✓ Sortez les calculs de coordonnées des quatre moteurs de rendu vers la couche de géométrie, pour qu'elle prenne l'état du modèle plus une taille et renvoie les limites de l'anneau, les angles de balayage, les rectangles de cellule, les rectangles de nœud et les positions des libellés. Testez unitairement à cet endroit la conversion valeur vers angle, la normalisation d'une plage inversée, le hit testing et l'élimination hors de la région visible, là où aucun contrôle Wisej n'est nécessaire.
- ✓ Écrivez `RenderMetrics` avec une méthode `Record` qui prend un nom de surface, le temps écoulé au `Stopwatch`, la largeur et la hauteur de sortie et le nombre d'objets dessinés, puis appelez-la depuis l'adaptateur `Paint` de la jauge et depuis le moteur de rendu hors écran. Relevez une mesure de référence, appliquez une optimisation que vous pouvez défendre, et relevez de nouveau les mêmes chiffres.

- ✓ Ajoutez les chemins dégradés : renvoyez un visuel neutre « No data » lorsque le modèle est vide, sortez immédiatement sur une surface de taille nulle, bornez ou rejetez une plage invalide avant de dessiner, et résolvez les polices par une chaîne de repli pour qu'une famille manquante produise un contenu de remplacement lisible plutôt qu'une exception. Enveloppez l'export dans un `try / catch` afin qu'un échec parvienne à l'utilisateur sous forme de message et non d'image blanche.
- ✓ Construisez le tableau accessible qui liste toutes les valeurs montrées par les graphiques, donnez à la jauge une mesure visible ainsi qu'une `AccessibleDescription`, puis rédigez `CapstoneNotes.md` avec le schéma des couches, l'endroit où chaque moteur de rendu s'exécute et ce qui transite sur le réseau, la matrice de tests pour Windows, les conteneurs Linux et les cibles Hybrid, et les mesures avant et après.

Avant de déclarer le projet de synthèse terminé, déroulez le scénario de démonstration de bout en bout : changez une valeur de télémétrie et regardez les quatre surfaces concorder, redimensionnez ou recréez le Canvas et montrez que `Redraw` le reconstruit, faites défiler la grille et expliquez le compromis entre dessin par votre code et HTML, faites glisser et zoomez un nœud de topologie et expliquez le hit testing par coordonnées inverses, exportez un PNG et dites à voix haute où `System.Drawing.Managed` s'exécute, puis présentez les mesures et la limite d'optimisation que vous avez choisie. La note d'architecture doit aussi une alternative écartée par surface : soyez prêt à dire pourquoi la jauge est dessinée plutôt que thémée, pourquoi la colonne de grille est dessinée par votre code plutôt qu'en HTML, pourquoi la topologie est un Canvas plutôt qu'une image dessinée par le serveur, et pourquoi l'export est hors écran plutôt qu'une capture d'écran du navigateur.

Livrables attendus

- ✓ Couches Models, Geometry, Renderers et adaptateur, avec un unique `OperationsModel` qui pilote les quatre surfaces
- ✓ Helper `RenderMetrics` enregistrant la durée de rendu, la taille du bitmap et le nombre d'objets dessinés par surface
- ✓ Mesures avant et après pour une optimisation, avec le raisonnement qui l'a fait choisir
- ✓ Chemins d'échec pour des données vides, une taille nulle, une plage invalide et une police manquante, qui se dégradent au lieu de lever une exception
- ✓ `CapstoneNotes.md` avec l'architecture, la frontière serveur/navigateur, la matrice de tests et un tableau de données accessible

Questions de révision

- ✓ Votre couche de géométrie prend le modèle et une taille et renvoie des formes, mais un coéquipier veut qu'elle renvoie un `GraphicsPath` pour raccourcir le moteur de rendu de la jauge. Que cela coûte-t-il au moteur de rendu Canvas et aux tests unitaires, et où placeriez-vous la limite à la place ?
- ✓ Le PNG exporté et le Canvas à l'écran divergent de quelques pixels sur l'agent de compilation Linux, mais concordent sous Windows. Quelles mesures et quel type de test vous diraient s'il s'agit d'un défaut ou d'une différence de plateforme attendue ?

- ✓ Un relecteur affirme que le tableau accessible est redondant puisque chaque jauge a déjà une info-bulle. Expliquez ce que le tableau apporte à un utilisateur au clavier seul ou avec lecteur d'écran que l'info-bulle n'apporte pas, et ce que cela implique pour les pastilles de gravité codées par la seule couleur dans la grille.



Réflexe dessin : la surface d'abord

Choisissez la surface de rendu avant d'écrire le moindre appel de dessin. Demandez-vous où les pixels sont produits et ce qui transite sur le réseau : un gestionnaire Paint de contrôle s'exécute sur le serveur et expédie une image, CellPaint fait de même à l'intérieur d'une cellule de grille, Canvas envoie des commandes de dessin que le navigateur exécute et ne conserve rien après un redimensionnement, et System.Drawing.Managed produit des octets d'image sans aucun contrôle. Gardez le modèle sémantique dans des objets C#, faites de chaque moteur de rendu une fonction déterministe de ce modèle, n'invalidiez que lorsqu'une dépendance visuelle a changé, et laissez toujours un équivalent textuel de tout ce qui compte et n'existe que sous forme de pixels.