Rapport d'Expert : Analyse Architecturale des Systèmes de Canevas Infini et Stratégies de Synchronisation

I. Introduction : Formalisation du Problème de Désynchronisation et Solution Architectural

Le développement d'un canevas infini (Infinite Canvas) est une entreprise complexe qui exige un équilibre précis entre la cohérence mathématique et la performance du rendu. L'objectif fondamental est de fournir un espace de travail dont les limites sont théoriquement illimitées, permettant une navigation fluide (panoramique et zoom).[1] La difficulté soulevée par l'utilisateur, à savoir la désynchronisation entre le fond (grille, arrière-plan) et les objets (blocs, formes, texte), est le symptôme le plus fréquent d'une architecture de vue non unifiée.

I.1. Définition et Exigences d'un Canevas Infini Efficace

L'efficacité perçue par l'utilisateur repose sur deux piliers techniques interdépendants. Premièrement, la **Synchronisation Rigoureuse**, où le fond, la grille, et tous les objets doivent se déplacer et s'adapter au zoom de manière parfaitement unifiée. Toute divergence, même minime, rompt l'illusion de l'espace unique. Deuxièmement, la **Performance Soutenue**, qui nécessite le maintien d'un taux de rafraîchissement élevé (idéalement 60 images par seconde) même lorsque la quantité de contenu augmente considérablement.[3] Des applications comme Logseq Whiteboards ou AFFiNE Edgeless Mode démontrent l'impératif de cette performance pour une expérience utilisateur sans friction.[5]

I.2. Diagnostic de la Désynchronisation : L'Omission du Modèle Dual

Le problème de désynchronisation survient typiquement lorsque le développeur tente de gérer indépendamment le mouvement du fond et celui des objets, souvent en appliquant des manipulations DOM ou des calculs de décalage distincts. Par exemple, si la grille est déplacée en ajustant background-position et que les objets sont déplacés en modifiant leur propriété left/top CSS sans une source de vérité unique pour les valeurs de transformation, la synchronisation est perdue lors du zoom ou des opérations de pan rapides.

La résolution architecturale de ce dilemme impose l'adoption d'une source de vérité unique : la **Matrice de Caméra**. L'architecture d'un canevas infini doit être conceptualisée comme un système où une "Caméra" (le Viewport) se déplace et zoome sur un monde fixe (le Monde/Page). Cette Matrice de Caméra unique est responsable de la transformation de toutes les coordonnées mondiales en coordonnées d'écran. Les applications professionnelles telles que Tldraw décrivent explicitement la nécessité de gérer le positionnement du viewport.[7] La transformation du viewport (Pan et Zoom) est l'unique état qui doit être appliqué à *tous* les éléments affichables — qu'il s'agisse de la grille d'arrière-plan, des formes vectorielles ou des blocs de texte. Si la caméra du fond se déplace sans que la même transformation ne soit appliquée aux objets au premier plan, la désynchronisation est inévitable.

II. Fondation Géométrique et Architecturale : Le Modèle Dual des Coordonnées et la Matrice de Caméra

Le secret d'une synchronisation parfaite réside dans la séparation logique stricte entre l'espace où les objets existent (le Monde) et l'espace où ils sont affichés (l'Écran).

II.1. Définition des Espaces de Coordonnées Essentiels

L'architecture nécessite la définition et la gestion active de deux espaces de coordonnées distincts :

1. **Coordonnées du Monde/Page (World Coordinates) :** Ce sont les coordonnées absolues où les objets du canevas "vivent". Elles sont indépendantes de l'état de zoom, de pan, ou de la taille de la fenêtre. Un bloc positionné à $(1000, 500)$ dans les

coordonnées du monde y restera toujours, assurant la permanence de la structure et des relations d'objets, quelle que soit la vue utilisateur.[7] Ces coordonnées sont essentielles pour la logique métier, le stockage des données, et la gestion des relations hiérarchiques, comme la structure en arbre des blocs dans Logseq.[8]

2. **Coordonnées de l'Écran (Screen/Viewport Coordinates) :** Ces coordonnées sont relatives à la fenêtre du navigateur ou à l'élément conteneur. Elles sont utilisées pour le rendu final en pixels et, de manière cruciale, pour capter les événements d'entrée utilisateur (position du clic, début du glisser-déposer). L'origine $(0, 0)$ est généralement le coin supérieur gauche du conteneur d'affichage.

II.2. Le Cœur de la Synchronisation : La Matrice de Transformation de la Vue (Camera Matrix)

La Matrice de Caméra ($M_{Caméra}$) est la structure de données centrale du système. Il s'agit d'une matrice 3x3 qui agrège l'état global du viewport, y compris la translation (pan), l'échelle (zoom), et la rotation.

$$P_{Screen} = M_{Caméra} \times P_{World}$$
Cette matrice sert de fonction de mappage universelle, transformant les positions fixes des objets dans le Monde (P_{World}) en positions affichables à l'Écran (P_{Screen}). Tous les objets visibles, ainsi que la grille de fond, doivent être rendus en utilisant cette unique matrice de transformation pour garantir la cohérence.

Table 1 : Distinction des Systèmes de Coordonnées pour le Canevas Infini

Caractéristique	Coordonnées de l'Écran (Viewport/Screen)	Coordonnées du Monde (World/Page)
Définition	Relatives à la fenêtre visible (Pixels). Origine $(0, 0)$ en haut à gauche.	Absolues, permanentes, définissent la position réelle des objets.
Impact du Zoom/Pan	Inchangées. Utilisées pour les événements d'entrée	L'espace est fixe ; l'affichage dépend

	(clic, glisser).	uniquement de $M_{Caméra}$.
Utilisation Principale	Calcul des événements utilisateur et rendu de l'UI non-canvastique.	Stockage de la position, de la taille, et de la rotation des blocs de contenu.[7]

II.3. Gestion des Interactions Utilisateur et la Matrice Inverse

Une fois la matrice de caméra définie, la désynchronisation des interactions devient le prochain point critique. Lorsqu'un utilisateur clique sur un objet, les coordonnées de ce clic sont fournies dans l'espace de l'Écran. Or, pour que la logique de l'application puisse identifier et manipuler l'objet, elle doit connaître sa position par rapport aux coordonnées du Monde.

Ce processus de sélection ou de manipulation nécessite de convertir le point de clic de l'Écran vers le Monde. Cette conversion est réalisée par l'application de la matrice inverse de la caméra, $M_{Caméra}^{-1}$:

$$P_{World} = M_{Caméra}^{-1} \times P_{Screen}$$
Sans ce calcul précis, le point de clic apparent de l'utilisateur serait incorrectement décalé par l'état du zoom, provoquant un décalage lors de la tentative d'édition, de sélection, ou de glisser-déposer. La matrice inverse est donc fondamentale pour garantir que les événements utilisateur dans l'espace d'affichage affectent correctement les objets dans leur espace logique (le Monde).

III. Le Langage des Transformations : Matrices Affines pour la Cohérence

L'utilisation des matrices de transformation affines est la méthode technique universelle adoptée par les systèmes graphiques 2D (y compris CSS, SVG, Canvas 2D et WebGL) pour manipuler des systèmes de coordonnées de manière cohérente et performante.

III.1. Structure et Composition des Transformations Affines

La matrice de transformation 2D la plus courante est une matrice 3x3 qui permet de combiner la translation, l'échelle, la rotation et l'inclinaison. Sa forme standard est :

$$M = \begin{pmatrix} a & c & e \\ b & d & f \\ 0 & 0 & 1 \end{pmatrix}$$
Dans cette structure, les valeurs (a, d) représentent les facteurs d'échelle sur les axes X et Y, tandis que (e, f) définissent la translation (l'offset ou le pan).[9] Pour garantir que $M_{Caméra}$ soit toujours la source unique de vérité, le Pan et le Zoom ne doivent pas être appliqués comme des manipulations séquentielles de style "position, puis zoom", mais doivent être multipliés dans une seule matrice globale. Cette composition matricielle unique est ensuite appliquée à l'ensemble de la scène, assurant une seule opération de rendu et prévenant les erreurs d'arrondi cumulatives souvent observées dans les systèmes qui gèrent les transformations par des variables séparées.

III.2. Implémentation via les Technologies Web

La puissance des matrices affines réside dans leur applicabilité directe aux APIs graphiques du web :

- **Canvas 2D :** L'API fournit la méthode CanvasRenderingContext2D.transform(a, b, c, d, e, f).[10] Cette fonction multiplie la matrice de transformation courante du contexte par les arguments fournis, déplaçant et mettant à l'échelle toutes les opérations de dessin subséquentes. Utiliser cette approche garantit que les formes, les lignes de grille, et le texte sont dessinés sous la transformation de la caméra.[10]
- **CSS / DOM :** Pour les objets basés sur le DOM ou SVG, la matrice de la Caméra est appliquée sur le conteneur parent (le "World Container") via la propriété transform: matrix(a, b, c, d, e, f).[9] Cette approche bénéficie d'une accélération matérielle par le GPU pour les transformations, ce qui est crucial pour la fluidité.
- **SVG :** Les éléments SVG gèrent nativement les transformations via l'attribut transform, utilisant la même convention matricielle.[9]

III.3. Précision Arithmétique et Rendu

Une considération majeure lors de l'utilisation des matrices est la gestion des nombres à virgule flottante. La désynchronisation (même légère) peut persister si les systèmes de conversion entre le Monde et l'Écran n'utilisent pas une haute précision. La standardisation

sur $M_{Caméra}$ appliquée partout minimise les erreurs.

Un choix critique dans le design des moteurs de rendu concerne l'affichage des lignes de grille. Pour garantir la netteté des lignes (pour éviter le flou de l'anti-aliasing), certains systèmes peuvent choisir d'arrondir les coordonnées finales à l'entier (pixel snapping) avant le dessin. Cependant, l'utilisation de l'arrondi pour la netteté introduit inévitablement de petits "sauts" (jumps) perceptibles pendant les opérations de zoom continu, car les lignes sont légèrement décalées lors du changement d'échelle.[12] La cohérence mathématique fournie par la matrice permet d'isoler ce compromis au niveau du rendu final, et non au niveau de la logique de positionnement des objets. La plupart des applications visant la fluidité maximale préfèrent le rendu en sous-pixels (flottants) pour les objets et la caméra, acceptant un léger flou au profit d'un mouvement parfait.

IV. Algorithmes de Manipulation de la Vue (Pan & Zoom Efficaces)

La synchronisation ne dépend pas seulement de la structure matricielle, mais également des algorithmes précis utilisés pour modifier cette matrice en réponse aux entrées utilisateur.

IV.1. Algorithme Fondamental du Panoramique (Panning)

Le panoramique (panning) est l'opération la plus simple, correspondant à une translation. Il implique la mise à jour des composantes e et f de la matrice $M_{Caméra}$.

Le calcul est direct : étant donné un changement de position du curseur ou du doigt ($\Delta X_{Screen}, \Delta Y_{Screen}$), le nouvel offset mondial (e', f') est calculé en ajoutant ce déplacement d'écran à l'offset précédent :

$$e' = e_{Old} + \Delta X_{Screen}$$
$$f' = f_{Old} + \Delta Y_{Screen}$$
Pour la grille (si elle est dessinée dynamiquement sur un Canvas), le panoramique doit également recalculer les coordonnées de début et de fin de dessin (startX, startY, finishX, finishY) basées sur l'offset actuel pour garantir que seules les parties visibles du canevas sont rendues, simulant l'infini sans latence.[13]

IV.2. Algorithme Critique du Zoom Centré

L'erreur la plus fréquente, qui mène à la désynchronisation, est de modifier le facteur d'échelle S (composantes a et d) sans ajuster proportionnellement les facteurs de translation e et f. Cela décale l'intégralité de la vue.

Le principe essentiel du zoom centré est de garantir que le point de l'écran sur lequel l'utilisateur effectue le zoom (le $P_{Ancrage, Screen}$) reste fixe *après* l'application du nouveau facteur d'échelle.

Étapes Mathématiques pour un Zoom Centré (ou Zoom-au-Curseur) :

1. **Identifier l'Ancrage du Monde :** Convertir la position de l'entrée utilisateur ($P_{Ancrage, Screen}$) en coordonnées du monde ($P_{Ancrage, World}$) en utilisant la matrice inverse actuelle $M_{Caméra, Old}^{-1}$.
2. **Mise à Jour de l'Échelle :** Mettre à jour le facteur d'échelle S_{New}.
3. **Correction d'Offset (La Clé de la Synchronisation) :** Le nouveau décalage (e_{New}, f_{New}) doit compenser le déplacement du point d'ancrage du monde dû à la mise à l'échelle.

La détermination du nouveau décalage est l'étape la plus critique. Elle doit prendre en compte le changement d'échelle lors de la mise à jour des offsets e et f.[14] Si l'on applique S_{New} à $P_{Ancrage, World}$, on obtient une position différente dans l'espace Screen transformé. Le décalage T_{New} est calculé pour ramener ce point transformé à la position originale $P_{Ancrage, Screen}$. C'est l'omission de cette correction d'offset qui provoque le "saut" indésirable de la vue après le zoom.

Cas Spécifique : Le Pinch-Zoom Tactile

Pour les périphériques tactiles, le facteur de zoom ($ZoomRatio$) est dérivé de la distance relative entre les doigts de l'utilisateur. Le calcul de la distance entre deux points touchés est effectué en utilisant le théorème de Pythagore : la distance actuelle entre les touches est comparée à la distance précédente.[14] L'ancrage ($P_{Ancrage, Screen}$) est alors fixé au point médian entre les deux doigts.[14]

V. Techniques de Rendu pour le Fond de Canevas Infini

Le fond du canevas, souvent une grille ou un motif répétitif, est l'élément le plus susceptible de se désynchroniser si son rendu n'est pas directement lié à la matrice $M_{Caméra}$. L'analyse révèle deux approches principales pour le rendu d'une grille infinie, performante et synchrone.

V.1. Rendu par CSS Background (Performance et Simplicité)

Cette technique exploite les capacités natives d'accélération GPU du navigateur. Elle est souvent considérée comme la solution la plus efficace pour les grilles simples.[15]

Mécanisme : Une image de fond (souvent un petit SVG ou Data URI définissant un point ou une ligne) est appliquée au conteneur principal, qui contient tous les objets du canevas.

Synchronisation :

1. **Zoom :** L'échelle de la grille est contrôlée par la propriété CSS background-size. Cette valeur doit être mise à jour dynamiquement en tant que $GridSize \times Scale_{Caméra}$ (où $Scale_{Caméra}$ est la composante a ou d de $M_{Caméra}$).
2. **Pan :** Le mouvement de la grille est géré par la propriété background-position, directement définie par l'offset de la Caméra (e et f).[15]

L'avantage majeur est la performance. Le navigateur gère la répétition et le rendu du fond de manière hautement optimisée, réduisant l'effort du développeur et le coût en ressources CPU.[15] Le code nécessaire est souvent plus concis et moins sujet aux erreurs de synchronisation.

V.2. Rendu Dynamique par Canvas 2D (Précision et Flexibilité)

Cette approche est utilisée lorsque le fond est plus complexe (par exemple, des motifs dynamiques, des zones de couleur).

Mécanisme : La grille est dessinée manuellement sur un élément Canvas 2D positionné sous les objets principaux du canevas.

Le Secret de l'Infini (Rendu Virtuel de la Grille) : Pour éviter de dessiner une grille de taille arbitrairement grande, la simulation de l'infini passe par le calcul précis de la zone visible (frustum). Le moteur doit calculer les coordonnées exactes des lignes ou points de grille qui

tombent dans la zone visible du viewport, plus un léger tampon pour le lissage du mouvement.

Cette méthode nécessite des calculs trigonométriques complexes pour déterminer les coordonnées de début et de fin du dessin (startX, startY, finishX, finishY) en s'assurant qu'elles sont des multiples de la taille de la grille (gridSize) et qu'elles tiennent compte du décalage de la caméra (currentOffsetPositions).[13]

Pour les implémentations de canevas qui utilisent principalement le DOM pour les objets, l'approche CSS Background est techniquement supérieure pour la grille simple en raison de sa performance et de sa synchronisation directe avec les transformations CSS appliquées au conteneur principal. Seule la nécessité d'un fond dynamique complexe justifie la complexité d'un rendu Canvas 2D dynamique.

VI. Choix des Moteurs de Rendu pour les Objets et Optimisation de la Performance

L'efficacité d'un canevas infini, particulièrement sous forte charge, dépend du choix du moteur de rendu pour les objets (texte, blocs, formes). Ce choix détermine la capacité de l'application à monter en charge.

VI.1. Analyse Comparative des Moteurs de Rendu d'Objets

Moteur	Avantages Clés	Inconvénients pour le Canevas Infini	Seuil de Performance / Cas d'Utilisation
HTML/CSS (DOM/SVG)	Gestion événementielle native [16] ; excellent rendu de texte ; intégration facile avec les frameworks modernes.	CPU-bound ; le re-layout est coûteux ; performance limitée au-delà de 10 000 éléments.[3]	Applications de faible à moyenne densité.

Canvas 2D	API 2D simple [16] ; bonne vitesse d'initialisation [3] ; utilise l'API matricielle.[10]	CPU-bound ; gestion manuelle des événements (hit testing) ; performance insuffisante à très haute échelle.[17]	Applications 2D intermédiaires ; MVPs.
WebGL (GPU)	Accélération matérielle (GPU) ; performance soutenue à haute échelle (50k+ éléments) [3] ; cohérence cross-platform.[16]	Temps d'initialisation plus long [3] ; complexité de développement et de débogage élevée.[16]	Projets exigeant une fluidité maximale et une haute densité (ex: outils CAD [16]).

L'analyse des benchmarks académiques et communautaires illustre un point de bascule critique. Pour des visualisations dépassant 10 000 éléments, Canvas et SVG peuvent chuter sous les 30 images par seconde (FPS), tandis que WebGL maintient sa performance élevée grâce à l'accélération GPU.[3] Un projet qui est destiné à gérer des dizaines de milliers de blocs ou d'annotations doit nécessairement s'orienter vers une architecture optimisée par GPU (WebGL) pour garantir l'efficacité à long terme. La décision architecturale concernant le moteur de rendu doit donc être prise très tôt, car elle est difficilement réversible.

VI.2. Stratégies d'Optimisation Obligatoires : Rendu Virtuel

Même le moteur le plus rapide (WebGL) sera ralenti si l'application tente de rendre l'intégralité du contenu du Monde en continu. Pour garantir l'efficacité quel que soit le nombre total d'objets, l'implémentation de techniques de rendu sélectif est essentielle.

Le **Rendu Virtuel** (souvent appelé *Virtual Rendering* ou *Frustum Culling*) est l'unique façon de simuler véritablement un canevas infini efficace.

Principe d'Implémentation : L'application doit d'abord calculer la zone visible (frustum) dans les coordonnées du Monde en utilisant les dimensions de l'écran et la matrice $M_{Caméra}$. Elle filtre ensuite la liste complète des objets stockés en coordonnées mondiales, ne transmettant au moteur de rendu que les objets dont la boîte englobante intersecte la zone visible, plus un petit tampon.

L'expérience montre que l'absence de rendu virtuel conduit à un lag important et à une forte consommation de mémoire pour les grands documents.[4] Par exemple, AFFiNE a identifié ce problème pour ses documents volumineux, nécessitant la mise en œuvre de cette optimisation pour maintenir une expérience utilisateur fluide, surtout pour les utilisateurs intensifs.[4]

Le concept de Rendu Virtuel est donc un impératif technique. Un canevas infini sans culling est intrinsèquement inefficace. Il permet de s'assurer que le nombre d'éléments effectivement dessinés reste sous un seuil gérable (par exemple, moins de mille objets), assurant la rapidité de l'interface indépendamment du nombre total d'objets existant dans la base de données.

VII. Études de Cas, Modèles Architecturaux et Recommandations d'Intégration

Les applications de référence modernes fournissent des modèles architecturaux éprouvés pour résoudre les problèmes de synchronisation et d'échelle.

VII.1. Le Modèle Tldraw : Architecture Découplée

Tldraw, une bibliothèque largement utilisée pour créer des expériences de canevas infini [19], est un excellent exemple d'architecture à double coordonnée.[7]

Cohérence Matricielle et Hiérarchie : Tldraw gère une distinction claire entre les coordonnées de l'Écran, qui gèrent le positionnement du viewport, et les coordonnées de la Page, qui conservent les relations entre les objets indépendamment du zoom et du pan.[7]

De plus, Tldraw utilise un système où chaque forme maintient sa propre matrice de transformation. Pour les objets regroupés, ces matrices se *composent* hiérarchiquement. La transformation appliquée au groupe parent est répercutée mathématiquement aux enfants. Cette structure est essentielle pour que les opérations complexes (rotation d'un groupe, ou déplacement d'un élément dans un groupe zoomé) restent cohérentes sans introduire de désynchronisation interne.[7]

VII.2. Recommandations d'Intégration et Plan d'Action

Pour intégrer un système de canevas infini fonctionnel et efficace, le projet doit s'articuler autour de la Matrice de Caméra comme état central.

Étape 1 : Isoler l'État de la Matrice de Caméra

Définir une variable d'état unique, $M_{Caméra}$, qui encapsule toutes les transformations (scale a, d et translation e, f). Cette matrice est la seule source de vérité pour la navigation et doit être mise à jour uniquement via les algorithmes de pan et de zoom centrés (Section IV).

Étape 2 : Synchronisation des Objets

L'application de $M_{Caméra}$ aux objets doit être systématique. Si le projet utilise des éléments DOM/SVG pour les blocs, il faut appliquer la matrice $M_{Caméra}$ via CSS transform: matrix(...) sur le conteneur englobant. Si un Canvas 2D est utilisé, la méthode context.transform(...) est appelée une seule fois par frame avant de dessiner tous les objets visibles.

Étape 3 : Rendu du Fond Cohérent

Pour la grille de fond, il est fortement recommandé d'utiliser l'approche CSS Background.[15] Il suffit de lier les propriétés CSS background-position et background-size directement aux composantes de translation et d'échelle de $M_{Caméra}$ (respectivement e, f et a, d). Cette méthode garantit une synchronisation intrinsèque et performante avec la transformation des objets.

Étape 4 : Mise en Œuvre du Rendu Virtuel et de l'Évolutivité

Pour garantir l'efficacité à long terme et la capacité à gérer un grand nombre de blocs, le Rendu Virtuel (culling) est impératif.[4] Le moteur doit implémenter une fonction de test de visibilité qui filtre les objets en coordonnées mondiales avant de les envoyer au moteur de rendu. Si l'objectif est de dépasser le seuil de 10 000 éléments, une transition vers un moteur accéléré par GPU (WebGL) sera nécessaire, en s'assurant que la logique de la matrice $M_{Caméra}$ reste la même, découplée du moteur de rendu sous-jacent.

Conclusion et Recommandations

Le problème de la désynchronisation du fond et des objets dans un canevas infini est un problème de référence de coordonnées mal géré. La solution réside dans une architecture graphique qui traite la vue comme une **Caméra Virtuelle**, dont l'état est totalement et uniquement défini par une **Matrice de Transformation Affine** ($M_{Caméra}$).

L'analyse technique révèle que l'efficacité et la fonctionnalité d'un tel système reposent sur l'adhérence aux principes suivants :

1. **Unification Matricielle :** Utiliser une matrice $M_{Caméra}$ unique pour transformer *tous* les éléments à afficher (grille, formes, texte) des coordonnées du Monde aux coordonnées de l'Écran.
2. **Gestion de l'Interaction :** Utiliser la matrice inverse ($M_{Caméra}^{-1}$) pour convertir les entrées utilisateur (clics, glisser) de l'espace de l'Écran vers l'espace du Monde pour la manipulation d'objets.
3. **Algorithme de Zoom Précis :** L'algorithme de zoom doit impérativement inclure une correction d'offset post-mise à l'échelle pour maintenir le point d'ancrage fixe, évitant ainsi le "saut" de la vue.
4. **Optimisation d'Échelle :** Pour une application efficace à grande échelle, le Rendu Virtuel (Frustum Culling) est obligatoire, quelle que soit la technologie de rendu choisie (DOM/Canvas/WebGL), afin de limiter le nombre d'éléments actifs au seul contenu visible par l'utilisateur.

Pour le projet immédiat, la meilleure approche pour intégrer l'interface demandée est d'isoler l'état $M_{Caméra}$ et d'appliquer cette matrice via les transformations CSS sur le conteneur principal. Pour le fond, l'utilisation d'une image de fond CSS dont la taille et la position sont mises à jour par les composantes de $M_{Caméra}$ est la méthode la plus simple et la plus performante pour garantir une synchronisation immédiate et fluide.

Sources des citations

1. tldraw: Infinite Canvas SDK for React, consulté le novembre 3, 2025, https://tldraw.dev/

2. AFFiNE: A Truly Wonderful Open Source Notion Alternative With a Focus on Privacy, consulté le novembre 3, 2025, https://news.itsfoss.com/affine/
3. Comparing Canvas vs. WebGL for JavaScript Chart Performance - DigitalAdBlog, consulté le novembre 3, 2025, https://digitaladblog.com/2025/05/21/comparing-canvas-vs-webgl-for-javascript-chart-performance/
4. Implement Virtual Rendering in AFFiNE · Issue #13616 · toeverything/AFFiNE - GitHub, consulté le novembre 3, 2025, https://github.com/toeverything/AFFiNE/issues/13616
5. Personal Knowledge Management App | AFFiNE Review (First Impression), consulté le novembre 3, 2025, https://gameandtechfocus.com/personal-knowledge-management-app-affine-first-impression/
6. NEW: Logseq Whiteboards | Get Started! - YouTube, consulté le novembre 3, 2025, https://www.youtube.com/watch?v=y3FqWPONN6s
7. Layout and composition - tldraw: Infinite Canvas SDK for React, consulté le novembre 3, 2025, https://tldraw.dev/features/composable-primitives/layout-and-composition
8. A whiteboard for the main concepts in Logseq - Look what I built, consulté le novembre 3, 2025, https://discuss.logseq.com/t/a-whiteboard-for-the-main-concepts-in-logseq/18423
9. Working with SVG Transformations - NickNagel.com, consulté le novembre 3, 2025, https://dr-nick-nagel.github.io/blog/trans-matrix.html
10. CanvasRenderingContext2D.transform() - Les API Web - MDN Web Docs, consulté le novembre 3, 2025, https://developer.mozilla.org/fr/docs/Web/API/CanvasRenderingContext2D/transform
11. CSS transforms - MDN Web Docs, consulté le novembre 3, 2025, https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_transforms
12. js canvas simulate infinite, pan- and zoomable grid - Stack Overflow, consulté le novembre 3, 2025, https://stackoverflow.com/questions/74307643/js-canvas-simulate-infinite-pan-and-zoomable-grid
13. Implementing Infinite Grid Dots in Vue Canvas with Panning System - Stack Overflow, consulté le novembre 3, 2025, https://stackoverflow.com/questions/77649396/implementing-infinite-grid-dots-in-vue-canvas-with-panning-system
14. Infinite HTML canvas with zoom and pan | Sandro Maglione, consulté le novembre 3, 2025, https://www.sandromaglione.com/articles/infinite-canvas-html-with-zoom-and-pan
15. Creating endless grid background in Fabric.js canvas with zoom and pan features, consulté le novembre 3, 2025, https://community.latenode.com/t/creating-endless-grid-background-in-fabric-js

-canvas-with-zoom-and-pan-features/37450
16. [Feature Request]: Improve the rendering files speed, on web interface and application for AFFiNE · Issue #13793 - GitHub, consulté le novembre 3, 2025, https://github.com/toeverything/AFFiNE/issues/13793
17. tldraw/tldraw: very good whiteboard SDK / infinite canvas SDK - GitHub, consulté le novembre 3, 2025, https://github.com/tldraw/tldraw
18. WhiteBoards Requests - Feature Requests - Logseq, consulté le novembre 3, 2025, https://discuss.logseq.com/t/whiteboards-requests/16288