Convergence of Contexts: Architecting Scalable Hybrid Frontend Systems in 2026

Executive Summary

The architectural paradigm of the modern web has shifted decisively towards "Hybrid Frontend Systems"—applications that seamlessly merge the structured, accessible document object model (DOM) with high-fidelity, hardware-accelerated graphics contexts (Canvas/WebGL/WebGPU). This report provides an exhaustive engineering analysis of the architectural patterns required to build, scale, and maintain these systems using the Next.js App Router ecosystem in 2026.

We analyze the dissolution of the boundary between "creative coding" and "application development." Where these disciplines once operated in silos—resulting in distinct, incompatible tech stacks—they now converge within the React reconciliation loop. However, this convergence introduces significant complexity regarding lifecycle management, main-thread contention, and asset delivery.

Key findings indicate that the Next.js App Router's nested layout system is the fundamental primitive for solving the "context destruction" problem, enabling persistent 3D scenes that survive route navigation. We evaluate the transition from JavaScript-driven animation libraries to the native View Transitions API, demonstrating a critical performance advantage in decoupling visual transitions from the CPU-bound initialization of heavy 3D assets. Furthermore, we define the 2026 standard for asset transmission, transitioning from simple file compression to GPU-ready formats like KTX2 and Draco, and explore how the Speculation Rules API has revolutionized preloading strategies by allowing background execution of future routes.

Finally, this report proposes a rigorous "Feature-Sliced" directory structure and a dual-state management pattern (utilizing Zustand for global app state and Valtio for high-frequency reactive interactions) that bridges the divide between the React DOM reconciler and the React Three Fiber render loop. This blueprint ensures that "creative" codebases remain maintainable at enterprise scale, capable of delivering 60fps experiences on mobile devices while satisfying the strict Core Web Vitals requirements of the modern web.

---

1. The Hybrid Convergence: Historical Context and

Technical Imperatives

The evolution of frontend architecture has historically been a pendulum swing between interactivity and accessibility. In the early 2020s, the "Creative Web" was characterized by "scroll-jacking," heavy loaders, and isolated WebGL experiences that felt distinct from the browser's native navigation. These experiences, while visually impressive, often suffered from poor SEO, accessibility failures, and fragile state management.

By 2026, the user expectation has matured. The novelty of 3D has been replaced by a requirement for "spatial interfaces" that feel as responsive as native applications. This demands a Hybrid System: a unified architecture where a 3D product configurator, a WebGL data visualization, or an immersive brand background coexists with a standard, indexable HTML interface.

1.1 The Fundamental Tension: DOM vs. Canvas

The core engineering challenge in a hybrid system is the reconciliation of two disparate rendering models:

1. **The DOM (Document Object Model):** A retained mode graphics system. The browser manages the painting of pixels based on a tree of objects (HTML elements). Updates are event-driven and typically occur only when state changes.
2. **The Canvas (WebGL/WebGPU):** An immediate mode graphics system (managed via libraries like Three.js). The application is responsible for clearing and redrawing the entire scene every frame (typically 60 or 120 times per second).

Merging these worlds within a framework like React (which is designed primarily for the DOM's lifecycle) creates friction. React's reconciliation process—calculating diffs and applying updates—competes for the main thread with the WebGL render loop. If React takes 50ms to diff a complex DOM tree during a route change, the 3D animation stutters, dropping frames and breaking the illusion of immersion.

1.2 The "Context Destruction" Problem

In traditional routing architectures (like the Next.js Pages router), navigating from one route to another destroys the component tree of the previous page and mounts the new one. For DOM elements, this is acceptable; re-creating a <div> is cheap.

For a WebGL context, however, destruction is catastrophic for performance.

- **The Cost of Disposal:** Destroying a 3D scene involves releasing GPU memory buffers, disposing of textures, and cleaning up event listeners.
- **The Cost of Re-initialization:** Mounting a new 3D scene requires re-compiling shaders (a blocking synchronous operation in WebGL), re-uploading geometry to the GPU, and re-parsing asset files.

This cycle of destruction and reconstruction results in a "flash of white" or a severe frame drop on every page load, rendering the "seamless app-like feel" impossible. The solution lies in **Persistence Architecture**—keeping the heavy 3D context alive while the lightweight DOM content swaps around it.

---

2. Architecting for Persistence: The App Router Paradigm

The Next.js App Router, introduced as stable in recent years and matured by 2026, provides the architectural primitives necessary to solve the Context Destruction problem natively. Unlike the flat hierarchy of the Pages router, the App Router utilizes a nested route tree defined by **Layouts** and **Templates**.

2.1 The Persistent Layout Pattern

A layout.tsx component in Next.js preserves its state and avoids re-rendering when a user navigates between routes that share that layout.[1] This behavior is the cornerstone of hybrid architecture.

Architectural Blueprint: The Global Scene Container

To achieve a persistent 3D world, the <Canvas> component must be hoisted to the highest common ancestor of all routes that require 3D content—typically the root layout (app/layout.tsx).

Implementation Strategy:

1. **Root Layout (app/layout.tsx):** Renders the global <Canvas> component. This component acts as the "Stage." It is mounted once when the application loads and never unmounts until the user leaves the application entirely.
2. **Z-Index Layering:** The Canvas is positioned via CSS to sit behind the DOM content (position: fixed; inset: 0; z-index: -1;).
3. **Event Passthrough:** A critical implementation detail is event handling. The DOM container overlaying the Canvas must have pointer-events: none to allow clicks and hovers to pass through to the 3D scene. Interactive DOM elements (buttons, links) must explicitly re-enable interactions with pointer-events: auto.

By decoupling the 3D Scene's lifecycle from the Page's lifecycle, we ensure that the WebGL context remains active. The "spinning cube" on the home page doesn't stop spinning when the user clicks "About"; it simply transitions to a new state.

2.2 The "Tunneling" Pattern for Route-Specific Content

While the Canvas is global, the *content* within it is often route-specific. The Home page might need a 3D Hero Model, while the Pricing page needs a 3D chart. Since these components are defined in separate route files (app/page.tsx, app/pricing/page.tsx), they cannot naturally be children of the layout.tsx Canvas.

To bridge this gap, modern hybrid architectures utilize a **Tunneling** or **Portal** pattern (often implemented via libraries like tunnel-rat).

The Mechanism:

1. **The Tunnel Setup:** A tunnel is created with two ends: <In> and <Out>.
2. **The Outlet:** The <Tunnel.Out /> is placed inside the persistent <Canvas> in the Root Layout.
3. **The Inlet:** The <Tunnel.In> is placed inside the route component (page.tsx).

When the user visits the Home page, the Home component mounts. It renders <Tunnel.In><HeroModel /></Tunnel.In>. The tunnel transports the <HeroModel /> to the <Tunnel.Out /> inside the Canvas.

When the user navigates to Pricing, the Home component unmounts (removing the <HeroModel />), and the Pricing component mounts, rendering <Tunnel.In><PricingChart /></Tunnel.In>.

Implication: The WebGL context, camera, and lighting (if global) remain untouched. Only the specific geometry for the page is swapped. This operation is lightweight compared to a full context reload, enabling instant, seamless transitions.

2.3 Layouts vs. Templates: A Critical Distinction

Architects must distinguish between layout.tsx and template.tsx.

- **Layouts:** Persist across routes. Use this for the Canvas.
- **Templates:** Re-mount on navigation. Use this for DOM elements that *should* trigger enter/exit animations (like a fading page header).[3]

Using a Template for the Canvas would negate the entire persistence strategy, causing the scene to reset on every navigation.

---

3. Router Logic and Intelligent Asset Loading

In a text-based web, "page weight" is measured in kilobytes of JavaScript and CSS. In a hybrid web, it is measured in megabytes of geometry and textures. The router logic must be sophisticated enough to conceal this weight from the user.

3.1 Code Splitting at the Interaction Boundary

Next.js provides automatic code splitting by route. However, hybrid applications often contain heavy 3D logic that isn't needed immediately upon route load but is required for specific interactions (e.g., a "Customize" button that opens a 3D editor).

Strategy: Interaction-Based Dynamic Loading Instead of importing the 3D editor at the top level, developers should use next/dynamic to load the chunk only when the interaction intent is signaled.[4]

TypeScript

```
// Architectural Pattern: Lazy 3D Feature
const ProductConfigurator = dynamic(
  () => import('@/features/configurator/CanvasScene'),
  {
    ssr: false, // Never render 3D on the server
    loading: () => <SceneLoader /> // Lightweight placeholder
  }
);

export default function Page() {
  const = useState(false);

  return (
    <>
      <button onClick={() => setShowConfig(true)}>Customize</button>
      {showConfig && <ProductConfigurator />}
    </>
  );
}
```

This pattern reduces the Initial Bundle Size, improving the Total Blocking Time (TBT) metric. The ssr: false flag is crucial: React Server Components cannot execute WebGL code. Attempting to render Three.js classes on the server will result in hydration mismatches and build failures.

3.2 The Speculation Rules API (2026 Standard)

The traditional <link rel="prefetch"> tag has been superseded by the **Speculation Rules API**, a browser-native feature that allows for granular control over preloading behavior.[6]

For hybrid apps, this API acts as a "Predictive Engine." It allows the developer to instruct the browser to pre-load resources or even pre-render entire pages based on heuristics (like the user hovering over a link).

Implementation for Hybrid Apps:

- **Prefetch Strategy:** We inject speculation rules to download the JSON data and JavaScript chunks for the next route's 3D assets.
- **Prerender Caution:** While "Prerender" (rendering the page in a background tab) is powerful for static sites, it is dangerous for hybrid apps. Spinning up a *second* WebGL context in a background tab can exhaust the GPU limit (browsers often limit contexts to 8-16 active contexts) and crash the active foreground scene.
- **Recommendation:** Use prefetch speculation rules for hybrid routes. Reserve prerender only for lightweight, text-only routes.

Example Speculation Rule Injection:

JSON

```
<script type="speculationrules">
{
 "prefetch": [
  {
   "source": "list",
   "urls": ["/projects/interactive-case-study"],
   "requires": ["anonymous-client-ip-when-cross-origin"]
  }
 ]
}
</script>
```

This ensures that the heavy .glb files and texture maps associated with the next project begin downloading *before* the user clicks, masking the network latency.[9]

---

4. The 2026 Asset Delivery Standard: The "JPEG of 3D"

The performance of a hybrid system is bound by the efficiency of its assets. A raw .obj file or a .png texture is an architectural failure in 2026. The industry standard has coalesced around the **glTF (GL Transmission Format)** container, augmented by advanced compression standards.

4.1 The Geometry Pipeline: Draco Compression

Geometry (the mesh data) can be massive. **Draco** compression is a library developed by Google that compresses geometry data by ~90-95%.[10]

- **Mechanism:** It quantizes vertex positions (reducing precision where invisible to the eye) and compresses connectivity data.
- **Decoding:** The decoding process happens on the client. Crucially, Three.js loaders (like useGLTF in React Three Fiber) offload this decoding to a **Web Worker**. This prevents the main thread from freezing while the model is parsed, maintaining UI responsiveness.
- **Architecture Rule:** All .glb assets in the public/ folder must be processed via a CI/CD pipeline using gltf-transform to apply Draco compression automatically before deployment.

4.2 The Texture Pipeline: KTX2 and Basis Universal

The most common cause of crashes in hybrid mobile web apps is **VRAM Exhaustion**.

- **The Problem:** A 200KB JPEG image is highly compressed on disk. However, the GPU cannot read JPEG directly. It must be decompressed into a raw bitmap. A 4K texture ($4096 \times 4096 \times 4$ bytes) consumes roughly **67 MB** of VRAM. A scene with 10 such textures consumes nearly 700 MB, causing mobile browsers to crash (reload the page).
- **The Solution: KTX2** textures (using the Basis Universal standard).
- **The Innovation:** KTX2 textures remain compressed *in VRAM*. The GPU reads them directly using specialized hardware decoders.
- **Format Transcoding:** The Basis Universal format is "universal" because it is transcoded at runtime to the specific format supported by the user's hardware (e.g., ASTC for Apple/Android devices, BC7 for Desktop PCs).[12]
- **Impact:** Memory usage drops by 6x to 8x. A 67 MB texture becomes ~8 MB. This is the difference between a functional app and a crashing one on an iPhone.

Architectural Requirement:

The asset pipeline must convert all source textures (PNG/JPG) to .ktx2.

- **UASTC:** Use for "Hero" textures (high quality, normals, UI elements).
- **ETC1S:** Use for "Background" textures (high compression, roughness maps, environment maps).

4.3 Table: 2026 Asset Standards Comparison

Asset Type	Legacy	2026	Compressi	VRAM	Decoding

	Format	Standard	on	Impact	Thread
Model	.obj / .fbx	.glb (glTF 2.0)	Draco	Moderate	Web Worker
Texture	.png / .jpg	.ktx2	Basis Universal	**Low (6-8x reduction)**	Web Worker
Environment	.hdr / .exr	.exr (half-float)	None / DWAA	High	Main Thread
Delivery	Direct File	CDN + Speculation	Gzip/Brotli	N/A	Network

---

5. The Animation Interface: View Transitions vs. Framer Motion

Transitioning between routes in a hybrid app is a high-risk operation. The CPU is spiking due to React reconciliation and Three.js initialization. If the animation logic also runs on the main thread, the result is "jank" (stuttering).

5.1 The View Transitions API: The S-Tier Standard

The **View Transitions API** is the native browser standard for page transitions in 2026.[14] It fundamentally changes the architecture of navigation animations.

The Mechanism:

1. **Snapshot:** When a navigation is triggered, the browser captures a raster screenshot of the current page.
2. **Freeze:** The rendering is paused.
3. **Update:** The DOM is updated to the new route.
4. **Snapshot New:** The browser captures the new state.
5. **Compositor Animation:** The browser animates between the two screenshots (cross-fade, slide, morph) on the **GPU Compositor Thread**.

Why it wins for Hybrid Apps: Because the animation happens on the Compositor Thread, it is immune to Main Thread blocking. Even if the JavaScript thread is locked up for 200ms parsing a massive GLTF file for the new route, the visual cross-fade remains buttery smooth at 60fps.[14]

Next.js Integration: Libraries like next-view-transitions patch the Next.js router to trigger document.startViewTransition automatically.[17]

5.2 Framer Motion: The Interaction Layer

While View Transitions handle the "Page" level, **Framer Motion** remains the standard for "Component" level interactions.[16]

- **Layout Animations:** When a user clicks a card and it expands, Framer Motion's FLIP (First, Last, Invert, Play) engine calculates the transform required to simulate the layout change.
- **3D Integration:** The framer-motion-3d library allows developers to treat Three.js meshes like DOM nodes.

  JavaScript
  ```
  <motion.mesh
    animate={{ x: isActive? 1 : 0 }}
    transition={{ type: "spring" }}
  />
  ```

  This unifies the animation language. The same spring physics driving the HTML button can drive the 3D button, creating a cohesive "feel" across the hybrid boundary.

5.3 Decision Tree for Animation Tools

Requirement	Recommended Tool	Execution Context	Reason
Route Navigation	View Transitions API	GPU Compositor	Immune to JS main thread blocking; native performance.
UI Micro-interactions	Framer Motion	Main Thread (JS)	Rich physics, gesture support (drag/hover).
Complex 3D Sequences	GSAP (GreenSock)	Main Thread (JS)	Timeline control, precise sequencing, no React overhead.
High-Frequency	React Spring /	Main Thread (JS)	Efficient

Updates	Maath		frame-loop integration for direct math updates.

---

6. State Management: The Nervous System

A hybrid app effectively runs two applications simultaneously: the React DOM app and the React Three Fiber (R3F) app. While they share a build pipeline, they render to different roots. Bridging the state between them is critical.

6.1 The Bridge Pattern

React Context does not automatically permeate the boundary between the react-dom renderer and the react-three-fiber renderer because they are separate reconciliation trees. While R3F v8+ supports Context Bridging, complex state management requires an external store that exists outside the React tree.[19]

6.2 The Dual-Store Architecture

The industry best practice for 2026 leverages a dual-store approach: **Zustand** for global application state and **Valtio** for high-frequency interaction state.[20]

Zustand: The Global Backbone

Zustand is an immutable, hook-based state manager. It is predictable and debuggable (via Redux DevTools).

- **Role:** Manages "Macro" state.
  - Current active route.
  - User theme preference (Dark/Light).
  - Asset loading progress (Global loader).
  - UI Overlay visibility (Menu open/closed).
- **Next.js Alignment:** Zustand stores must be created per-request or carefully scoped in Server Side Rendering (SSR) environments to prevent state pollution between users. The pattern of creating a store within a generic React Context provider is recommended for RSC compatibility.[21]

Valtio: The Reactive Proxy

Valtio uses JS Proxies to create mutable state.

- **Role:** Manages "Micro" state and 3D interactions.
  - Mouse position (normalized coordinates).

    - Scroll progress (0 to 1).
    - Product Configuration (Color: Red, Material: Leather).
- **Advantage in 3D:** Valtio allows for direct mutation (state.rotation.x = 0.5) inside the render loop without the boilerplate of actions/reducers. Its "fine-grained subscription" model means that if a component only reads state.color, it will not re-render when state.rotation changes. This is vital for performance in a scene with hundreds of objects.[20]

6.3 Transient Updates vs. State Updates

For 60fps animations (like a model following the mouse), updating a React state variable (which triggers a full component re-render) is too slow.

Transient Updates: The architecture should allow writing directly to refs.

- **Pattern:** Use a store (like Valtio) to hold the value. Inside useFrame, read the value and apply it to the mesh.current.rotation. Do *not* trigger a React render. This bypasses the Reconciler entirely, communicating directly with the WebGL layer.

---

7. Component Structure for Creative Developers

The rigid file-system routing of Next.js can clash with the experimental, iterative nature of creative coding. A scalable architecture requires a "Feature-Sliced" directory structure that enforces separation of concerns while keeping related logic colocated.[23]

7.1 The "Renderer-Aware" Directory Structure

We strictly separate components based on their rendering target. Mixing HTML tags into a Canvas component causes runtime errors.

Plaintext

```
src/
├── app/                 # Next.js App Router (Routing & Layouts)
│   ├── layout.tsx        # Global Layout (mounts the Canvas)
│   ├── page.tsx          # Home Page (mounts <Tunnel.In>)
│   └── projects/         # Project Routes
│
├── components/           # Shared Primitives
│   ├── dom/              # HTML-only Components
│   │   ├── Button.tsx    # Accessible <button>
│   │   └── Navigation.tsx      # Semantic <nav>
│   ├── canvas/                # WebGL-only Components
│   │   ├── Scene.tsx          # <Canvas> setup
│   │   ├── Lights.tsx         # <ambientLight>, <spotLight>
│   │   └── Effects.tsx        # <EffectComposer>
│   └── layout/                # Structural Bridges
│       └── View.tsx           # The Tunnel Component (Bridge)
│
├── features/                  # Domain-Specific Logic (Colocated)
│   └── product-configurator/
│       ├── dom/               # UI Overlays (Color Picker)
│       ├── canvas/            # 3D Model Logic (ShoeMesh)
│       ├── store.ts           # Feature-local State (Zustand/Valtio)
│       ├── glsl/              # Shaders
│       │   ├── vertex.glsl
│       │   └── fragment.glsl
│       └── assets/            # Feature-specific assets (optional)
│
├── lib/                       # Utilities
│   ├── hooks/                 # Custom Hooks (useScroll, useMouse)
│   └── shaders/               # Shared Shader Chunks (Noise, Math)
│
└── public/                    # Static Assets (CDN Root)
    └── assets/
        ├── models/            # Optimized GLB files
        └── textures/          # KTX2 files
```

7.2 The Philosophy of Colocation

In the features/ directory, we group code by *business logic* rather than *file type*. A "Product Configurator" feature requires a 3D model, a 2D UI overlay, and a shared state to sync them.

- **Benefit:** If we delete the product-configurator folder, the entire feature is gone. No orphaned CSS files or unused 3D models left behind.
- **Shader Management:** Shaders are code. We use .glsl files with a loader (like glslify) to allow imports. vertex.glsl can import noise.glsl from src/lib/shaders. This brings software engineering principles (modularity, reuse) to graphics programming.[26]

---

8. Performance Engineering and Future Horizons

Scaling a hybrid app isn't just about adding features; it's about maintaining the "Frame Budget." A browser has 16.6ms to render a frame for 60fps.

8.1 React Server Components (RSC) as a Performance Tool

RSCs cannot render 3D, but they are essential for *preparing* the 3D scene.[27]

- **Waterfall Elimination:** In a client-side app, the browser loads JS -> parses JS -> fetches API data -> parses JSON -> loads GLTF.
- **RSC Optimization:** The Server Component fetches the data (database) and resolves the asset URLs *before* sending HTML to the client. The client receives the locations of the GLTF files immediately in the initial HTML payload. The 3D scene can begin loading assets the moment the JS hydrates, effectively parallelizing the network requests.

8.2 The Rise of WebGPU

As of 2026, the transition from WebGL2 to WebGPU is mature. Three.js supports the WebGPURenderer.[11]

- **Compute Shaders:** WebGPU unlocks "Compute Shaders," which allow the GPU to perform general-purpose calculations. This enables massive particle systems (millions of particles) or complex physics simulations to run entirely on the GPU, freeing up the CPU Main Thread for React's reconciliation work.
- **TSL (Three Shader Language):** To support both WebGL (legacy fallback) and WebGPU (modern standard), we use TSL. This node-based material system compiles to *both* GLSL and WGSL. Adopting TSL is the primary "Future-Proofing" strategy for 2026 creative codebases.

8.3 Core Web Vitals in 3D

- **INP (Interaction to Next Paint):** Heavy 3D hydration can block the main thread, causing poor INP scores.
  - **Mitigation:** Use React 19/20 concurrency. Wrap the 3D scene hydration in startTransition. This tells React that the 3D scene is "low priority" compared to the user clicking a navigation button.
- **CLS (Cumulative Layout Shift):** The Canvas must have explicit dimensions.
  - **Mitigation:** The DOM container for the Canvas should have fixed CSS aspect ratios or absolute positioning to ensure it reserves space before the WebGL context initializes.

Conclusion

The architecture of a scalable Hybrid Frontend System in 2026 is defined by the intelligent separation of concerns. We utilize the Next.js App Router's **Persistent Layouts** to maintain the integrity of the WebGL context. We leverage the **View Transitions API** to decouple visual continuity from main-thread processing. We adopt a strict **Asset Pipeline** based on KTX2 and Draco to respect the limitations of mobile hardware. And we employ a **Feature-Sliced** directory structure to tame the complexity of managing two distinct rendering paradigms.

This blueprint transforms the "Creative Web" from a niche of experimental, fragile demos into a robust, enterprise-grade platform capable of delivering the next generation of spatial user experiences.

End of Report.

Referências citadas

1. Getting Started: Project Structure | Next.js, acessado em fevereiro 16, 2026, https://nextjs.org/docs/app/getting-started/project-structure
2. Persistent Layout Patterns in Next.js - Adam Wathan, acessado em fevereiro 16, 2026, https://adamwathan.me/2019/10/17/persistent-layout-patterns-in-nextjs/
3. Routing: Pages and Layouts - Next.js, acessado em fevereiro 16, 2026, https://nextjs.org/docs/13/app/building-your-application/routing/pages-and-layouts
4. Code splitting with dynamic imports in Next.js | Articles | web.dev, acessado em fevereiro 16, 2026, https://web.dev/articles/code-splitting-with-dynamic-imports-in-nextjs
5. Mastering Code Splitting in Next.js App Router - DEV Community, acessado em fevereiro 16, 2026, https://dev.to/ahr_dev/mastering-code-splitting-in-nextjs-app-router-2608
6. Speculation Rules API 2026: Prefetch and Prerender for WordPress | performance | WPPoland, acessado em fevereiro 16, 2026, https://wppoland.com/en/speculation-rules-api-wordpress-woocommerce-performance-2026-en/
7. Speculation Rules API - Web APIs | MDN - Mozilla, acessado em fevereiro 16, 2026, https://developer.mozilla.org/en-US/docs/Web/API/Speculation_Rules_API
8. Boost Speed Speculation Rules API: Prerendering, Prefetching - Telerik.com, acessado em fevereiro 16, 2026, https://www.telerik.com/blogs/boost-site-speed-speculation-rules-api-guide-prerendering-prefetching
9. the web performance experts - Speculation Rules API: Instant Page Loading - Catch Metrics, acessado em fevereiro 16, 2026, https://www.catchmetrics.io/blog/speculation-rules-api-instant-page-loading
10. What is glTF? | A Beginner's Guide, acessado em fevereiro 16, 2026, https://wpconfigurator.com/blog/what-is-gltf/
11. 100 Three.js Tips That Actually Improve Performance (2026) - Utsubo, acessado em fevereiro 16, 2026, https://www.utsubo.com/blog/threejs-best-practices-100-tips
12. KTX2 compression artifacts - Questions - three.js forum, acessado em fevereiro 16, 2026, https://discourse.threejs.org/t/ktx2-compression-artifacts/66037
13. Khronos KTX 2.0 Textures Enable Compact, Visually Rich, glTF 3D Assets, acessado em fevereiro 16, 2026,

https://www.khronos.org/news/press/khronos-ktx-2-0-textures-enable-compact-visually-rich-gltf-3d-assets
14. Mastering Smooth Page Transitions with the View Transitions API in 2026 - DEV Community, acessado em fevereiro 16, 2026, https://dev.to/krish_kakadiya_5f0eaf6342/mastering-smooth-page-transitions-with-the-view-transitions-api-in-2026-31of
15. What's new in view transitions (2025 update) | Blog | Chrome for ..., acessado em fevereiro 16, 2026, https://developer.chrome.com/blog/view-transitions-in-2025
16. The Web Animation Performance Tier List - Motion Magazine, acessado em fevereiro 16, 2026, https://motion.dev/magazine/web-animation-performance-tier-list
17. shuding/next-view-transitions: Use CSS View Transitions ... - GitHub, acessado em fevereiro 16, 2026, https://github.com/shuding/next-view-transitions
18. Comparing the best React animation libraries for 2026 - LogRocket Blog, acessado em fevereiro 16, 2026, https://blog.logrocket.com/best-react-animation-libraries/
19. React-Three-Fiber General Pattern? - Questions, acessado em fevereiro 16, 2026, https://discourse.threejs.org/t/react-three-fiber-general-pattern/30834
20. State Management: Zustand vs Valtio - Caisy, acessado em fevereiro 16, 2026, https://caisy.io/blog/zustand-vs-valtio
21. Setup with Next.js - Zustand, acessado em fevereiro 16, 2026, https://zustand.docs.pmnd.rs/guides/nextjs
22. Proper way to save server-side property to Zustand store : r/nextjs - Reddit, acessado em fevereiro 16, 2026, https://www.reddit.com/r/nextjs/comments/1hrqcmc/proper_way_to_save_serverside_property_to_zustand/
23. Next js project structure: Master the setup for scalable Next.js apps - Magic UI, acessado em fevereiro 16, 2026, https://magicui.design/blog/next-js-project-structure
24. Level Up Your Next.js Project: A Folder Structure That Sparks Joy (and Sanity!) - Medium, acessado em fevereiro 16, 2026, https://medium.com/@mahartha.gemilang/level-up-your-next-js-project-a-folder-structure-that-sparks-joy-and-sanity-0793e3885f84
25. Best Practices for Organizing Your Next.js 15 2025 - DEV Community, acessado em fevereiro 16, 2026, https://dev.to/bajrayejoon/best-practices-for-organizing-your-nextjs-15-2025-53ji
26. Mastering Next.js 15+ Folder Structure: A Developer's Guide | by TechTales by Hari, acessado em fevereiro 16, 2026, https://medium.com/@j.hariharan005/mastering-next-js-15-folder-structure-a-developers-guide-b9b0461e2d27
27. Mastering Code Splitting in Next.js App Router | by Abdul Halim | Medium, acessado em fevereiro 16, 2026, https://medium.com/@ahr.web.pro/mastering-code-splitting-in-next-js-app-router-3b9f6bf9f40b

28. Vercel Customers and Case Studies, acessado em fevereiro 16, 2026, https://vercel.com/customers