The 2026 High-Performance WebGL Stack: A Comprehensive Architectural Report for React Developers

1. Introduction: The Post-WebGL Era and the React Ecosystem

The landscape of real-time 3D graphics on the web has undergone a paradigmatic shift as of early 2026. The convergence of **Next.js 15**, **React 19**, and the maturation of **WebGPU** has established a new baseline for high-performance immersive web applications. The era of pure WebGL reliance is fading, replaced by a hybrid stack where **Three Shading Language (TSL)** acts as the lingua franca, enabling seamless execution across next-generation GPU pipelines and legacy WebGL 2 contexts.[3]

For React developers, the challenge has shifted from mere implementation to architectural orchestration. Integrating **React Three Fiber (R3F)** into the React Server Components (RSC) paradigm introduced by Next.js 15 requires a rigorous separation of concerns—serializing scene graphs on the server while deferring the heavy lifting of rendering and interaction to optimized client-side bundles. Simultaneously, the visual fidelity demanded by modern users—characterized by fluid liquid distortions, volumetric noise gradients, and cinematic post-processing—must be balanced against the thermal and battery constraints of mobile devices.[5]

The transition to 2026 standards is not merely an upgrade of dependencies; it is a fundamental rethinking of the rendering pipeline. The browser is no longer just a document viewer but a high-performance application runtime capable of compute-heavy tasks previously reserved for native applications. This report defines the definitive "High-Performance WebGL Starter Kit" for 2026, exhaustively analyzing the integration of the R3F ecosystem with the Next.js App Router, detailing the migration from GLSL to TSL for forward-compatible shaders, and prescribing a strict asset optimization pipeline using **glTF-Transform**, **Draco**, and **KTX2**.

---

2. Core Architecture: Next.js 15 and React Three Fiber Integration

The release of Next.js 15 stabilized the App Router and solidified the role of React Server Components (RSC), fundamentally changing how 3D applications are structured. In 2026, the naive approach of importing a <Canvas> component directly into a server page is a known anti-pattern that leads to hydration errors and performance bottlenecks. The high-performance stack requires a distinct "Hollow Shell" architecture.

2.1 The "Hollow Shell" Architecture and RSC Serialization

React Three Fiber (R3F), by definition, binds to the host environment's canvas element, a DOM node that does not exist on the server. Consequently, R3F is strictly a client-side library. However, Next.js 15's strict RSC enforcement means developers must explicitly delineate the boundary between the server-rendered application shell and the interactive 3D scene.[7]

The "Hollow Shell" pattern treats the 3D scene as a leaf node in the render tree. The server handles data fetching—such as product details, texture URLs, and scene graph configurations—and passes this data as serialized props to a Client Component wrapper. This separation allows the server to handle the logic of *what* to render, while the client handles *how* to render it.

Server Responsibility:

- **Data Fetching:** Retrieving glTF URLs, environment map paths, and configuration objects from a CMS or database.
- **Device Detection:** Analyzing the User Agent to determine initial quality settings (e.g., opting for lower-resolution textures for mobile devices) before the JavaScript bundle even loads.
- **Asset Resolution:** Generating signed URLs for protected assets.

Client Responsibility:

- **Hydration:** Initializing the React tree on the client.
- **Renderer Initialization:** Setting up the WebGPURenderer (or WebGLRenderer fallback).
- **Frame Loop Management:** Executing the 60fps (or higher) render loop via requestAnimationFrame.

To prevent hydration mismatches—a common issue when random values (like initial particle positions) differ between server HTML and client JavaScript—developers must use dynamic imports with Server Side Rendering (SSR) disabled for the 3D canvas component.[9]

Implementation Strategy: The Safe Import Pattern

JavaScript

```
// Architecture Diagram: The Safe Import Pattern
```

```
import dynamic from 'next/dynamic'

const Scene3D = dynamic(() => import('@/components/canvas/Scene'), {
 ssr: false,
 loading: () => <div className="absolute inset-0 bg-neutral-900" />,
})
```

This pattern ensures that the heavy Three.js bundle (often exceeding 200KB even with tree-shaking) is not loaded during the initial HTML navigation, significantly improving First Contentful Paint (FCP) and preventing the "Uncaught Error: window is not defined" crashes common in isomorphic rendering.[10] By deferring the load, the main thread remains free to handle the hydration of the LCP (Largest Contentful Paint) elements, ensuring the page feels responsive immediately.

2.2 React 19 Compiler and the Frame Loop

The introduction of the React Compiler in React 19 (fully integrated into Next.js 15) has eliminated the need for manual memoization (useMemo, useCallback) in standard DOM components.[2] The compiler automatically memoizes components and hooks, reducing unnecessary re-renders. However, within the R3F loop, manual optimization remains critical because the compiler optimizes the *component tree* re-renders, not the *frame loop* (useFrame).

The 2026 best practice is to strictly separate reactive state (React) from mutable state (Three.js). A common pitfall for developers migrating from standard React development is attempting to drive animation via state updates.

- **React State:** Use for high-level scene orchestration (e.g., changing the active 3D model, toggling day/night mode, updating UI overlays).
- **Mutable Refs:** Use for high-frequency updates (60fps animation). Never call setState inside a useFrame loop, as this triggers the React reconciler 60 times a second, destroying performance.[3]

Table 1: State Management Strategies in R3F (2026)

Feature	**React State (useState)**	**Mutable Refs (useRef)**	**Zustand / Valtio**
Update Frequency	Low (Interaction-based)	High (Frame-based)	Mixed (Transient updates)
Trigger Re-render	Yes	No	Configurable

Use Case	UI Overlay, Scene Config	Animation, Physics	Global Game State, Uniforms
Performance Cost	High (Reconciliation)	Near Zero	Low (Selector-based)

2.3 Transient State Management with Zustand

For complex applications, passing props deep into the 3D scene graph is inefficient and leads to "prop drilling." **Zustand** remains the de facto state manager for R3F in 2026 due to its transient update capabilities. Transient updates allow components to subscribe to state changes via refs or direct access without triggering a React render cycle. This is essential for binding UI controls (like a slider) to shader uniforms.[12]

Transient Update Pattern:

Instead of binding a component to a state value, components subscribe to the store directly inside the frame loop.

JavaScript

```javascript
// 2026 Pattern: Transient State Access
const useStore = create((set) => ({
  distortion: 0,
  setDistortion: (v) => set({ distortion: v })
}))

function ShaderPlane() {
  const materialRef = useRef()
  useFrame(() => {
    // Direct access avoids React render cycle
    // Reading directly from the store state without a hook subscription
    materialRef.current.uniforms.uDistortion.value = useStore.getState().distortion
  })
  return <mesh><shaderMaterial ref={materialRef} /></mesh>
}
```

This architecture ensures that the UI runs at the speed of React (event-driven), while the 3D scene runs at the speed of the GPU (frame-driven), with Zustand acting as the high-performance bridge between the two contexts.

---

3. The Rendering Backend: WebGPU and TSL

The defining characteristic of the 2026 web graphics stack is the transition from WebGL 2 to **WebGPU**. With Safari shipping stable WebGPU support in late 2025 (v26), the API is now universally available across major platforms, including iOS, iPadOS, and macOS.[3] This shift is not merely an API change; it unlocks compute shaders, storage textures, and more efficient binding models that were impossible in WebGL.

3.1 The WebGPURenderer Migration

Three.js r171+ introduced a "zero-config" WebGPURenderer that automatically falls back to WebGL 2 if WebGPU is unavailable.[3] This capability allows developers to target a single renderer API while maintaining compatibility with older devices.

Critical Initialization Step:

Unlike the synchronous WebGLRenderer, the WebGPURenderer requires an asynchronous initialization phase to request the GPU adapter and device. If this step is skipped, the renderer will fail silently or throw errors upon the first draw call.

JavaScript

```javascript
// 2026 Standard Initialization within R3F
import { WebGPURenderer } from 'three/webgpu';

// In the Canvas component
<Canvas
 gl={async (props) => {
  const renderer = new WebGPURenderer({
   antialias: false,
   powerPreference: 'high-performance'
  });
  await renderer.init(); // Mandatory await in 2026
  return renderer;
 }}
>
 {/* Scene Content */}
</Canvas>
```

R3F v9+ abstracts this complexity via an async gl prop factory, allowing the Canvas to handle the async setup internally.[15]

3.2 TSL: The Three Shading Language

The shift to WebGPU necessitates a move away from raw GLSL strings. **Three Shading Language (TSL)** is the new standard, enabling developers to write shader logic in JavaScript/TypeScript that compiles to **WGSL** (for WebGPU) or **GLSL** (for WebGL fallback).[3]

Why TSL is Non-Negotiable in 2026:

1. **Portability:** A single TSL codebase runs on both WebGPU and WebGL backends. Raw GLSL requires maintaining separate shaders or complex transpilers.
2. **Tree-Shaking:** TSL constructs are modular JavaScript functions. Unused shader nodes are eliminated at build time, reducing bundle size.[16]
3. **Composability:** TSL allows shader logic to be composed like React components. Functions can be passed as arguments, creating a library of reusable visual effects.[3]
4. **Type Safety:** Being TypeScript-based, TSL offers autocomplete and type checking for shader uniforms and attributes, eliminating a specific class of runtime errors common in GLSL development.

Table 2: Comparative Analysis - GLSL vs. TSL

Feature	**GLSL (Legacy)**	**TSL (2026 Standard)**
Language	C-like String	JavaScript/TypeScript Nodes
Compilation	Runtime (Driver)	Build-time + Runtime
Backend	WebGL Only	WebGPU + WebGL 2
Uniforms	Manual Management	Automatic Binding
Debugging	String parsing errors	JS Stack Traces
Optimization	Driver-dependent	Node-graph optimization

3.3 Performance Benchmarks: The "Force WebGL" Trap

Early adoption benchmarks in late 2025 indicated that WebGPURenderer could be slower than WebGLRenderer in simple scenes due to driver maturity and overhead. However, specifically forcing WebGL 2 via forceWebGL: true on the WebGPURenderer often resulted in significant performance degradation (5-10x slower) compared to the native WebGLRenderer.[17]

Recommendation:

For the 2026 stack, the WebGPURenderer should be used with its default settings. If a device does not support WebGPU, the automatic fallback is generally more performant than manually forcing the legacy backend within the new architecture. For extremely simple scenes targeting low-end hardware, the legacy WebGLRenderer remains a valid, albeit deprecating, choice.

---

4. Visual Effects: Liquid Distortion and Noise Gradients

The "2026 Starter Kit" visual identity is defined by organic, fluid motion and high-fidelity texture manipulation. These effects leverage TSL to ensure they run efficiently on the GPU.

4.1 Liquid Distortion (The "Hover" Effect)

The liquid distortion effect, a staple of high-end immersive sites, relies on vertex displacement and fragment shader UV manipulation. In 2026, this is implemented using **flow maps** and **RGB channel splitting** (chromatic aberration) driven by mouse velocity.[18]

Mechanism:

1. **Flow Map Computation:** A generated texture or data buffer that records the "trail" of the mouse cursor. The Red and Green channels store the X and Y velocity of the mouse.
2. **Vertex Displacement:** The geometry (typically a dense plane) is displaced along the Z-axis based on the flow map intensity, creating a physical "ripple."
3. **Fragment Distortion:** The texture lookup for the image is offset by the flow map values. Crucially, the Red, Green, and Blue channels are offset by slightly different amounts, creating a prismatic color separation at the edges of the ripple.[19]

GLSL Implementation Logic (Legacy/Reference):

The core fragment logic involves calculating a normalizedSpeed and using it to offset texture lookups. This logic is crucial for understanding the TSL port.

```
OpenGL Shading Language
```

```
// fragment.glsl
uniform sampler2D uTexture;
uniform vec2 uMouse;
uniform float uVelo;
varying vec2 vUv;

void main() {
  float mouseDistance = length(vUv - uMouse);
  float circle = smoothstep(0.4, 0.0, mouseDistance);
  vec2 distortedUv = vUv + circle * uVelo * 0.1;

  // RGB Split / Chromatic Aberration
  float r = texture2D(uTexture, distortedUv + 0.01 * uVelo).r;
  float g = texture2D(uTexture, distortedUv).g;
  float b = texture2D(uTexture, distortedUv - 0.01 * uVelo).b;

  gl_FragColor = vec4(r, g, b, 1.0);
}
```

TSL Implementation (2026 Standard):

In TSL, this logic is reconstructed using texture(), vec2(), and arithmetic nodes. The TSL compiler optimizes the texture fetches for the specific GPU architecture (e.g., using specific samplers in WGSL).

JavaScript

```javascript
// TSL Implementation of Liquid Distortion
import { Fn, texture, uv, vec2, float, smoothstep, length, uniform } from 'three/tsl';

const liquidEffect = Fn(([inputUV, mousePos, velocity]) => {
  const dist = length(inputUV.sub(mousePos));
  const circle = smoothstep(0.4, 0.0, dist);
  const distortion = circle.mul(velocity).mul(0.1);

  const distortedUV = inputUV.add(distortion);

  // Chromatic Aberration Nodes
  const tex = texture(uTexture);
  const r = tex.sample(distortedUV.add(vec2(0.01).mul(velocity))).r;
  const g = tex.sample(distortedUV).g;
  const b = tex.sample(distortedUV.sub(vec2(0.01).mul(velocity))).b;

  return vec3(r, g, b);
});
```

4.2 Noise Gradients (The "Grain" Aesthetic)

Procedural noise gradients provide a tactile, filmic quality that reduces the "digital" feel of WebGL renders. In 2026, **Simplex Noise** and **Perlin Noise** are expensive to compute per-pixel on high-DPI mobile screens. The optimized stack uses **dithering** and **pre-computed blue noise textures** or efficient hash functions.[20]

Efficient Grain Implementation:

Instead of computing noise for every pixel, the high-performance stack uses a high-frequency "White Noise" function based on fract(sin(dot(...))). This is computationally cheap and effective for grain.

TSL Implementation of High-Performance Grain:

JavaScript

```javascript
// TSL: Optimized White Noise for Grain
import { Fn, vec2, fract, sin, dot, float } from 'three/tsl';

const whiteNoise = Fn(([uv]) => {
 const k = vec2(12.9898, 78.233);
 return fract(sin(dot(uv, k)).mul(43758.5453));
});

// Usage in material
const grain = whiteNoise(uv().mul(resolution));
const finalColor = color.mix(vec3(grain), 0.05); // 5% mix
```

This TSL function compiles to a single instruction set on modern GPUs, negligible in cost compared to texture lookups.[20]

4.3 The "Reveal" Effect

Combining noise with radial gradients creates the popular "Reveal" effect. This technique uses a uProgress uniform to drive a threshold function.

1. **Displacement:** UVs are perturbed by a noise function to create organic edges.
2. **Masking:** A radial gradient decreases in size as uProgress increases.
3. **Mixing:** The image alpha is determined by smoothstep(mask, noise).

This combination allows for transitions that look like burning paper, dissolving ink, or evaporating liquid, all controllable via a single 0-1 float value.[21]

---

5. Asset Pipeline and Optimization

No amount of shader optimization can save a scene choked by uncompressed assets. The 2026 stack enforces a rigorous pipeline based on **glTF-Transform**, **Draco**, and **KTX2**.[22]

5.1 The glTF-Transform Workflow

glTF-Transform is the industry-standard tool for optimizing 3D assets programmatically. It replaces manual work in Blender, ensuring reproducibility. The CLI allows for a "build step" for 3D assets, similar to Webpack/Turbopack for JS.[23]

The "Golden Standard" Pipeline Command:

Bash

```bash
gltf-transform optimize input.glb output.glb \
 --compress draco \
 --texture-compress ktx2 \
 --simplify 0.5 \
 --weld 0.0001
```

Pipeline Stages:

1. **Prune & Dedup:** Removes unused nodes, materials, and accessors. Merges duplicate data.
2. **Weld:** Merges vertices that are spatially identical, vital for smooth shading and reducing vertex count.
3. **Simplify:** Reduces mesh complexity (decimation) while preserving topology.
4. **Texture Compression (KTX2):** This is the most critical step for GPU memory (VRAM).

5.2 KTX2 vs. WebP: The VRAM Battle

A common misconception is that WebP is "optimized" for WebGL. While WebP reduces *file size* (network transfer), it must be fully decoded into raw RGBA data on the CPU before being uploaded to the GPU. This causes main thread jank and consumes massive VRAM (e.g., a 4K texture is ~64MB uncompressed).[24]

KTX2 (Basis Universal):

KTX2 textures remain compressed *in VRAM*. The GPU decodes them on the fly.

- **Network:** Comparable size to JPEG/WebP (using UASTC or ETC1S supercompression).
- **VRAM:** 4-6x smaller footprint than decoded WebP/JPEG.
- **Performance:** Eliminates the "upload to GPU" stall that causes frame drops during loading.

Recommendation: Use **ETC1S** for color textures (diffuse/albedo) where slight artifacts are acceptable. Use **UASTC** for normal maps and metallic/roughness maps where precision is paramount.[26]

5.3 Draco Compression

Draco compresses geometry (vertex positions, normals, UVs). It requires a WASM decoder on the client.

- **Trade-off:** Increases main thread load (decoding) but drastically reduces file size.
- **2026 Best Practice:** Use Draco for all static geometry. Avoid Draco for morph targets if using older decoders, though 2026 decoders largely support it. Ensure the DracoDecoderModule is lazy-loaded or loaded via a Web Worker to keep the UI responsive.[27]

Table 3: Asset Optimization Techniques and Benefits

Technique	Function	Primary Benefit	Trade-off
Draco	Geometry Compression	Reduced File Size (Network)	CPU Decode Time
KTX2 (UASTC)	Texture Compression	Reduced VRAM Usage	Encoding Time (Build)
Meshopt	Geometry/Animation	High Performance Decoding	Less Compression than Draco

Dedup/Prune	Clean-up	Reduced Draw Calls/Nodes	None

6. Mobile Performance and Optimization Strategies

High-end shaders and post-processing can decimate mobile battery life. The 2026 stack prioritizes **Adaptive Performance**.

6.1 The "Golden Rule" of Draw Calls

The target is **<100 draw calls per frame.**[3]

- **InstancedMesh:** For identical geometry (foliage, particles, UI icons).
- **BatchedMesh:** Introduced in Three.js r159 and standard in 2026, this allows merging *different* geometries into a single draw call, provided they share the same material. This supersedes the old method of manually merging buffer geometries, as BatchedMesh allows for per-object culling and transformation.[3]

6.2 GPGPU Particles (Compute Shaders)

CPU-based particle systems (looping through arrays in JS) bottleneck at ~50,000 particles. Compute Shaders (WebGPU) allow for millions.

Implementation:

Use TSL instancedArray to create GPU-persistent buffers. The logic for position updates (velocity, gravity, collision) runs entirely on the GPU via a Compute Node.

JavaScript

```javascript
// TSL Compute Shader Pattern
const updateParticles = Fn(() => {
  const velocity = storage(velocityBuffer, 'vec3', instanceIndex);
  const position = storage(positionBuffer, 'vec3', instanceIndex);

  position.addAssign(velocity); // Update position based on velocity
  //... collision logic...
}).compute(particleCount);
```

This frees up the main thread entirely for React's reconciliation.[3]

6.3 Post-Processing and Battery Life

Post-processing is the single most expensive operation on mobile.

1. **Selective Bloom:** Do not apply bloom to the whole scene. Use SelectiveBloom to target specific emissive meshes. However, note that SelectiveBloom can be *more* expensive than global bloom if implemented inefficiently. The 2026 preferred method is "Emissive Thresholding"—pushing material colors above (1.0, 1.0, 1.0) and using a global bloom that only picks up these "super-white" values.[28]
2. **Adaptive DPR:** Use dpr={}. On high-density mobile screens (DPR 3+), capping at 2 is visually indistinguishable but saves 50% fragment processing power.[30]
3. **On-Demand Rendering:** For non-game apps (e-commerce, portfolios), set <Canvas frameloop="demand">. The scene only renders when the camera moves or props change. This drops battery usage to near zero when the user is idle.[31]
4. **Movement Regression:** Automatically lower DPR or disable expensive shaders (like transmission) while the camera is moving, then restore quality when static.
   JavaScript
   ```
   <PerformanceMonitor onDecline={() => setDpr(1)} onIncline={() => setDpr(2)} />
   ```

---

7. The 2026 Starter Kit Specification

Based on the research, the recommended "Starter Kit" for High-Performance WebGL in 2026 consists of the following configuration:

7.1 Dependencies

- **Core:** react@19, next@15, three@0.175+ (ensuring WebGPU stability).
- **R3F Ecosystem:** @react-three/fiber@9, @react-three/drei, @react-three/postprocessing.
- **State:** zustand (for transient updates).
- **Build/Asset:** @gltf-transform/cli (dev dependency).

7.2 Project Structure

/src

/components

/canvas # R3F Components (Client-only)

Scene.tsx # Main Canvas entry

Effects.tsx # Post-processing (Selective Bloom)

Fluids.tsx # TSL Fluid distortion

/dom # Next.js Server Components (Overlays)

/app

page.tsx # Server Component (Fetches data, imports Scene dynamically)

/public

/assets

/models # KTX2/Draco optimized GLBs

7.3 Configuration Defaults

- **Renderer:** WebGPURenderer with powerPreference: "high-performance".
- **Color Space:** SRGBColorSpace (linear workflow).
- **Tone Mapping:** ACESFilmicToneMapping (cinematic contrast).
- **Shadows:** SoftShadows or AccumulativeShadows for static bakes; ContactShadows for dynamic objects. Avoid expensive PCSS on mobile.

---

8. Conclusion

The high-performance WebGL stack of 2026 is defined by discipline. It rejects the "load it and see" approach in favor of a structured, optimized pipeline. By leveraging **Next.js 15** for efficient delivery, **WebGPU/TSL** for future-proof rendering, and **glTF-Transform** for asset hygiene, React developers can deliver cinema-quality 3D experiences that run smoothly on the constrained hardware of the mobile web. The transition to TSL and Compute Shaders is not optional—it is the necessary evolution to unlock the next generation of interactive web experiences. This stack provides the framework not just to render 3D, but to integrate it deeply and performantly into the fabric of the modern web.

Referências citadas

1. Next.js 15, acessado em fevereiro 16, 2026, https://nextjs.org/blog/next-15
2. 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
3. Introduction to TSL - Arie M. Prasetyo - Medium, acessado em fevereiro 16, 2026,

https://arie-m-prasetyo.medium.com/introduction-to-tsl-0e1fda1beffe
4. React-Three-Fiber: Enhancing Scene Quality with Drei + Performance Tips - Medium, acessado em fevereiro 16, 2026, https://medium.com/@ertugrulyaman99/react-three-fiber-enhancing-scene-quality-with-drei-performance-tips-976ba3fba67a
5. post-processing decrease performance so much on mobile. : r/Unity3D - Reddit, acessado em fevereiro 16, 2026, https://www.reddit.com/r/Unity3D/comments/u4jlgd/postprocessing_decrease_performance_so_much_on/
6. Server Components - Rendering - Next.js, acessado em fevereiro 16, 2026, https://nextjs.org/docs/13/app/building-your-application/rendering/server-components
7. Getting Started: Server and Client Components - Next.js, acessado em fevereiro 16, 2026, https://nextjs.org/docs/app/getting-started/server-and-client-components
8. The Future is 3D: Integrating Three.js in Next.js | Artekia Blog, acessado em fevereiro 16, 2026, https://www.artekia.com/en/blog/future-is-3d
9. Persistent 'ReactCurrentOwner' TypeError with React Three Fiber in Next.js 15.3.3 App (Firebase Studio) - Stack Overflow, acessado em fevereiro 16, 2026, https://stackoverflow.com/questions/79657951/persistent-reactcurrentowner-typeerror-with-react-three-fiber-in-next-js-15-3
10. React.js Best Practices In 2026 - AWS Builder Center, acessado em fevereiro 16, 2026, https://builder.aws.com/content/35mjuFWn4hSGCK6JjaZHFIGrzPG/reactjs-best-practices-in-2026
11. Performance pitfalls - Introduction - React Three Fiber, acessado em fevereiro 16, 2026, https://r3f.docs.pmnd.rs/advanced/pitfalls
12. How to improve three.js performance, with react-three-fiber? - Questions, acessado em fevereiro 16, 2026, https://discourse.threejs.org/t/how-to-improve-three-js-performance-with-react-three-fiber/69562
13. WebGPU Three.js Migration Guide 2026 - Utsubo, acessado em fevereiro 16, 2026, https://www.utsubo.com/blog/webgpu-threejs-migration-guide
14. v9 Migration Guide - React Three Fiber, acessado em fevereiro 16, 2026, https://r3f.docs.pmnd.rs/tutorials/v9-migration-guide
15. Three.js Shading Language - GitHub, acessado em fevereiro 16, 2026, https://github.com/mrdoob/three.js/wiki/Three.js-Shading-Language
16. WebGPU performance issue - Questions - three.js forum, acessado em fevereiro 16, 2026, https://discourse.threejs.org/t/webgpu-performance-issue/87939
17. Creating a Glitch Hover Effect with React Three Fiber and GLSL | by Milad Ghamati, acessado em fevereiro 16, 2026, https://miladghamati.medium.com/creating-a-glitch-hover-effect-with-react-three-fiber-and-glsl-2bd62e390f38
18. Three.js image distortion · GitHub, acessado em fevereiro 16, 2026, https://gist.github.com/damien-hl/a592b87f5a23adc80ec0aefbdf0c786f

19. 10 Noise Functions for Three.js TSL Shaders, acessado em fevereiro 16, 2026, https://threejsroadmap.com/blog/10-noise-functions-for-threejs-tsl-shaders
20. How to Code a Shader Based Reveal Effect with React Three Fiber & GLSL | Codrops, acessado em fevereiro 16, 2026, https://tympanus.net/codrops/2024/12/02/how-to-code-a-shader-based-reveal-effect-with-react-three-fiber-glsl/
21. Bundler plugins for optimizing glTF 3D models - GitHub, acessado em fevereiro 16, 2026, https://github.com/nytimes/rd-bundler-3d-plugins
22. Command-line quickstart - glTF Transform, acessado em fevereiro 16, 2026, https://gltf-transform.dev/cli
23. WebP vs KTX2 for web textures - Questions - three.js forum, acessado em fevereiro 16, 2026, https://discourse.threejs.org/t/webp-vs-ktx2-for-web-textures/58651
24. Expected KTX2 file sizes #443 - donmccurdy glTF-Transform - GitHub, acessado em fevereiro 16, 2026, https://github.com/donmccurdy/glTF-Transform/discussions/443
25. donmccurdy/glTF-Transform: glTF 2.0 SDK for JavaScript and TypeScript, on Web and Node.js. - GitHub, acessado em fevereiro 16, 2026, https://github.com/donmccurdy/glTF-Transform
26. KHRDracoMeshCompression - glTF Transform, acessado em fevereiro 16, 2026, https://gltf-transform.dev/modules/extensions/classes/KHRDracoMeshCompression
27. Bloom - React Postprocessing, acessado em fevereiro 16, 2026, https://react-postprocessing.docs.pmnd.rs/effects/bloom
28. How really selectiveblooming works in both R3F or three? - Questions, acessado em fevereiro 16, 2026, https://discourse.threejs.org/t/how-really-selectiveblooming-works-in-both-r3f-or-three/49478
29. AdaptiveDpr - Drei, acessado em fevereiro 16, 2026, https://drei.docs.pmnd.rs/performances/adaptive-dpr
30. Scaling performance - Introduction - React Three Fiber, acessado em fevereiro 16, 2026, https://r3f.docs.pmnd.rs/advanced/scaling-performance