ChatGPT

WebGPU & TSL Particle Engine Resources

🗄 Open-Source Projects & Demos (TSL + WebGPU)

- **Bruno Simon's TSL Sandbox:** A personal collection of **Three.js r177** experiments using the new **Three Shading Language (TSL)**. It includes numerous particle demos (e.g. *attractors, flow fields, cursor-responsive particles, morphing point clouds*) built on the **WebGPU renderer** [1] [2]. This sandbox showcases TSL-based compute shaders (for physics) and NodeMaterials (for rendering) in action. Bruno also published a standalone **TSL Particles System** demo (with code on GitHub) as a template for GPU particle systems [3]. These examples demonstrate high-count particle updates on the GPU, custom node-based materials, and creative effects (fireworks, galaxy, etc.) using Three.js r177+.

- **Three.js Official TSL Examples:** The Three.js **r177** examples library features many TSL/WebGPU demos – often contributed by Bruno and others. For instance, **"TSL Compute Attractors (Particles)"** shows particles orbiting gravitational points using a TSL compute shader, and **"TSL Galaxy"** or **"TSL Angular Slicing"** demonstrate NodeMaterial-based visual effects. These were originally done in GLSL and have been ported to TSL for WebGPU [4]. Notably, the *Angular Slicing* shader (from Bruno's ThreeJS Journey course) was converted to TSL and integrated, even adding a postprocessing bloom in the example [4]. The official examples illustrate best practices for structuring **TSL code** and using Three's `Fn()` to define GPU functions for particles and post-effects.

- **"Linked Particles" by Christophe Choffel:** A striking WebGPU/TSL demo where particles link together with filament-like trails. This experiment (open-sourced by Choffel as *webgpu-tsl-linkedparticles*) was added to Three.js examples as **"VFX Linked Particles"**, complete with a **bloom post-process node** for glow [5] [6]. It uses TSL to spawn and update particles in a flow-field, storing attributes in GPU buffers, and leverages TSL's built-in bloom **post-process node**. The code showcases multi-pass rendering entirely in TSL: after each frame's particle positions are computed, a bloom pass is applied via a `BloomNode` – all on the GPU.

- **R3F WebGPU Starter (Anderson Mancini):** An open-source starter template for **React Three Fiber (R3F)** with Three.js r177 WebGPU. It demonstrates setting up `<Canvas>` with `WebGPURenderer` and applying **postprocessing using TSL nodes** [7] [8]. The demo (see **r3f-webgpu-post-processing** on Vercel) renders a simple scene with a custom NodeMaterial and a TSL-based post-effect (like a bloom or distortion). The repo shows how to integrate Three's new `three/webgpu` module into React, including awaiting `renderer.init()` and extending Three types in R3F. It's a great starting point for building React apps with WebGPU + TSL, including how to attach NodeMaterials in JSX and stack multiple render passes.

- **Other Notable Repos:** *TSLFX* (by @verekia) is an emerging library of pre-made shader effects for TSL (see **"TSLFX"** below). Also, Matt Frawley's **React-Three-TSL Playground** (from Pragmattic blog) provides a minimal Next.js + R3F setup with TSL NodeMaterials. It includes an interactive example of

a hovering sphere using **MeshStandardNodeMaterial** with custom `colorNode` and `positionNode` logic [9]. This can serve as a reference for hooking TSL nodes into React state and props. Another example is Michael Kellogg's **Photons2** (a Three.js particle engine rewrite), which predates TSL but could be updated – it shows basic GPGPU particle structure (position/velocity textures) albeit in WebGL.

📄 Guides & Tutorials (TSL Nodes, Compute & Pipelines)

- **Wawa Sensei's WebGPU/TSL Lessons (React):** A comprehensive tutorial series on using WebGPU with R3F, culminating in a lesson **"GPGPU Particles with TSL & WebGPU"** [10]. This step-by-step guide (with code) walks through creating hundreds of thousands of particles that morph between shapes (e.g. forming a 3D fox model, then a text logo) using pure GPU computation. It covers using TSL's `storage()` buffers for particle data, writing compute functions with `Fn()`, and running them via `renderer.computeAsync()` each frame [11]. The tutorial also integrates R3F hooks (using `useFrame` to dispatch computes) and shows how to use a `SpriteNodeMaterial` for efficient point sprites [12] [13]. Wawa's earlier **"WebGPU/TSL"** intro lesson covers setting up R3F's `<Canvas>` with a WebGPURenderer and using Three's built-in NodeMaterials (Basic/Standard NodeMaterial) in JSX. Together these resources are invaluable for learning TSL's API (e.g. `color()`, `mix()`, `uv()` nodes) and React integration.

- **Codrops "Matrix Sentinels" Tutorial (May 2025):** A detailed *TSL particle trails* guide by tutorial author *MisterPrada*, showing how to create particles with persistent **tails** (like streaking "sentinels") [14]. It uses TSL compute shaders to maintain a history buffer of positions for each particle, essentially implementing a multi-step position buffer (a technique similar to storing a trail of each particle). The article's code demonstrates setting up **multiple storage buffers** (e.g. a large `positionStoryBuffer` for trails [15]), an **init compute** that seeds random start positions [16], and an **update compute** that moves "head" particles with noise while shifting trailing segments from the history [17] [18]. It also attaches a custom `positionNode` in the MeshStandardNodeMaterial to offset each instance based on its trail state (enlarging the head of each snake) [19]. This tutorial is notable for showcasing **TSL's ability to handle multi-pass logic**: one compute shader to record history, another to update positions, and then a NodeMaterial using that data – all in sync. It's a great reference for **complex particle behaviors (motion trails, flow-field movement)** done entirely with TSL nodes and Three r177.

- **Pragmattic Blog – R3F + TSL NodeMaterial Guide:** *Matthew Frawley's* article *"React Three Fiber with WebGPU and TSL"* (March 2025) provides a concise walkthrough for newcomers to TSL in React [20]. It explains how to **extend Three's WebGPU classes in R3F**, switch the Canvas to use WebGPURenderer, and create a simple NodeMaterial-driven component. The example builds a **hover-responsive sphere**: two color gradients (base and hover) are defined with `color()` nodes and mixed by a uniform, and the sphere's vertex position is slightly offset when hovered (using `positionWorld` minus a vector) [9] [21]. This is all done in JS/TS instead of GLSL. The blog (and its associated GitHub playground) is useful for understanding **basic NodeMaterial customization** – e.g. replacing a standard material's color with a TSL node network – and how to hook that into React state (toggling the uniform on pointer over). It also discusses some quirks (like needing to call `extend(THREE)` and R3F's `frameloop="never"` during async renderer init). Overall, a good starting point for **TSL node syntax** in a React context.

- **Three.js Forum & Reddit Q&As:** The community has shared tips as well. Notably, *react-three-fiber* author **@drcmda** provided an example CodeSandbox showing R3F + TSL in action [22] – it uses `import * as THREE from 'three/webgpu'; import * as TSL from 'three/tsl';` then `extend(THREE)` so that `<meshStandardNodeMaterial {...} />` JSX works. He notes that as long as you use a Three.js version that includes TSL, R3F will reflect it (since R3F just renders Three scenes) [23] [24]. One caveat he mentioned is that TSL is essentially a "new fork" of the materials/shaders, so **many Drei shaders or custom shader code need updates** – e.g. Drei's shader-based effects won't work out-of-the-box until rewritten in TSL [25]. This is an important point: migrating to TSL means using NodeMaterials or writing effects with the new node system (old `ShaderMaterial` code won't run on WebGPU). The forum also has threads on using **PassNode** and postprocessing with TSL (since the postprocessing API changed under the hood to utilize nodes). These community insights can help avoid pitfalls when integrating TSL in existing Three.js projects.

▤ Libraries & Extensions for TSL Engines

- **TSLFX Library (Verekia): TSLFX** is an early-stage open-source library providing **pre-built shader effects and utilities** for Three.js TSL [26]. It includes a collection of VFX "primitives" (e.g. an **"impact" shockwave effect, glowy trails, noise warps) and even some geometric** SDF shapes **(signed distance functions like circles, hearts, etc.) that you can use as building blocks [27]. The library is designed to work with both WebGPU and WebGL backends. For example, you can import an** `impact()` **effect from *tslfx*, which returns a** `{ uniforms, nodes }` **object – you then apply those nodes to a NodeMaterial's inputs (like colorNode) and update the provided uniform (like time) each frame [28] [29]. TSLFX also provides a system to** chain multiple effects **with timing controls (their *Timing* example shows sequencing effects with delays for complex animations [30]). The docs show integration both in vanilla Three and in React (extending MeshBasicNodeMaterial for R3F) [31] [32]. This library can jump-start a TSL-powered engine with ready-made** visual effects, procedural noise, and SDFs** – useful for adding explosions, force-field flashes, etc., on top of your particle simulations.

- **Node-Based PostProcessing:** Three.js's node system isn't limited to materials – it also enables **node-driven post-processing passes**. In TSL, one can create effects that automatically perform multi-pass rendering under the hood. For instance, Three r177 introduces a `gaussianBlur()` node function that encapsulates a **two-pass Gaussian blur**; you can call it in a material or as a standalone post effect, and TSL manages the render-to-texture and ping-ponging for you [33]. Similarly, there are node classes like `BloomNode`, SSAO, etc., which integrate with the WebGPU renderer's new PostProcessor. The *Linked Particles* example cited earlier uses `bloom()` simply by importing `BloomNode.js` and calling `renderer.addPipeline(bloom(...))` (or in that case, a one-liner `import { bloom } from 'three/addons/tsl/display/BloomNode.js';`) [6]. For developers, this means you don't need a separate EffectComposer – you can incorporate post FX as NodeMaterial components or pipeline nodes. This is **highly relevant for a particle engine**, where you may want glow, motion blur, DOF, etc. The TSL approach ensures these effects work in WebGPU and play nicely with NodeMaterials. (Keep in mind the node-based postprocessing is still evolving; some forum discussions note quirks with `PassNode` when porting certain effects [34]).

- **Control & Animation Tools:** Building a "complete" engine involves more than just the render loop – you'll likely need UI controls and scripted animations. **Tweakpane** (or Drei's <Leva> alternative) can

be easily used to tweak TSL uniform values at runtime. For example, you could expose particle parameters (e.g. *attractor strength, particle size, field turbulence*) and update the corresponding `uniform(...).value` – since TSL nodes use real JS objects for uniforms, changes take effect immediately. Likewise, **GSAP** or any animation library can tween these uniform values or even do timeline sequencing of effect triggers. (We didn't find a dedicated example of GSAP + TSL in the wild, but conceptually it's straightforward: e.g., tween a `uniform(0)` to `1` over time to fade in an effect, or animate a `vec3` uniform controlling emission color). One community example is TSLFX's timing system, which essentially does scheduling of multiple effect uniforms – something you could also achieve by driving TSL node uniforms via GSAP timelines.

- **Drei & Fiber Integration:** Aside from materials, most Drei helpers work normally in WebGPU (e.g. <OrbitControls>, <Stats>, camera rigs, etc., since those are not shader-dependent). As drcmda noted, anything that **uses custom shaders will need a rewrite** or workaround. For instance, Drei's effects like <MeshWobbleMaterial> or <shaderMaterial> wrappers won't function until rewritten with TSL NodeMaterials. The community is actively adapting – expect Drei to eventually offer TSL-compatible versions of some essentials. In the meantime, you can still use Drei for non-shader components and write your own NodeMaterials for any visual effects. Another tip: use React's context or zustand store to share state (like physics parameters) between your React UI controls and the underlying TSL uniforms – this makes hooking up things like **audio reactivity** easier. (No official TSL audio-reactive demo was found, so this remains an exercise for the engine developer – e.g. analyzing an audio stream with Web Audio API and feeding frequencies into a TSL uniform array for particle behaviors akin to an equalizer visual.)

▤ Advanced GPU Particle Techniques & Examples

- **Fluid Simulations (SPH, MPM, FLIP):** A highlight of WebGPU's power is real-time fluid physics with particles. Developer **Matsuoka-601** shared two impressive open-source fluid demos: *WebGPU-Ocean* and *WaterBall*, each simulating tens of thousands of water particles in real-time [35] [36]. His accompanying article *"High Performance WebGPU Fluid Simulations"* explains that the first demo uses **MLS-MPM (Moving Least Squares Material Point Method)** – a hybrid particle-grid approach – to achieve 100k+ particles at 60 FPS, while the second combines that with a **screen-space fluid rendering** technique for visualizing the liquid [36] [37]. The article provides deep insights into implementing **SPH (Smoothed Particle Hydrodynamics)** vs. **MPM** on the GPU. Key takeaways: SPH is pure particle-based and required an efficient neighbor search (he implemented a GPU hashing method) [38], but even so, SPH struggled beyond ~30k particles on integrated GPUs. By switching to MLS-MPM (which is related to PIC/FLIP and avoids neighbor search by using a grid), he achieved ~100k particles in real-time [39] [40]. These projects (available on GitHub via the links in the article) are goldmines for learning advanced GPU physics algorithms. While they are raw WebGPU (written in WGSL), they could inspire a TSL-based engine: e.g. one could implement an SPH solver in TSL compute nodes, or use TSL's **StructuredBuffer/Storage** APIs to do MPM transfers. The **Three.js r177 Node API** even allows writing compute in a style similar to WGSL but in JS (using `storage()` buffers and looping with `instanceIndex`). For instance, TSL's author @sunag demonstrated a basic 1000-particle gravity simulation using a `StructuredArray` and a Fn compute that updated positions with gravity and ground collision [41] [42]. Adapting full SPH or MPM in TSL would be a challenge, but these resources prove it's feasible on the WebGPU backend.

- **Reaction-Diffusion & Cellular Automata:** A different class of "particle" system is actually grid-based – e.g. reaction-diffusion on a texture (which can create organic patterns that one might apply to particles or as a post-effect). There is an excellent Codrops tutorial by **Robert Leitl** on implementing a **Reaction-Diffusion compute shader** with WebGPU [43]. The tutorial (May 2024) shows how to set up a **ping-pong compute pipeline** with two storage textures (for chemical concentrations), then run many iterations of the RD algorithm (the Gray-Scott model) entirely on the GPU, finally rendering the result to screen [44] [45]. While it doesn't use Three.js, the provided code (on GitHub via Codrops) uses **webgpu-utils** and demonstrates concepts directly applicable to Three's node system: using **storage textures**, workgroup dispatch, and double-buffering. In a TSL context, one could recreate this by using `storageTexture()` nodes and TSL's `compute()` functions (since TSL can create compute passes as well). The result could be used for particle effects – e.g. driving particle colors or positions based on a diffusing chemical field. Reaction-diffusion can also produce **interesting masks and patterns for post-processing**, which might be part of an engine's "fields" or "biological systems" module (as alluded to in the vision docs).

- **Physarum Slime Mold (Agents + Field):** Simulating **Physarum polycephalum** (slime mold) involves thousands of agent particles leaving chemical trails that diffuse – a great test of GPU particle-feedback loops. A standout example is **SuboptimalEng's WebGPU Slime Simulation** [46] (inspired by Sebastian Lague's coding adventure). This project, written in TypeScript with WGSL, runs ~1 million agents that follow pheromone gradients on a 2D grid. The GitHub repo includes a live demo and a 5-minute devlog explaining the pipeline. Essentially it uses two compute passes: one to move agents (reading from a trail map and writing their new positions & deposits), and one to diffuse/decay the trail map (a blur on a texture) – a classic "agents + field" approach. The result is mesmerizing emergent network patterns. While this wasn't built with Three.js, it's directly relevant: a TSL-based engine could incorporate a **slime mold mode** by using a similar technique (e.g. a compute node that updates a trail texture and a particle positions buffer each frame). Three r177's support for **storage buffers and textures** makes this possible. (Indeed, the *vision docs* mention Physarum in a "BiologicalField" context – this is how you'd do it). The slime simulation code is a useful reference for **WGSL compute logic**, and it underscores WebGPU's ability to handle *millions* of lightweight agents. One could imagine integrating this with R3F by rendering a point cloud or mesh instances for the agents, or even sampling the trail texture to influence other particles.

- **Flocking (Boids) and Emergent Behavior:** Classic boids (flocking birds) can be achieved with GPGPU as well – Three.js had an old WebGL GPGPU *birds* demo (1000 boids with position/velocity textures). With TSL, one could implement boid rules in a compute shader similarly, but benefiting from clearer syntax. We didn't find a published TSL boids demo yet, but the building blocks are evident in other examples: Bruno's **"Boids.js"** in the vision likely follows the Reynolds rules (separation, alignment, cohesion) and could be coded as a TSL compute node acting on a **StorageInstancedBufferAttribute** of velocities. Another angle is *flow-field following*, which is related – e.g., Bruno's sandbox "particles-flow-field" example shows particles steered by a 3D noise field (the code likely samples curl noise in a Fn and updates velocity). Flocking and flow-field behaviors fall under **emergent particle motion**, and the key is lots of parallel vector math, which WebGPU excels at. We can leverage libraries like `mx_fractal_noise_vec3` (Three's built-in noise node) for vector fields, or even feed in vector field textures. The **Linked Particles** demo mentioned earlier actually uses a noise field to move particles and spawns them in bursts, which feels akin to a flocking behavior. So, while boids aren't explicitly showcased in official TSL examples yet, the techniques are known and ready to implement in a custom engine.

- **Shape Morphing & Volume Effects:** The engine vision mentions shape-shifting and SDFs – these can be achieved by treating a mesh or volume as a target for particles. **Point-cloud morphing** is demonstrated in Wawa's GPGPU lesson: particles sample positions from a 3D model (the fox) and smoothly interpolate to new positions (text letters) [10]. The implementation uses a texture or buffer of target positions and lerps each particle over time (resetting when lifecycle ends). This kind of morphing can be extended with **Signed Distance Fields**: e.g., you could define a 3D volume via an SDF and move particles along its gradient or confine them inside. TSLFX includes some 2D SDF nodes (circle, heart), and for 3D one could port primitives (sphere SDF, box SDF, etc.) into TSL. **Volume rendering** on WebGPU (for things like smoke or fire) might involve particles as well – either many small particles (sprite particles) or a volumetric raymarch. No specific TSL volume-raymarch example is out yet, but three.js did get a `NodeMaterial` version of MeshStandardMaterial that could be tricked into raymarching with TSL's `Loop()` nodes. For now, most volumetric demos (smoke, fire) are still done in GLSL or as texture animations. But given TSL's flexibility, one could attempt a volume simulation (like a grid of density updated by a compute shader) and then render it via particles or raymarch in a shader – definitely advanced territory, but WebGPU can handle it.

- **Audio Reactivity:** We didn't find a dedicated TSL example for audio-driven particles, but it's certainly on the table. Typically, audio reactivity means using FFT or waveform data (via the Web Audio API Analyser) to influence particle properties (e.g. particle size, color, emission rate to the beat). In a TSL context, one could feed an array of spectrum values into a uniform or storage buffer. For instance, you could have a TSL uniform like `audioSpectrum = uniform(new Float32Array(N))` and then update that array each frame from JS with the latest FFT. Your compute shader or material nodes can then access `audioSpectrum` to modulate behavior – e.g., use bass frequencies to shake particle positions, or map amplitude to particle color intensity. A simpler route is to use audio as an input texture – some Three.js demos have used a "sound texture" where each x coordinate corresponds to a frequency bin. TSL can sample textures via `textureLoad()` even in compute, so that's viable. While we lack a concrete code sample to cite for this, the engine builder can look to prior art like **Shader Park** or **GLSL sound visualizers** for inspiration. The key point: WebGPU has no issue handling this – it's just data – and R3F can easily connect audio analysis to the TSL uniforms. The result could be anything from particles dancing to music, to procedurally generating emitters based on song structure.

▤ Relevant Three.js r177+ Updates (Changelog Highlights)

Three.js r177 (and the surrounding releases) was a game-changer for GPU-based engines. The new **TSL (Three Shading Language)** system introduced in r177 provides a unified, high-level way to write both materials and compute shaders in JavaScript. According to the official release notes and examples:

- **TSL NodeMaterials:** r177 added NodeMaterial variants of all major materials (MeshBasicNodeMaterial, StandardNodeMaterial, etc.), which allow assigning `*.node` properties instead of writing GLSL. These NodeMaterials are **renderer-agnostic** – the same code works on WebGL or WebGPU. For instance, you can take a MeshStandardNodeMaterial and plug in a custom `colorNode` (e.g., a mix of two colors by UV) and it will compile to WGSL for WebGPU or GLSL for WebGL automatically [47] [48]. This makes it much easier to extend or modify shading models (no more patching shader chunks). The node system is also tree-shakable (import only what you use)

and typesafe with TS definitions. Essentially, Three.js now has a built-in analog of Unity's Shader Graph or Unreal's Material Editor, but also exposed as JavaScript APIs.

- **Compute Shader Support:** TSL introduced the ability to create compute shaders in JS via `Fn()` and calling `.compute(workgroupSize)` [41] [42]. The new `WebGPURenderer` has methods like `computeAsync()` to dispatch these. The Nodes system provides WGSL built-ins like `instanceIndex`, `deltaTime`, and lets you declare **StorageBuffers** easily (e.g. `storage( new StorageBuffer(...), 'vec3')`). The official examples include TSL compute usage (the attractors and particles demos use it for particle motion). This means Three.js can now handle GPGPU logic without external libraries – perfect for particle engines (physics, flocking, etc.). Notably, **StructuredArray** was added as a helper to interleave multiple data fields (position, velocity, etc.) in one buffer [49] [42], which is then accessed in TSL via `particles.element(index).get('position')` syntax. This abstracts a lot of WebGPU's boilerplate and allows more focus on the simulation logic.

- **WebGPU Renderer Maturity:** By r177, Three's WebGPURenderer had stabilized enough to support **shadows, morph targets, skeletal animation, instancing, and more**, approaching feature parity with WebGL. Many bugs with module imports and build configurations were fixed – for example, initial issues where bundlers struggled with `three/webgpu` have been resolved [24]. This is evident from the fact that CodeSandbox can now run three/webgpu (as pointed out on Reddit) and that Vite+R3F setups work after updating to Three ≥0.177. The **WebGPU capabilities** check (`WebGPU.isAvailable()`) is provided to gracefully fallback if needed [50]. Also, **async GPU init**: calling `renderer.init()` returns a promise; R3F patterns involve waiting to start the render loop until this resolves [51] [52]. These changes mean that a cutting-edge engine can reliably target WebGPU in browsers like Chrome 113+ and easily switch back to WebGL if not available.

- **Post-Processing Pipeline:** Three.js overhauled its postprocessing to be compatible with WebGPU via Nodes. Instead of the old EffectComposer, r177+ uses a Graph-like approach. The `EffectComposer` from three-stdlib is not (yet) WebGPU-compatible, but the new `NodeFrame` / `Renderer.compute()` paradigm covers a lot. The changelog for r177 mentions new node-based passes and several demo effects (like TSLHalftone, TSLEarth – which likely show full-screen shader effects as NodeMaterials). As a developer, you can create a **PassNode** which essentially is a miniature shader that runs in screen space after the scene. The *Interactive 3D with BatchedMesh* article (Oct 2024) on Codrops gives a breakdown of using **BatchedMesh** and mentions "exploring the new post-processing pipeline with TSL" [53], implying one can now do things like bloom, depth-of-field, color grading via TSL. This is an evolving area; for now, basic things like bloom, blur, tone mapping are available as nodes (with more complex ones like SSAO or SSR in progress).

- **Changelog & Community Resources:** The Three.js changelog for r177 (accessible via threejs.org) enumerates all changes – importantly the introduction of *TSL (NodeMaterial) as stable*, and deprecations of some older WebGPU beta features. The Three.js Twitter (X) account shared demos around that time, e.g. a WebGPU grass rendering using TSL (which ties into TSLFX's grass example) and others like liquid simulations by community members [54]. Also, since r173+ started the TSL rollout, there have been Q&A threads ("Why TSL is interesting" on the forum [55]) and even a **YouTube video "Intro to WebGPU & TSL"** by Johnny Mass (Oct 2024) to help people migrate. All these underscore that *now (2025)* is an excellent time to build a GPU particle engine with Three.js – the pieces (WebGPU, TSL, compute, React integration) have fallen into place in the past year.

Sources:

- Bruno Simon's TSL Sandbox (examples list) [1] [2]; "TSL Particle System" repo [3]
- Three.js Examples & Bruno's contributions [4] (TSL attractors, slicing)
- *Linked Particles* example (Three.js r177) [5] [6]
- Anderson Mancini's R3F WebGPU Starter (GitHub README) [7] [8]
- Wawa Sensei – *GPGPU Particles with TSL* (Lesson text) [10] [11]
- Codrops – *"Matrix Sentinels"* TSL tutorial [14] [18]
- Pragmattic Dev – R3F + TSL blog (Matthew F.) [20] [9]
- Reddit r/threejs – drcmda CodeSandbox + comments [22] [25]
- **TSLFX** library README [26] [29]
- Three.js Wiki – TSL Post-processing (gaussian blur node) [33]
- Fossies (Three.js r177 example code) – BloomNode import in linkedparticles [6]
- Matsuoka's WebGPU Fluid article (Codrops) [36] [37]
- Codrops – WebGPU Reaction-Diffusion tutorial [44] [45]
- SuboptimalEng's Slime Simulation (GitHub README) [46] [56]
- Three.js Wiki – "Why TSL" (notes on Node sharing) [57] and forum discussions.

---

1 2 GitHub - brunosimon/three.js-tsl-sandbox
https://github.com/brunosimon/three.js-tsl-sandbox

3 GitHub - brunosimon/three.js-tsl-particles-system
https://github.com/brunosimon/three.js-tsl-particles-system

4 23 TSL in codesandbox & three-fiber V9 : r/threejs
https://www.reddit.com/r/threejs/comments/1i8v1zt/tsl_in_codesandbox_threefiber_v9/

5 6 50 three.js: examples/webgpu_tsl_vfx_linkedparticles.html | Fossies
https://fossies.org/linux/three.js/examples/webgpu_tsl_vfx_linkedparticles.html

7 8 GitHub - ektogamat/r3f-webgpu-starter: React Three Fiber WebGPU Post Processing by Anderson Mancini. A very simple scene to demonstrate how to integrate Threejs WebGPU with React Three Fiber using Post Processing effects.
https://github.com/ektogamat/r3f-webgpu-starter

9 20 21 React Three Fiber with WebGPU and Three Shading Language (TSL) Node Material | Pragmattic
https://blog.pragmattic.dev/react-three-fiber-webgpu-typescript

10 11 12 13 GPGPU particles with TSL & WebGPU - Wawa Sensei
https://wawasensei.dev/courses/react-three-fiber/lessons/tsl-gpgpu

14 15 16 17 18 19 Matrix Sentinels: Building Dynamic Particle Trails with TSL | Codrops
https://tympanus.net/codrops/2025/05/05/matrix-sentinels-building-dynamic-particle-trails-with-tsl/

22 24 25 Setup React-Three-Fiber with TSL : r/threejs
https://www.reddit.com/r/threejs/comments/1iko4kg/setup_reactthreefiber_with_tsl/

26 27 28 29 30 31 32 GitHub - verekia/tslfx: TSLFX • Early-stage collection of VFX, utils, and SDFs for Three.js Shading Language (TSL)
https://github.com/verekia/tslfx

33 57 Three.js Shading Language · mrdoob/three.js Wiki · GitHub
https://github.com/mrdoob/three.js/wiki/Three.js-Shading-Language

34 PassNode Issues - Questions - three.js forum
https://discourse.threejs.org/t/passnode-issues/68499

35 36 37 38 39 40 WebGPU Fluid Simulations: High Performance & Real-Time Rendering | Codrops
https://tympanus.net/codrops/2025/02/26/webgpu-fluid-simulations-high-performance-real-time-rendering/

41 42 49 ALLin1THREE_TSL_NODE.md
file://file-Sh9vzxdbnKBh4uZst614tX

43 44 45 Reaction-Diffusion Compute Shader in WebGPU | Codrops
https://tympanus.net/codrops/2024/05/01/reaction-diffusion-compute-shader-in-webgpu/

46 56 GitHub - SuboptimalEng/slime-sim-webgpu: Slime mold simulation with WebGPU and TypeScript.
https://github.com/SuboptimalEng/slime-sim-webgpu

47 48 51 52 WebGPU / TSL - Wawa Sensei
https://wawasensei.dev/courses/react-three-fiber/lessons/webgpu-tsl

53 webgpu | Codrops
https://tympanus.net/codrops/tag/webgpu/

54 Three.js (@threejs) / X
https://twitter.com/threejs/

55 Why TSL (three.js shading language) is so interesting! - Discussion
https://discourse.threejs.org/t/why-tsl-three-js-shading-language-is-so-interesting/56306