ChatGPT

WebGPU Node-Based Particle Engine (Three.js r177+) Documentation

Introduction to the WebGPU & TSL Ecosystem

Welcome to the **WebGPU Node-Based Particle Engine** documentation. This engine harnesses the latest Three.js **r177** features – notably the **WebGPU renderer** and **Three Shading Language (TSL)** node system – to enable high-performance, flexible particle simulations entirely on the GPU. We target creative coders, VFX engineers, and scientific visualization developers who need complex, real-time particle effects.

Three.js WebGPU & TSL Overview: Three.js's WebGPU pipeline introduces a new shader system built on nodes (TSL). Instead of writing raw GLSL or WGSL, you construct shaders by composing **NodeMaterial** building blocks. TSL is *renderer-agnostic* – the same node shader can compile to either WGSL (WebGPU) or GLSL (WebGL) as needed [1]. In practice, this means you write shader logic in JavaScript/TypeScript using Three.js's node APIs, and Three.js handles generating the GPU code for the active backend. The WebGPURenderer will automatically fall back to WebGL2 on unsupported devices, **preserving node-based materials across both** (with some limitations) [2] [3]. With TSL, developers can focus on creative shader design without worrying about GLSL vs WGSL differences.

WebGPURenderer Setup: By default, Three.js's core includes only WebGL; to use WebGPU you must import the dedicated module and initialize the new renderer. For example, in a React Three Fiber (R3F) app you can import WebGPU classes from `'three/webgpu'` and override R3F's default renderer via the `<Canvas>` `gl` prop [4]. You might do:

```
import * as THREE from 'three/webgpu';
<Canvas gl={(canvas) => new THREE.WebGPURenderer({ canvas, antialias: true })}>
    {/* ... */}
</Canvas>
```

Call `renderer.init()` once to request adapter/device access, then proceed as usual (r177's WebGPURenderer internally handles acquiring a GPUDevice) [5] [6]. After init, rendering runs on WebGPU. The engine automatically shares code paths with WebGL when needed – e.g. uniform values and node logic don't change, thanks to TSL's abstraction.

R3F and Drei Integration: If you use React, **React Three Fiber (R3F)** provides a declarative way to construct the scene graph and manage rendering loops. Our engine's components are compatible with R3F, and you can integrate them as JSX elements or use hooks like `useFrame` for per-frame updates. The **@react-three/drei** library complements R3F with handy helpers – for instance, orbit controls, environment lighting, and UI widgets. For example, to add camera controls you might import Drei's `OrbitControls` and include `<OrbitControls makeDefault />` in your JSX [7], or in plain Three.js use `new OrbitControls(camera, renderer.domElement)`.

Animation & UI Tools: We leverage popular tools for animation and parameter tuning. **GSAP** (GreenSock Animation Platform) can animate any numeric property over time, which is ideal for smoothly transitioning simulation parameters, camera moves, or material properties. For example, you could tween a particle emitter's rate or a field strength with `gsap.to(engine.settings, { duration: 2, emissionRate: 500 })`. Complex timeline sequences, easings, and scroll-based triggers can all be orchestrated via GSAP. For live parameter control, **Tweakpane** offers a modern GUI panel for sliders, color pickers, and inputs. It's invaluable for debugging and art-directing the simulation: you can expose parameters like *particle size*, *gravity*, *wind force*, etc., and adjust them in real-time with minimal code. Together, GSAP and Tweakpane allow animators and developers to drive the particle system interactively – from syncing with music to manual fine-tuning – without modifying core code.

Staying Current (r177 Updates): Three.js r177 brings numerous improvements relevant to our engine. Notably, the NodeMaterial system and WebGPURenderer have matured significantly. As of r177, NodeMaterials are primarily designed for WebGPU; legacy WebGL support was removed in r164 to streamline development [8] [2], but the WebGPURenderer now transparently falls back to WebGL2 so your node-based shaders still run on older devices [3]. The TSL node library is continuously optimized – for example, automatic multi-render-target support was added for advanced effects (so a single pass can output color, normal, depth, etc.) [9]. Three.js's changelogs also highlight new NodeMaterial APIs and bug fixes. We incorporate these latest patterns: using the new `MeshStandardNodeMaterial`, `SpriteNodeMaterial` for particles, the `PassNode` system for post-processing, and updated loader and animation interfaces. Throughout this documentation, we will reference relevant r177+ APIs and changes. For instance, r177 ensures color management defaults to sRGB, so our examples call `texture.colorSpace = THREE.SRGBColorSpace` when loading images, and we utilize the new `renderer.setAnimationLoop()` for WebGPU-safe animation loops.

With the ecosystem introduced, let's dive into the architecture of the engine's components and how to use them.

JSM Component Architecture and Project Structure

To keep the engine modular and maintainable, all components are built as **ES6 JavaScript Modules (JSM)**. This means each module can be imported individually (e.g. `import ParticleEngine from './core/ParticleEngine.js'`) and supports tree-shaking for optimal bundling. The recommended project structure groups related functionality into folders, for example:

```
src/
 ├─ core/
 │   ├─ ParticleEngine.js       // Main engine orchestrator
 │   ├─ PBS.js                 // Physics solver (Position-Based Simulation
utilities)
 │   ├─ Emitter.js             // Particle emitters/spawners
 │   └─ ...other core classes
 ├─ materials/
 │   ├─ ParticleMaterial.js    // Material for rendering particles (node-
based)
 │   ├─ EffectMaterial.js      // Material for full-screen postprocessing
effects
 │   └─ ...other custom NodeMaterials
 ├─ shaders/
 │   ├─ NoiseFieldNode.js      // Example: a TSL node for noise vector field
 │   ├─ BoidsComputeNode.js    // Compute shader node for Boids behavior
 │   └─ ...other shader logic packaged as nodes
 ├─ systems/
 │   ├─ PhysicsSystem.js       // Manages physics update loop (could integrate
multiple solvers)
 │   ├─ FieldSystem.js         // Handles global fields (wind, vortex, etc.)
affecting particles
 │   ├─ InteractionSystem.js   // Handles user interactions (e.g. attractors
from input devices)
 │   └─ PostProcessing.js      // Sets up postprocessing passes chain using
PassNodes
 ├─ examples/
 │   ├─ basicUsage.js          // Minimal example: create engine, spawn
particles
 │   ├─ morphTargets.js        // Example showing shape morphing
 │   ├─ audioReactive.js       // Example linking audio FFT to particle
behavior
 │   └─ ...etc.
```

Each file exports a self-contained class or set of functions. For instance, `ParticleEngine.js` exports the main `ParticleEngine` class; `PBS.js` might export helper functions for position-based constraints; `NoiseFieldNode.js` exports a function that returns a Three.js Node representing a noise field in the shader. By organizing this way, developers can **"hot-swap"** modules – meaning you can modify or replace one part (say, the physics solver) without affecting others, as long as you adhere to the interfaces. During development, tools like Vite or Webpack can pick up module changes and live-reload just that portion of the code.

Core vs. Node Modules: We distinguish between *core logic* (JavaScript that orchestrates things, e.g. deciding when to spawn particles or call a compute shader) and *node-based shader logic* (TSL code that runs on the GPU). Shader code is encapsulated in Node classes/graphs under `shaders/` or within Material classes. For example, `ParticleMaterial.js` might create a `PointsNodeMaterial` (or `SpriteNodeMaterial`) and assign its `colorNode`, `positionNode`, etc., linking it with data buffers. This separation means you could swap out a shader by importing a different Node or Material module, without altering the engine loop.

File Naming and Standalone Design: All modules use clear naming and export defaults where appropriate. Each component should be usable on its own: for example, `EffectMaterial.js` (for post FX) could be imported and applied to a scene even without the full engine, if someone just wants the Bloom or DOF effect. This standalone design is facilitated by Three.js's examples infrastructure – our modules mirror that approach (like how Three's `examples/jsm` provide modular pieces). We avoid single huge scripts; instead, multiple small modules keep concerns separated (rendering, physics, interaction, etc.).

Hot-Swappability: Because each module is independent, you can iterate rapidly. If you're tweaking the physics solver, you only need to reload `PhysicsSystem.js` and the engine will use the new behavior. For instance, you might have multiple physics solvers (XPBD, SPH, etc.) in development and switch which one `PhysicsSystem` imports. Similarly, `ParticleMaterial` could have variants (for example, a basic one vs. a PBR one) and you can configure the engine to use a different material class for rendering without touching the core. This modular architecture is crucial for experimentation in creative coding and ensures the system can be extended easily by adding new modules.

In summary, the directory structure and JSM components make the engine **extensible and maintainable**. Next, we'll explain each major part of the engine in depth, starting with the core particle engine and its GPU compute pipeline.

Core Engine: ParticleEngine and GPU Compute Pipeline

At the heart of the system is the `ParticleEngine` class. This core class is responsible for orchestrating particle simulation and rendering, managing data buffers on the GPU, and scheduling compute and render operations each frame.

Data Buffers: Unlike CPU-based engines, our particle engine keeps particle state entirely in GPU memory using buffer objects. Each particle might have properties such as *position*, *velocity*, *age*, *mass*, *color*, etc. These are stored in GPU buffers (either as shader storage buffers or textures). For example, we might allocate a float32 array for positions (`positionsBuffer`) and another for velocities (`velocitiesBuffer`), each with `numParticles` elements. Three.js's TSL provides abstractions to treat these as **attribute or storage nodes**. In code, you can create an `AttributeNode` or use `instanceStorage` (for per-instance data) to pass these buffers to shaders. WebGPU's flexibility means we are not limited to textures for storage – we can use large raw storage buffers for arbitrary data. TSL even allows us to define **structured buffer types** and manipulate them easily in node code. The engine's initialization will create these buffers and upload any initial data (for example, initial positions might be randomly distributed in a sphere or on a shape).

Compute Shaders via TSL Nodes: The **simulation step** (updating particle positions, velocities, etc.) is done with GPU compute. Three.js doesn't expose a raw compute pipeline API directly; instead, we leverage TSL's ability to run code in the compute stage. Specifically, TSL provides a `Fn` (function) node constructor that can define a snippet of shader logic and then **invoke it as a compute shader** by calling `.compute(count)` [10] [11]. For example, to initialize particles we might do:

```
import { Fn, vec3, rand, uniform, instanceIndex } from 'three/tsl';

const computeInit = Fn( () => {
    // Pseudocode: assign random spawn position and reset age
    spawnPos.assign( vec3( rand(), rand(), rand() ).mul(6.0).sub(3.0) );
    velocity.assign( vec3(0.0) );
    age.assign( rand() );
} )().compute(numParticles);
```

This defines a GPU function that, for each particle index from 0 to `numParticles-1`, writes random values to that particle's spawn position, zeroes its velocity, and sets a random age. The `instanceIndex` is a built-in node providing the current index within the compute dispatch. We then call `compute(numParticles)` to create a **ComputeNode** representing this operation on the given number of instances. Finally, we instruct the renderer to execute it with `renderer.computeAsync(computeInit)` [11] (in an R3F context, `useThree(state => state.gl)` gives the renderer). The engine does this at startup to seed initial state.

This approach leverages Three.js's node system to greatly simplify GPGPU workflows. In earlier WebGL approaches, one had to encode data into textures and write separate fragment shaders to ping-pong them. With TSL, we directly run a compute function that can read and write to buffers in a straightforward manner [12]. The node system handles generating the appropriate WGSL code and ensures no read-write conflicts by using multiple buffers if needed.

Update Loop and Compute Pipelines: After initialization, `ParticleEngine` sets up an *animation loop* (using `WebGPURenderer.setAnimationLoop` or R3F's `useFrame` in a React context) to advance the simulation each frame. In each tick, one or more **compute shaders** update the particle data. For example, a basic update might increment positions by velocity and decrease life ages:

```
const computeUpdate = Fn( () => {
    // Simple Euler integration
    velocity.addAssign( acceleration.mul(deltaTime) );
    position.addAssign( velocity.mul(deltaTime) );
    age.addAssign( deltaTime );
    If( age.greaterThan(lifetime), () => {
         age.assign(0.0);
         position.assign(spawnPos); // respawn
    });
} )().compute(numParticles);
```

Here we use a `deltaTime` uniform node (TSL provides `deltaTime` automatically as the time since last frame [13]) and simple physics integration. We also show a conditional (the `If` node) to reset particles that exceeded their lifetime – if age > lifetime, set age to 0 and teleport the particle back to a spawn position [14]. After constructing this compute node, each frame the engine calls `renderer.compute(computeUpdate)` (or schedules it via an onBeforeRender callback) so that GPU updates happen before drawing.

Often, multiple compute passes are needed. For example, one pass might compute forces (like a physics solver computing new velocities), and another might integrate those velocities to update positions. With WebGPU, we can run many small compute dispatches per frame if needed. Our engine's `ParticleEngine` manages an array of compute tasks that should run each frame (populated by the physics system or field system). It ensures they execute in the correct order – thanks to WebGPU's command scheduling, the GPU will run them sequentially as given.

Orchestration and Rendering: Once compute updates complete for the frame, the engine renders the particles. Rendering is handled by a **ParticleMaterial** (see next section) attached to a Three.js object – typically we use a `Points` or `Sprite` system to draw many particles efficiently. In Three.js with nodes, there is a convenient `PointsNodeMaterial` (or using a `<sprite count={...}>` in R3F which internally instantiates an instanced points cloud [15]). The `ParticleEngine` sets up a single mesh that represents all particles (either a large `InstancedMesh` of quads for each particle, or a `Points` geometry). The geometry can be something like a unit quad or just point indices, and the custom material's shader will handle fetching each particle's data from the buffers to display it.

We attach our data buffers to this mesh in the form of attributes or via the NodeMaterial's inputs. For example, if using `SpriteNodeMaterial` (which is a material for point sprites in the node system), we can supply a `positionNode` that is the sum of a base spawn position and an offset (for moved position) [16]. In our code above, `spawnPos` and `position` might correspond to buffers, and we tell the material to use `spawnPos + offset` as the particle's rendered position. Similarly, a `colorNode` can be provided (e.g. each particle could have a color or one derived from other data), and a `sizeNode` (scale) to control sprite size. The engine handles creating these nodes and linking them to the material's inputs.

Finally, the `ParticleEngine` ensures everything updates in sync: it may call `computeUpdate` and then immediately trigger `renderer.render(scene, camera)`. Because the WebGPURenderer batches commands, the compute shaders will execute and their results will be available by the time the draw call runs (the engine internally manages GPU sync). The result is a smooth animation loop fully on GPU. If WebGPU is not available, the fallback ensures the compute nodes run in a WebGL fragment shader behind the scenes, enabling broad compatibility [3].

In summary, **ParticleEngine.js** sets up buffers, defines TSL compute functions for init and per-frame updates, and ties those into the render loop. All heavy lifting – moving particles, solving physics – is done in parallel on the GPU for potentially **hundreds of thousands** of particles at high FPS [12]. Next, we'll cover the materials used to render these particles and the post-processing effects pipeline.

Particle Materials and Post-Processing Effects

Visual quality is as important as simulation. The engine provides custom **NodeMaterials** for rendering particles and for full-screen post-processing effects, leveraging Three.js's node graph for flexibility.

Particle Rendering Material

ParticleMaterial (Sprite/Points NodeMaterial): For drawing particles, we use Three.js NodeMaterials designed for sprites or points. Three.js r177 includes `SpriteNodeMaterial` and `PointsNodeMaterial` classes which correspond to `THREE.SpriteMaterial`/`THREE.PointsMaterial` but with node-based customization. In our engine, `ParticleMaterial.js` creates an instance of `SpriteNodeMaterial` (sprites billboards are useful for particles because they always face the camera and can use textured point shapes). We then assign its inputs:

- **positionNode:** A node that supplies the position of each particle. Typically this is something like `spawnPosition + offsetPosition` as mentioned. We created those as buffer-backed nodes in the compute pipeline. By assigning `material.positionNode = spawnPos.add(offsetPos)`, the material's vertex shader will use that sum as the particle's world position [17].
- **colorNode:** We can supply either a uniform color or per-particle colors. In a simple case, `uniform(color("white"))` gives every particle a base color [18]. More dynamically, we might have an attribute for color or derive it from other data (age could map to color, for instance).
- **size/scaleNode:** This controls particle size. For example, to randomize size we could use a TSL `range(min,max)` node (which gives a random scalar per instance) [19] [20]. We assign `material.scaleNode = vec3(range(0.001, 0.01))` so each particle gets a random scale. If we want size to change with time, we could instead compute a size factor in the update shader and store it in a buffer.

Additionally, we set **blending and transparency** on the material: for additive effects (like glowing particles), use `material.blending = THREE.AdditiveBlending; material.transparent = true; material.depthWrite = false;` [21]. Depth write is off so that bright particles don't obscure others and blending looks correct. The NodeMaterial honors these render states just like a normal material.

One key advantage of NodeMaterials is that we can incorporate custom effects easily. For instance, if we want a fresnel highlight on each particle, we can multiply the particle color by a fresnel node (which depends on view angle). Or we could apply a texture map to each sprite by setting `material.mapNode = texture(spriteTexture, UV)` using TSL's `texture()` function. The TSL node library includes many such nodes (noise, trigonometric, lighting models, etc.), letting us create rich particle appearances without leaving JavaScript. For example, a per-particle animated color could be done by `material.colorNode = colorA.mix(colorB, sin(time + instanceIndex))` to oscillate between two colors.

EffectMaterial and Postprocessing Nodes

Beyond particle rendering, our engine supports a suite of **post-processing effects** using Three.js's new node-based postprocessing pipeline. Instead of the old EffectComposer with passes that use separate shaders, in WebGPU we use **PassNode** and related node classes to define fullscreen effects in the same node graph system as materials.

EffectMaterial (PassNode based): `EffectMaterial.js` in our structure encapsulates a custom post-processing pass. Under the hood, a **PassNode** is essentially a node that can render a full-screen quad and produce an output texture (or multiple, for MRT). We can create a PassNode by writing a TSL function that processes input textures (like the scene color, depth, etc.) and returns a new pixel color. For example, a simple grayscale effect could be:

```
const grayscaleFn = tslFn( () => {
    const color = viewportSharedTexture(); // get rendered color of the scene
    const luminance = color.r.mul(0.299).add( color.g.mul(0.587) ).add(
color.b.mul(0.114) );
```

```
    return vec4( luminance, luminance, luminance, 1.0 );
});
```

Here `viewportSharedTexture()` is a TSL function that fetches the current frame's color buffer [22]. We compute luminance by an NTSC weight and output a vec4. Wrapping this in a PassNode would yield a postprocess pass that turns the scene grayscale. In code, we might do: `const grayPass = new PassNode(grayscaleFn());`.

Our `EffectMaterial` class likely handles chaining multiple such passes. It could compose nodes for **Depth of Field (DOF)**, **Bloom**, **Motion Blur**, **Screen-Space Reflections (SSR)**, etc. Many advanced effects require multiple inputs and multiple render targets (G-buffers). Fortunately, Three.js nodes support multi-render-target automatically: you can request that the scene render normals, depth, velocity etc. in one go [9]. For instance, a **Motion Blur** effect might need a velocity buffer. Our engine can configure the primary scene pass (ParticleMaterial and any other materials) to output velocity (each particle's last frame position minus current, encoded as a color) alongside the beauty color. Three.js's r167+ PassNode system can set this up so that by the time the motion blur node runs, it has access to a texture with velocities [9].

Let's outline some core effects and how we implement them with nodes:

- **Depth of Field (DOF):** We simulate camera focus blur by blurring parts of the image based on depth. With nodes, we can get a linear depth buffer via `viewportLinearDepth()` [23]. A DOF PassNode might compute a circle-of-confusion value from depth (how out-of-focus a pixel is) and then sample the scene color with an aperture blur. We could utilize a provided `gaussianBlur()` node – TSL has a built-in example of a double-pass Gaussian blur node that can be invoked in one line [24]. For DOF, one strategy is:
- Use a PassNode to create a blurred version of the scene (maybe two passes: one for near-blur, one for far-blur).
- Use another node to interpolate between sharp and blurred color based on each pixel's depth relative to focal range.

The key is that the node system allows *creating new render passes on the fly* in shader code. The `gaussianBlur()` function, for instance, internally does two passes (horizontal/vertical) but as a single node call from the user perspective [24]. We leverage that for DOF.

- **Bloom:** Bloom adds a glowing halo to bright parts of the image. Traditionally, one would extract bright areas, blur them, and add back. In our engine, we do similar using nodes:
- A PassNode to extract brightness (threshold filter on `viewportSharedTexture()`).
- A `BlurNode` (e.g. gaussian blur node) to spread the bright highlights [24].
- Finally, blend the blurred highlights additively onto the original image (this can be done in the same PassNode by just returning `originalColor + blurredColor * intensity`).

We can configure bloom radius/intensity via uniform values in the node graph. Three.js's node library might already include a BloomNode or similar (if not, it's easy to construct as above).

- **Motion Blur:** Requires the scene's velocity buffer (we can get per-pixel motion vectors either from previous vs current position in the shader or by rendering velocity in an MRT). A MotionBlur PassNode would, for each pixel, sample along the motion vector direction across a few steps from the previous frame's image. In TSL, we can achieve this by using `viewportSharedTexture()` (current frame) and also possibly storing the last frame's color in a persistent buffer. Alternatively, some implementations accumulate motion blur over frames. For simplicity, one can approximate motion blur by taking the current color and blending it with the previous frame's color buffer using the velocity as weight (this requires storing last frame as a texture uniform to the next frame's pass). The node system's flexibility allows custom strategies, though it might require writing a bit of custom logic or multiple pass nodes.

- **SSR (Screen Space Reflections) & SSSR:** Screen-space reflections use the depth and normal buffers to reflect the scene on shiny surfaces. Implementing SSR in nodes is complex but doable:

- Use the normal and depth to compute a reflection ray for each pixel of a reflective material.
- March that ray in screen space (sample along the ray direction in the color buffer or in a hierarchical depth buffer).
- If it hits, use that color; if not, fall back to environment map or nothing.

With TSL, one approach is to do a few fixed steps of ray marching in a loop. As of now, TSL doesn't allow dynamic loops, but you can unroll a small fixed loop with nodes. Another simpler approach: there is a library by community (like 0beqz's SSRPass for WebGL) – in our engine context, SSR could be integrated as an optional add-on using a specialized node or even a custom WGSL shader if needed. We mention SSR because high-end visuals often demand it (reflections of particles or environment). Modern engines consider it essential [25]. In practice, SSR in a particle-heavy scene might not be critical (since particles are volumetric and light-scattering rather than mirror-like), but if particles land on surfaces, SSR would make puddles or shiny floors reflect them, enhancing realism.

- **Other Effects (SSAO, etc.):** Our architecture can support SSAO (ambient occlusion) by rendering a normal+depth G-buffer and running an AO PassNode (sampling depth neighborhood). Because we focus on particles and volumetrics, AO might be less relevant (particles typically simulate their own shading). But if needed, the pattern is similar: nodes to generate required buffers and nodes to combine them.

Composing Passes: We typically chain these effects. The engine's `PostProcessing.js` (or similar) can instantiate multiple PassNodes and feed one's output into the next. In Three.js's new pipeline, this might be as simple as adding multiple nodes to an array and letting the renderer know to apply them in sequence (or using a `NodeFrame` to manage it). For example:

```
const dofPass = new PassNode( dofFunction() );      // Node that blurs based on
depth
const bloomPass = new PassNode( bloomFunction() );  // Node that adds bloom
bloomPass.inputTexture = dofPass;  // use DOF output as input to bloom
renderer.addPass( dofPass );
renderer.addPass( bloomPass );
```

The specifics of API may differ, but conceptually that's how we string them together. The final output after the last pass goes to screen.

Custom Effects with TSL: One of the biggest benefits of TSL is the ease of writing new effects. We can tap into *renderer buffers* like the scene color or depth with one line (as shown), and even create multi-pass effects encapsulated in one function. For example, a **glitch effect** might periodically offset the screen pixels or drop scanlines – that can be written as a small function using sin/cos on UV coordinates and sampling `viewportSharedTexture()`. Another creative effect: **edge detection outline** – sample depth, compare with neighbors to find edges, then draw an outline. The forum example of someone porting an OutlinePass to nodes [26] [27] illustrates how one can create complex multi-step post fx (masking objects, blurring edges) using custom Node classes. While that was a work-in-progress, it shows that with some ingenuity, anything from filmic tonemapping to color grading to CRT filters can be done by combining node operations.

In summary, **EffectMaterial** and the postprocessing node pipeline allow us to achieve high-quality visuals (DOF, Bloom, Motion Blur, SSR, etc.) in a unified, GPU-accelerated way. Modern engines expect these out-of-the-box [25], and our engine's node-based approach means they integrate seamlessly with the particle rendering. Next, we'll explore the physics and field subsystems – how the engine drives particle motion through various simulation models.

Physics, Fields, and Fluid Simulation Systems

Particle effects often require physics-based motion to look natural. Our engine includes a **Physics System** supporting multiple solver types, as well as customizable **force fields** and even fluid dynamics, all implemented in GPU compute shaders (via TSL nodes).

Physics Solvers (XPBD, SPH, MPM, Boids)

We provide or outline several physics simulation techniques that can be applied to particles:

- **XPBD (Extended Position-Based Dynamics):** XPBD is an advanced constraint solver that iteratively adjusts particle positions to satisfy constraints (distance springs, volume preservation, etc.) [28]. In our engine, XPBD can be used for soft bodies or cloth-like particle systems. Implementing XPBD on GPU involves iteratively running a constraint relaxation shader multiple times per frame. For instance, if you have particles connected by springs, a compute shader can enforce each spring's rest length by moving the two particles accordingly. XPBD adds compliance (stiffness control) and can handle constraints like rigid rods, bending, etc. Our `PBS.js` (Position-Based Simulation) module provides building blocks: for example, a function to solve distance constraint between two particle indices. You can call this in a loop within a `Fn` node. Because GPU programming excels at parallel, you might handle each constraint in parallel, but note constraints on the same particles need multiple iterations for convergence. In practice, we might alternate constraint sets or run the compute shader a few times. XPBD's advantage is stability with larger time steps, which is useful if the frame rate varies. It also can generate forces if needed (for coupling with other systems). We offer an XPBD example configuration where particles form a cloth: neighbor constraints maintain the fabric structure, and a separate gravity force is applied each frame, all via compute.

- **SPH (Smoothed Particle Hydrodynamics):** SPH is a fluid simulation method treating particles as fluid elements, computing densities and pressures to make them flow. In our engine, an SPH solver would involve for each particle:

- Compute density by summing a kernel function of distance to neighboring particles.
- Compute pressure based on density (equation of state).
- Compute forces from pressure and viscosity and apply to velocity.

Doing this on GPU for potentially thousands of neighbors is heavy. Efficient SPH requires spatial acceleration (like a grid or spatial hashing) to limit neighbor checks. We can implement a uniform grid in a compute shader: have another buffer that bins particles into cells, or use a sorting approach. A simpler but less optimal route is to limit influence radius and just brute-force compute in a single kernel using distance checks – feasible for moderate counts. For example, a TSL compute function for SPH might double-loop (which would be unrolled or use GPU subgroup operations) over nearby particles and accumulate density. WebGPU's compute shaders allow workgroup shared memory which could accelerate this, but TSL doesn't directly expose that yet – so an SPH node might be complex. Nonetheless, a basic SPH is within reach: our engine might include a `SPHComputeNode` that, given positions and an acceleration buffer, adds SPH pressure forces. We might cite known implementations for guidance [29] – e.g. using a uniform grid to find neighbors efficiently (as done in a BabylonJS WebGPU boids demo which could simulate millions by uniform grid [29]).

- **MPM (Material Point Method):** MPM is another technique combining particles with a background grid to simulate continuum materials (like soft solids, snow, or sand). In MPM, particles carry properties (mass, velocity, deformation gradient) and at each step deposit these onto a grid, the grid solves forces (e.g. elastic response), then particles are updated from the grid. Implementing full MPM is complex but doable on GPU. Our engine's modular approach means one could plug in an MPM solver module if needed for advanced effects like sand piles or cloth tear. A simplified MPM example: a snow simulation where particles stick together but also break under stress. We might not include a full MPM out of the box, but we design the system such that one can integrate it by writing appropriate compute nodes (for grid operations, one can use 2D/3D textures or buffers as grids). The `PhysicsSystem` can coordinate multiple passes (particle-to-grid, grid solve, grid-to-particle). Because this is an advanced topic, our documentation might outline how one would set this up rather than provide complete code.

- **Boids (Flocking Behavior):** Boids are an algorithm for simulating flocking (birds, fish) by applying three rules: separation (avoid crowding neighbors), alignment (steer towards average heading of neighbors), and cohesion (steer towards the group's center). We can simulate boids as particles with velocity and apply these rules each frame. Like SPH, this benefits from neighbor queries (each boid needs to consider nearby boids). On GPU, this again means either brute force O(N^2) or a spatial grid to limit neighbors. Impressively, GPU boids can reach millions with proper acceleration structures [29]. In our engine, we can provide a simpler boids implementation for a few thousand particles as a demo. We could have a `BoidsComputeNode` that for each particle, reads from a structured buffer of positions and velocities to accumulate the three desired steering vectors (with conditionals or weights for neighbors within a radius). For example, pseudo-code inside a compute shader:

```
alignment = vec3(0);
cohesion = vec3(0);
separation = vec3(0);
int neighborCount = 0;
for (int j = 0; j < numParticles; ++j) {
   if (j == i) continue;
   vec3 diff = position[j] - position[i];
   float dist2 = dot(diff,diff);
   if (dist2 < radius*radius) {
       neighborCount++;
       alignment += velocity[j];
       cohesion += position[j];
       separation -= diff / dist2;
   }
}
if (neighborCount > 0) {
   alignment = normalize(alignment/neighborCount);
   cohesion = normalize((cohesion/neighborCount) - position[i]);
}
// combine these vectors into acceleration
acceleration[i] = alignWeight*alignment + cohWeight*cohesion +
sepWeight*separation;
```

This logic can be translated into TSL nodes. One might use a technique of splitting the loop into batches to avoid exceeding shader loop limits. The engine could also demonstrate an optimization where space is divided into grid cells (like the referenced approach using a uniform grid to bring complexity down dramatically [29]). Our default boids example might not scale to millions, but we document that scaling trick for advanced users. The result of boids is very visually interesting emergent behavior – flocks, schools, or even crowd motion. We can easily combine it with other forces (wind fields, predator-prey behavior) by just adding more terms to the acceleration in the node.

Each of these solvers can be toggled or combined. The **PhysicsSystem** could maintain a list of active solvers, each providing a compute node or set of nodes to run. For example, you might run an XPBD solver for some constraints (like keeping a group of particles in a shape), then an SPH solver to add fluid pressure, then a boids rule to add some flocking tendency – all on the same particles, producing a hybrid behavior. Because all solvers ultimately output forces or position adjustments, they can simply sum their contributions into the particle velocity or position.

Field Systems: Forces and Emission Fields

Beyond explicit physics solvers, our engine allows arbitrary **force fields** to influence particles. Fields are continuous functions of position (or other particle attributes) that produce a force or velocity. They can simulate wind, turbulence, attraction, etc.

Some field types and their implementation:

- **Vector Fields:** A vector field assigns a velocity or acceleration vector to every point in space. This could be as simple as a constant wind (`v(x) = (windX, windY, windZ)`) or something spatially varying. For example, a gravity field centered at the origin would be `v(x) = -g * (x / |x|)` (attractive toward center). In the engine, vector fields are implemented as functions (TSL nodes) of position. You might write a node function `vectorField(pos)` that returns a vec3. This can be used in the compute shader: `velocity.addAssign(vectorField(position).mul(deltaTime))`. Because it's node-based, we can combine fields easily. If you want wind plus gravity, just add those vector results. Our `FieldSystem.js` can hold a collection of active fields and sum them for each particle each frame.

- **Vortex Fields:** A vortex field causes rotation around an axis or point. For instance, a vortex around the Y-axis could be `v(x,y,z) = ( -ω*(z - z0), 0, ω*(x - x0) )` for a vortex centered at (x0, *, z0) with angular speed ω. This will make particles orbit that axis. We implement it as a node: given position, compute perpendicular velocity. Another example is a swirl around a point** in 3D, which might involve a vector cross product with a direction vector to get a circular motion. The engine might include a `VortexFieldNode` where you input the center and strength.

- **Noise Fields (Flow Noise, Curl Noise):** Noise is widely used for natural-looking motion (like fire, dust, magic effects). A 3D Perlin or Simplex noise field can drive particle acceleration randomly but smoothly. For example, `v = noise3D(position + time)` could give a Perlin noise vector that changes over time, making particles drift in a turbulent flow. Even better is **curl noise**, which is divergence-free and creates swirling motion (often used for fluids and smoke). We can precompute a curl noise vector field or compute it on the fly in the shader (taking spatial derivatives of a noise scalar field). Three.js's node library might have a noise node; if not, we can implement one using permutations or a 3D texture of noise. A `NoiseFieldNode` can output a vector given position and time, and we scale it by a "strength" uniform. By combining curl noise with an updraft vector, you get a believable rising smoke effect for example. These fields are purely functional – easy to add to any simulation.

- **Physarum (Slime Mold) Simulation:** Physarum is an example of an agent-based simulation where particles (agents) move based on pheromone fields they secrete, creating vein-like patterns. While not a traditional "field" in the sense of a static function, it involves a *trail field* (usually a grid/texture) that agents read and write. We mention Physarum because it's popular in generative art. Implementation: Each particle has an orientation and moves forward depositing some "trail" into a trail texture. The trail diffuses and evaporates over time (this can be done with a compute shader on a texture – essentially a blur and fade). Agents turn based on local trail concentration (towards higher concentration or away, depending on model, to form networks). In our engine, one could set up a Physarum system by using an offscreen texture as the field and an additional compute shader for the trail. The particles themselves move according to simple rules (if front sensor detects more trail on left vs right, turn accordingly, etc.). This requires multiple passes: one pass to move particles and mark where they deposit, and one pass to diffuse the trail. TSL can handle this with storage textures (we can use `StorageTexture` nodes for the trail). We won't detail full code here, but rest assured the engine can accommodate it – it's a mix of particle updates and a field update. The result is fascinating biological patterns.

For each field or specialized simulation like Physarum, our system's approach is to encapsulate it in a node or module. You might have `NoiseFieldNode`, `VortexFieldNode`, `PhysarumSolver` etc. The `FieldSystem` can invoke these. For example, `FieldSystem.update()` might construct a composite acceleration for each particle by adding up contributions from all active fields (each being a node expression). This composite acceleration is then passed to the physics update compute to apply to velocity.

To illustrate with a small shader snippet, here's how you might apply a gravity towards (0,0,0) and a noise field:

```
const gravityCenter = uniform(vec3(0,0,0));
const gravity = position.negate().normalize().mul(9.8); // 9.8 m/s^2 towards
origin
const turbulence = simplexNoise3D(position.add(time));   // 3D noise vector
const fieldAccel = gravity.add( turbulence.mul(2.0) );   // combine, weight
noise by 2
velocity.addAssign( fieldAccel.mul(deltaTime) );
```

This would make particles get pulled inward while also jittering around due to noise. Such combinations are endless and are part of what makes the engine appealing to creative coders: you can "layer" behaviors without rewriting core code, simply by enabling different field nodes.

Shader Code Examples: We include examples in the documentation for representative cases. For instance:

- **Boids steering (pseudo-TSL):** (as explained above) showing how alignment, cohesion, separation would be computed. Due to length, we might not list the entire loop code, but we describe it conceptually, as done, and note that advanced GPU techniques can handle many neighbors [29] .

- **XPBD distance constraint:** as pseudo-code:

  ```
  // positions and prevPositions for Verlet integration
  let idxA = constraint.a, idxB = constraint.b;
  let restLength = constraint.rest;
  let pA = positions[idxA], pB = positions[idxB];
  let delta = pB.sub(pA);
  let dist = length(delta);
  let diff = (dist - restLength);
  let correction = delta.normalize().mul(0.5 * diff); // split correction
  pA.addAssign(correction);
  pB.subAssign(correction);
  ```

  This might run in a loop for all constraints. In TSL, you'd use something like a storage buffer of constraint data and iterate; however, TSL might not have native support for variable-length loops over an array of constraints easily. A workaround is to dispatch a compute shader with one thread per constraint, which is doable if constraints are separate objects (like cloth springs). That might be a separate compute node outside the main particle loop.

The documentation provides these snippets as guidance so developers can implement or adjust them. We emphasize that **GPGPU techniques** let us run these heavy physics calcuations in real-time. For instance, GPU-based neighbor search can allow **tens of thousands of interacting particles** in fluid or flocking simulations that would be impractical on the CPU [30] .

Finally, we note that the Physics and Field systems are optional – you can mix and match. If you just want a swirling particle field with no inter-particle physics, you might use only a vortex and noise field and turn off XPBD/SPH solvers. If you want a realistic fluid, you focus on SPH and maybe some viscosity fields. The engine's modular design supports this flexibility.

Lighting, Cameras, and Rendering Techniques

Although particles are often self-illuminated or simple in shading, our engine supports advanced rendering features to integrate particles into rich 3D scenes.

Lighting Systems (Tiled & Clustered Lighting)

When dealing with many light sources (imagine hundreds of small light-emitting particles or interactive lights in a scene), the traditional forward lighting approach (looping over all lights for each pixel) becomes a bottleneck [31] [32]. We incorporate **clustered lighting** techniques to handle numerous lights efficiently [33].

Clustered Forward Rendering: The idea is to subdivide the camera's view frustum into a 3D grid of clusters (for example, 16×9×24 clusters covering the view). In a preprocessing compute pass, we assign each light to the clusters it affects (by checking light bounding volume overlap with clusters) [33]. Then, the fragment shader for particles (or any material) only iterates over the lights in the cluster of the pixel being shaded, instead of all lights. This drastically reduces computations when lights are distributed, as demonstrated: 1024 lights with naive loop ~50 FPS vs. clustered ~240 FPS on the same hardware [34] [35].

In our engine, we can implement clustered lighting by adding a WebGPU compute step each frame: 1. If lights moved or camera moved, recompute cluster-light lists. We divide the frustum (TSL might not have built-in for this, but we can do it in JS or a compute shader) and for each light record an index in a cluster buffer. 2. In the particle material's node graph, instead of using Three.js's default lighting, we use a custom lighting node that reads from a **structured buffer of lights** and uses the cluster index (which can be computed from the fragment's position/depth) to only sum relevant lights.

We likely use simpler lighting models for particles (like basic Lambertian or even unlit with additive glow), but for volumetric effects or if particles should respond to scene lights (say you have point lights that color the particles), this is important. Three.js might not have cluster lighting built-in yet for WebGPU, but experiments exist [36]. We can incorporate code or ideas from Toji's WebGPU clustered shading demo [37] or Usnul's implementations. Specifically, in TSL one could write a function `applyLights(clusterId, fragPos, normal)` that loops through a list of light indices for that cluster and accumulates illumination. Our engine can maintain buffers `clusterOffsets` and `clusterLightIndices` akin to known clustered shading data structures. This is an advanced option; for many simpler scenes, standard lighting with a few lights is fine, but it's good to have scalability.

Tiled Shading: A simpler variant is 2D tiling (screen space tiling) where you divide the screen into tiles (e.g. 16×16 pixels) and assign lights to tiles based on their screen projection (often used in Forward+ rendering). This we could also do, but clustered (adding depth slicing) is more robust for 3D. Either way, a compute shader can cull lights per tile/cluster.

For now, we document cluster lighting as an available technique if the user needs *lots of lights*. If a scene only has say 5-10 lights, default is fine. If they want *hundreds*, enabling cluster lighting in engine settings will engage the compute pass and a specialized nodeMaterial chunk.

NodeMaterial for Meshes and Volumes

While our focus is particles, the engine can coexist with mesh objects in the scene. **Mesh NodeMaterials** allow you to apply the same node-based shader logic to regular geometries. For instance, if you want your particles to interact or have the same style as some meshes (say, a character dissolving into particles), you can use NodeMaterials on those meshes to achieve effects like vertex animations, custom lighting, etc.

Three.js r177 provides `MeshBasicNodeMaterial`, `MeshStandardNodeMaterial`, etc., which mirror the normal materials but let you override or extend parts via nodes. An example use: You want a mesh to fade out where it intersects particles (for a dissolve). You could use a distance field or some shared texture. NodeMaterial makes it easier – e.g. you could feed the particle system's density field as a uniform to a mesh's fragment node to modulate opacity. Because all share TSL, it's easy to link them.

For volume rendering (e.g. rendering a 3D scalar field or smoke volume), NodeMaterials can help but volume rendering often involves ray marching, which might not be trivial to express in pure nodes due to loops. However, one can sample a 3D texture (if you have one representing the volume density) along a ray in increments. If performance allows, you can unroll e.g. 32 steps. Alternatively, another approach: use *compute shaders to render volumes to slices or point clouds*. For instance, you can spawn particle sprites inside a volume and make them semi-transparent to approximate a volumetric effect (sometimes called *particle slicing*). Our particle engine is actually great for volumetrics: you can treat each particle as a voxel or point in the volume. If you have an SDF (Signed Distance Field) or density field, you could initially populate particles in that shape. For dynamic volumes (like evolving smoke), you might actually simulate it with particles (SPH or vortex methods) rather than a voxel grid.

However, for completeness, if one wanted a true volume raymarch using NodeMaterial, you'd likely use a `ShaderNode` (literal code node) or multiple nested `If` to simulate a loop. This is an advanced use beyond typical nodes. We mention that with NodeMaterials, one could create custom effects like **ray-marched metaballs or SDFs** by constructing the ray traversal in the shader. Also, WebGPU allows bind groups with storage textures, so you could bind a 3D texture of density and do binary search for surface, etc.

In summary, NodeMaterials ensure that any **mesh** or **volume** in the scene can share the same shading language as the particles. This unification is beneficial; for example, you can have a **NodeMaterial for floor** that reads particle impacts (maybe via a shadow map or a compute result) to darken areas where particles landed – creativity is the only limit.

LOD and Transparency Handling

Level of Detail (LOD): For particle systems, LOD usually means reducing detail for distant or offscreen particles. Because our simulation is GPU-based, it's often cheap to keep all particles updating. But rendering could be heavy if you have millions of particles. Strategies: - Spawn fewer particles further away. We could implement a rule where the emitter spawns particles proportional to camera distance (or use multiple emitters for near/far). - Use **fading**: as particles get distant, make them smaller or fade out, effectively letting the rasterizer do less work. This can be done by a node that sets particle opacity based on distance (using camera position uniform and particle pos to compute distance, then perhaps `material.opacityNode = smoothstep(farFadeStart, farFadeEnd, distance)`). - Switch to simpler rendering at distance: For instance, nearby particles might use a detailed material (textured, lit), while far ones use a cheaper material (flat color or even rendered as a single billboard representing many). This requires grouping particles by distance – could be done if we had multiple particle systems for different LOD ranges. Another trick is **imp posters** (impostors) – render far particles as one combined sprite. Our engine doesn't automatically do that, but the extensible design means you can create a far-distance particle renderer (perhaps render the whole system to a textured quad when far away).

- If your particle count is extremely high, consider using **multi-rate update**: you can update nearby particles every frame, and far ones every 2nd or 3rd frame to save computation. The engine can support this by having multiple engines or by storing a "last updated" timestamp per particle and skipping as needed. Given WebGPU's power, often this micro-optimization isn't needed unless pushing limits.

Transparency Sorting: Particles are often transparent (for additive glow or alpha blending). Proper rendering of many transparent particles is tricky because of depth sorting issues. Traditional solutions include: - **Depth Sorting** each frame (sort particles by depth on CPU) – not feasible for GPU-driven particles as data is on GPU and count is large. - **Order-Independent Transparency (OIT):** Using techniques like depth peeling or weighted blended OIT. WebGPU allows multiple render targets, which we can use for OIT. One method is **Weighted Blended OIT** where each fragment contributes a weighted color and weight accumulation, avoiding explicit sorting. We could implement this by rendering particles with a custom blending that accumulates color and alpha in floating point targets, then do a final composite. Three.js doesn't have that by default, but with a custom NodeMaterial we could output premultiplied values to two MRTs (one for accumulated color, one for accumulated alpha) and then in a final pass divide color by alpha. - **Depth Peeling:** Multi-pass technique: render and peel one layer at a time. This is costly beyond 2-3 layers. Likely not used here due to complexity.

Our engine's default approach for transparency is *either* additive blending (which doesn't require sorting, as addition is commutative) – great for fire, sparks, magical glows – or alpha blending with approximate OIT. If using **AdditiveBlending**, we bypass sorting issues (everything just adds up, though depth write is off so they don't occlude correctly, but visually it works for light effects). For **regular translucent particles** (like smoke that should alpha-blend), a robust solution is weighted blended OIT. We might include an option to enable OIT which internally uses an extra pass or MRT.

In documentation, we advise: if you encounter sorting artifacts (e.g. some smoke particles rendering in front of others incorrectly), consider using the engine's OIT mode. This will use a dual-depth buffer accumulation approach to better approximate correct blending. This area is quite technical, so our engine's default might just leave depthWrite off and let blending handle it, which works "well enough" for many particle clouds especially with additive.

MRT G-Buffer and Deferred Techniques

We already touched on MRT (Multiple Render Targets) when discussing post-processing. To elaborate, enabling an MRT G-buffer means: - In one render pass, output multiple textures: typically color, normal, depth (and perhaps motion vectors). Three.js's PassNode auto-MRT can configure materials to output needed data [9]. - We could then do **deferred shading**: i.e., calculate lighting in a separate fullscreen pass using those G-buffers. Deferred shading is less common for particles (because particles are usually simple shading), but if we had many lights it's an option. Actually, combining deferred shading with cluster lighting can handle truly massive light counts.

In context, if a user wanted particle-mesh interaction with many lights and maybe shadows, a deferred approach could be beneficial. For example, render all particles to G-buffer (normal might be undefined for points, but we can assign a normal for each sprite e.g. facing camera or use some proxy), then compute lighting per pixel. This is probably overkill for most particle scenes, but the engine is flexible to do it if needed. We might include a setting `engine.renderMode = 'deferred'` that would trigger using a deferred pipeline (the engine would render nodes to G-buffer MRT then call a lighting PassNode). The cluster lighting we discussed is actually a variant of forward+, but it also benefits from MRT if we do SSR or AO.

Cameras: The engine doesn't enforce a special camera – you can use any Three.js Camera (Perspective, Orthographic). We ensure that any camera-related uniforms (projection matrix, etc.) are provided to the node materials. For VR/AR (WebXR), the engine can be used with XR cameras as well; R3F helps here by managing the XR loop. One note: when using VR, the engine might need to run the simulation once per frame for both eyes but typically you simulate once and render twice (stereo). That's fine since the particle state is global and both eyes see the same simulation. The engine is compatible with VR since WebGPURenderer supports VR multi-view rendering.

Shadows: Casting shadows from particles is a challenge (many small semi-transparent objects). By default, we might not cast shadows from particles as it's expensive and often not needed (e.g. fire particles casting shadows would produce flickering noise). If needed, one can approximate shadows by having major emitters cast a single shadow blob or use raymarching for volumetric shadows. Standard Three.js shadow maps won't handle thousands of points well. Instead, one could render particles to a shadow map as regular geometry (point sprites in shadow pass). Our NodeMaterials could be configured to output to depth in a shadow pass (though WebGPU shadows might not fully support node materials yet). For now, we assume particles **don't cast or receive standard shadows** (except maybe receive via some tricks). Instead, use other lighting cues like ambient occlusion (SSAO might darken areas of high particle density) or simply accept that particles add light but don't block it (common in effects).

To sum up this section: we equip the engine with **modern rendering techniques** to handle complex scenes. Clustered lighting ensures lots of lights are possible with little cost [38] [33]. NodeMaterials allow custom shading on meshes and even volumes, unifying the shading language across the scene [39]. We handle transparency through either simple means or advanced OIT if needed, and we leverage MRT for any effect requiring multiple buffers (depth, normals, etc.) which opens the door to deferred shading and screen-space effects. These features ensure the particle engine can integrate into a larger rendering pipeline and scale to demanding visual scenarios.

Motion, Morphing, and “Emotion” Systems

This engine not only simulates physics but also higher-level behaviors like morphing shapes or propagating abstract “emotions” through particles. These systems allow for storytelling and design-driven effects beyond basic physics.

Shape Morphing and SDF Transitions

Shape Shifting Particles: One stunning effect is to have a cloud of particles form one shape, then flow into another shape (e.g. dissolve a statue into particles that then swirl and reassemble into a logo). Our engine supports shape morphing by using **target position buffers**. Essentially, we can preload target shapes (as point clouds) and assign each particle a target position for a given shape. The ParticleEngine can then drive particles to move towards those targets. This can be done smoothly and even sequentially for multiple morphs: - Prepare N target datasets (each a set of 3D points representing a shape). - For morph, for each particle, pick a point from target shape (some mapping: often easiest is to sort points and map index-to-index, but one can also randomly assign or do nearest match for continuity). - In a compute shader, apply a steering force towards the target: `velocity += (targetPos - currentPos) * morphStrength`. - Optionally, when near target, zero velocity to "stick" the particle in place forming the shape.

We can control morph progression via a uniform (morph time 0 to 1). Alternatively, do abrupt switch: one moment target is shape A, next is shape B, and particles will naturally move. For smoother interpolation, one could blend targets or assign intermediate waypoints.

A concrete example: Morphing from a 3D text shape to another. We have arrays of positions for each shape. We feed the second shape's array as a texture or buffer and gradually interpolate positions. Node approach: we could actually store both shapes' positions in node data and do `currentTarget = mix(shape1Pos, shape2Pos, morphFactor)` in the shader, then move towards `currentTarget`. But an easier approach is simply swapping out the target buffer and letting the simulation move particles.

SDF Morphing: Using Signed Distance Fields (SDFs) is another approach. If you have an SDF representation of shapes (a function that for any point gives distance to surface), you can move particles according to the SDF's gradient. For instance, to form an SDF shape, you spawn particles randomly and then apply a force `F = -∇(distance(p))` which drives them toward the surface (the zero level-set). They will settle on the surface of the shape. By changing the SDF over time (interpolating between two SDFs), particles will move accordingly – effectively morphing. SDF morphing is smooth and doesn't require explicit point correspondences. In practice, computing ∇distance on the fly can be done if you have an analytic SDF or a grid; TSL can encode analytic SDFs (like sphere, box, etc.) fairly easily with nodes (GLSL-like operations). For more complex shapes, one might precompute SDF in a 3D texture and sample it. WebGPU being modern can sample 3D textures in shader, which node system can wrap.

The engine could provide some **SDF primitives** (sphere, box, torus, etc.) and allow blending. For example, to morph a sphere into a box, you could use a time parameter `t` and define a blended SDF: `d(p) = (1-t)*sphereSDF(p) + t*boxSDF(p)`. Not strictly correct SDF blending, but as an approximation it can guide particles. Alternatively, at t<0.5 use one shape's SDF, then switch to the other – but that might cause a jump. Some creative code can be used for smooth morph, like a signed distance union or intersection formula.

We mention SDF morphing as a powerful technique to achieve continuous transformation of particle formations. It's especially useful for volumetric shapes or when you want particles to fill not just the surface but volume of an object.

Motion Drivers and Kinetic Sculptures

Motion drivers are external inputs or procedural rules that drive particle motion in a coordinated way. Examples: - **Orbiting Paths:** Make particles move along prescribed paths (circles, curves). You can set velocity or position based on a mathematical formula rather than free physics. E.g., assign each particle an angle variable that increments each frame so they orbit a center. - **Spline Morphing:** If you have an animation path (Bezier or spline), particles could move along it or between multiple paths. For instance, a dancing particles effect where each particle oscillates around a moving anchor point on a spline. - **Kinetic Typography or Sculpture:** If particles form letters or shapes that move (like a text that waves, or a logo that twists), you can treat that as morphing frames. Possibly use a rig – e.g., have a hidden skeleton or morph targets and drive particle targets accordingly.

Our engine can incorporate these by either direct formula in the shader or by reading from a buffer that is updated on CPU. For instance, if you have an animated model, you could each frame set target positions to the model's vertex positions, so particles swarm to match the model's pose.

We encourage creative usage: the engine can mix physical and scripted motion. You might use GSAP or keyframes to animate a "ghost" object and have particles chase it. One could implement a "follow leader" behavior where one particle or an invisible object moves on a predefined path (like a spiral) and all other particles are attracted to it (like a moving gravity well). These are motion drivers at a conceptual level. Implementation is often simply computing a velocity towards the driver.

"Emotion" Propagation and Behavioral Dynamics

The term "emotion" in simulation can be thought of as any internal state that particles carry and influence each other with, analogous to how an emotion or signal spreads in a crowd. This could represent things like heat, charge, or synchronization phases (like fireflies blinking in unison), or truly some artistic notion of emotion (e.g. "excitement" value).

Propagation Mechanism: Typically, to propagate a scalar through particles, you'll have particles influence neighbors or a field. A simple model: - Each particle has an attribute E (emotion level). - Differential rule: E increases or decreases based on neighbor values or external stimulus. - Could use a diffusion-like update: each particle looks at neighbors' E, and tends towards their average (smoothing out differences) – that would simulate a diffusive spread of whatever the value is. - Alternatively, like the fireflies synchronization: if one particle "flashes" (E spikes), neighbors might also spike after a delay, creating a wave of flashes.

Implementation on GPU: you can do it similar to how one would for any neighbor-based interaction (like boids or SPH): in a compute shader, for each particle loop over neighbors within radius and sum or check their E. For large counts, again the uniform grid acceleration helps.

A specific example drawn from nature is the *Kuramoto model* for synchronization (though that's continuous phase math). A discrete alternative: if any neighbor's E > threshold, raise my E. This kind of rule can make a cascading event (like one particle "infects" others). This can be used for effects like explosion chains or lighting up a grid of particles.

Emotion as Color or Motion Modifier: In visuals, we might map this emotion value to particle color, size, or movement. For instance, you could simulate a "shockwave" through a particle cluster by having an initial particle get high energy (red and expanding) and all others gradually do the same outwards. Or imagine "mood" particles that change color when near a certain object and that spreads.

Practical Use Case – Firefly Sync: The earlier Floids example combined boids with syncing fireflies [40]. Each particle had a phase for blinking. They adjust their phase based on others (if they see a neighbor flash, they slightly advance their own clock). Over time, they synchronize and all blink together periodically. We can implement this: give each particle an "oscillator" (phase angle that increases each frame). If phase > 2π, flash (emit light) and reset phase to 0. And for sync, if neighbors flashed, add a small jump to our phase. This can all be done in a compute shader per frame. Eventually, clusters of particles start aligning their phase, resulting in waves of synchronized blinking. It's an "emotion propagation" where emotion = readiness to blink.

Buffering and Nodes: To support such systems, we ensure particles can have arbitrary custom data in buffers (like an `emotion` float). Then, either through additional compute passes or by expanding our main update shader, we include rules for updating that data. One might create a dedicated `EmotionSystem` that updates a buffer of emotion values based on neighbor influence. If the number of particles is not too high, it can be part of the main update loop. For larger, maybe a separate pass focusing only on that scalar.

Additionally, one can combine emotion with physics. Perhaps particles with high emotion move faster or repel others more, etc. Our node-based approach makes it straightforward to incorporate such couplings: just multiply a force by the emotion value or similar.

Visualization: We often map internal states to visual properties. If emotion is like "temperature," we could use a NodeMaterial to color hot particles red and cool ones blue. If it's like "happiness," maybe size or brightness changes.

In summary, "emotion" systems enable **emergent behaviors and pattern formation** in the particle ensemble, beyond physical forces. They are highly customizable: you define what the scalar or state means and how it influences others. Because our engine is built for creative coding, we give you the hooks to play with these ideas: add a buffer, write a node function to update it, and plug it into colors or forces. With some experimentation, one can simulate things like chemical reactions (A/B reaction-diffusion using two "emotion" chemicals diffusing among particles) or social behaviors (some particles act as leaders with a state that others follow).

Interaction and Animation Integration

One of the strengths of this engine is how it can respond to live inputs and orchestrate complex animations. We provide multiple integration points for external interactions:

- **MIDI Controller Input:** MIDI devices (or MIDI files) can control simulation parameters. For example, you can map a MIDI knob to the gravity strength or emission rate. Using Web MIDI API in a web context, you'd listen for control change messages and update a uniform or engine setting accordingly. The engine might have a small helper to bind MIDI notes to triggers – e.g., note C4 could spawn an extra burst of particles, or drum kicks could emit shockwaves in a water simulation. This allows live performances: a musician can literally "play" the particle system. Setup usually involves onMIDIMessage events setting values that our Tweakpane (or internal state) uses. We ensure that all engine parameters (like fields strength, solver toggles, etc.) are exposed in a way that is easy to manipulate at runtime (simple JS setters).

- **Audio FFT (Sound Reactivity):** By analyzing audio frequencies (with Web Audio API's AnalyserNode), we can drive the particles to dance to music. Common technique: get an FFT array of, say, 64 or 256 bands. Then map certain frequency bands to forces or colors. For instance, bass (low frequencies) could modulate a vertical bounce or expansion (making the whole particle cloud throb), while treble (high freq) might produce small jittery movements or color sparkles. In our engine, you might create an AudioSystem that every frame updates some uniforms like `audioBassLevel`, `audioMidLevel`, etc., which the particle shader can sample. For example, the particle size node could be `baseSize + audioBassLevel * 0.5` so they pulse with the beat. Or a field force could be influenced by music (imagine a radial explosion force when a kick drum hits). As an example from a tutorial [41] [42], you create an Analyser, get frequency data each frame, and then could feed that into a TSL uniform. Because WebGPU can handle frequent uniform updates, doing this 60 times a second is fine. We might provide utility to smooth the FFT data (smoothing constant) and maybe pick out specific ranges. The Codrops visualizer article [43] [44] demonstrates blending GSAP-driven animation with audio – in our case, we integrate by directly affecting shader parameters or spawning events.

- **Webcam Skeletons (Pose Detection):** Using libraries like TensorFlow or MediaPipe, one can get human skeletal joints from a webcam in real time. Our engine can use that data to influence particles – a popular example is particles flowing around a person's silhouette or emanating from their hands. Concretely, if you get coordinates for 2D joints (which can be projected to 3D with some assumptions or use depth camera for actual 3D), you can feed them into the simulation as attractor points. For instance, each hand could be an attractor field that pulls particles (so you can "slosh" particles by moving hands). Or the skeleton can act as an emitter – each joint emitting particles as you move (think of a dancing person leaving trails). Implementation: you'd use pose detection in JS which gives you say 33 points of a body. You then either convert those to world coordinates in the Three.js scene (if using webcam image behind, calibrate to some scale). Then in the engine, for each such point, create a field like `attractorField(point, strength)` or if you want collisions, you might even remove particles that hit the body outline. We likely don't implement the vision part (that's user's integration), but our docs advise it's doable and common. We ensure our engine can update hundreds of attractor points – if needed we can pack them into a buffer and in the shader loop through them adding forces (though 33 points is trivial).

- **VR Controllers and Pointer Interaction:** In interactive installations or VR, users might want to touch or manipulate the particle system. We can represent a controller or mouse pointer as a sphere or ray that affects particles:

- **Pointer repulsion/attraction:** The simplest: take the cursor 3D position (with raycasting if needed to find where in scene) and apply a radial force from that point. So when the user moves the pointer through the particle cloud, it pushes them out of the way (or pulls them in if attractive). This is a matter of adding a term in the update compute: `vec3 diff = particlePos - pointerPos; float dist = length(diff); if(dist < R) velocity += diff/dist * forceStrength;`.
- **Controller collision:** In VR, controllers have position and maybe a shape. We can do a soft collision – treat the controller as a sphere or capsule and push particles outside it. We could implement that via

a simple distance check per particle each frame, or use an SDF (e.g. an SDF for the hand shape, and push particles out of negative distance).
- **Painting with Particles:** If the app allows, user could spawn particles by pressing a button, injecting particles at the controller's position with initial velocity along a flick. The engine's Emitter module likely listens for such events (click or VR trigger) to emit bursts.

Our engine's design uses an `InteractionSystem` which can receive events or queries from the outside (could be callbacks like `onPointerMove`, `onControllerMove`). It then updates global fields or triggers in the simulation. Possibly it directly writes to some buffers (for example, if one wanted to pick up a particle, one might write its velocity to zero or pin it to controller position – that's advanced but possible with an ID mechanism).

GSAP Animation Integration: We touched on this earlier – GSAP is perfect for smoothly transitioning any numerical value. In our engine, many aspects can be animated: - Animate emitter properties: e.g. ramp up the emission rate from 0 to 1000 over 5 seconds for a dramatic reveal. - Animate uniform values: e.g. gradually change the target shape morph factor, or fade a field's strength. - Timeline sequences: e.g. at time 0, start with particles forming text "HELLO", then at 2 seconds morph to "WORLD", then at 4 seconds explode outward. GSAP timelines can coordinate these by scheduling function calls or uniform changes at specific times.

Since GSAP works with JS objects, we might expose a convenient API: `engine.tween({property: value})` that internally uses GSAP. Or simply instruct users to do `gsap.to(engine.particleMaterial.uniforms.someValue, { ... })`. As an example, if we want an explosion, we can animate a "globalExplosionForce" uniform from 0 to some high value and back to 0 quickly; the shader could apply a radial push when that value is non-zero. This approach separates the animation logic (done by GSAP) from the simulation logic (done by GPU), which is a good separation of concerns.

Tweakpane for Live Control: During development or for interactive art pieces, a Tweakpane UI can be provided to the end user or developer to adjust parameters on the fly. Our engine can integrate by simply passing references: - For each parameter (say gravity, wind, particle size, count), we add a Pane input that binds to the engine's config object. Changing a slider calls our code to update the corresponding uniform or setting. - Because Tweakpane can also have buttons and color pickers: we could allow color themes to be changed (e.g. particle base color), or allow toggling on/off certain solvers/effects with checkboxes.

For example, one might have a checkbox "Enable Bloom" – when unchecked, our PostProcessing system removes or disables the bloom pass. Or a slider "Boid Cohesion" that sets the weight of the cohesion force in the boids solver.

This is extremely useful for fine-tuning and also for demos to clients or live performances (one could imagine a DJ live-tweaking the visuals).

All these interactions assume the engine is running in an environment (browser) where such inputs are accessible, which is our case.

React Integration (R3F specifics): If using R3F, hooking these up is straightforward: - Use `useFrame` to update anything per frame (like reading audio data and setting uniforms). - Use React state or context for user inputs (buttons toggling features can directly call engine methods). - Drei and other libraries can supply some premade controls (like `PointerLockControls` or VRControls).

We ensure that our engine's state is not deeply hidden – e.g., `ParticleEngine` could have properties like `engine.params = { gravity: 1, wind: 0.5, ... }` which are either plain or made observables such that external code can modify them easily.

In summary, the engine is **built to be interactive**. Whether it's being driven by a music track, controlled by a user with a mouse/VR, or orchestrated in a scripted animation, we provide the hooks: - Real-time input influence via fields and triggers. - Smooth animations via GSAP on parameters. - On-screen controls via Tweakpane for quick adjustments.

By combining these, developers can create complex live experiences: imagine particles that respond to a live drummer (via MIDI or audio), swirl around a dancer's silhouette (via webcam skeleton), and all of it can be adjusted in real-time with a UI panel during rehearsal. This fusion of multimedia inputs is exactly what this engine targets.

Optimization and Extensibility

To get the best performance and to extend the engine for new use cases, we need to consider optimization strategies at both the shader and system level.

Performance Tuning Tips

- **Optimal Particle Count:** WebGPU is powerful, but it's still possible to overload the GPU with millions of particles. Find a balance between count and frame rate. Use GPU profiling (if available) to check where the bottleneck is – compute or fragment rendering. Often, if particles are large on screen, fragment shading can be heavy (overdraw). If they are tiny and many, the vertex or compute might be heavier. Adjust particle size and count accordingly.

- **Batch Computation:** When running multiple compute passes (for physics, fields, etc.), try to combine steps into one shader if possible to reduce overhead. For example, if you can update velocity and position in one kernel, do so instead of two (unless a solver's algorithm logically requires separate steps).

- **Workgroup Size and Dispatch:** WebGPU compute shaders operate in workgroups. TSL likely picks a default, but if you write low-level WGSL you'd choose something like 64 or 128 threads per group for GPU efficiency. If extending with custom WGSL, tune the workgroup size to match your GPU. Also, dispatch enough groups to cover particles; if numParticles is not divisible by group size, a few threads idle – that's fine.

- **Avoid Unnecessary Readbacks:** Ensure you don't read GPU data to CPU each frame (no `renderer.readRenderTargetPixels` or such) as that stalls the GPU. The engine as designed keeps everything GPU-side. Only read back if absolutely needed (like for some analysis or if you allow CPU picking of a particle); and even then, consider doing it infrequently or asynchronously.

- **GPU Memory and Buffer Formats:** Large particle counts mean large buffers. Use the smallest data type sufficient for each property. For instance, if a property is just a boolean or small int, consider packing it (multiple bools in one float, or use 16-bit normalized values if range allows). WebGPU supports 16-bit floats (half) and even 8-bit for some formats – TSL might abstract some of this. If using textures for storage, use `R32F` or even `RGBA16F` if appropriate. Also free any buffers/textures not needed (the engine should dispose of buffers when particle systems are destroyed to avoid VRAM leaks).

- **Dynamic vs Static Data:** If some data is static (e.g., target shape positions that never change), flag those buffers as static or put them in a texture so they can be sampled efficiently. Mutable data like positions and velocities go in storage. Separating static target info from dynamic info can avoid copying unchanging data every frame.

- **Parallelization and Async:** The WebGPU command submission is asynchronous to the CPU. Take advantage by doing minimal CPU work per frame – basically just feeding uniform updates and let the GPU do heavy work. Don't introduce CPU bottlenecks like sorting particles each frame (which we avoid entirely by doing everything in compute). If you have to update some data (like attractor position from VR controller), that's trivial math, so it's fine.

- **FPS and V-Sync:** If you need more performance and tearing isn't an issue (for an offline render for example), you could run uncapped FPS or at higher refresh if monitors allow. But typically, stay vsynced at 60 or 90 (VR) for fluid visuals. If the GPU is hitting limits, consider lowering resolution of the canvas (for example, use `renderer.setPixelRatio(0.5)` to render at half res, then scale up – particles might still look okay with a slight blur and you double performance). Or use **foveated rendering** in VR if needed (a future WebGPU possibility).

Level of Detail (LOD) Strategies

We mentioned some LOD in the rendering section. Additional strategies: - **Hierarchical Systems:** For extremely large scale (imagine simulating a galaxy with dust at planet scale and sparkles up close), you might create hierarchical particle systems. Far-away clusters could be represented as single particles in a higher-level system. This is more manually implemented but can be done – e.g., simulate small subsystems separately and treat each subsystem as one entity in the main scene. - **Conditional Compute:** If certain groups of particles become inactive or out of view, skip updating them. One could maintain an "active count" and only dispatch that many. If particles have finite lifespans and die, you could reuse them or decrement a counter. Our engine could allow a max pool and an active count that changes. In GPU, implementing an efficient "kill" of particles (removing them) is tricky because parallel removal is hard. But you can mark them inactive with a flag and have the shader ignore them (which still does some work per particle, though). A better is using an **indirect draw** approach described next.

GPU-driven Rendering (Indirect Draw and Culling)

WebGPU supports indirect draw calls, where the GPU can decide how many instances to render. In our context, imagine we have a buffer where one value is `activeParticleCount`, and we use `drawIndirect` so that only that many instances are drawn. We can update that count on GPU (for instance, each frame a compute shader outputs the current count of alive particles after spawning or dying). This avoids overdrawing a bunch of dead/inactive ones.

Three.js's WebGPURenderer may not yet directly expose indirect draws in the high-level API, but it's likely possible via lower-level or coming soon. Our engine can be ready to take advantage: for example, for a particle emitter where particles continuously die and spawn, we can keep a head index. But unless dealing with huge numbers, keeping them all and just not rendering some might be okay. If one is motivated: one could maintain a list of alive particle indices and render using that via a custom shader that moves dead ones off-screen or something.

- **Frustum Culling on GPU:** Usually Three.js does CPU frustum culling per object, not per particle. If you have a truly immense area and particles are concentrated in pockets, you could do compute side culling: flag particles outside the camera frustum and either eliminate them or don't draw. But since particles are points, the cost to draw off-screen ones isn't that bad (the rasterizer will discard them anyway after transform).

If one had extremely many particles, culling might help, but at that scale probably other strategies are also needed (like splitting particles into spatial bins that are objects which can be culled as whole – but then you lose global compute ease).

WebGPU Limits and Considerations

WebGPU has some known limits: - **Workgroup and Dispatch Limits:** e.g., maximum threads in a workgroup (typically 256 or 512 depending on GPU), maximum total workgroups (often large, like 65535 in each dimension). If you have more particles than threads, you just dispatch more groups, so not an issue unless you tried to use a single dispatch with >some millions threads which is not allowed. Better to chunk it. - **Buffer Binding Limits:** There's a max number of storage buffers you can bind at once (maybe around 8 or 12 depending on tier). Our engine packs data to not exceed these. We might use 1 storage buffer with struct containing all particle data instead of separate buffers for pos, vel, etc., to use fewer bindings. Or use textures for some. - **Memory:** Keep an eye on total GPU memory usage. 1 million particles with a 32-byte struct each is 32 MB, which is fine. But if you start adding multiple fields (pos, vel, prevPos, color, etc.), it adds up. Also rendering a million points can be heavy on the rasterizer – but WebGPU should handle it better than WebGL. Nonetheless, test on target hardware. If going for millions, ensure to have options to downscale gracefully.

- **Compute/Render Sync:** WebGPU can run compute and render passes asynchronously if they don't depend, but in our case they do (we need compute results for render). So we effectively serialize them each frame: compute then render. That's expected. One could overlap compute of next frame with render of current if you double-buffered data and ran one frame behind, but complexity likely isn't worth it. The engine just does sequential which is simpler and usually fine.

Buffer Aliasing and Ping-Pong

Buffer aliasing refers to reusing the same memory for different purposes at different times to save space. In a GPU context, if you know two buffers aren't needed simultaneously, you could use one allocation for both. Our engine could do some of this automatically for postprocessing targets (like reuse one texture for both ping-pong passes if possible). However, WebGPU and Three.js examples often handle double buffers explicitly. For example, for ping-pong (like Physarum or any simulation in texture), we use two textures and alternate each frame (compute writes to B while rendering uses A, then swap). This is straightforward and the engine provides utility for it.

We could optimize by using a single texture and computing in place with barriers, but WebGPU forbids reading and writing the same resource in one pass (hence the error that user encountered [45]). So ping-pong is the way, which we do.

If memory is tight, one could even reduce resolution of certain buffers (for instance, do physics on a lower resolution grid or fewer particles when not needed).

Multi-GPU Scalability: While web content typically runs on one GPU (the discrete or integrated one in system), the concept of multi-GPU load is not directly applicable in browsers yet (no explicit multi-adapter in WebGPU at the moment). However, if you were in a native context or future scenario where multiple GPU work could be split (e.g. one GPU does physics, another does rendering), it's complex to coordinate and outside our scope. Instead, if extremely high particle counts are needed for offline rendering, one might manually split simulation into two separate instances and then merge outputs – but that's manual.

From an extensibility perspective, important is that the engine's core classes are not monolithic black boxes – they can be subclassed or replaced: - Want a new solver? You can add a module and register it with PhysicsSystem to run its compute. - New node type? You can extend Three.js's Node classes or use `ShaderNode` for custom code and integrate it. - The engine tries to follow Three.js patterns so devs familiar with Three can adapt quickly (for example, using `onBeforeRender`, `onAfterRender` hooks on objects if needed, or using the existing NodeMaterial interfaces).

Plugin System: We might even allow plugins – e.g., a developer could write a plugin that adds a new type of field and just drop it in without editing core. This could be as simple as pushing a new compute node into a list.

Documentation and Debugging: For optimization, debugging tools are vital. Using the browser's devtools with WebGPU (still evolving) or enabling Three.js's debug modes can help. Also, visualizing data can be done by mapping buffers to colors drawn on screen (for example, output velocity as color to see flow patterns).

We could include a debug toggle that, when on, renders an alternate view like the depth buffer or the cluster grid or the particle IDs. This can be done by swapping the material or output in a PassNode for debugging. It's mentioned here so developers know they can inspect intermediate results if needed.

Summarizing Extensibility

The architecture is built to grow: - New **Particle Types**: Could add support for different particle primitives (e.g., particle ribbons or trails). Already, an example might be adding a `Trails.js` system where each particle leaves a line – implemented either by a second set of particles or by a geometry that is updated in compute (treat a trail as many connected segments). - **Rendering Effects**: If future Three.js adds new Node-based effects (like volumetric light shafts, lens flares), those can be slotted into PostProcessing. - **Integration**: If using frameworks like **Three.js + React + Redux** etc., the engine can simply be an orchestrated set of stateful objects. Already with R3F, it fits in as described. We keep side effects controlled (i.e., our compute nodes don't accidentally conflict – which could happen if two nodes write same buffer; we prevent that by design).

- **MaterialX and GLSL Export**: NodeMaterials align with standard material graphs; Three.js has an emerging MaterialX loader (though at one point NodeMaterial examples were removed to rework them [46]). Our engine could potentially export or import material graphs (for example, load a MaterialX and apply it to particles). This is advanced and dependent on Three's support.

- **Interoperability**: The engine's output is just Three.js objects in a scene, so any other Three.js features (like environment mapping, background, fog) should still work normally (except where not yet in WebGPURenderer – e.g. fog might not be implemented in WebGPU at time of writing, but NodeMaterial can emulate fog by blending color based on distance easily).

Finally, we encourage profiling and incremental complexity: start with simpler simulations and gradually layer on physics and effects, verifying performance at each step. The documentation provides guidance, but actual tuning might depend on the target device (a high-end GPU can handle a lot, mobile less so – WebGPU on mobile is coming, with possibly less capacity).

In essence, this engine is a **platform for creativity** that should scale from quick experiments to production scenes. By following these optimization tips and understanding the underlying systems, developers can push it to its limits and even contribute new ideas back into the ecosystem.

Examples and Usage Scenarios

Let's put everything together with some **runnable examples** that illustrate key components of the engine. Each example is a self-contained scenario demonstrating how to set up and use various modules:

1. Basic Particle Simulation (Hello Particles)

Objective: Create a simple particle fountain using `ParticleEngine` with gravity and an emitter.

Code:

```
import * as THREE from 'three/webgpu';
import { ParticleEngine } from './core/ParticleEngine.js';
import { ParticleMaterial } from './materials/ParticleMaterial.js';
import { Emitter } from './core/Emitter.js';

// Initialize Three.js scene, camera, WebGPURenderer...
const camera = new THREE.PerspectiveCamera(75, window.innerWidth/
window.innerHeight, 0.1, 100);
camera.position.set(0, 5, 20);
const scene = new THREE.Scene();
const renderer = new THREE.WebGPURenderer({ antialias: true });
document.body.appendChild(renderer.domElement);
await renderer.init();  // ensure WebGPU is ready

// Create particle engine with N particles
const engine = new ParticleEngine({ count: 10000 });
scene.add(engine.particles);  // engine.particles is a Points object with our
ParticleMaterial
```

```
// Set up an emitter that spawns particles upward from origin
const emitter = new Emitter({
    rate: 100,                // particles per second
    initialize(particle) {
        // particle is an object with properties we can set for initial state
        particle.position.set(0, 0, 0);
        // give a random upward velocity
        particle.velocity.set(
            (Math.random()-0.5)*1,
            Math.random()*5 + 5,      // upward bias
            (Math.random()-0.5)*1
        );
        particle.age = 0;
        particle.lifetime = 5.0;      // live for 5 seconds
    }
});
engine.addEmitter(emitter);

// Add a simple gravity field (pointing down on Y)
engine.fieldSystem.addField({
    apply(particle, delta) {
        particle.velocity.y -= 9.81 * delta;
    }
});

// Animation loop
renderer.setAnimationLoop((time, frame) => {
    const delta = frame ? frame.deltaTime : 0.016; // (in R3F, use provided
delta)
    engine.update(delta);
    renderer.render(scene, camera);
});
```

Explanation: We set up a `ParticleEngine` with 10k particles, attach it to the scene (it internally creates a Points mesh). We create an `Emitter` that spawns particles at the origin with random upward velocity (like a fountain spray). The engine's `fieldSystem` is used directly here with a custom gravity field that affects velocity each frame. On each animation frame, we call `engine.update(delta)` which advances the simulation (running compute shaders under the hood) and then render the scene. The result is a fountain of particles that rise up and fall back down due to gravity, with new ones emitted continuously at the specified rate. This basic example combines **emitter, field (gravity), and engine update**.

2. Shape Morphing Between Models

Objective: Demonstrate morphing a particle cloud from one 3D model shape to another using **target position buffers**.

Setup: Assume we have two meshes loaded (e.g. a fox model and a 3D text), and we sample a set of points on their surfaces.

```
import { ParticleEngine } from './core/ParticleEngine.js';
import { MeshSurfaceSampler } from 'three/examples/jsm/math/
MeshSurfaceSampler.js';  // helper to get random points on mesh surfaces

// Suppose foxMesh and textMesh are loaded Three.js Mesh objects
const foxSampler = new MeshSurfaceSampler(foxMesh).build();
const textSampler = new MeshSurfaceSampler(textMesh).build();
const numParticles = 50000;
const shapeA = [], shapeB = [];
for(let i=0; i<numParticles; i++){
    const p = new THREE.Vector3();
    foxSampler.sample(p);
    shapeA.push(p.x, p.y, p.z);
    textSampler.sample(p);
    shapeB.push(p.x, p.y, p.z);
}

// Create engine and set initial positions to shapeA
const engine = new ParticleEngine({ count: numParticles });
engine.initPositions(shapeA);  // hypothetical method to set particle positions
engine.particleMaterial.colorNode = THREE.color('orange');  // start color
scene.add(engine.particles);

// Assign target positions for shapeB in a buffer on GPU
engine.setMorphTargets(shapeA, shapeB);

// Use a custom field to drive particles towards the interpolated target
engine.fieldSystem.addField({
    apply(particle, delta) {
        // particle has current pos and an assigned target pos (interpolated by
morphProgress)
        const target = particle.currentTarget;
        const toTarget = target.clone().sub(particle.position);
        // apply acceleration towards target
        particle.velocity.add( toTarget.multiplyScalar(2 * delta) );
        // simple damping
        particle.velocity.multiplyScalar(0.95);
    }
});

// Animate the morphProgress from 0 to 1 over 3 seconds using GSAP
gsap.to(engine, { morphProgress: 1, duration: 3, ease: "power2.inOut" });

// Also animate color from orange to blue for effect
```

```
gsap.to(engine.particleMaterial.uniforms.color.value, { r:0, g:0.5, b:1,
duration: 3 });

// In engine.update(), ensure to interpolate currentTarget = (1-
morphProgress)*shapeA + morphProgress*shapeB for each particle.
```

Explanation: We sample points on two meshes to create shapeA and shapeB arrays. We initialize particles at shapeA positions. We then provide these shapes to the engine as morph targets. The engine has a property `morphProgress` (0 to 1) and internally sets each particle's `currentTarget` accordingly each frame. We add a field that pulls particles toward their currentTarget. By using GSAP, we tween `engine.morphProgress` from 0 to 1 over 3 seconds, causing the particles to gradually move to shapeB's configuration. Simultaneously, we tween the particle color from orange to bluish to enhance the effect. The end result: an orange fox made of particles smoothly transforms into blue text. This example shows coordination of shape data, a custom field (target attraction), and GSAP animation.

3. Boids Flocking with Audio Reactive Behavior

Objective: Create a flock of boids (particles following flocking rules) that also react to music volume (speeding up when music is loud).

Setup: We'll use the boids solver and the Web Audio API.

```
const engine = new ParticleEngine({ count: 10000, particleMaterial: new
ParticleMaterial({ color: 0xffffff }) });
scene.add(engine.particles);

// Configure boids behavior
engine.physicsSystem.enableBoids({
    neighborhoodRadius: 5,
    separationWeight: 1.5,
    alignmentWeight: 1.0,
    cohesionWeight: 1.0,
    maxSpeed: 5,
    maxForce: 0.05
});
// This boids solver will update velocities each frame considering neighbors [29] .

// Add a mild random noise field to avoid too rigid movement
engine.fieldSystem.addField(new NoiseField( { scale: 0.1, strength: 0.5 } ));

// Audio setup (get microphone or audio element FFT)
const audioCtx = new (window.AudioContext)();
const analyser = audioCtx.createAnalyser();
analyser.fftSize = 256;
navigator.mediaDevices.getUserMedia({audio:true}).then(stream => {
    const source = audioCtx.createMediaStreamSource(stream);
    source.connect(analyser);
});

// Each frame, get volume and adjust boid speed
function getVolumeLevel() {
    const data = new Uint8Array(analyser.frequencyBinCount);
    analyser.getByteFrequencyData(data);
    let avg = 0;
    for(let i=0;i<data.length;i++){ avg += data[i]; }
    avg /= data.length;
    return avg/255;  // normalized 0-1
}

renderer.setAnimationLoop(() => {
    const vol = getVolumeLevel();
    // Map volume to a speed factor between 1 and 3
    const speedFactor = 1 + 2 * vol;
    engine.physicsSystem.setBoidsMaxSpeed(5 * speedFactor);
    engine.physicsSystem.setBoidsMaxForce(0.05 * speedFactor);
    // Optionally adjust particle color or size with volume
    engine.particleMaterial.uniforms.size.value = 1 + vol * 2;
    engine.update(timeStep);
    renderer.render(scene, camera);
});
```

Explanation: We initialize 10k boids with a ParticleMaterial. We then enable a Boids solver on the physicsSystem with given parameters (radius, weights) – presumably the engine will run a compute shader each frame computing the alignment, cohesion, separation forces [29]. We add a slight NoiseField to keep the motion natural. For audio, we use the microphone (or it could be music playback) and an AnalyserNode to get frequency data. We compute an average volume (`avg` of frequency bins) each frame. We then map that volume to a speedFactor. Using engine's API, we adjust the boids solver's `maxSpeed` and `maxForce` in real-time based on volume – so when loud, boids can move faster and turn sharper, essentially making them more frenetic. We also tie volume to particle size (or could be color, here size is changed via a uniform if ParticleMaterial supports it). Then we call `engine.update` each loop. As a result, when music is quiet, the flock glides calmly; when it gets loud or a beat drops, the flock speeds up and possibly particles grow, giving a pulsating flock that visually represents the audio. This example combines **physics (boids)**, **audio input**, and **dynamic parameter adjustment**.

4. Interactive Fluid with Cursor Interaction (SPH Fluid)

Objective: Simulate a simple 2D fluid and allow the user to stir it with the mouse.

Setup: We'll treat particles as fluid (SPH) and use pointer position to add velocity.

```
const engine = new ParticleEngine({ count: 20000 });
scene.add(engine.particles);
```

```
// Enable SPH solver in 2D (e.g., confine to XZ plane for a top-down fluid)
engine.physicsSystem.enableSPH({
    kernelRadius: 1.0,
    restDensity: 30.0,
    stiffness: 0.5,
    viscosity: 0.1
});
// We assume this sets up compute to calculate density & pressure forces.

// Constrain motion mostly to a plane (if needed, we could just zero out Y
velocity each frame via a field)
engine.fieldSystem.addField({
    apply(p, dt) { p.velocity.y = 0; }
});

// Interaction: on pointer move, add a radial outward velocity from pointer
renderer.domElement.addEventListener('pointermove', (e) => {
    // Normalize pointer to world coordinates in XZ plane:
    const x = (e.clientX / window.innerWidth) * 20 - 10;
    const z = (e.clientY / window.innerHeight) * 20 - 10;
    engine.fieldSystem.addField({
        // a one-tick field:
        apply(p) {
            const dx = p.position.x - x;
            const dz = p.position.z - z;
            const dist2 = dx*dx + dz*dz;
            if(dist2 < 4.0) {  // if within radius ~2
                const force = 5.0;
                p.velocity.x += force * (dx);
                p.velocity.z += force * (dz);
            }
        }
    }, true); // 'true' indicates remove after one application
});
// (This simplistic approach injects a field on each pointer event that pushes
particles away from the cursor position.)

renderer.setAnimationLoop(() => {
    engine.update(1/60);
    renderer.render(scene, camera);
});
```

Explanation: We turn on an SPH solver with certain parameters (radius, restDensity, etc.). The engine now each frame calculates density and pressure for each particle and accelerates them accordingly, simulating incompressible fluid behavior (in a basic way). We confine movement to XZ plane by zeroing Y velocity (effectively making it a 2D simulation, like looking down at water). For interaction, we add an event listener for pointer moves: it computes a world coordinate (here just mapping screen to a fixed plane region). On each event, we add a temporary field to the fieldSystem that, for the next `engine.update`, will push particles outward from that point (within a radius). We make it a one-time field (assuming our `addField` can take a flag to auto-remove it after one use). This effectively creates ripples or swirls where the user moves the mouse, as if stirring the fluid. The combination demonstrates an **SPH fluid** solver and **user input** injection.

5. WebGPU Compute Shader Extensibility (Custom Compute Example)

Objective: Show how to extend the engine with a custom compute shader – for instance, a reaction-diffusion on particles' color.

Scenario: Each particle has a chemical A and B, which react (like Gray-Scott reaction-diffusion) while particles move. This is niche, but demonstrates extensibility.

```
// Assuming engine is set up with particles and some initial distribution of
chemical A and B in particle properties

// Define a custom compute node for reaction-diffusion
import { Fn } from 'three/tsl';
const feed = 0.055, kill = 0.062, diffusionA = 1.0, diffusionB = 0.5;
const reactionCompute = Fn( () => {
    // Let's say particle.chemA and chemB are accessible (previous frame values)
    // We'll need to fetch neighbor average for diffusion - for simplicity,
assume we computed that already or ignore diffusion in this demo.
    const a = chemA;
    const b = chemB;
    const reaction = a.mul(b.mul(b));       // reaction term: A * B^2
    const da = diffusionA * laplacianA - reaction + feed * (1 - a);
    const db = diffusionB * laplacianB + reaction - (kill + feed) * b;
    chemA.addAssign( da.mul(deltaTime) );
    chemB.addAssign( db.mul(deltaTime) );
} )().compute(numParticles);
renderer.computeAsync(reactionCompute);
```

(This code would be part of engine.update or a custom system; it's a bit pseudo since doing laplacian on a particle graph is non-trivial without neighbors, but assume maybe each particle's neighbors approximate that or we ignore diffusion for sake of example.)

Explanation: We used `Fn` to define an update for two chemical species A and B per particle with Gray-Scott equations. We would integrate that into the engine's update loop (likely after moving particles). This shows we can inject entirely custom compute logic into the pipeline, not originally provided by engine's presets. Reaction-diffusion would normally be done on a grid, but here we conceptualize it on moving points, which could yield interesting visuals (particles changing color over time forming patterns).

While this example is theoretical (needs neighbor coupling), it emphasizes that if a developer wants to implement something new, they can write their own `Fn` compute node, use any particle properties or global uniforms, and execute it via `renderer.computeAsync`. The engine architecture doesn't lock you in – you can always extend with lower-level code when needed.

Each example above is **runnable** with minimal setup (assuming proper imports and that the engine modules exist with the described API). They illustrate different facets: basic usage, advanced morphing, flocking with audio, fluid with interaction, and extensibility with custom compute.

By studying these examples and the preceding detailed documentation, developers should feel equipped to utilize and adapt the WebGPU Node-Based Particle Engine for their own creative projects. The combination of Three.js's cutting-edge WebGPU support and a well-architected modular engine opens up new possibilities for real-time VFX and scientific simulations in the browser, with performance and visual fidelity previously hard to achieve on the web. Enjoy creating and pushing the boundaries of particle simulations!

Sources: The design and techniques discussed are informed by the Three.js documentation and community examples on WebGPU and NodeMaterials [1] [2], Three.js's official wiki on the shading language [47] [48], and practical demonstrations of GPGPU particle systems [49] [12] and clustered lighting approaches [33], among others, ensuring the engine employs state-of-the-art methods.

---

[1] [4] [5] [6] WebGPU / TSL - Wawa Sensei
https://wawasensei.dev/courses/react-three-fiber/lessons/webgpu-tsl

[2] [3] [8] [46] What is going on with NodeMaterials? All the examples are gone - Questions - three.js forum
https://discourse.threejs.org/t/what-is-going-on-with-nodematerials-all-the-examples-are-gone/66451

[7] [10] [11] [12] [13] [14] [15] [16] [17] [18] [19] [20] [21] [30] [49] GPGPU particles with TSL & WebGPU - Wawa Sensei
https://wawasensei.dev/courses/react-three-fiber/lessons/tsl-gpgpu

[9] PassNode: Implement Auto-MRT. · Issue #28749 · mrdoob/three.js · GitHub
https://github.com/mrdoob/three.js/issues/28749

[22] [23] [24] [39] [47] [48] Three.js Shading Language · mrdoob/three.js Wiki · GitHub
https://github.com/mrdoob/three.js/wiki/Three.js-Shading-Language

[25] [36] Shade - WebGPU graphics - Showcase - three.js forum
https://discourse.threejs.org/t/shade-webgpu-graphics/66969

[26] [27] PassNode Issues - Questions - three.js forum
https://discourse.threejs.org/t/passnode-issues/68499

[28] [PDF] Detailed Rigid Body Simulation with Extended Position Based ...
https://matthias-research.github.io/pages/publications/PBDBodies.pdf

[29] [WebGPU] Millions of Flocking Boids with BabylonJS : r/GraphicsProgramming
https://www.reddit.com/r/GraphicsProgramming/comments/13cemv5/webgpu_millions_of_flocking_boids_with_babylonjs/

[31] [32] [33] [34] [35] [38] Clustered Rendering on WebGPU - Showcase - three.js forum
https://discourse.threejs.org/t/clustered-rendering-on-webgpu/81042

[37] toji/webgpu-clustered-shading: Personal experimental ... - GitHub
https://github.com/toji/webgpu-clustered-shading

[40] Floids - Multi-Agent Boids+Fireflies Simulation - Showcase - three.js forum
https://discourse.threejs.org/t/floids-multi-agent-boids-fireflies-simulation/41690

[41] [42] [43] [44] Coding a 3D Audio Visualizer with Three.js, GSAP & Web Audio API | Codrops
https://tympanus.net/codrops/2025/06/18/coding-a-3d-audio-visualizer-with-three-js-gsap-web-audio-api/

[45] Compute Shader TSL - Questions - three.js forum
https://discourse.threejs.org/t/compute-shader-tsl/80277