# PRIMORDIA

## A 3D Emergence Laboratory

Two-scale particle life with Barnes–Hut gravity
and a piloted articulated creature

**Technical Report and User Manual**

Interactive Graphics course

| | |
|---|---|
| **Application type** | Real-time interactive 3D web application |
| **Environment** | WebGL via three.js (r128), plain ES6 JavaScript |
| **External assets** | None — all geometry, textures and animation are generated procedurally |
| **Entry point** | `index.html` |

A real-time simulation in which two force laws acting at different spatial scales — short-range, type-dependent "chemistry" and long-range, mass-dependent gravity — produce emergent structure that the user can inspect, perturb, and swim through with a hand-animated creature.

# Contents

# 1 Introduction

## 1.1 What the application is

PRIMORDIA is a real-time, interactive 3D laboratory for *emergence*: the phenomenon in which a large number of agents obeying very simple local rules produce complex global structure that was never explicitly programmed. The user does not build the structures — the user changes the rules, and then watches, steers and disturbs whatever the rules produce.

The simulation superimposes two force laws that act at deliberately different spatial scales, which is the conceptual core of the project:

- **Short-range "chemistry".** Every particle carries a discrete *type* (a colour). An *asymmetric* matrix $A$ specifies how strongly type $i$ is attracted to or repelled by type $j$ within a small radius $r_{\max}$. Because $A_{ij} \neq A_{ji}$ in general, the system is non-conservative and produces the chasing, cycling and self-propelling behaviour characteristic of *particle life*. This term is responsible for local structure: cells, membranes, filaments, worms.
- **Long-range gravity.** Every particle also carries a mass derived from its radius, and attracts every other particle with an inverse-square law. This term is responsible for global structure: collapse, clumping, orbits, spiral arms.

Neither force alone is novel. Their superposition is what makes the application interesting to explore: gravity assembles the coarse skeleton, chemistry decorates it, and the two fight each other in ways that are genuinely difficult to predict from the parameter values alone.

Naive evaluation of both laws would cost $O(N^2)$ per frame, which is far too slow for the 1,000–3,000 particles we target at 60 Hz. Each law is therefore accelerated by the data structure that suits its own spatial scale — a uniform grid for the short-range term, a Barnes–Hut octree for the long-range one. Section 5 describes both in detail.

Finally, the soup is inhabited. A hierarchical, self-animated creature (the *Grazer*) is built as a real three.js scene graph with four articulated legs running a procedural gait. The user pilots it with the keyboard; it is a solid obstacle that particles bounce off, and it emits its own local force field, so it is a physical participant in the simulation rather than a decorative overlay.

## 1.2 Design goals

1. **Everything procedural.** No mesh files, no image files, no imported animation clips. Every vertex, every texel and every joint angle in the application is computed in JavaScript at run time. This was a self-imposed constraint, and it shapes many of the technical decisions in this report.
2. **Real-time at meaningful $N$.** The simulation must stay interactive with thousands of mutually interacting particles, which rules out the naive all-pairs approach and motivates the two acceleration structures.
3. **Everything is live.** No parameter change requires a reload. Particle count, type count, world size, and the rule matrix can all be edited while the simulation runs, and the affected buffers and data structures are rebuilt in place.
4. **The rules are visible.** The user can display the octree subdivision, the gravitational field as animated streamlines, and a wireframe lattice of space that deforms under mass and under the user's own interventions. The acceleration structure is not hidden — it is part of the exhibit.
5. **Zero-friction deployment.** Opening `index.html` from disk must be sufficient. No build step, no bundler, no local server, no `npm install`.

### 1.3 How to read this document

Section 2 describes the execution environment; Section 3 lists everything used but not written by us; Section 4 explains the code structure; Section 5 is the technical core (physics, acceleration structures, rendering, the creature); Section 6 documents every implemented interaction; Section 7 is the user manual proper, and can be read on its own by someone who only wants to run the application.

## 2 Description of the environment used

### 2.1 Rendering environment

The application is a **client-side web application** rendering through **WebGL**, accessed via the **three.js** library (revision 128) rather than through raw WebGL calls. Concretely:

- The rendering context is created by `THREE.WebGLRenderer`, which selects a WebGL 2 context when the browser provides one and silently falls back to WebGL 1 otherwise. The application uses no feature that requires WebGL 2, so it runs correctly on both.
- The renderer is configured with `antialias: true` (MSAA on the default framebuffer) and `powerPreference: 'high-performance'`, which asks the browser to select the discrete GPU on dual-GPU laptops.
- The device pixel ratio is clamped, `setPixelRatio(Math.min(devicePixelRatio,2))`. On a high-DPI display an unclamped ratio would quadruple the fragment load for a barely perceptible gain; the clamp is a deliberate, and very effective, performance decision.
- Output is written in sRGB (`outputEncoding = THREE.sRGBEncoding`) so that lighting is composited in linear space and converted once at the end, which is what makes the emissive teals and pinks read correctly rather than washing out.

We chose three.js over hand-written WebGL deliberately. The project's intellectual content is the *simulation* — the force laws, the grid, the octree, the articulated model and the gait — and all of that is written from scratch. Delegating context creation, shader compilation and matrix bookkeeping to a mature library let us spend the available time on the parts of the project that are actually ours. Nothing in the physics or in the model hierarchy is inherited from the library: three.js is used as a rendering back end and a scene-graph container, not as a simulation framework.

### 2.2 Language and execution model

The application is written in **plain ES6+ JavaScript**. There is no TypeScript, no JSX, no transpilation and no bundling. The consequences are worth stating explicitly because they influenced the code structure:

- **No build step.** The source that ships is the source that runs, which makes the project trivial to inspect and to grade.
- **Classic scripts, not ES modules.** The JavaScript files are loaded as ordinary `<script src=...>` tags in dependency order and share a single global lexical scope. This is a conscious trade-off: ES modules would give explicit import graphs, but browsers refuse to load modules over the `file://` protocol for security reasons, which would force every user to start a local web server. Classic scripts keep the "double-click `index.html`" promise. The dependency order is documented in `index.html` and in each file's header comment (Section 4).

- **Single-threaded.** The whole simulation runs on the main thread, inside the animation-frame callback. There are no Web Workers. This bounds the achievable particle count, and Section 8 discusses it as the first candidate for future work.

### 2.3 Runtime requirements and platform support

| Requirement | Detail |
|---|---|
| Browser | Any modern browser with WebGL: Chrome, Firefox, Edge, Safari |
| GPU | Any GPU capable of WebGL 1; integrated graphics are sufficient |
| Network | Needed *once*, to fetch three.js from the CDN (see §3) |
| Server | Not required — the `file://` protocol works |
| Installation | None |
| Input | Mouse + keyboard (full feature set); touch (camera only) |

Table 1: Runtime requirements.

The application is responsive: below a viewport width of 560 px the control console expands to full width and the HUD is hidden, and the camera accepts one-finger drag (orbit) and two-finger pinch (zoom). Creature piloting and the drag tools require a keyboard and mouse respectively, so a desktop browser is the intended target.

## 3 External libraries, tools and models

This section lists exhaustively everything in the delivered project that was **not** written by the team.

### 3.1 Libraries

| Library | Version | Source | What it is used for |
|---|---|---|---|
| three.js | r128 | `cdnjs.cloudflare.com` (single `<script>` tag) | The *only* runtime dependency. Used for: WebGL context and render loop (`WebGLRenderer`); scene graph (`Scene`, `Group`, `Object3D`) and its transform composition; `PerspectiveCamera` and projection; primitive geometry generators (`SphereGeometry`, `CylinderGeometry`, `BoxGeometry`, `TorusGeometry`, `EdgesGeometry`); the standard PBR material (`MeshStandardMaterial`) and its shaders; `InstancedMesh` for batched particle drawing; lights; `CanvasTexture`; `BufferGeometry` for the line visualisations; and the math types (`Vector2/3`, `Color`, `Plane`, `Raycaster`). |

Table 2: Third-party libraries. There is exactly one.

That is the complete list. No UI framework (React, Vue), no control-panel library (dat.GUI, lil-gui, Tweakpane), no physics engine (cannon.js, ammo.js), no math library (gl-matrix), no utility library (lodash), no bundler (webpack, Vite, Rollup), no package manager, no CSS framework. The control console, the attraction-matrix editor, the HUD and the camera controller were all

written for this project; in particular we did *not* use three.js's own `OrbitControls`, because it ships as a separate add-on file and our camera has to interoperate with the poke tools, which share the same mouse buttons.

## 3.2 Models, meshes and animation

**None.**

There are no imported 3D models, no `.obj`/`.gltf`/`.fbx` files, no rigs, and no imported animation clips or keyframe data anywhere in the project.

Every piece of geometry in the scene is assembled at run time from three.js's parametric primitive generators:

- **Particles** — one `SphereGeometry(1, 10, 8)`, drawn $N$ times by an `InstancedMesh`.
- **The Grazer** — roughly two dozen spheres, cylinders and a torus, composed into a hierarchy of `Groups` (Section 5.9).
- **The boundary cage** — `EdgesGeometry` of a `BoxGeometry`.
- **Octree, field and lattice visualisations** — `BufferGeometry` objects whose vertex buffers we fill by hand every frame.

The Grazer's walk cycle is likewise not an imported clip: it is a closed-form function of a phase accumulator, evaluated per leg per frame (Section 5.10).

## 3.3 Textures

**None imported.**

All three texture maps are **painted procedurally into an off-screen** `<canvas>` at start-up and uploaded via `THREE.CanvasTexture`.

Three 128 × 128 maps are generated (§5.2): an *albedo* map (radial gradient plus 900 random speckles), a *normal* map (70 overlapping radial gradients perturbing the tangent-space normal), and a *roughness* map (1,400 random grey squares). Generating them costs a few milliseconds once and keeps the project asset-free.

## 3.4 Fonts

The CSS font stacks name Inter, SF Mono, JetBrains Mono and Roboto Mono, but no font is *fetched*: there is no `@font-face` rule and no Google Fonts link. Each stack ends in a generic family (`system-ui`, `sans-serif`, `monospace`), so the named fonts are used only if the user's machine already has them and the OS default is used otherwise. The application therefore carries no font dependency.

## 3.5 Conceptual credits

The short-range rule model is our implementation of the well-known *particle life* / "Clusters" idea popularised by Jeffrey Ventrella and by Hunar Ahmad (`hunar4321`). We credit the idea in the application footer. No code was taken: the original demonstrations are 2D, use all-pairs evaluation, and have neither variable mass nor gravity. Our contribution is the extension to 3D, the addition of a mass-dependent long-range term with a Barnes–Hut octree, the uniform-grid acceleration of the short-range term, the live visualisations, and the piloted articulated creature.

The Barnes–Hut algorithm itself is the standard published method [1]; the implementation in `js/octree.js` is ours.

### 3.6 Development tools

| Tool | Role |
|---|---|
| Text editor | Writing the source |
| Browser developer tools | Debugging, profiling, frame timing |
| LaTeX | This document |

Table 3: Development tools. No compiler, bundler, package manager or asset pipeline is part of the workflow.

## 4 Software architecture

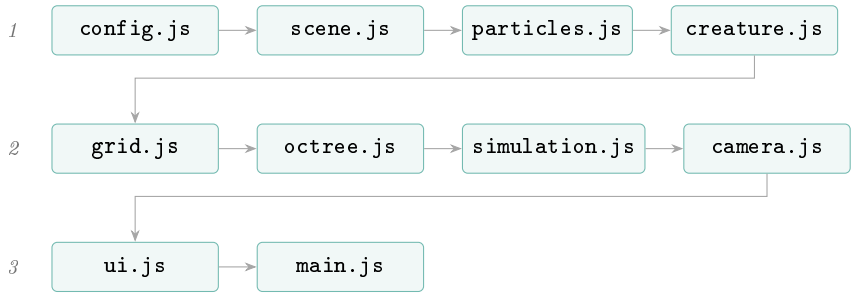
### 4.1 File organisation

The project is deliberately flat and readable: one HTML entry point, one stylesheet, ten JavaScript files, each with a single responsibility.

| File | Lines | Responsibility |
|---|---|---|
| `index.html` | 45 | Markup for the console, HUD and pilot hint; declares script load order. |
| `css/style.css` | 128 | All presentation: console panel, HUD, sliders, toggles, matrix grid, responsive rules. |
| `js/config.js` | 54 | `CFG` (every tunable), `Poke` (intervention state), `PALETTE`. Single source of truth. |
| `js/scene.js` | 88 | Renderer, camera object, lights, boundary cage, poke marker, procedural textures. |
| `js/particles.js` | 56 | Particle buffers, seeding, attraction matrix, `InstancedMesh`. |
| `js/creature.js` | 163 | Grazer model hierarchy, procedural gait, keyboard piloting. |
| `js/grid.js` | 38 | Uniform grid (counting sort) and the particle-life force kernel. |
| `js/octree.js` | 124 | Barnes–Hut octree: build, mass/COM pass, traversal. |
| `js/simulation.js` | 282 | The integrator and the three debug visualisations; mesh sync. |
| `js/camera.js` | 71 | Orbit/pan/zoom, screen-to-world unprojection, mouse and touch bindings. |
| `js/ui.js` | 293 | Widget factory, control console, matrix editor, impulses, presets. |
| `js/main.js` | 57 | Frame loop, HUD, resize handling, bootstrap. |

### 4.2 Dependency order

Because the files are classic scripts sharing one global scope, **load order is part of the design**. Each file may use anything declared in the files above it. The order encoded in `index.html` is exactly the dependency graph:




`config.js` comes first because every other file reads `CFG`; `main.js` comes last because it bootstraps. Only `main.js` executes significant work at load time — the others declare state and functions,

with a handful of self-contained initialisation calls (building the cage, generating the textures, attaching input listeners).

The bootstrap is five lines and shows the whole start-up sequence:

```
allocParticles(N); allocOctree(); rebuildGridDims();
randomizeMatrix(); seedParticles(); buildMesh(); buildCreature(); buildLattice(); buildPanel();
applyCam();
requestAnimationFrame(frame);
```

### 4.3 Data layout: structure of arrays

Particles are **not** objects. There is no `Particle` class and no array of instances. State is held in parallel typed arrays — a *structure of arrays* (SoA) layout:

```
function allocParticles(n){
  px=new Float32Array(n); py=new Float32Array(n); pz=new Float32Array(n);
  vx=new Float32Array(n); vy=new Float32Array(n); vz=new Float32Array(n);
  ptype=new Uint8Array(n); psize=new Float32Array(n); pmass=new Float32Array(n);
}
```

This is the single most important performance decision in the project, for three reasons:

1. **Cache locality.** The inner loop of `simulate()` streams sequentially through `px`, `py`, `pz`. With an array of objects each access would chase a pointer into a scattered heap allocation; here the next value needed is almost always in the cache line already fetched.
2. **No garbage collector pressure.** Typed arrays are allocated once, at start-up or on a population change. In steady state the simulation allocates *nothing* per frame, so the GC never runs, so there are no collection pauses — which in an animation is the difference between smooth and stuttering.
3. **Predictable types.** `Float32Array` elements cannot be boxed or change representation, which lets the JIT emit unboxed floating-point code for the whole inner loop.

The same discipline is applied to the octree (§5.6), which is a pooled SoA structure of thirteen typed arrays rather than a tree of node objects, and to the gravity field samples.

## 5 Technical aspects

### 5.1 Scene, camera and lighting

The scene is a dark volume with exponential fog (`FogExp2`, density 0.006) matching the background colour `0x05070d`, so distant particles dissolve into the background instead of ending abruptly at the far plane — a cheap and effective depth cue in a scene that is otherwise a cloud of similar-looking spheres.

The camera is a `PerspectiveCamera` with a 55° vertical field of view and a near/far range of 0.5 to 2000 world units. It is never manipulated directly; it is a dependent variable of the orbit state described in §6.1.

Lighting is a deliberately conventional four-light rig plus two accent lights:

| Light | Type | Colour / intensity | Purpose |
|---|---|---|---|
| `ambient` | Ambient | `0x33405a`, 0.55 | Lifts shadowed sides out of black |
| `sun` | Directional | `0xbcd3ff`, 0.85 | Key light, cool daylight from above |
| `rim` | Directional | `0x2a4a7a`, 0.50 | Back/rim light, separates silhouettes |
| `p1` | Point | `0x37e0c8`, 1.60 | Teal accent; orbits the origin each frame |
| `p2` | Point | `0xff6bb0`, 1.20 | Pink accent, static |

The two point lights have finite range (260 and 220 units) and physical quadratic decay. `p1` is animated on a slow circle of radius 40:

```
p1.position.set(Math.sin(t*0.3)*40, 30, Math.cos(t*0.3)*40);
```

Its moving highlight sweeping across a static clump is often the clearest visual signal that a structure is three-dimensional rather than a flat sprite cloud. The accent pair can be toggled off from the console.

### 5.2 Procedural texture generation

Three maps are painted into 128×128 off-screen canvases at start-up and wrapped in `CanvasTexture`:

- **Albedo** — an off-centre radial gradient (white → light grey → blue-grey) with 900 random white speckles at $\leq 10\%$ alpha. It is kept *light* on purpose: the per-instance colour multiplies this map, so a dark albedo would mute the type colours. Tagged `sRGBEncoding`, since it is colour data rather than measured data.
- **Normal** — filled with the neutral tangent-space normal $(128, 128, 255)$, then overlaid with 70 radial gradients of random radius whose red and green channels are randomly offset by up to $\pm 60$. Each blob is a small bump. Applied at `normalScale` $= (0.6, 0.6)$ on particles and a much subtler $(0.25, 0.25)$ on the creature body.
- **Roughness** — mid-grey with 1,400 random $2 \times 2$ squares of varying value, giving micro-variation in specular response so that a clump of identically-coloured particles does not look like a single moulded object.

Total cost is a few milliseconds once at start-up, and the reward is that the project ships with no image files at all.

### 5.3 Particle representation and instanced rendering

Each particle $i$ has position $\mathbf{x}_i$, velocity $\mathbf{v}_i$, a type $t_i \in \{0, \dots, T-1\}$, a radius $s_i$ and a mass $m_i$. Radii are drawn at seed time from a distribution deliberately skewed toward small values:

```
const base=0.55, sv=CFG.sizeVariance;
const s = base*(1-sv) + base*sv*(0.4 + Math.pow(Math.random(),2.2)*3.4);
psize[i]=s;
pmass[i]=s*s*s; // volume-like mass
```

The exponent 2.2 biases `Math.random()` toward zero, so the population is mostly small particles with a few large ones — and since $m_i = s_i^3$, those few large ones carry disproportionate gravitational mass. This is what makes the gravity presets interesting: a handful of heavy seeds act as attractors around which the light majority organises, exactly as in the astrophysical case being evoked. The `sizeVariance` slider interpolates from a uniform population ($sv = 0$) to a highly varied one ($sv = 1$).

Initial positions are uniform *by volume* inside a sphere of radius $0.7\,h$, which requires the cube-root inversion of the radial CDF:

```
const L=Math.sqrt(a*a+b*b+cc*cc)||1; // random direction
const r=Math.cbrt(Math.random())*R; // cbrt => uniform density, not centre-heavy
px[i]=a/L*r; py[i]=b/L*r; pz[i]=cc/L*r;
```

Omitting the `cbrt` would concentrate particles near the centre, seeding a gravitational collapse before the user has touched anything.

**Rendering.** All $N$ particles are drawn by a single `THREE.InstancedMesh` — one sphere geometry, one material, one draw call. Two details matter:

- The instance matrix buffer is flagged `DynamicDrawUsage`, telling the driver the buffer changes every frame so it can choose the right memory residency.
- Per-instance colour is written once per type change via `setColorAt`, not per frame. Only the transform buffer is re-uploaded each frame, by `syncMesh()`.

```
function syncMesh(){
  for(let i=0;i<N;i++){
    dummy.position.set(px[i],py[i],pz[i]);
    dummy.scale.setScalar(psize[i]);
    dummy.updateMatrix();
    mesh.setMatrixAt(i, dummy.matrix);
  }
  mesh.instanceMatrix.needsUpdate=true;
}
```

A single reused `Object3D` (`dummy`) composes each matrix, so this function allocates nothing. The sphere is only $10 \times 8$ segments — 80 faces. At $N = 3000$ that is a quarter of a million triangles, which is comfortable; at $32 \times 16$ it would be a million, which is not. The normal map restores the surface detail that the coarse tessellation gives up.

## 5.4 Short-range chemistry: the force law

Let $\hat{r} = r/r_{\max}$ be the normalised distance between two particles, $\beta$ the repulsion-core fraction, and $a = A_{t_i t_j}$ the (signed) attraction coefficient. The scalar force profile is piecewise linear:

$$F(\hat{r}, a) = \begin{cases} \dfrac{\hat{r}}{\beta} - 1, & 0 \le \hat{r} < \beta \quad \text{(universal repulsion)} \\ a\left(1 - \dfrac{\left|\,2\hat{r} - 1 - \beta\,\right|}{1 - \beta}\right), & \beta \le \hat{r} < 1 \quad \text{(typed interaction)} \\ 0, & \hat{r} \ge 1 \quad \text{(no interaction)} \end{cases} \tag{1}$$

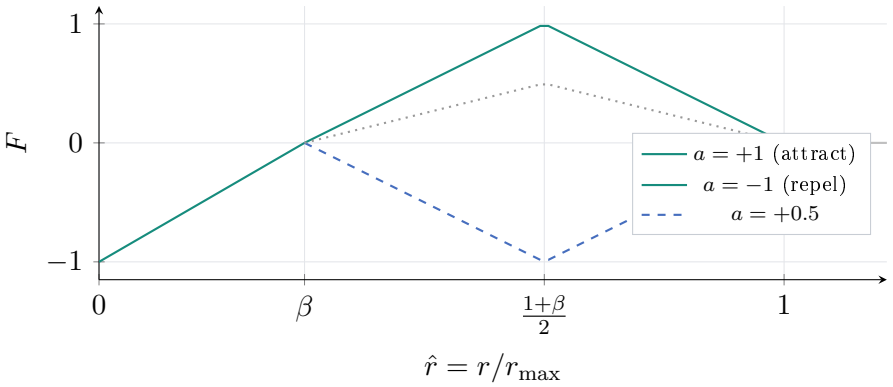



Figure 1: The particle-life force profile, drawn with $\beta = 0.3$. Below $\beta$ the force is a type-independent repulsion that reaches $-1$ at contact; between $\beta$ and 1 it is a triangular lobe scaled by the matrix entry $a$, peaking at $\hat{r} = (1 + \beta)/2$; beyond $r_{\max}$ it is exactly zero, which is what makes the uniform grid valid.

Three properties of this law carry the whole design:

1. **The repulsion core is universal.** For $\hat{r} < \beta$ the profile does not depend on $a$ at all. Every particle pushes every other particle away at very close range, regardless of type, which is what prevents the entire population from collapsing to a single point under mutual attraction. The `beta` slider is therefore effectively an incompressibility control.

2. **The interaction is compactly supported.** $F$ is exactly zero beyond $r_{\max}$ — not asymptotically small, but zero. This is the precondition for the uniform grid: a particle can only be influenced by particles within $r_{\max}$, so a grid with cell size $r_{\max}$ makes the $3 \times 3 \times 3$ neighbourhood a *complete* search domain, not an approximation.

3. **The matrix is asymmetric.** $A_{ij}$ and $A_{ji}$ are independent, so Newton's third law is deliberately violated. If red chases green while green flees red, the pair self-propels and momentum is not conserved. This is physically false and behaviourally essential: it is the source of every chasing, cycling and locomotive pattern in the system.

   The accumulated short-range acceleration on particle $i$ is

$$\mathbf{a}_i^{\text{chem}} = r_{\max}\, k \sum_{j\,:\,r_{ij}<r_{\max}} F\!\left(\frac{r_{ij}}{r_{\max}},\, A_{t_i t_j}\right) \hat{\mathbf{d}}_{ij}, \qquad \hat{\mathbf{d}}_{ij} = \frac{\mathbf{x}_j - \mathbf{x}_i}{r_{ij}} \tag{2}$$

where $k$ is the `forceStrength` slider. Note the $r_{\max}$ factor: it keeps the effective force scale roughly constant as the user drags the interaction radius, so changing $r_{\max}$ changes the *geometry* of the patterns without simply making everything more violent.

### 5.5 Uniform grid acceleration

Evaluating the sum above naively is $O(N^2)$. Since the kernel has compact support, we bin particles into a uniform grid of cell size $r_{\max}$ and search only the 27 neighbouring cells.

The grid resolution adapts to both the world size and the interaction radius, capped to bound memory:

```
gN = Math.max(1, Math.floor(world / CFG.rMax));
gN = Math.min(gN, 64); // <= 262144 cells
gCells = gN*gN*gN;
```

**Counting-sort build.** The grid is rebuilt from scratch every frame — there is no incremental update, which would be more complex and, at these particle counts, slower. The build is a three-pass counting sort into flat typed arrays, and allocates nothing:

```
function buildGrid(){
  cellCount.fill(0);
  for(let i=0;i<N;i++){ const c=cellIndex(px[i],py[i],pz[i]); cellOf[i]=c; cellCount[c]++; }
  let acc=0;
  for(let c=0;c<gCells;c++){ cellStart[c]=acc; acc+=cellCount[c]; }
  cellStart[gCells]=acc;
  const cursor=cellStart.slice(0,gCells);
  for(let i=0;i<N;i++){ const c=cellOf[i]; sortedIdx[cursor[c]++]=i; }
}
```

Pass 1 histograms the cells, pass 2 turns the histogram into a prefix sum, and pass 3 scatters particle indices into `sortedIdx` so that the members of cell $c$ occupy the contiguous slice `sortedIdx[cellStart[c] .. cellStart[c+1])`. The whole build is $O(N + \text{cells})$, and the resulting layout means the inner loop walks neighbours contiguously in memory.

The query then decomposes the linear cell index back into $(i_x, i_y, i_z)$ and sweeps the 27-cell neighbourhood; the HUD reports the number of pairs actually evaluated per frame, which typically sits between one and two orders of magnitude below $N^2$.

**Complexity.** With roughly constant density, the expected cost is $O(N)$ — each particle examines a bounded number of neighbours — against $O(N^2)$ for the naive version. At $N = 3000$ that is the difference between $9 \times 10^6$ pair tests per frame and a few tens of thousands.

### 5.6 Long-range gravity: Barnes–Hut octree

Gravity has infinite support: every particle attracts every other, so the grid trick does not apply and an exact evaluation is irreducibly $O(N^2)$. We use the Barnes–Hut approximation [1]: a distant *group* of particles is replaced by a single pseudo-particle at the group's centre of mass.

**Structure.** The octree is a pooled SoA structure — thirteen typed arrays indexed by node id, not a graph of objects:

```
maxNodes = Math.max(256, N*4+64);
nCx,nCy,nCz : Float32Array(maxNodes) // node centre
nHalf : Float32Array(maxNodes) // half-width
nMass : Float32Array(maxNodes) // total mass
nComX/Y/Z : Float32Array(maxNodes) // centre of mass
nBody : Int32Array(maxNodes) // body index if leaf, -1 otherwise
nInternal : Uint8Array(maxNodes) // 0 = leaf, 1 = internal
nDepth : Uint8Array(maxNodes)
nChild : Int32Array(maxNodes*8) // 8 child ids, -1 = absent
```

Allocation happens once, on a population change. Rebuilding the tree every frame then costs *zero* allocations: `buildOctree()` simply resets `nodeCount=0` and re-uses the pool. Children are created lazily by `childOf()`, so an empty region of space costs nothing.

**Build.** Bodies are inserted one at a time into a root cube covering the whole world. The insertion loop handles the standard subtle case — inserting into an occupied leaf requires pushing the existing body down as well, and if both bodies land in the same octant the process must recurse:

```
if(nInternal[node]===0){
  if(nBody[node]===-1){ nBody[node]=i; return; } // empty leaf: done
  const e=nBody[node]; nBody[node]=-1; nInternal[node]=1; // occupied: subdivide
  if(nHalf[node]<0.02) return; // coincident-body guard
  const ce=childOf(node, octant(node,px[e],py[e],pz[e])); nBody[ce]=e;
  const ci=childOf(node, octant(node,px[i],py[i],pz[i]));
  if(ci===ce){ node=ci; continue; } // same octant: go deeper
  nBody[ci]=i; return;
}
```

The `nHalf < 0.02` guard is essential: two exactly (or nearly) coincident bodies would otherwise subdivide forever, hanging the browser. Below that cell size we drop the body from the tree for that frame — an invisible error at this scale, and a hard bound on depth. A second guard caps the insertion loop at 64 iterations.

**Mass pass.** A post-order recursion aggregates mass and centre of mass:

$$M_{\text{node}} = \sum_{c \in \text{children}} M_c, \qquad \mathbf{X}_{\text{node}} = \frac{1}{M_{\text{node}}} \sum_{c \in \text{children}} M_c \, \mathbf{X}_c \tag{3}$$

**Traversal and the opening criterion.** For each particle the tree is walked with an explicit stack (an `Int32Array(4096)` reused across calls — recursion here would mean a JS call frame per node and no way to bound depth). A node of width $s$ at distance $d$ is accepted as a single pseudo-particle when

$$\frac{s}{d} < \theta \qquad \Longleftrightarrow \qquad s^2 < \theta^2 d^2 \tag{4}$$

and is otherwise opened and its children pushed. The squared form is what the code evaluates, avoiding a square root per node test:

```
const s = nHalf[node]*2;
if(s*s < theta2*d2){ /* far enough: treat as one body */ }
else { /* push the 8 children */ }
```

The contribution of an accepted node is the softened inverse-square law

$$\mathbf{a}_i \mathrel{+}= \frac{G\,M_{\text{node}}\,(\mathbf{X}_{\text{node}} - \mathbf{x}_i)}{(\|\mathbf{X}_{\text{node}} - \mathbf{x}_i\|^2 + \varepsilon^2)^{3/2}} \tag{5}$$

where $\varepsilon$ is the `softening` parameter. Softening is not cosmetic: without it, two particles passing close together produce an acceleration $\propto 1/r^2 \to \infty$, and with a finite time step they are slingshotted to infinity, destroying the simulation. The $\varepsilon^2$ term bounds the force at $GM/\varepsilon^2$ and keeps the integrator stable. Note that $\varepsilon$ is included in the *force* evaluation but not in the *opening* test, which is the standard formulation.

**$\theta$ and the accuracy/speed trade-off.** $\theta$ is exposed as a slider because it *is* the algorithm's central dial:

| $\theta$ | **Behaviour** | **Cost** |
|---|---|---|
| $\to 0$ | Every node is opened; degenerates to exact all-pairs | $O(N^2)$ |
| 0.5 | High accuracy, more nodes visited | Slower |
| 0.8 | Default; the usual accuracy/speed compromise | $O(N \log N)$ |
| 1.6 | Aggressive approximation, visible clumping artefacts | Fastest |

The HUD's octree node count makes the trade-off legible in real time: raising $\theta$ visibly cuts the number of nodes visited and raises the frame rate.

A second entry point, `gravityAccelAt(root,x,y,z,out)`, evaluates the same field at an *arbitrary* point rather than at a particle. It is identical except that it omits the self-exclusion test, and it is what drives the field-arrow and lattice visualisations (§5.12).

## 5.7 Integration and stability

The two force terms, plus the creature field and the user's intervention field, are summed per particle and integrated with **semi-implicit (symplectic) Euler** — velocity is updated first, then position is updated using the *new* velocity, which is markedly more stable for orbital motion than explicit Euler at the same cost.

Friction is applied as an exponential decay expressed through a *half-life* rather than a raw coefficient:

$$\mu = 2^{-\Delta t / T_{1/2}}, \qquad \mathbf{v} \leftarrow \mu\,\mathbf{v} + \mathbf{a}\,\Delta t \tag{6}$$

```
const mu = Math.pow(0.5, dt/CFG.frictionHalfLife);
```

This formulation is framerate-independent by construction — a half-life of 45 ms means velocity halves every 45 ms of *simulated* time regardless of how many steps that took — and it gives the user a control with a physical meaning ("motion halves in this long") instead of an arbitrary number.

Three further stability measures:

- **Time-step clamp.** `simulate(1/60 * Math.min(2, rdt*60))` caps the step at 1/30 s, and the raw frame delta is itself clamped at 50 ms. After a stall (the user switches tabs and returns) a huge $\Delta t$ would blow the simulation apart; the clamp trades a moment of slow motion for survival.

- **Velocity clamp.** Speeds are limited to 90 units/s. This is the last line of defence against a numerical explosion.
- **Two-pass update.** Forces are computed for *all* particles from the start-of-step positions before *any* position is written. A single fused loop would make particle $j$ see particle $i$'s already-updated position, introducing an index-dependent asymmetry — the result would depend on particle numbering, which is exactly the kind of bug that is invisible until it isn't.

### 5.8 Boundary conditions

Two modes are available, toggled live.

**Reflecting walls (default).** Positions are clamped to the cube and the normal velocity component is reversed with restitution 0.6:

```
if(px[i]>h){ px[i]=h; vx[i]*=-0.6; }
```

**Toroidal wrap.** Space becomes a 3-torus: a particle leaving one face re-enters through the opposite one. This is more than a position remap — the *forces* must wrap too, or structures would tear at the seams. Two things change:

- **Minimum-image convention.** A separation longer than half the box is replaced by the shorter way round:

  ```
  if(ddx> h) ddx-=worldW; else if(ddx<-h) ddx+=worldW;
  ```

  so a particle near the $+x$ face genuinely feels its neighbour across the boundary as a near neighbour.
- **Periodic grid neighbourhood.** Cell indices wrap modulo `gN`, and the 27-cell sweep therefore crosses the boundary. This is enabled only when `gN` $\geq 3$; below that a cell would be its own neighbour in two directions and pairs would be double-counted.

The position remap uses a floor rather than a conditional subtraction:

```
px[i] -= worldW*Math.floor((px[i]+h)/worldW);
```

so that a particle overshooting by more than one box width in a single step still lands correctly, instead of flickering at the edge.

Gravity is deliberately *not* wrapped: a periodic gravitational field would require an Ewald summation, which is far beyond the scope of a real-time browser demo. With wrap enabled, gravity is therefore an approximation — a documented and visible one.

### 5.9 The Grazer: hierarchical modelling

The creature demonstrates the classical hierarchical-modelling requirement: a tree of transforms in which each child's world transform is the composition of its own local transform with all of its ancestors'. We use three.js's scene graph as the transform tree, so composition is handled by the library, but the hierarchy, the joint layout and the motion are ours.

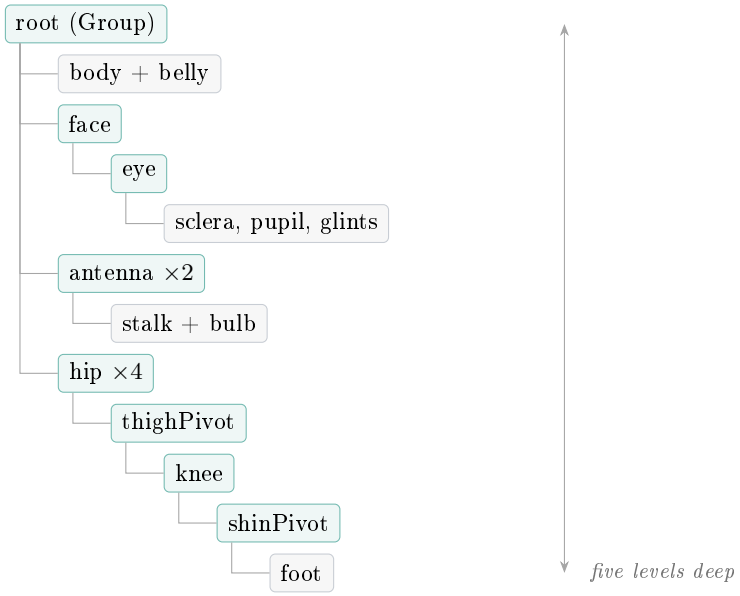



Figure 2: The Grazer's transform hierarchy. Rotating `thighPivot` carries the knee, the shin and the foot with it; rotating `shinPivot` carries only the foot. Depth from root to foot is five levels.

**Welded joints by construction.** The subtlety in building an articulated limb from primitives is that a `CylinderGeometry` is generated centred on the origin and aligned with $+y$, which is the wrong frame for a bone. Rotating such a mesh about a joint would swing it about its middle. The geometry is therefore baked into the correct frame *once*, at build time:

```
const thighGeo = new THREE.CylinderGeometry(0.2, 0.16, L1, 10);
thighGeo.rotateZ(-Math.PI/2); // lie along +x instead of +y
thighGeo.translate(L1/2, 0, 0); // span 0..L1, so the joint is at the origin
```

Now each segment spans $0 \ldots L$ along its local $+x$ with its pivot at the local origin, and the next joint is simply a child `Group` at $(L, 0, 0)$:

```
const knee = new THREE.Group(); knee.position.set(L1,0,0); thighPivot.add(knee);
```

Because the knee is a *child* of the thigh's pivot, it inherits every rotation of the thigh automatically. The segments cannot come apart — not because the animation is careful to keep them together, but because the hierarchy makes separation unrepresentable. This is the whole point of hierarchical modelling, and it is why the geometry pre-transform is worth the two extra lines.

The four hips are placed on the body surface at radius 2.45 (greater than the body's 2.6× scaled radius at that height) and rotated so that each limb's local $+x$ points radially outward, so thighs sweep away from the body and never intersect it.

**Solid body.** The Grazer is not a ghost. After integration, particles found inside its bounding radius are projected onto the surface and given an outward impulse:

```
const bR = cBodyR + psize[i]; // creature radius + particle radius
if(dd2 < bR*bR && dd2 > 1e-6){
  const d=Math.sqrt(dd2), nx=ddx/d, ny=ddy/d, nz=ddz/d;
  px[i]=cx+nx*bR; py[i]=cy+ny*bR; pz[i]=cz+nz*bR; // project onto the surface
  const vn = vx[i]*nx + vy[i]*ny + vz[i]*nz;
  if(vn<0){ const j=-vn*1.4; vx[i]+=nx*j; vy[i]+=ny*j; vz[i]+=nz*j; }
}
```

The impulse $-1.4\,v_n$ leaves a residual normal velocity of $-0.4\,v_n$, i.e. a restitution of 0.4. The `vn<0` test applies the impulse only to particles actually moving *into* the surface, without which a particle sliding along the surface would be repeatedly kicked. The collision proxy is a sphere, not the visual mesh — exact per-triangle collision against an articulated body would cost more than the entire rest of the frame, and at this scale the difference is imperceptible.

### 5.10 Procedural gait

The walk cycle is a closed-form function of a phase accumulator. There is no keyframe data anywhere.

```
const gaitSpeed = 5 + moving*0.5; // legs cycle faster when moving faster
c.phase += dt*gaitSpeed;
for(const leg of c.legs){
  const ph = c.phase + leg.phase; // per-leg constant offset
  const lift = Math.max(0, Math.sin(ph)); // half-wave: stance vs swing
  const swing = Math.cos(ph);
  leg.thighPivot.rotation.z = 0.18 + swing*0.24 - lift*0.10;
  leg.thighPivot.rotation.y = swing*0.16;
  leg.shinPivot.rotation.z = -1.7 + lift*0.5;
}
```

Each leg $k$ carries a constant phase offset $2\pi k/4$, so the four legs are evenly distributed around the cycle and the gait reads as a coordinated walk rather than four independent twitches. The half-wave rectification $\max(0, \sin)$ is what distinguishes the two halves of the cycle: during swing ($\sin > 0$) the leg lifts, and during stance ($\sin \le 0$) `lift` is pinned to zero so the foot stays planted — a lift term of sin alone would make the leg burrow into the ground on the downstroke. The constant $-1.7$ radians on the shin is the resting knee bend that gives the silhouette its insect-like character.

`gaitSpeed` is tied to the creature's speed, so the legs cycle faster when the user accelerates. This is a crude one-way coupling rather than true foot-locking, but it removes almost all of the "skating" impression for a single multiplication.

Three secondary motions layer on top, all closed-form: a hover bob ($\sin(2.2t)$ on the root), springy antennae that lean back in proportion to speed (`lean = min(0.5, moving*0.02)`), and a stochastic blink implemented as a countdown that squashes the eye's $y$ scale for 140 ms and then re-arms at a random interval of 2.5–5 s. Heading is not snapped but critically damped toward the target, `rotation.y += (heading - rotation.y)*min(1, dt*6)`, so turns ease instead of popping.

### 5.11 The creature's force field

Within a radius $R$ the Grazer applies a quadratic-falloff radial force plus a tangential swirl:

$$\mathbf{a} \mathrel{+}= \underbrace{\sigma\, S\, \frac{(1 - d/R)^2}{d}\, \mathbf{d}}_{\text{attract/repel}} + \underbrace{0.25\, S\, \frac{(1 - d/R)}{d}\, (-d_z,\, 0,\, d_x)}_{\text{swirl about } y} \tag{7}$$

with $\sigma = \pm 1$ the mode and $S$ the strength slider. The quadratic falloff makes the field vanish smoothly at $r = R$ rather than cutting off with a discontinuity that would be visible as a hard shell. The swirl term is the reason gathered particles orbit the creature instead of piling into a static ball — it converts pure radial infall into an accretion disc, which is far more legible as motion. The aura sphere drawn around the creature is scaled to exactly $R$, so the visible bubble is an honest depiction of the field's reach, and its colour tracks the mode (cool blue for attract, warm orange for repel).

### 5.12 Visualising the invisible

Three optional visualisations expose structure that is normally internal.

**Octree wireframe.** A recursive walk emits the twelve edges of each occupied node's cube into a `BufferGeometry`, capped at depth 6 (deeper levels are visually indistinguishable and expensive to emit). Empty leaves are skipped. The result makes the Barnes–Hut adaptivity immediately obvious: cells are large and sparse in empty space and subdivide finely exactly where mass concentrates.

**Gravitational field streamlines.** The field is sampled on an $n \times n \times n$ lattice ($n =$ `fieldN`, up to $20^3 = 8000$ samples) via `gravityAccelAt`. Each sample is drawn as *two* line segments sharing a geometry: a faint full-length vector showing structure, and a bright pulse that streams along it toward the mass. The pulse is parameterised by a travelling wave

```
let f = ((t*speed + FLD.ph[k]) % 1 + 1) % 1; // 0..1 travel parameter
const glow = (0.45 + 0.55*Math.sin(f*Math.PI)) * (0.5 + 0.5*s);
```

where the per-sample phase offset `FLD.ph[k]` is a linear function of position, $0.05x + 0.045y + 0.04z$. Because the coefficients are unequal and irrationally related, the resulting wave sweeps diagonally across the volume and never resynchronises into a distracting pulse-in-unison. Colour interpolates teal → orange with field strength, magnitudes are normalised against the frame's maximum, and samples below 4% strength are collapsed to a degenerate point rather than culled — keeping the vertex count constant so the buffer is never reallocated. Rendering uses additive blending with `depthWrite:false`, so overlapping streamlines accumulate brightness instead of z-fighting.

The sample *directions* are rebuilt only every fourth frame (`vizTick&3`), while the pulse is redrawn every frame. This is the key optimisation: the expensive part (8000 octree traversals) runs at 15 Hz, while the cheap part (writing 8000 line positions) runs at 60 Hz, and the eye cannot tell.

**Deforming space lattice.** A $9^3$ grid of nodes, connected into a wireframe by $\sim$ 1,944 segments, is displaced from its rest position by the local gravitational acceleration (scaled $\times 7$) and by the user's active poke field, with the total displacement capped at 9 units so that a violent warp bends space instead of shredding the lattice. Rest positions are stored separately from current positions, so the deformation is always computed from the undeformed state and never accumulates drift. This is the most direct answer to "what is gravity doing to space?" that the application offers, and it is the intended companion to the Warp tool.

## 5.13 The frame loop

```
function frame(now){
  const rdt = Math.min(0.05, (now-last)/1000); last=now;
  let octRoot = -1;
  if(CFG.running) octRoot = simulate(1/60*Math.min(2, rdt*60)); // physics
  pilot(rdt); // read keyboard, move creature
  animateCreature(rdt, t); // gait, antennae, blink, aura
  syncMesh(); // particles -> instance matrices
  if((CFG.showField||CFG.showLattice) && CFG.gravity && octRoot<0) octRoot = buildOctree();
  ... // visualisations
  if(CFG.follow && creature.root) cam.target.lerp(creature.root.position, 0.05);
  applyCam();
  renderer.render(scene, camera);
  hud.innerHTML = ...;
  requestAnimationFrame(frame);
}
```

Two details are worth pointing out. First, `simulate()` *returns* the octree root it built, so the visualisations can reuse it rather than rebuilding; the tree is only constructed independently when the simulation is paused but a visualisation still needs it. Second, the follow camera uses `lerp(..., 0.05)` rather than assignment, giving a critically-damped chase that lags slightly behind the creature — an instant snap would make the whole world jitter with the gait's hover bob.

### 5.14 Performance summary

| Stage | Complexity | Notes |
|---|---|---|
| Grid build | $O(N + \text{cells})$ | Counting sort, allocation-free |
| Short-range forces | $O(N)$ expected | 27-cell sweep, constant density |
| Octree build | $O(N \log N)$ | Pooled nodes, allocation-free |
| Gravity traversal | $O(N \log N)$ | Iterative, shared explicit stack |
| Mesh sync | $O(N)$ | One reused `Object3D` |
| Rendering | 1 draw call for $N$ particles | `InstancedMesh` |
| Field samples | $O(n^3 \log N)$ | Throttled to every 4th frame |
| Lattice | $O(9^3 \log N)$ | Only when enabled |

The HUD exposes the live picture: frame rate, particle count, pairs evaluated per frame, octree node count, and the current mode (*chem* or *2-scale*). The steady-state allocation rate is zero, so no GC pause ever interrupts the animation.

## 6 Implemented interactions

The application is interactive in five distinct registers: the user can *look* (camera), *touch* (drag tools), *strike* (one-shot impulses), *inhabit* (piloting the creature), and *legislate* (editing the rules themselves). This section documents each.

### 6.1 Camera control

The camera is driven by a spherical orbit state, hand-written rather than taken from three.js's `OrbitControls`:

```
const cam = { target:new THREE.Vector3(0,0,0), theta:0.7, phi:1.15, radius:130 };
function applyCam(){
  const st=Math.sin(cam.phi), r=cam.radius;
  camera.position.set(
    cam.target.x + r*st*Math.sin(cam.theta),
    cam.target.y + r*Math.cos(cam.phi),
    cam.target.z + r*st*Math.cos(cam.theta));
  camera.lookAt(cam.target);
}
```

| Input | Action | Detail |
|---|---|---|
| Left-drag (Orbit tool) | Orbit | $\theta \mathrel{-}= 0.005\,\Delta x$, $\phi \mathrel{-}= 0.005\,\Delta y$ |
| Right-drag | Orbit | Always orbits, whatever tool is selected |
| Middle-drag | Pan | Translates `cam.target` in the view plane |
| Shift + left-drag | Pan | Alternative for trackpads without a middle button |
| Scroll wheel | Zoom | Multiplicative, $\times 1.08$ per notch |
| One-finger drag | Orbit | Touch |
| Two-finger pinch | Zoom | Touch |

Several deliberate details:

- **$\phi$ is clamped** to $[0.08,\ \pi - 0.08]$. Reaching the poles exactly makes the view direction parallel to the up vector and `lookAt` produces a degenerate basis — the camera flips. The clamp keeps a sliver of angle in reserve.
- **Zoom is multiplicative**, not additive, and clamped to $[20, 500]$. An additive zoom crawls when far away and overshoots through the subject when close; a multiplicative one feels linear at every scale.

- **Pan speed scales with radius** (`cam.radius*0.0016`), so a given mouse movement covers a constant fraction of the screen regardless of zoom.
- **Panning breaks follow mode.** If the user pans while the camera is following the creature, follow silently disables itself and the toggle in the console updates. Without this the camera would fight the user's input and lose, which reads as a bug rather than as a feature.
- **Right-drag always orbits**, so the user never becomes trapped in a tool: there is always a way to move the view without changing the tool selection. The context menu is suppressed on the canvas to make this possible.

### 6.2 Direct manipulation: the drag tools

Four tools share the left mouse button, selected in the *Intervene in space* section of the console.

| Tool | Effect of a left-drag |
|---|---|
| **Orbit** | Rotates the camera (the default; no effect on the simulation). |
| **Push** | Radial outward force within the brush sphere; blasts a cavity in the soup. |
| **Pull** | Radial inward force; gathers particles into the cursor. |
| **Warp** | A violent composite: swirl about the view axis, inward collapse, and compression along the view axis, producing a lensing/folding effect. |

**Where is the cursor in 3D?** The mouse gives a 2D position; the force needs a 3D one. We unproject the cursor onto the plane through the orbit target whose normal is the camera's view direction — i.e. the plane the user perceives as "the screen, pushed into the scene":

```
function screenToWorld(clientX, clientY, out){
  _ndc.x = ((clientX-rect.left)/rect.width )*2 - 1; // pixels -> NDC
  _ndc.y = -((clientY-rect.top )/rect.height)*2 + 1; // (y flips)
  _ray.setFromCamera(_ndc, camera);
  camera.getWorldDirection(_n);
  _plane.setFromNormalAndCoplanarPoint(_n, cam.target);
  const r = _ray.ray.intersectPlane(_plane, _hit);
  if(r){ out.copy(_hit); return true; }
  return false;
}
```

This choice matters: because the plane is always camera-facing and passes through the orbit target, the tool acts at the depth the user is looking at, and it behaves identically from every viewing angle. The whole function reuses five module-level scratch objects and allocates nothing, which matters because it runs on every `mousemove`.

**Force models.** With $d$ the distance to the poke centre, $R$ the brush radius, $\text{fall} = 1 - d/R$, and $I$ the global intensity multiplier:

$$\text{Push/Pull:}\quad \mathbf{a} \mathrel{+}= \pm 420\, I\, \frac{\text{fall}^2}{d}\, \mathbf{d} \tag{8}$$

$$\text{Warp:}\quad \mathbf{a} \mathrel{+}= \underbrace{620\, I\, \text{fall}^2\, (\hat{\mathbf{n}} \times \mathbf{d})}_{\text{swirl about the view axis}} \underbrace{- 520\, I\, \frac{\text{fall}^2}{d}\, \mathbf{d}}_{\text{collapse}} \underbrace{- 300\, I\, \text{fall}\, (\mathbf{d} \cdot \hat{\mathbf{n}})\, \hat{\mathbf{n}}}_{\text{axial compression}} \tag{9}$$

The swirl axis $\hat{\mathbf{n}}$ is captured as the camera's forward vector at the moment the drag begins, so the vortex always spins in the plane of the screen — the user sees a whirlpool face-on rather than edge-on. The quadratic falloff again avoids a hard edge at $r = R$. A translucent sphere with a rotating ring marks the brush in the scene, sized to $R$ and coloured by mode (warm for push, teal for pull/warp), so the region of influence is never guesswork.

**Brush radius** is a slider (5–60 units); **Intensity** (0.2–6×) is a master multiplier applied to the drag tools *and* to every impulse below.

### 6.3 One-shot impulses

Six buttons apply an instantaneous velocity change to every particle, centred on the current orbit target — i.e. on whatever the user is looking at. Unlike the drag tools, these modify velocity (or, for Fold, position) directly rather than adding a force.

| Impulse | Effect |
|---|---|
| **Shockwave** | Radial outward kick, $40I/(1+0.05d)$ — an explosion whose strength decays with distance. |
| **Implode** | Radial inward kick, $45I/(1+0.04d)$. |
| **Vortex** | Tangential kick about the vertical axis through the epicentre, $35I$, plus a small random vertical component that keeps the disc from becoming perfectly flat. |
| **Warp** | Swirl about the view axis combined with inward collapse, both $\propto 1/(1+0.03d)$. |
| **Fold** | A genuine reflection of space: every particle in the far half is mirrored onto the near half, position *and* velocity. |
| **Jitter** | A uniform random kick of magnitude up to $40I$ per axis — thermal noise, useful for shaking a structure out of a local minimum. |

**Fold** is the most interesting of the six because it is a transformation of space rather than a force. Taking $\hat{\mathbf{n}}$ as the view axis and $a = \mathbf{d}\cdot\hat{\mathbf{n}}$ the signed distance from the fold plane, every particle with $a > 0$ is reflected:

```
if(along>0){
  px[i]-=2*along*ax; py[i]-=2*along*ay; pz[i]-=2*along*az; // mirror position
  const vn = vx[i]*ax + vy[i]*ay + vz[i]*az;
  vx[i]-=2*vn*ax; vy[i]-=2*vn*ay; vz[i]-=2*vn*az; // mirror velocity
}
```

Reflecting the velocity as well as the position is what makes the result coherent: a mirrored particle continues along the mirrored trajectory, so folding a rotating galaxy produces two counter-rotating halves that then interact — rather than a pile of particles moving in incoherent directions.

### 6.4 Piloting the Grazer

| Key | Action |
|---|---|
| W / S | Move forward / backward along the creature's heading |
| A / D | Strafe left / right, *and* turn the heading (±1.8 rad/s) |
| Q / E | Ascend / descend |
| Shift | Boost (2.2× acceleration) |
| H | Show/hide the control console |

Movement is force-based rather than positional. The input assembles an acceleration in the creature's local frame, which is then integrated with heavy exponential damping:

```
if(active){
  if(keys['a']||keys['d']) c.heading += (keys['d']?-1:1)*dt*1.8;
  accel.normalize().multiplyScalar(60*boost);
}
c.vel.addScaledVector(accel, dt);
c.vel.multiplyScalar(Math.pow(0.02, dt)); // strong damping, framerate-independent
c.root.position.addScaledVector(c.vel, dt);
c.root.position.clamp(min, max); // stay inside the world
```

The `normalize()` before scaling is what stops diagonal movement from being $\sqrt{2}$ times faster than axis-aligned movement — the classic bug. The damping uses the same framerate-independent exponential form as the particle friction. The result has weight: the creature accelerates and coasts rather than teleporting, which also means it stirs the soup as it goes.

A hint bar listing the controls fades in whenever the creature is moving and fades out when it stops, so the controls are discoverable without a permanent overlay.

**Follow camera.** A toggle makes `cam.target` chase the creature's position each frame with a 5% lerp. As noted above, panning cancels it.

### 6.5 Editing the rules: the attraction matrix

The most consequential interaction in the application is also the smallest widget. The matrix editor is a $T \times T$ grid of coloured cells with a row and column header of type swatches; **rows act on columns**, so cell $(i, j)$ is the force type $i$ feels toward type $j$.

- **Colour encodes value**: green for attraction, red for repulsion, near-black for neutral, with the channel intensity proportional to $|A_{ij}|$.
- **Clicking a cell cycles** it through five discrete values: $+1 \to +0.5 \to 0 \to -0.5 \to -1 \to +1$. The implementation snaps to the nearest step first, so a cell holding a random value like 0.37 from *Reroll* still cycles sensibly.
- **Changes are instantaneous**: `A` is read directly by the force kernel every frame, so the soup reorganises as the user clicks — no apply button, no reseed.
- **A tooltip** reports the exact value, e.g. `type 2 -> 3 (-0.50)`.

The asymmetry is the pedagogical point, and the editor is built to make it discoverable: setting $A_{01} = +1$ and $A_{10} = -1$ — one click each — makes type 0 chase type 1 while type 1 flees, and the pair immediately begins to self-propel across the volume. Making the matrix symmetric instead produces static clumps. A user can find this out in ten seconds without reading anything.

### 6.6 The control console

The console is a collapsible panel of nine groups, all collapsed by default so that the interface is not a wall of sliders. Every control carries a one-line explanation underneath it. It is built by a small widget factory (`slider`, `toggle`, `btn`, `group`, `withDesc`) — around 30 lines of DOM code that replaces a UI dependency.

| Group | Control | Effect |
|---|---|---|
| **Simulation** | Run / Pause | Freezes physics; rendering and camera stay live |
| | Reset | Re-scatters particles, keeps the rules |
| | Reroll | New random rule matrix, keeps the particles |
| **Attraction rules** | The $T \times T$ matrix | See §6.5 |
| **Chemistry** | Force strength (0.2–8) | Overall power of the type rules |
| | Interaction radius (3–20) | $r_{\max}$; rebuilds the grid live |
| | Repulsion core (0.05–0.6) | $\beta$; incompressibility |
| | Friction half-life (0.01–0.2) | Damped and syrupy → lively |
| **Gravity** | Enable gravity | Switches the HUD to *2-scale* mode |
| | Strength $G$ (0–4) | |
| | $\theta$ (0.2–1.6) | Barnes–Hut accuracy ↔ speed |
| | Softening (0.5–6) | Stability at close range |
| | Show octree | Draws the subdivision |
| | Show gravity field | Animated streamlines |
| | Field density (3–20) | Samples per axis ($n^3$ total) |
| **Population** | Particles (200–3000) | Reallocates buffers on release |
| | Types (2–8) | Resizes the matrix and reseeds types |
| | Size variance (0–1) | Uniform → highly varied masses |
| **Intervene** | Intensity (0.2–6×) | Master multiplier for all interventions |
| | Tool: Orbit/Push/Pull/Warp | Left-drag behaviour |
| | Brush radius (5–60) | |
| | Six impulse buttons | See §6.3 |
| **World & render** | World size (40–160) | Rescales positions and rebuilds the cage |
| | Wrap edges (torus) | Periodic boundaries + minimum image |
| | Accent point lights | The teal/pink pair |
| | Show space lattice | Deforming wireframe of space |
| **Grazer** | Show creature | Also disables its force field |
| | Follow camera | |
| | Field: Attract/Repel | Aura colour follows the mode |
| | Field radius (6–40) | Matches the visible aura |
| | Field strength (0–400) | 0 = harmless wanderer |
| **Presets** | Nine one-click regimes | See §6.8 |

**Live reconfiguration.** Several of these controls rebuild live state, which is where the interesting engineering hides:

- **Particle count** reallocates all nine typed arrays, the octree pool and the grid, then reseeds and rebuilds the `InstancedMesh` (whose instance count is fixed at construction). It is bound to the `change` event rather than `input`, so it fires once when the slider is released instead of on every pixel of the drag.
- **Interaction radius** changes the grid cell size, so `rebuildGridDims()` runs on every slider step — cheap, since it only reallocates index arrays.
- **World size** is the subtlest: rather than clipping particles into the new volume, it *scales all positions* by the ratio of new to old half-width, so structures survive the resize instead of being

crushed at the walls. The cage and the lattice are then rebuilt.

### 6.7 Presets

Nine presets reconfigure the rules, gravity and size distribution in one click. Each is a genuinely different regime rather than a cosmetic variation:

| Preset | Types | Gravity | $r_{\max}$ | Strength | Character |
|---|---|---|---|---|---|
| Cells | 5 | off | 10 | 3.2 | Soft membrane-like blobs |
| Chase | 4 | off | 12 | 4.5 | Asymmetric hunting, moving trails |
| Galaxies | 3 | $G = 1.6$ | 8 | 1.5 | Spiral, orbiting clumps |
| Web | 6 | off | 9 | 3.8 | Connected filament network |
| Orbits | 3 | $G = 2.4$ | 6 | 0.9 | Slow stable orbits |
| Worms | 3 | off | 14 | 5.0 | Wriggling chains |
| Crystal | 4 | off | 8 | 2.6 | Rigid lattice-like solid |
| Nebula | 5 | $G = 0.7$ | 11 | 2.2 | Diffuse swirling clouds |
| Swarm | 3 | off | 13 | 4.2 | Flocking drift |

Each preset also sets $\beta$, the friction half-life and the size variance, rerolls the matrix, reseeds, and rebuilds the panel so every slider reflects the new state. Because the matrix is rerolled, clicking the same preset twice gives two different worlds *of the same character* — the preset fixes the regime, not the outcome.

## 7 User manual

### 7.1 Running the application

1. Open `index.html` in any modern browser — double-clicking the file is enough; no server, no installation, no build step.
2. An internet connection is needed the first time, to fetch three.js from the CDN. To work fully offline, download `three.min.js` and change the first `<script src>` in `index.html` to point at the local copy.
3. The simulation starts running immediately with 1000 particles, 5 types, a random rule matrix and gravity off.

### 7.2 The screen

- **Left** — the control console. Click a section header to expand it; click ◀ (or press H) to hide the whole panel, and the ≡ tab to bring it back.
- **Top right** — the HUD: frame rate, particle count, pairs evaluated per frame, octree node count, and the current mode (*chem* = chemistry only, *2-scale* = chemistry + gravity).
- **Bottom centre** — the pilot hint bar, which appears while the Grazer is moving.
- **Centre** — the simulation, inside a faint wireframe cage marking the world boundary.

### 7.3 Quick reference

| Input | Action |
|---|---|
| Left-drag | Orbit (or use the active tool) |
| Right-drag | Orbit (always) |
| Middle-drag / Shift+left-drag | Pan |
| Scroll | Zoom |
| W A S D | Pilot the Grazer |
| Q / E | Grazer up / down |
| Shift | Boost |
| H | Toggle the console |
| One finger / two fingers | Orbit / pinch-zoom |

### 7.4 A guided tour

The following sequence takes about ten minutes and exercises every subsystem.

1. **Watch first.** Let the default random matrix run for thirty seconds and orbit around it. Whatever structure appears was not designed — it is the matrix expressing itself.

2. **Change one rule.** Open *Attraction rules* and click a single cell until it is bright green, then click its mirror cell until it is bright red. Watch those two colours begin to chase each other. This is asymmetry, and it is the engine of every moving pattern in the system.

3. **Try the presets.** Open *Presets* and work down the list. Compare *Crystal* (rigid, locked) with *Worms* (crawling chains) — same code, same particles, different numbers.

4. **Turn on gravity.** Click *Galaxies*. The mode indicator in the HUD switches to *2-scale*. Now enable *Show octree*: the cubes are large in empty space and subdivide tightly around the clumps. This adaptivity is why the frame rate survives.

5. **Break the octree on purpose.** With the octree visible, drag $\theta$ from 0.8 up to 1.6 and watch the node count in the HUD collapse and the frame rate climb — then drag it down to 0.3 and watch the opposite. You are looking directly at the accuracy/speed trade-off.

6. **See the field.** Enable *Show gravity field* and raise *Field density* to about 12. Bright pulses stream along each vector toward mass, with a wave sweeping diagonally across the volume.

7. **Bend space.** Enable *Show space lattice*, then select the **Warp** tool and drag through the middle of a clump. The lattice twists and collapses around the cursor. Raise *Intensity* to 4× and try again.

8. **Fold the universe.** Click *Fold*. The far half of space is mirrored onto the near half; the two halves then collide and interact.

9. **Go for a swim.** Set *Field: Attract*, *Field strength* to $\sim 300$, enable *Follow camera*, and pilot with W A S D. Particles gather into an orbiting disc around the Grazer and are dragged along behind it. Switch to *Repel* to plough a tunnel through the soup instead.

10. **Push the limits.** Raise *Particles* to 3000 and watch the HUD. If the frame rate drops, raise $\theta$, lower the field density, or turn off the visualisations — and note which one buys the most.

### 7.5 Troubleshooting

| Symptom | Cause and remedy |
|---|---|
| Black screen, nothing renders | three.js failed to load from the CDN. Check the network, or switch to a local copy of `three.min.js`. |
| Low frame rate | Lower *Particles*; raise $\theta$; lower *Field density*; disable the octree/field/lattice visualisations. |
| Everything collapses to a dot | $G$ too high or *Repulsion core* too low. Raise $\beta$, lower $G$, or raise *Softening*. |
| Particles fly apart violently | *Force strength* or *Intensity* too high; friction half-life too long. The velocity clamp prevents a true explosion but the result is still noise. |
| Nothing happens when I drag | The active tool is *Orbit*. Select Push, Pull or Warp — or just use right-drag to orbit. |
| The camera will not follow | Follow mode disables itself whenever you pan. Re-enable it in the *Grazer* section. |
| No creature visible | *Show creature* is off, or it has been piloted out of view. Enable *Follow camera* to find it. |

## 8 Limitations and future work

We state the limitations plainly; each is a considered trade-off rather than an oversight.

1. **Single-threaded, CPU-bound.** The whole simulation runs on the main thread. Beyond roughly 3000 particles the force evaluation, not the rendering, is the bottleneck. The natural fixes are a Web Worker with a `SharedArrayBuffer` (which would need COOP/COEP headers, and therefore a server — breaking the `file://` promise), or moving the integration to the GPU via transform feedback or a compute pass, which is a substantial rewrite.

2. **Gravity is not periodic.** With *Wrap edges* enabled, the short-range term uses the minimum-image convention but gravity does not: it still sees the non-wrapped configuration. A correct treatment needs an Ewald summation, which is out of scope for a real-time browser application.

3. **The creature's collision proxy is a sphere**, not its visual silhouette. Particles bounce off an invisible ball rather than off the legs. Exact collision against an articulated body would dominate the frame budget for an effect that is barely perceptible at this scale.

4. **The gait is not foot-locked.** Leg cycle speed tracks body speed, but feet are not constrained to be stationary while planted, so a slight skate remains at high speed. Proper foot-locking needs inverse kinematics.

5. **Momentum is not conserved** — by design. The asymmetric matrix violates Newton's third law, which is the entire point of the chemistry model, but it does mean the system has no conserved quantity to validate against.

6. **Bodies dropped at extreme density.** The octree's coincident-body guard drops a particle from the tree when a cell shrinks below 0.02 units. This introduces a small, bounded gravitational error in extremely dense clumps, and is preferable to an infinite subdivision that would hang the browser.

7. **Instance count is fixed at construction**, so changing the particle count rebuilds the `InstancedMesh`. This causes a brief hitch, which is why the control fires on slider release rather than during the drag.

**Natural extensions.** GPU-side integration; a proper IK solver with foot locking; multiple creatures with flocking behaviour among themselves; recording and replaying parameter trajectories; saving and loading matrices as shareable URLs; and soft-body or bonded-particle chains built on the existing neighbour grid.

## 9 Conclusions

PRIMORDIA sets out to demonstrate that complexity does not require complicated rules. Two force laws — a piecewise-linear typed kernel and a softened inverse-square law — acting at two different spatial scales are sufficient to produce cells, chains, filaments, orbits, spirals and swarms, none of which appears anywhere in the source as an explicit construct.

The engineering that makes this interactive rather than merely correct is the pairing of each force law with the acceleration structure suited to its scale: a uniform grid, valid precisely because the short-range kernel has compact support, and a Barnes–Hut octree, valid because a distant cluster is well approximated by its centre of mass. Both are built on pooled typed arrays, so the steady-state allocation rate is zero and the animation is never interrupted by a garbage collection pause. The user-facing payoff of that work is the $\theta$ slider and the octree visualisation: the approximation is not hidden behind the curtain, it is on stage, and the user can watch the trade-off it makes.

The project satisfies its self-imposed asset-free constraint completely: every vertex, texel and joint angle is computed in JavaScript at run time, and the entire application consists of one HTML file, one stylesheet and ten JavaScript files with a single third-party dependency.

## References

[1] J. Barnes and P. Hut, *A hierarchical $O(N \log N)$ force-calculation algorithm*, Nature, vol. 324, pp. 446–449, 1986.

[2] J. Ventrella, *Clusters — artificial life with simple particle interaction rules.* `http://www.ventrella.com/Clusters/`

[3] H. Ahmad (`hunar4321`), *Particle Life — a simple simulation of emergent behaviour.* `https://github.com/hunar4321/particle-life`

[4] three.js — a JavaScript 3D library, revision 128. `https://threejs.org`

[5] S. J. Aarseth, *Gravitational N-Body Simulations: Tools and Algorithms*, Cambridge University Press, 2003. (Reference for softening and time-step stability.)