Map Rules

This document is the **authoritative specification of the game world's data model and rules**. It is the ground truth for how maps are stored, how blocks behave, and how gameplay (pathfinding, movement, editing, generation) derives from that data.

It binds **humans and AI agents alike**. If code, docs, or habits contradict this document, this document wins. If a rule genuinely needs to change, change it here first, then update the code — never the reverse.

Implementation status is marked per section (`[built]`, `[partial]`, `[planned]`) so readers know what exists today vs what the rules call for.

---

1. Purpose & scope

The model must serve these goals, all from **one** data structure:

- **Final Fantasy Tactics maps, 1:1** — integer-height terrain, slopes, steps, thin (.2) blocks, water, bridges.
- **Space-Engineers-style building** — a broad set of block shapes, placed/rotated/removed by an editor, with arbitrary fractional heights.
- **Rolling hills of any height** — continuous terrain surfaces, not grid-quantized.
- **Dynamic terrain** — explosions and moves can carve holes in walls and terrain.
- **Procedural generation** — deterministic from a seed: hills, cliffs, rocks, buildings; fixed-size, round, and open-world maps.
- **An in-game map editor** — place / rotate / delete blocks.

`[built: chunked voxel storage, codec, box + smooth renderers]` — the rest of the rules describe the full model; parts marked planned are not yet implemented.

---

2. Guiding principles

1. **The data layer is the ground truth.** Every block's exact size and shape lives in the map data. Rendering never modifies it.
2. **Render never feeds back.** Smoothing, shading, props, and cosmetics are derived in the render layer and can never influence rules or map data.
3. **One structure.** A map is a single voxel grid. "Terrain" and "blocks" are *conceptual* roles of voxels, not separate data structures.

4. **No derived storage.** Anything derivable from the voxel grid (column tops, surfaces, adjacency) is computed or cached, never stored as map data.
5. **Deterministic.** Everything derived from the map + seed (generation, cosmetics) reproduces identically given the same inputs.
6. **Append-only catalog.** Tile types are added, never reordered or removed (maps reference them by index).
7. **Extensible shapes.** A block shape is an enum value plus geometry. Adding a shape is additive and safe.
8. **The map stores facts; rules decide.** Walkability, movement cost, hazard damage, and LoS are *judgments* the rules layer derives from the data — never stored in the map.

---

3. The voxel model

3.1 Storage

- The map is a **3D grid of 1×1×1 cells**.
- Cells are grouped into **chunks of 16³** (`VoxelMap.CHUNK_SIZE`), keyed by chunk coordinate `Vector3i`.
- Fixed maps occupy a bounded set of chunks (an optional `bounds` AABB); open-world maps are unbounded and generate chunks on demand.
- `[built]` — `src/map/voxel_map.gd`.

3.2 Voxel encoding

Each cell holds at most **one block**. A cell is either empty or a block:

```
empty                       0
block  { type, shape, rot, height }
```

Packed as 3 bytes little-endian: - byte 0 — `type + 1` (1-indexed; `0` = empty, so catalog index 0 never collides with "empty") - byte 1 — `shape` (4 bits) | `rot` (4 bits) - byte 2 — `height` (0..255 → fraction 0..~1.0, 1/256 resolution); `0` means "use the shape default"

`[partial]` — the 2-byte form (type/shape/rot) is built; the `height` byte is **planned** (schema 4).

3.3 Height = continuous surfaces

`height` is the fraction of its cell that a block occupies, measured from the block's anchored edge:

- `FULL` occupies the whole cell.
- `HALF` is a block anchored at the bottom, occupying `height` of the cell (default 0.5).
- `SLAB` is a block anchored at the top, occupying `height` of the cell (default 0.2).

A column's **surface height** is `cell_y + height` — a continuous float. Terrain hills therefore rise smoothly (0.8, 0.9, 1.0, 1.3, 1.7…) with no quantization. `[planned]`.

---

4. Materials & tile types

A tile type (catalog entry) is the shared identity of a **material**. It describes *what the material is* — not *how the world treats it*.

Material facts (ground truth):

field	meaning
`id`	stable name (`StringName`)
`solid`	physically occupies its volume
`fluid`	physically non-solid; can flow later (water)
`shape`	canonical block shape (voxels may override per cell)
`height`	canonical fractional height (voxels may override)

Rules-layer properties (NOT ground truth — the rules engine owns these):

`walkable`, `move_cost`, `hazard` / `hazard_damage`, and the behavioral flags (`BLOCKS_LOS`, `IS_PLATFORM`, `BLOCKS_JUMP`, `REQUIRES_FLIGHT`). These are *judgments* a rules layer consults by material. They may currently live on `TileType` for convenience, but they are **rules data, never map data** — the map never carries them.

Contract: the catalog is **append-only by index**. Maps store type indices; never reorder or remove entries. `[built, partial]` — facts live on `TileType`; the rules-layer split is built (`src/rules/material_rules.gd` owns passability/cost/hazard/flags); per-type `height` is still planned.

---

5. Block shapes

The shape enum defines the geometry of a block within its 1×1×1 cell. **Every shape must be a closed volume** (no open faces) so it renders cleanly.

shape	volume
`FULL`	whole cell
`HALF`	bottom block of `height`
`SLAB`	top slab of `height`
`SLOPE`	full-length ramp (+x)

shape	volume
`HALF_SLOPE`	half-length ramp
`STAIR`	stepped block
`WEDGE`	corner wedge (closed)
`CORNER`	L-shaped inner corner

- **Rotation** in 90° steps (`rot` 0–3 meaningful; geometry treats `rot % 4`).
- Directional shapes (`SLOPE`, `HALF_SLOPE`, `STAIR`, `WEDGE`, `CORNER`) rotate around Y.
- The shape gallery map (`shape_gallery.vmap`, planned) shows every shape in isolation and is the visual reference.

`[built: FULL/HALF/SLAB/SLOPE/WEDGE]` — HALF_SLOPE/STAIR/CORNER and the closed WEDGE are planned. `src/render/shape_geometry.gd`.

---

6. Terrain

Terrain is **voxel columns**: a run of solid voxels rising from the map base (`y=0`) to a column top. The top voxel carries the fractional `height`.

- **Rolling hills** = columns whose tops vary continuously (`surface = y + height`).
- **Cliffs/steps** = adjacent columns whose tops differ.
- **Slopes** = a surface that ramps between adjacent column tops (FFT-style), *or* a `SLOPE` block where a discrete slope tile is wanted.
- **The base** is implicitly `y=0`; generators fill solid voxels from the base up.

`[partial]` — column filling exists via FULL/HALF stacks; continuous `height` tops are planned.

---

7. Blocks & placement

Blocks are voxels placed by the editor, generator, or gameplay:

- Blocks **snap to cells** (integer grid). A block occupies `[cell_y, cell_y + height)`.
- **Blocks replace terrain.** Placing a block clears any terrain voxels under its footprint; removing it restores terrain (generator- or map-provided). This is the Space-Engineers rule.
- A **building's footprint** may flatten sloped terrain under it (placement levels the columns it touches) — chosen per-building by the placement rule.
- Buildings always rest on integer cell bounds; they never "stack on" a fractional block in the same column.

`[planned]` — no placement engine yet.

---

8. Surfaces, adjacency & movement

These are the rules rules-engine and pathfinding will consume. All are derived from the voxel grid, never stored. The engine receives **only facts** — surface heights, block shapes, materials, fluid state, adjacency — and **produces the verdicts** (walkable? cost? blocked?). Verdicts are never written back into the map.

8.1 Surfaces

A **surface** is a place a unit can stand: - **Column top:** `surface(x, z) = y + height` of the top voxel in column `(x, z)` (terrain). - **Block top:** the top face of any solid block (stand on a roof, a bridge, a wall cap).

`surface(x, z)` returns `{ height: float, type }`. `[planned]`.

8.2 Adjacency

Rules query neighbors directly: `voxel_at(x, y, z)` plus its 6 (or 26) neighbors — a plain grid read. Anything about an adjacent block (its shape, height, type, fluid state) is available; the rules decide what it *means*.

8.3 Movement thresholds (current rule)

Movement cost between adjacent columns compares their surface heights. **Rise** = `surface(neighbor) - surface(current)`. The current thresholds:

- **Rise ≤ 0.25** — free step onto a flat surface (or up a very low lip).
- **0.25 < rise ≤ 1.0** — needs a supporting slope/step (the approach side has a `SLOPE`/`STAIR`/partial block, or the columns are separated by a ramp surface), or costs extra move.
- **Rise > 1.0** — blocked, unless the actor has a jump/lift ability that explicitly allows it.

These numbers are a **deliberate, tunable rule** — they live in one place (the rules module) and must not be scattered. `[built: src/rules/movement_rules.gd]`.

8.4 Pathfinding

Pathfinding runs over walkable surfaces: - Nodes = `(x, z, surface)` (a column top, or a block top). - Edges = adjacent surfaces within the movement thresholds (8.3). - Blocked by: non-walkable surfaces, `solid` blocks between surfaces, hazards that forbid entry. - `surface()` results are cached per map and invalidated only where edited.

Built as a **budgeted 4-directional Dijkstra** over the surface graph: `PathEvaluator` (`src/pathfinding/path_evaluator.gd`) composes terrain rules (passability/cost), climb rules (`MovementRules`), and unit mobility (`UnitProfile` traits/statuses — fly, float, swim, slow/stop) into per-step verdicts; `Pathfinder` (`src/pathfinding/pathfinder.gd`) returns either the lowest-cost path to a destination (player move, NPC) or the full reachability bubble (the FFT movement range), and reports every hazardous cell crossed so movement/combat can apply the rule's damage and statuses. Surfaces currently derive from block shape geometry (schema 3); the `height` byte (schema 4) will refine them. `[built: src/pathfinding/*, src/rules/*]`.

---

9. Fluids

- Water (and future fluids) is a **non-solid** `fluid` **voxel type** (`solid = false`, `fluid = true`).
- Pools are clusters of fluid voxels filling low columns.
- Fluid flow simulation is **future**; the data contract only requires a fluid voxel with a type and height. `[built: water type is fluid]`.

---

10. Dynamic edits

Any code may read or modify voxels:

- `set_voxel(x, y, z, block)` / `clear_voxel(x, y, z)` — the editor, abilities, and explosions all go through these.
- **Explosions** clear a voxel sphere (walls *and* terrain), carving craters.
- **Surface-cache invalidation:** editing a column invalidates that column's cached `surface()`.
- Edits are never "render changes" — they are data changes that the renderer reflects.

`[built: set/clear; planned: cache invalidation]`.

---

11. Procedural generation contract

Generators write ordinary voxels; they are subject to the same rules.

- **Deterministic** from `(seed, position)`. Same seed → same map, everywhere, always.
- **Terrain:** per-column height from noise → fill columns from the base (rule 6).
- **Structures:** stamp voxel shapes (buildings, trees, rocks — those occupying voxel space) on top (rule 7).
- **Map shapes:** square = bounded chunk set; **round** = square bounds with a circular playable mask (columns outside the circle stay empty); **open world** = unbounded, chunks generated on demand from `(seed, chunk_coord)`.

- Generators must produce *valid* maps (rule 12).

`[planned]`.

---

12. Codec & validation

- Binary schema versioned in the header; loaders reject mismatches loudly.
- A map is **invalid** if: a voxel's type index exceeds the catalog, `shape` is unknown, `rot` ≥ 8, or chunk data is malformed. Loaders must reject invalid maps, never silently repair them.
- The voxel grid is the only serialized form. `[built]`.

---

13. Rendering discipline

Two views exist; both are derived from the same data:

- **RAW** — the map data literally: every block at its exact shape/height, flat per-type colors. This is the truth view; rules are debugged against it.
- **RENDER** — cosmetics only: the **exact same voxel shape geometry** as the debug view, plus a seeded per-cell tint, rocks/foliage props, day/night, and the top-grid shader. **Effects never change geometry** — slopes stay slopes, voids stay holes; only colors and props are added.

`[built: debug + render views — identical shape geometry, render adds tint + props]`.

---

14. Rules for agents

Non-negotiable conventions for anyone (human or AI) working on this project:

1. **Never store derived data in the map.** Surfaces, adjacency, LoS, cached geometry — computed or cached, never serialized.
2. **Never let rendering feed back into logic.** No rule may depend on what the render layer does.
3. **The catalog is append-only.** Add types; never reorder or remove.
4. **Movement/surface rules live in one place.** Thresholds (§8.3) are a single tunable module, not scattered constants.
5. **Follow this document.** When a rule changes, change `MAP_RULES.md` first, then the code, then mark the code section `[built]`.
6. **Validation rejects, never repairs.** Corrupt maps fail loudly.
7. **The map is facts, not rules.** Never encode walkability or behavior verdicts into map data; the rules layer owns all judgments.

8. **Simulation is deterministic.** No engine RNG (`randi`, `randomize`) in any simulation path — use `DeterministicRng`, seeded from the match. Sim math stays in integers and exact fractions.
9. **Rule data is versioned.** `RulesetVersion` hashes the catalog + material rules + movement rules; a match pins it, and clients that don't match refuse to join.
10. **State changes are commits.** Every mutation — including mid-combat map edits — flows through the sequence-numbered commit log. Nothing mutates simulation state out-of-band.

Revision log: created as the foundation spec. Supersedes informal conventions discussed before it existed. Added the facts-vs-rules principle (§2.8, §14.7) and built the rules layer (`MaterialRules`/`MovementRules`/`UnitProfile`) plus the pathfinding pipeline (`PathEvaluator`/`Pathfinder`) per §8. Added the determinism contract (§14.8-14.10), the deterministic RNG, ruleset versioning, and the commit log (multiplayer foundation); combat is specified in `COMBAT.md`.

Revision #10
Created 2026-08-15 18:17:04 UTC by Fifthdread
Updated 2026-08-15 19:00:03 UTC by Fifthdread