Operation Tactics

Game bible for operation_tactics — an FFT-like tactics RPG built in Godot. Vision, systems, pipelines, and baselines, synced from the repo (tools/sync_wiki.py).

- Game Bible
  - Part I — The Game
  - Part II — Systems
  - Part III — Pipelines & Workflows
  - Part IV — Baselines (rules that never break)
  - Part V — Glossary
- Reference
  - Map Rules
  - Project Context
- Combat
  - Combat

Game Bible

Part I — The Game

This is the high-level reference for **Operation Tactics** (codename `operation_tactics`, real name TBD) — a Final Fantasy Tactics–style tactical RPG built in Godot.

Think of this as the game's bible: it describes the whole game and how its systems and pipelines work, in plain language. It is the starting point for anyone (human or agent) working on the project.

How this relates to the other docs:

Doc	Purpose
This bible	The big picture: vision, every system, every pipeline, the baselines. Read this first.
`MAP_RULES.md`	The authoritative, precise spec of the world's data model and rules. This bible summarizes it; MAP_RULES wins on any conflict.
`CONTEXT.md`	The operational handbook: current state, engineering standards, architecture quick-reference, shipping. Agents read it before every task.
`AGENTS.md`	Identity card: one-paragraph purpose and the hard rules.

The bible is mirrored to the wiki at `wiki.fifthdread.com/books/operation-tactics` by `tools/sync_wiki.py`. The repo version is the source of truth.

Vision & pillars

Operation Tactics is a **grid-based, turn-based tactical RPG**: you command a party of units on a battlefield that has real height, and every move and action is a decision.

The design pillars:

1. **Decisions, not reflexes.** Every turn is a puzzle — where to stand, which way to face, who to hit. Terrain, height, and hazard all matter.
2. **The map is honest.** What you see is what the data says. If a tile is lava, the rules say it is lava; nothing hides it. Players can trust the battlefield.
3. **Grow through doing.** Units learn and improve through what they actually do in combat (classic job-point style), not just through leveling up.
4. **One world, one set of rules.** The player, the enemies, and the AI all play by the same rules. The map stores facts; everyone's rules derive from the same facts.

5. **Deterministic by construction.** The simulation is a pure function of the shared facts and the rules. A match cannot de-sync by design — which is what makes multiplayer a natural consequence rather than an add-on.

Core loop

A battle plays out in a rhythm:

1. **Read the field.** The camera sits above the battlefield; you see elevation, hazards, and where everyone stands.
2. **Pick a unit.** Each unit gets a turn in a rotating order.
3. **Move.** The game shows a **movement bubble** — every tile the unit can reach within its move budget, considering terrain cost and height. You pick a tile.
4. **Act.** Attack, cast, use an item, or wait. Range and line-of-sight are computed from where the unit ended up.
5. **End turn.** Effects tick (hazards, statuses, day/night), the next unit goes, and the field is read again.

Beyond battles: a story-driven campaign, branching characters, and between-battle progression (shops, loadouts, jobs). Those systems are planned; the battle mechanics below are the growing core.

Player experience

- **Camera:** an isometric-style tactical camera. Hold to pan, wheel to zoom, rotate 45° at a time. F1 toggles between two views of the same battlefield.
- **Two views of the same truth: DEBUG** shows the map data literally — every block at its exact size, shape, and position, flat colors. **RENDER** shows the game look — the **exact same geometry** (slopes stay slopes, holes stay holes) with seeded per-cell tint, rocks and foliage, lighting, day/night. Both are derived from the same data; RENDER only adds cosmetics, never geometry.
- **Feedback without lies:** the top grid, hover highlighting, and selection all read from the data, so what you point at is exactly what exists.

Part II — Systems

The world & how it's stored

What it is. The battlefield is a 3D grid of small cells called **voxels**. A voxel is either empty or holds one **block**: a material (grass, lava, water, wall…), a **shape** (full cube, half block, slab, slope, wedge…), a rotation, and later a fractional height. Terrain is just columns of blocks rising from the ground; a building is blocks stacked on top. There is no separate "terrain" structure — one grid holds everything.

The facts rule. The map stores only *facts*: what material is where, in what shape and orientation. It never stores judgments like "walkable" or "costs 5." Those live in the rules layer (below). This is the single most important idea in the project: **the map is what is; rules are how it behaves.**

How it's stored. - The grid is split into 16×16×16 **chunks**, so a map can be small (a demo) or effectively endless (an open world) with the same code. - Files are saved by a **codec** that packs the grid into a compact binary format (`.vmap`), versioned so old files fail loudly rather than load wrong. - Materials are catalogued in a **registry** (currently built in code, one day an on-disk catalog). Maps reference materials by index, so the catalog is **append-only** — entries are never reordered or removed.

Where it lives. `src/map/` (`voxel_map.gd`, `voxel_codec.gd`, `tile_type.gd`, `tile_type_registry.gd`, the demo/test registries), `assets/maps/` for map files, `tools/` for the builders that create them.

Status: `[built]` — storage, codec, catalog, and demo/hills/test maps. The per-voxel fractional `height` byte is `[planned]` (schema 4).

The rules layer

What it is. A separate area of the code that decides how the world *treats* each material. It turns map facts into gameplay answers: can a unit stand here, what does it cost to move here, does standing here hurt, what statuses does it apply?

How it works. Three rule modules compose together:

1. **Material rules** — one entry per material: passable or not, movement cost, hazard (lava, water, poison, fire, spikes) and its damage, status effects applied on contact, and behavioral flags.
2. **Movement rules** — the tunable thresholds for climbing: how high a step is free (0.25), the ceiling a normal unit can climb (1.0), how much extra a climb costs, and the surcharge for wading without swim.

3. **Unit mobility** — each unit's move budget, plus traits (float, fly, swim) and status effects (slow, stop, immobilize) that change how the rules apply to that unit.

Rules are data (editor-editable Godot resources) — the intent is that they can be tuned later from a debug/config screen without touching code.

Where it lives. `src/rules/` (`material_rules.gd`, `terrain_rule.gd`, `movement_rules.gd`, `unit_profile.gd`).

Status: `[built]`.

The pathfinding pipeline

What it is. The engine that answers "how does this unit get there?" — and, for the player, "where *can* this unit go?" It is called every time a player (or an NPC) wants to move.

How it works. Four steps:

1. **Read the field.** From the map it computes a **surface** for every column — the exact height a unit would stand on. The map supplies only facts: heights, materials, shapes.
2. **Compose the verdict.** For every candidate step, it asks the rules layer: is this step passable, what does it cost, is there a hazard? The rules consider the material, the height difference (a rise costs extra; one above the ceiling is blocked), and the unit's traits/statuses (fly ignores terrain, float ignores height and ground effects, swim skips the wading surcharge).
3. **Search.** A **budgeted search** fans out from the unit's position, tracking how many move points each tile costs. It stops when it has spent the unit's entire move budget. This single pass produces the **movement bubble** (every reachable tile) — and, given a destination, the **lowest-cost path** to it.
4. **Report.** The result is the ordered list of tiles to walk, the total cost, and **every hazardous tile crossed** — so the movement system can apply the damage and status effects as the unit actually walks through them.

The same search serves the player (bubble + path) and the AI (which uses the bubble to pick where to move). It is fully deterministic and runs headless — the whole pipeline is unit-tested.

Where it lives. `src/pathfinding/` (`path_evaluator.gd`, `pathfinder.gd`, `path_result.gd`).

Status: `[built]` — flat steps, climbing, walls, water, hazards, flight/float/swim, status effects, movement bubble, hazard report. Jumping is `[planned]` (the rule supports a ceiling extension).

Multiplayer & determinism

What it is. The goal that shapes the whole architecture: the game is a **deterministic simulation** — given the same facts and the same rules, every client produces the same outcomes on its own. Multiplayer doesn't require syncing the battlefield every frame; it only requires syncing *what*

changed, and letting every client derive the rest.

Why the architecture is built for it. Because the map stores only facts, the sync surface is tiny. Because the rules derive behavior, every client computes the same move costs, the same movement bubble, the same damage from the same facts — nothing behavioral is transmitted. Because cosmetics are seed-driven, every screen looks the same too.

How a match stays in sync (the determinism contract):

1. **Pinned rules.** A match carries a **ruleset version** — a deterministic hash of the catalog, the material rules, and the movement rules. A client whose ruleset doesn't match refuses to join; otherwise it would derive different outcomes from the same facts.
2. **Ordered commands.** Every state change — including every map edit — flows through the **commit log**: one canonical, append-only, sequence-numbered stream. A host arbitrates order; every client applies commits in the exact same order. A gap or reorder in the sequence is a de-sync by definition, and it is caught the moment it arrives.
3. **Deterministic randomness.** All gameplay randomness comes from a single **seeded PRNG**, driven by the match's simulation seed and the commit sequence. The engine's own random number generator is never used in a simulation path.
4. **Sim-safe math.** Gameplay arithmetic stays in integers and exact fractions (move costs, budgets, half-steps), so every platform computes the identical result.
5. **Snapshot & reconnect.** A match can be captured as a snapshot (ruleset version, seeds, map, units, RNG position, turn state). A reconnecting client loads the snapshot and rejoins the commit stream — no state divergence.

The commit log is the game's spine: player moves, map edits, and (later) combat all record their decisions there; the simulation is the pure function that replays them.

Where it lives. `src/rules/deterministic_rng.gd`, `src/rules/ruleset_version.gd`, `src/net/commit.gd`, `src/net/commit_log.gd`.

Status: `[foundation built]` — deterministic RNG, ruleset versioning, and the commit-log shape exist. The snapshot codec and the host/peer protocol are `[planned]`.

Movement & turn flow *(planned)*

Once a path is chosen, the **movement phase** walks the unit along it, applying the reported hazards as it goes, then consumes the turn. The exact order — move → act → end turn — and the rules for facing, disengage, and counterplay are still being designed. The pathfinding pipeline is the substrate this is built on.

Combat

What it is. The battle system: turn order, movement, actions, damage, elements, statuses — and a battlefield that changes as you fight. The authoritative, precise spec is `COMBAT.md`; this is the short version.

How it works. Turn order runs on a deterministic **CT clock**: every tick, each unit's Charge Time grows by its Speed, and the unit with the highest CT over the threshold acts (overflow carries). A turn is the FFT shape: pick a destination from the movement bubble → face → choose an action → pick a target → confirm. Actions resolve range, line-of-sight, and height; damage runs the documented formulas through per-unit **element affinities**; statuses and terrain apply their effects on tick-based durations.

The map is a **living battlefield**. Blocks can be destroyed (materials carry integrity) or placed, field events are scheduled on the clock (rising lava, spreading fire, freezing water), and unsupported blocks collapse in a deterministic cascade. Every one of those mutations is a **commit** in the match's commit log — so the battlefield, the turns, and every derived outcome stay in sync across clients (see Multiplayer & determinism).

Where it lives. `COMBAT.md` (spec); `src/rules/elements.gd` + `src/rules/unit_profile.gd` (built); `src/combat/` (planned).

Status: `[designed]` — the spec is documented and the element/stat/commit primitives are built; the turn controller, damage math, targeting, field events, and collapse are `[planned]`.

Units, jobs & abilities *(planned)*

Units have stats and a move budget. Jobs/classes unlock abilities learned by doing. The mobility traits and statuses already exist in `UnitProfile` and will be joined by combat stats and an ability catalog.

Items & economy *(planned)*

Equipment, consumables (including the statuses like float/swim that the rules already understand), shops, and job-point progression between battles.

AI *(planned)*

Enemy AI will consume the **same reachability and cost model** the player uses — evaluating its movement bubble, scoring candidate tiles (threat, hazards, height advantage), then pathfinding. Because everything reads the same rules, the AI cannot "cheat" the terrain in ways the player can't.

Rendering & presentation

What it is. Everything visual. It is strictly a consumer: it reads map data and rules and *never* feeds back into them.

How it works. - The **voxel mesh** builds each block's shape as triangles and drops interior faces hidden by neighbors (so a solid hill doesn't render its inner walls). - The **render view** uses the exact same voxel shape geometry as the debug view (each block's true shape, face-culled interior boundaries) — only colors differ: a **seeded per-cell tint** adds gentle shade variation, and a

cosmetic layer scatters rocks and foliage on solid, hazard-free ground. Both are pure functions of the map's seed, so the same map always looks the same. The **day/night cycle** shades it all (sun, moon, clouds, stars). - The **theme** maps each material to a color (later, scenes/textures), keeping visuals fully separate from logic. - The **top grid** is a shader overlay drawn on the terrain: grid lines, hover highlight, and selection — used by the player and by debugging. - **Day/night** is astronomical: daylight length is computed from the configured latitude and month (Virginia in August ≈ 13.5h of sun, December ≈ 9.5h), the sun only lights the scene above the horizon, a separate moon follows its own phase-driven arc (brightness from the illuminated fraction, dimmed by cloud cover), and clear nights add faint starlight.

Where it lives. `src/render/` (`voxel_mesh.gd`, `terrain_mesh.gd`, `shape_geometry.gd`, `map_theme.gd`, `map_details.gd`, `map_view.gd`, `day_night.gd`, `top_grid.gdshader`).

Status: `[built]` — box + smooth views, cosmetic layer, theme, top grid, day/night, F1 toggle. Voxel-side picking/hover is `[partial]`.

Part III — Pipelines & Workflows

The data pipeline

The path every piece of battlefield information takes:

```
map files (facts)  ──►  rules layer (judgments)  ──►  consumers (pathfinding, combat, AI, rendering)
```

1. **Maps are facts.** A `.vmap` file (or a generator) holds voxels: material, shape, rotation, height.
2. **The registry names the materials** (append-only). The codec checks catalog versions so old maps can't silently load wrong.
3. **The rules layer interprets.** Consumers never read "walkable" from the map — they read facts and ask the rules.
4. **Consumers decide.** Pathfinding, and later combat and AI, produce verdicts (reachable, cost, damage) that are *reported*, never written back into the map.

The build & test pipeline

The guardrail that keeps the game trustworthy:

1. `./run_tests.sh` imports the project (so new classes register), then runs the whole headless test suite.
2. The suite covers the codec round-trips (byte-exact), the append-only catalog contract, and every subsystem's core behavior — including all of the pathfinding pipeline.
3. A **pre-commit hook** runs the suite; a commit that turns the suite red is rejected.
4. Tests are small, focused files under `tests/`, auto-discovered by name.

Rule: no red commits, ever. When behavior changes, the tests change with it.

The content & asset pipeline

How maps and content get made:

- **Builders** (`tools/*.gd`, run from Godot) construct demo maps, the test "museum" map, and the voxel demos programmatically, then pack them through the same codec any

game would use.
- **The test map** is the scenario museum — every hazard, elevation shape, layered structure, and wall the model can express — and its tests assert the map's shape so builders can't drift.
- Future **procedural generation** and the **in-game editor** are just other producers of the same map data, flowing through the same codec and renderer.

The docs & wiki pipeline

The documentation is versioned in the repo and mirrored to the wiki:

1. **Canonical docs live in the repo:** this bible, `MAP_RULES.md`, `CONTEXT.md`.
2. **`tools/sync_wiki.py`** reads a manifest (`tools/wiki_manifest.json`) and pushes each doc (and each bible part) to a BookStack book as chapters and pages — creating or updating in place.
3. The wiki is a **rendered view**; the repo is the source of truth. When the game changes, the docs change first (rule below) and the sync re-mirrors them.

The ship workflow

- Branch `main`; commits are concise, lowercase, imperative, focused on "why"; push frequently to the Forgejo remote.
- Repo uses the **SHA-256 object format** — never recreate it as sha1, never rely on push-to-create.
- When the game is ready to distribute on Arch Linux, a PKGBUILD makes it installable with `paru` (see `CONTEXT.md`).

Part IV — Baselines (rules that never break)

These are the invariants. If code, docs, or habits contradict them, the baseline wins.

1. **The map stores facts; rules decide.** No walkability, cost, hazard, or LoS verdict ever lives in map data.
2. **Render never feeds back.** No rule or map datum may depend on what the render layer does.
3. **No derived storage.** Anything computable from the map (surfaces, adjacency) is computed or cached, never serialized.
4. **Append-only catalog.** Materials are added, never reordered or removed; maps reference them by index.
5. **Versioned formats, loud failure.** Every binary format carries a schema version; loaders reject mismatches, never silently repair.
6. **Deterministic.** Everything derived from `(map, seed)` reproduces identically given the same inputs.
7. **One tunable movement module.** The climb thresholds live in one place, not scattered constants.
8. **Layered, one-way.** Data and logic never import rendering; scripts stay focused; no circular dependencies.
9. **Tests gate every change.** `./run_tests.sh` must pass before anything lands.
10. **Docs change first.** When a rule changes: update this bible / `MAP_RULES.md` first, then the code, then the wiki mirror.
11. **Deterministic simulation.** No engine RNG in simulation paths — gameplay randomness comes from `DeterministicRng`, seeded per match. Sim math stays in integers and exact fractions.
12. **Rules are versioned.** Rule data carries a `RulesetVersion`; a match pins it, and clients that don't match refuse to join.
13. **State changes are commits.** Every mutation — including map edits — flows through the append-only, sequence-numbered commit log. No out-of-band mutation.

Part V — Glossary

Term	Meaning (plain)
Voxel	A single cell in the 3D grid — empty, or holding one block.
Block	A voxel that exists: a material + shape + rotation (+ height, planned).
Chunk	A 16×16×16 group of voxels; maps are made of chunks.
Column	The stack of voxels at one grid position; terrain is columns.
Surface	A place a unit can stand, with an exact height.
Material	What a block is made of (grass, lava, water, wall…), named in the catalog.
Shape	The geometry a block occupies in its cell (full, half, slab, slope, wedge…).
Height	The fractional amount of a cell a block fills (planned per-voxel).
Registry / catalog	The append-only list of materials, referenced by index.
Codec	The code that packs/unpacks map files (`.vmap`).
Schema	The version number of the file format.
Facts	What the map stores: material, shape, rotation, height.
Rules layer	The modules that judge facts: passability, cost, hazards, mobility.
Move budget	The move points a unit has; each tile costs some.
Movement bubble	Every tile reachable within the move budget (FFT's movement range).
Hazard	A material that damages or affects a unit standing on it (lava, poison…).
Trait	A unit property that changes movement rules (float, fly, swim).
Status	A temporary effect on a unit (slow, stop, immobilize, wet…).
RAW view	The literal-data view of the map; the truth view.
RENDER view	The game-look view (smooth, lit, cosmetic).
Path	The ordered list of tiles a unit walks to reach a destination.

Term	Meaning (plain)
Affinity	A unit's damage multiplier for an element (0 immune, 2 weak, negative absorbs).
Charge Time (CT)	The clock that schedules turns: a unit acts when its CT reaches 100.
Commit	One ordered state change in the match log — a move, an action, a map edit.
Commit log	The append-only, sequence-numbered stream every client replays.
Element	One of the eight damage types (fire, ice, lightning, water, wind, earth, holy, dark).
Line of sight	Whether a straight line between two units is unblocked by solid terrain.
Ruleset version	The hash of rule data a match pins; clients that differ refuse to join.
Simulation seed	The seed driving the match's deterministic RNG.
Tick	One step of the global clock; CT and scheduled events advance per tick.
Deterministic RNG	The seeded PRNG used for all gameplay randomness.
Snapshot	A full capture of a match (ruleset, seeds, map, units, RNG, turn state) for reconnect.

Mirrored to the wiki by `tools/sync_wiki.py`. Canonical source: this file.

Reference

Reference

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`

shape	volume
`SLAB`	top slab of `height`
`SLOPE`	full-length ramp (+x)
`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`.

Reference

Project Context

Project knowledge. Agent-facing; read this before any work. Not the doc for humans.

Overview

A Final Fantasy Tactics-like tactical RPG (grid-based, turn-based combat with height, movement, and class/job systems) built in Godot 4.x with GDScript. **Codename:** `operation_tactics` — the real name is TBD. Nothing about the codename should leak into user-facing strings.

Goals

- Faithful FFT-style tactical combat: isometric(ish) grid, move/act/end-turn flow, terrain height, facing, and line-of-sight.
- Job/class system with learning abilities from action (JP), not just leveling.
- Branching/unique characters, story-driven campaign.
- One cohesive game, not a tech demo — a single unified codebase.

Current State

- Engine files exist (project.godot, directory skeleton). Map data core (registry, schema, binary codec), the 3D heightfield renderer (`MapView`/`MapTheme`), the seeded cosmetic layer (`MapDetails`), a demo map asset, and the custom test pipeline are in place. `main.tscn` boots the demo map with an FFT-style isometric camera (WASD pan / QE rotate / wheel zoom) and an **astronomical day-night cycle** (`src/render/day_night.gd`): daylight length computed from latitude + month (Virginia in August ≈ 13.5h), a sun that only lights above the horizon, a phase-driven moon with cloud-dimming, and clear-sky starlight. Tune `latitude_deg`/`month`/`day`/`cloud_cover`/`moon_phase`/`cycle_seconds` on the Sun node. **F2** toggles a free-fly spectator camera (`src/camera/free_cam.gd` — right-mouse look, WASD/QE move, wheel = speed, F3 recenters); it never touches the FFT camera's state, so toggling back is seamless.
- **Rules layer + pathfinding pipeline built** — facts-vs-rules split is done (`TileType` = facts; `MaterialRules`/`MovementRules`/`UnitProfile` = judgments) with a budgeted Dijkstra pathfinder (`Pathfinder`/`PathEvaluator`) providing FFT movement-bubble reachability, lowest-cost paths, hazard reports, and fly/float/swim/status support. See "Rules & Pathfinding pipeline".

- **Combat designed; multiplayer foundation built** — `COMBAT.md` is the authoritative battle spec (deterministic CT turn order, streamlined stats, the eight elements, living-map mutations as commits). Foundation primitives exist: `Elements`, `DeterministicRng`, `RulesetVersion`, `Commit`/`CommitLog` (with the commit vocabulary), and streamlined combat stats + affinities on `UnitProfile`. See "Combat & multiplayer foundation".
- **Procedural arena** — `voxel_arena.vmap` is the seeded showcase/test map: rolling noise terrain + stamped features covering all 10 materials, all 5 shapes, rotations 0-3, hazards, real void, multi-level climbing, and a pathable plaza. Generated by `src/map/arena_generator.gd` (`ArenaGenerator`), written by `tools/build_voxel_arena.tscn`, validated by `tests/map/voxel_arena_test.gd` (determinism + asset-in-sync + feature coverage). Now the default map in `main.gd`. Test suite is 117 green.
- Repository created on Forgejo and pushed. Repo uses the **SHA-256 object format** — never recreate it as sha1, and never rely on push-to-create for this repo (Forgejo's push-to-create ignores object format and would make a sha1 repo).

Engineering Standards

Applies to every agent and human working here. Goal: a codebase humans can maintain long-term, not one that merely "works."

Code quality

- **Human-first.** Idiomatic GDScript, clear names, logical file layout. Readability first, efficiency second — but never accept needless waste (hot paths: avoid per-frame allocations).
- **Concise.** Smallest correct change. No speculative generality, no backward-compat shims without a concrete need.
- **Reuse over copy.** Shared frameworks (tile registry, map codec; later combat math, pathfinding) live once and are used everywhere. Rule of three: third copy → extract.
- **Modular & layered.** Scripts stay focused (one responsibility); layers point one way — data/logic must not import rendering; cross-cutting concerns use signals. No spaghetti, no circular references.
- **Debuggable.** Keep logic in deterministic functions (pure input → output) where practical; avoid hidden global mutation.

Comment policy

- Comment the WHY and the intent: design decisions, non-obvious data formats, invariants, cross-module contracts. A reviewer should understand your intent.
- Never narrate WHAT the code does; no AI-style filler (e.g. `# get the player position`). No emoji. Code should look hand-written.

Testing (mandatory, Factorio-style)

- Before every commit: `./run_tests.sh` runs the full suite headless and must pass; a pre-commit hook enforces it. Never commit red.
- Suite covers: codec round-trip (byte-exact), registry append-only guard, and every subsystem's core behavior as it lands.
- Benchmark mode (`--bench`, planned) for perf-sensitive paths (combat sim, map load); checked for regressions, not just pass/fail.
- Test layout: `res://tests/**/*_test.gd`, custom runner (see `res://tests/run_tests.gd`). Tests extend the `TestCase` base and use its `eq/is_true/...` helpers.

Growing-project guardrails

- **Format/version drift** → schema version in every binary format; loaders reject mismatches loudly, never silently.
- **Duplication creep** → rule of three + review.
- **Scope creep** → anything new must fit the modular layout or the plan changes (and this doc gets updated).
- **Dead code** → no "just in case" branches; keep the tree lean.
- **Conventions drift** → this doc is the arbiter; update it when rules change.
- **Single-file bloat** → split when a script exceeds ~300 lines or has more than one job.

Map Data Architecture

`MAP_RULES.md` is the authoritative world-model spec — read it before touching anything map-related. It defines the voxel model, tile types, shapes, terrain/blocks, movement thresholds, fluids, dynamic edits, generation, validation, and the agent rules. This section is the quick-reference; MAP_RULES.md wins on any conflict.

Direction: voxel pivot. **`VoxelMap`** (`src/map/voxel_map.gd`): a chunked 16^3 grid of 1×1×1 voxels (open-world ready; columns are subsumed as vertical runs of solid voxels). Each voxel is `{type, shape, rot}` (2 bytes packed; `TileType.Shape` = FULL/HALF/SLAB/SLOPE/WEDGE…, per-voxel shape + rotation for the editor). Serialized by **`VoxelCodec`** (`voxel_codec.gd`, schema 3, `.vmap` files). **`VoxelMesh`** (`src/render/voxel_mesh.gd`) builds the combined shape mesh with face culling (interior faces shared with a full solid neighbor are dropped); **`VoxelView`** renders it. `main.gd` defaults to the voxel demo (`assets/maps/voxel_demo.vmap`, built by `tools/build_voxel_demo.tscn`) and loads `.vmap` as voxels vs `.otmap` as the legacy column model (still present until the column code is retired). Cosmetic layer, hover/select and picking are column-only for now.

Global design (see `res://src/map/` for implementation). Maps are **2D grids of columns**: each cell owns its entire z column, so interior-under-roof and bridges are expressible as stacked walkable surfaces.

- **`TileType`** (`tile_type.gd`) — one shared entry per tile kind: **material facts only** (`id`, `block_height`, `shape`, `solid`, `fluid`). Behavior (walkability, cost, hazards, flags) lives in the rules layer (`MaterialRules`), never here. **Logic only** — visuals live in a separate theme table (`type_id → scene/texture`).

- **`TileTypeRegistry`** (`tile_type_registry.gd`) — the **global catalog**. **Append-only**: never reorder/delete entries, maps reference types by index, so adding types is always safe. `version` guards against breaking changes; loaders reject `catalog_version > registry.version`.
- **`MapData`** (`map_data.gd`) — size, catalog_version, spawn points, and a column per cell: `{ elevation, ground (type index), layers: [{z, type, rot}] }`.
- **`MapCodec`** (`map_codec.gd`) — `StreamPeerBuffer` (little-endian) ↔ `PackedByteArray`. Binary layout is the contract (see file header): magic `OTMP` + schema version + catalog version + dims + spawns + per-cell: flags u8, elevation u16, ground u16, optional layer list. Map files use the **`.otmap`** extension.
- **`MapPicker`** (`src/map/map_picker.gd`) — resolves a cursor ray to the **specific block** hit (cell + surface height), not just the column: DDA over the grid + ray-vs-AABB on every block. View-aware (RENDER fills/ thin layers vs RAW data heights). `main.gd` feeds `hover`/ `selected` cell+height into the top-grid shader, so a click on a roof highlights the roof, a click on a cliff face highlights that block.
- **`MapTheme`** (`src/render/map_theme.gd`) — visual theme, `type_id → color` (later scene/texture). Kept out of `TileType` so map data stays renderer-agnostic; swapping the theme restyles the whole game.
- **`MapView`** (`src/render/map_view.gd`) — the single data→rendering bridge: one combined `ArrayMesh`. Two view modes (`MapView.ViewMode`, toggled with **F1** in `main.gd`): **RENDER** = the game look (terrain fills to the map base — the **bedrock rule**, no terrain voids; layers drawn thin; cosmetic layer on; no wireframe) and **RAW** = the map data literally (ground blocks at `[elevation, elevation + block_height]` with voids intact, layers at full data height, flat colors + wireframe, no cosmetics) — the geometry rules/pathfinding are written against the RAW view. Renderer direction is **3D heightfield**. The RENDER top grid is a **fragment shader** (`src/render/top_grid.gdshader`) on a duplicate of the terrain mesh: + crosses at lattice corners + dashed edges, uniform and anti-aliased, tinted per `hover_cell`/`selected_cell` uniforms. `main.gd` picks the cell under the cursor (hover on mouse move, blue select on left-click) and feeds the uniforms.
- **`MapDetails`** (`src/render/map_details.gd`) — the **cosmetic layer** (two-layer model): a pure deterministic function of `(map, seed)` producing per-cell tint variance, C0-continuous surface jitter (no seams), and scattered props (rocks/bushes/tufts via `MultiMesh`). Purely visual — never feeds back into map data or rules.
- **Two-layer model + seed contract** — `MapData.seed` is part of the map (codec field, schema 2). Ground truth is the column blocks (drives rules/pathfinding later); the cosmetic layer is derived from `(map, seed)` and regenerated freely — same seed, same result, everywhere. Determinism rules: integer-hash noise keyed by `(seed, x, y, z)`, fixed iteration order, no engine RNG. The seed can later drive other derived content (spawns, variants).
- **`DemoRegistry`** (`src/map/demo_registry.gd`) — canonical demo catalog; stand-in for a future on-disk catalog file (append-only by index). The demo map asset (`res://assets/maps/demo.otmap`, seed 12345) is generated by `tools/build_demo_map.tscn`.
- **`TestRegistry`** (`src/map/test_registry.gd`) + **test map** (`assets/maps/test_map.otmap`, 53×27, seed 20260815) — the scenario museum: every hazard, elevation shape, layered structure, wall/corridor, rotation, and spawn surface the model can express, laid out in 8×6 zones (see `tools/build_test_map.gd` — its coordinates are the contract;

`tests/map/test_map_test.gd` spot-checks each zone). View it with `godot --path . -- --map=res://assets/maps/test_map.otmap`.
- **Procedural maps** — a generator is just another producer of `MapData` (same shape as the demo builder); it flows through the same codec and renderer. Generators must produce valid `MapData` (indices within the registry, in-bounds elevations/layers).
- **In-game map editor** (planned) — mutates in-memory `MapData` and re-packs through the same codec; palette is the registry. Editor stays an orthogonal consumer of the schema.

Rules & Pathfinding pipeline

- **`MaterialRules`** (`src/rules/material_rules.gd`) — the rules layer: how the world *treats* each material. One `TerrainRule` per type id: `passable`, `move_cost`, `hazard`/`hazard_damage`, behavioral `flags`, `status_on_enter`. `for_registry()` yields the rule set for a catalog (defaults + per-id overrides — the single behavioral source for demo/test ids). All editor-editable as Resources.
- **`MovementRules`** (`src/rules/movement_rules.gd`) — the single tunable module for MAP_RULES §8.3 thresholds: `free_step` (0.25), `step_ceiling` (1.0), `climb_cost_per_step`, `water_cost_extra`.
- **`UnitProfile`** (`src/rules/unit_profile.gd`) — unit mobility: `move_budget`, traits (FLOAT/FLY/SWIM), status effects (slow/stop/immobilize modify the budget; float/swim statuses grant traits).
- **`PathEvaluator`** (`src/pathfinding/path_evaluator.gd`) — composes map facts (column surfaces from voxel shape geometry) × rules × unit state into per-step verdicts (`step_cost`, `hazard_at`).
- **`Pathfinder`** (`src/pathfinding/pathfinder.gd`) — budgeted 4-directional Dijkstra over the surface graph. `find_path()` → lowest-cost path to a destination; `reachability()` → the FFT movement bubble (cost field + parents). `PathResult` carries the ordered path, total cost, and every hazardous cell crossed (movement/combat applies the rule's damage/status as the unit traverses it).
- Pathfinding operates on `(x, y, surface)` nodes derived from the voxel grid; inter-surface transitions (stairs, ledges) are edge costs per MAP_RULES §8.3. Surface heights come from shape geometry until the schema-4 `height` byte lands.

Combat & multiplayer foundation

`COMBAT.md` is the authoritative battle spec — read it before touching combat. It locks the FFT baseline (turn order, turn flow, occupancy, targeting) and defines our divergences: deterministic CT scheduling, the eight-element system with affinities, and the living map (destruction/placement/field events/collapse, all as commits).

- **`Elements`** (`src/rules/elements.gd`) — the eight elements (fire/ice/lightning/water/wind/earth/holy/dark) + affinity helpers (`effective_multiplier`, `standing_multiplier`).
- **`UnitProfile`** (`src/rules/unit_profile.gd`) — now carries the streamlined stats (hp/mp/atk/def/mag/mdf/spd) and per-element `affinities` alongside mobility.

- `DeterministicRng` (`src/rules/deterministic_rng.gd`) — SplitMix64-seeded PRNG; the ONLY randomness source allowed in simulation paths (COMBAT.md §7.3, MAP_RULES §14.8).
- `RulesetVersion` (`src/rules/ruleset_version.gd`) — FNV-1a hash of catalog + material rules + movement rules; a match pins it, mismatched clients refuse to join.
- `Commit` / `CommitLog` (`src/net/`) — the append-only, sequence-numbered state-change stream (the shared clock of a match). The kind vocabulary (move_unit, set_block, remove_block, apply_damage, ...) is COMBAT.md §10.1, reflected as `Commit.KIND_*` constants.
- Turn controller, damage math, targeting, field effects, collapse, and the snapshot codec: `[planned]` (specified in COMBAT.md).

Conventions

- Godot 4.x GDScript; follow the `godot4` skill (theme/font override quirks, ConfigFile gotchas, explicit type annotations — type warnings treated as errors). Comments explain intent, not narration (see Engineering Standards).
- snake_case for scripts/vars, PascalCase for node names.
- Branch `main`; commits concise, imperative mood, lowercase, focused on "why"; push frequently.

Docs & Wiki

- **Canonical docs live in the repo**: `GAME_BIBLE.md` (the game bible — vision, systems, pipelines, baselines, glossary; read it first), `MAP_RULES.md` (authoritative world-model spec), `COMBAT.md` (authoritative battle spec), and this file.
- **Mirrored to the wiki** (`https://wiki.fifthdread.com/books/operation-tactics`) by `tools/sync_wiki.py` — the wiki is a rendered view, the repo is the source of truth.
- Sync (idempotent; creates missing pages, updates existing): `source ~/opencode/api_secrets.sh && python3 tools/sync_wiki.py`. `--check` previews without writing. Which docs map to which pages is declared in `tools/wiki_manifest.json`.

Shipping

Shipping = **commit and push** to the project repo.

- Remote: Forgejo — `ssh://git@forgejo.fifthdread.com:223/Fifthdread/operation_tactics.git`. **SHA-256 object format** — the repo must be pre-created on the server as sha256; push-to-create will not work for this repo.
- If the game is ready for distribution on Arch-based Linux: also update the **PKGBUILD** so `paru -S <pkg>` installs it.
- PKGBUILD lives in the `Fifthdread/pkgbuilds` repo at `~/opencode/pkgbuilds/` (one dir per package, `-git` variant recommended for an in-development game).

- Workflow: bump `pkgver=` (`rev-count.commit`), regenerate `makepkg --printsrcinfo > <pkg>/.SRCINFO`, commit `.SRCINFO` alongside, push. Full detail in the pkgbuilds project's CONTEXT.md/AGENTS.md.

Combat

Combat

This document defines the battle system of **Operation Tactics**: turn order, movement, actions, damage, elements, statuses, and a battlefield that changes as the fight goes on. It is the authoritative spec — `GAME_BIBLE.md` summarizes it, and nothing here is superseded except by a newer version of this file.

What is assumed (FFT baseline). Turn order, the per-turn flow, option/tile selection, pathfinding, and units standing on tiles all behave like Final Fantasy Tactics. We state those behaviors here so no one (human or agent) re-invents or "fixes" them.

What is ours. The map is a *living* battlefield (mutation during combat), an element/status system, deterministic CT scheduling, and a fully deterministic simulation pipeline. These have no FFT equivalent and are specified precisely.

---

1. Assumed baseline (FFT-shaped, stated once)

1. One unit per tile; a unit stands on its tile's surface (see MAP_RULES §7).
2. Movement is bounded by a **move budget** and shown as a **movement bubble** (the reachability set — see the pathfinding pipeline).
3. A unit's turn is: choose a destination (or hold) → face a direction → choose an action → pick a target → confirm.
4. The action menu is **Attack / Ability / Item / Wait**. "Wait" ends the turn without acting.
5. Occupied tiles are impassable — you cannot move through or stand on a tile another unit stands on.
6. Targeting considers both horizontal range and vertical height difference.
7. Anything not specified below is expected to follow FFT behavior unless it contradicts these rules.

2. Our divergences (defined here, no FFT equivalent)

Feature	Rule

Turn order	Deterministic CT scheduling (§3), not a fixed round order
The map	Mutable mid-combat; every change is a commit (§9, §10)
Elements	Eight-element system with per-unit affinities and terrain elements (§7)
Statuses	A status catalog with tick-based durations (§8)
Simulation	Fully deterministic: integer math, seeded RNG, commit-ordered (§10)

3. Turn order & the tick clock

The match has one global clock: an integer **tick counter** (`tick`, starting at 0, +1 per simulation tick). Everything scheduled in a battle — turns and world events alike — is keyed to this counter.

- Every tick, every unit's **Charge Time (CT)** increases by its **SPD**: `ct += spd`.
- A unit is **eligible** when `ct >= 100`.
- The **actor** is the eligible unit with the highest CT. Ties are broken by ascending `unit_id` (stable ids assigned at match setup) — deterministic, never random.
- After a unit acts, `ct -= 100`; the **overflow stays** (a unit at CT 137 acts and sits at 37). This is why fast units act more often.
- **Movement does not consume CT.** CT only schedules *when* a unit's turn comes up; the turn itself is free.
- Tick cadence is a presentation concern — the *logic* only cares that ticks are ordered.

`[planned]` — `src/combat/battle.gd` owns the tick counter, CT queue, and actor selection.

4. A unit's turn

When a unit is the actor:

1. **Move** — pick a destination tile inside the movement bubble (or hold position). The path is computed by the pathfinder; entering hazardous tiles applies their effects as the unit walks (§5, §9).
2. **Face** — pick one of the four directions. Facing determines which targets are in an action's arc and enables the back-attack bonus.
3. **Act** — choose an action from the menu (Attack / Ability / Item / Wait).
4. **Target** — if the action targets something, pick a tile or unit that satisfies range + line-of-sight + height reach (§6).
5. **Confirm** — the turn is committed as one ordered set of decisions and resolved (§10).

Waiting is "Act = Wait": the unit ends its turn immediately.

5. Movement & occupancy

- **Bubble:** the set of tiles reachable within the unit's Move budget, computed by the pathfinder's reachability pass. Terrain cost, climb, and unit mobility all apply (see the rules layer).
- **Blocking:** a tile occupied by another unit (friend or foe) is impassable for both movement *through* and *onto*. Ranged actions may still target the occupant.
- **Height:** a unit stands at its tile's surface height (voxel geometry; see MAP_RULES §8). Reach checks use the height difference between attacker and target.
- **Falling:** if the tile under a unit is destroyed or removed (§9), the unit **falls** to the highest remaining surface at or below its position. Fall damage is documented in the tunable combat module (baseline: `max(0, levels_fallen - 1)` × `fall_damage_per_level`). A unit with no surface below is removed from the battle.

`[planned]` — falling resolution; pathfinding and bubble logic are `[built]`.

6. Actions & targeting

An **action** is a data record with the following fields (defined once, referenced by the menu, AI, and damage math):

Field	Meaning
`kind`	attack / ability / item
`power`	base strength fed to the damage formula
`element`	one of the eight elements (or none)
`range`	max horizontal reach in tiles (manhattan distance)
`height_reach`	max absolute height difference (in levels)
`area`	target tile, radius-N blast, or line
`target_type`	enemy / ally / self / tile
`mp_cost`	MP spent (abilities)
`status`	status applied on hit (and its ticks)
`map_effect`	e.g. break block, create block, ignite (§9)

Reach: a target is in range when the manhattan tile distance ≤ `range` **and** the height difference ≤ `height_reach`. Area effects apply to every unit standing in the affected tiles.

Line of sight: single-target actions require an unblocked line between attacker and target. A line is unblocked if no solid block whose rule carries `BLOCKS_LOS` intersects it. Abilities may declare they ignore LOS. `[planned]` — voxel raycast.

Resolving an attack (§7) and **resolving a map effect** (§9) both happen during the action's apply step; neither writes anything except through commits.

7. Stats, damage & elements

7.1 Streamlined stats

Stat	Meaning
`hp` / `max_hp`	hit points
`mp` / `max_mp`	mana for abilities
`atk` / `def`	physical offense / defense
`mag` / `mdf`	magical offense / defense
`spd`	CT per tick (§3)
`move_budget`	movement points per turn (existing)
`jump_height`	max climbable rise (existing; jump ability later)

7.2 Damage baselines

Physical: `damage = max(1, (atk + action.power) - def) × modifiers` Magical: `damage = max(1, (mag + action.power) - mdf) × modifiers`

Modifiers (multiplicative, applied in order, each `[0, 2]`): element affinity (§7.5), height advantage (+`height_bonus_per_level` per level *above* the target), back attack (×`back_multiplier` when the target faces away), variance (`DeterministicRng` ±`variance_pct`). Final damage rounds to an integer.

Hit chance: `hit = accuracy - evasion` (from attacker action/stat minus defender stat), clamped to `[min_hit, max_hit]`; the roll uses `DeterministicRng.chance(hit / 100)`.

Every constant above lives in one tunable combat module (`src/combat/damage.gd`, planned) — the same discipline as `MovementRules`. `[planned]`.

7.3 Determinism

Combat randomness comes **only** from `DeterministicRng`, seeded from the match's simulation seed and fed by the commit sequence. The engine RNG is never used in a simulation path. Damage, hits, variance, and any future rolls are therefore identical on every client.

7.4 The eight elements

`fire, ice, lightning, water, wind, earth, holy, dark` (+ `none`).

There is **no fixed element-vs-element matrix.** An element matters through two mechanisms only:

1. **Affinities** — per unit, per element, a multiplier (below).
2. **Terrain element** — standing on an elemental tile grants resistance to that element (§7.6).

7.5 Affinities

Each unit has an affinity per element: a damage multiplier.

Multiplier	Meaning
`0`	immune
`< 0`	absorbs (heals for that much)
`0.5`	resists
`1`	normal
`2`	weak

An action's `element` is looked up against the target's affinities and the result multiplies damage (floor at 0 for immunity; negative values heal).

7.6 Terrain element & standing resistance

A `TerrainRule` may declare an element (e.g. lava is fire). While a unit stands on an elemental tile, its affinity for that element is treated as `min(affinity, 0.5)` — standing on your element protects you, and the same terrain element in the rules layer drives hazard damage (§9).

7.7 Element × status interactions

Condition	Effect
`wet` + fire element	+50% fire damage taken
`wet` + ice element	applies `freeze` (immobilize)
fire element on flammable terrain	ignites (field event, §9.4)
lightning on `wet`	+50% lightning damage taken

The table is small and explicit on purpose; additions require a documented rule first.

8. Statuses

A **status** is a named effect on a unit with a duration in **ticks** (the global clock), counted from application and decremented every tick. Durations are deterministic and never "until cured"

without a stated expiry.

Mobility statuses (built) — `slow`, `stop`, `immobilize`, `float`, `swim`, plus `wet`, `cold`, `hot` (see `UnitProfile`). `slow`/`stop`/`immobilize` already modify the movement budget; `float`/`swim` grant traits.

Combat statuses (planned) — `poison` (damage per tick), `blind` (accuracy penalty), `silence` (no abilities), `charm` (targets allies), `protect`/`shell` (halve physical/magical damage), `regen` (heal per tick), `freeze` (immobilize), `haste`.

Sources — terrain (hazard reports from the pathfinder), actions, field events. **Application** and **expiry** are commits (§10).

9. The living map

The battlefield is mutable, and **every mutation is a commit** (§10.2). Clients never mutate the map directly — they apply commits in order. This is what keeps a destructible battlefield in sync in multiplayer.

9.1 Destruction

A `TerrainRule` may be `destructible` with an `integrity` (hits to break). Damage against a block reduces integrity; an attack whose element matches the block's **weakness** (documented per material) deals double integrity damage. At zero, the block is removed (`remove_block` commit) and its support rules apply (§9.5). `[planned]` — `TerrainRule` fields.

9.2 Placement

Abilities may create blocks (walls, ice pillars, bridges). The voxel payload (position, type, shape, rotation) is a `set_block` commit. Placement follows the map's placement rules (MAP_RULES §6) — no overwriting solid space without a documented rule. `[planned]`.

9.3 Scheduled field events

The match can carry a schedule: `{tick, effect}` pairs. When the tick counter reaches one, it fires as a `field_event` commit. Examples: lava rising (raising a surface by adding blocks), fire spreading to flammable neighbors, water freezing to ice, a bridge collapsing at a plot beat. The schedule is part of the match setup and is deterministic. `[planned]`.

9.4 Collapse chains

When a block loses its support (the voxel below is removed or empty), gravity applies. Collapse is resolved as a deterministic flood from the removal point: supports are evaluated bottom-up, blocks that lose support fall, and a falling block either lands on a surface (becoming `set_block`) or continues down. Falling blocks and displaced units deal/apply documented impact effects. The

order is fixed by the resolution algorithm — never by UI or timing. `[planned]`.

9.5 Hazards & pathfinding

Hazards (hazard reports) already flow from the rules layer. Any map commit **invalidates the surface graph cache**, so the next movement bubble and the next hazard report reflect the new battlefield. `[built]` — hazard reporting; `[planned]` — commit-driven invalidation wiring.

10. The combat pipeline

Every battle step is a pure function of the commit log:

```
input (player/AI) ──► validate ──► Commit (seq, kind, payload)
   │                                   │
   └──────────── broadcast to all clients ◄┘
                                       │
        each client simulates, in order:
        move (path + hazard effects) → resolve action (range/LOS/affinity/status)
        → apply map mutations → tick statuses → fire due field events
        → CT tick → next actor
```

- **Order is the contract.** Commits are sequence-numbered and append-only (`CommitLog`); a gap or reorder is a de-sync by definition. The host arbitrates order; clients only ever apply.
- **Randomness** comes from `DeterministicRng` only.
- **It runs headless.** The whole pipeline is unit-testable without rendering (the pathfinder already is).

10.1 The commit vocabulary

kind	payload (sketch)
`move_unit`	`{unit_id, path, facing}`
`wait`	`{unit_id}`
`use_action`	`{unit_id, action_id, target_tile, target_unit}`
`set_block`	`{pos, type, shape, rot}`
`remove_block`	`{pos}`
`apply_damage`	`{unit_id, amount, element, source}`
`apply_status`	`{unit_id, status, ticks}`
`field_event`	`{tick, effect}`
`end_match`	`{outcome}`

Payloads are validated by the handler for their kind; the `Commit` class carries them as data.

11. AI

Enemy AI consumes the **same reachability and cost model** as the player (see the bible): it computes its movement bubble, scores candidate tiles (threat, hazard, height, cover), picks the best, then chooses an action whose targets are in range and LOS. The AI has no special rules — only its own scoring.

12. Framework layout

Module	Status
`src/rules/elements.gd` — the eight elements + affinity helpers	`[built]`
`src/rules/unit_profile.gd` — streamlined stats + affinities	`[built]`
`src/net/commit.gd` / `commit_log.gd` — commit + vocabulary	`[built]`
`src/rules/terrain_rule.gd` — element / destructible / integrity / flammable	`[planned]`
`src/combat/battle.gd` — tick clock, CT queue, actor selection	`[planned]`
`src/combat/turn.gd` — turn phases	`[planned]`
`src/combat/action.gd` — action data	`[planned]`
`src/combat/damage.gd` — formulas (single tunable module)	`[planned]`
`src/combat/target.gd` — reach/LOS resolution (uses pathfinder reachability)	`[planned]`
`src/combat/field_effect.gd` — scheduled events	`[planned]`
`src/combat/collapse.gd` — gravity resolution	`[planned]`

13. Agent rules

1. **Combat simulation is deterministic.** Integer CT/ticks, `DeterministicRng` only, no engine RNG in a sim path.
2. **All map mutations are commits.** Nothing mutates the map out-of-band, in combat or out.
3. **Occupancy is a hard rule.** One unit per tile; exceptions require a documented rule.
4. **Formulas live in one module.** Damage/hit/status constants are not scattered once `damage.gd` lands.

5. **Docs change first.** Update this file before changing combat code.

Status: `[designed]` *— the spec is documented; elements and unit stats are built; the turn controller, damage math, targeting, field events, and collapse are* `[planned]`*.*