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.
Affinity	A unit's damage multiplier for an element (0 immune, 2 weak, negative absorbs).

Term	Meaning (plain)
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.