Rogueima: the plan

A step-by-step ordering from Rogueima I MVP-4 toward something with NetWhack's shape. Each step is meant to be finishable on its own, and to leave the game playable when it lands.

For the reasoning behind the grouping, see the roadmap in `programs/rogueima/ROADMAP.md`. This page is the working list.

Part I — movement and the turn

Step 1. Collapse movement into one routine

`move_left`, `move_down`, `move_up` and `move_right` are four near-identical routines, each with its own edge check, its own `is_walkable` call and its own store. Replace them with a single

```
try_move  ; XL = dx, YL = dy
```

that computes the destination, bounds-checks it, asks `is_walkable` and commits. The four existing directions become four callers.

Nothing changes on screen. This is the step that makes the next one free, and it removes three copies of a bug surface.

Step 2. Eight-way movement

Add Y U B N for the diagonals, so the vi keys form the usual 3×3. With `try_move` in place each is two instructions and a call. Keep the arrow keys ($80-$83) mapped to the orthogonals, and consider the numpad digits as NetWhack uses them.

Four-way movement is the most noticeable difference between Rogueima and any other roguelike. This is an afternoon.

Step 3. A real turn loop

Today the player moves and the world does not answer. Establish the order: the player acts, every monster acts, the clock advances. The `T:` counter in the status panel already exists — make a successful action advance it, and give `move_mon` its turn immediately after.

Everything from here assumes monsters get a turn when you take one.

Step 4. Wait

`.` passes a turn. Trivial once Step 3 exists, and it is the only way to watch monster behaviour without

moving, which makes the next several steps testable.

Part II — depth

Step 5. Turn the map into a structure

`map1_data` is literal text — 40 lines of `.bytes "####…"`. That is fine for one static level and blocks everything else. Introduce a tile array with one byte of glyph and one byte of flags per cell, and a loader that expands the text into it when a level is entered.

Keep the text as the *source* format. It is readable, it diffs well, and hand-authored levels stay easy to write.

This is the pivotal step. Stairs, generated levels and line of sight all need per-tile storage, and doing any of them first means doing them twice.

Step 6. A level table

`map1_id`, `map1_name`, `map1_up`, `map1_down` and `map1_dim` already exist — the structure anticipates several levels and the stair coordinates are already declared. Generalise them into an array of level descriptors: id, name, dimensions, up and down stair positions, and a pointer to the tile data. One entry to begin with.

Step 7. Stairs

< and >. Entering a staircase switches the active level descriptor and places the player on the matching stair of the destination. The coordinates are already in the data.

Step 8. A second level

Hand-authored, in the same text format. Proves the machinery of Steps 5-7 before any of it depends on a generator.

Step 9. DungeonMaker

Rooms and corridors, seeded from the RNG so a level is reproducible from its depth and seed. Appendix III of *Writing Games in Assembly Language* sketches this already, and NetWhack's `DungeonMaker.java` (633 lines) is the working reference.

With 16 MB there is no reason to discard a level once made: an 80×40 map is 3,200 bytes, so a 26-level dungeon fits inside a single bank. Levels can simply persist.

Part III — sight

Step 10. Per-tile seen and visible bits

Add the two flags to the tile structure from Step 5, and teach the renderer three states: visible (bright), seen but not visible (dim), unseen (blank). Then mark every tile seen and visible, so **nothing changes on screen yet**.

Installing the machinery separately from switching it on keeps the next step small and makes a regression obvious.

Step 11. Line of sight

Each turn, clear `visible`, then walk a line from the player to every tile within a radius, stopping at anything opaque; mark what is reached visible and seen. NetWhack's `Level.visline()` and `makevis()` are the reference.

Integer Bresenham over tiles — not the PPU's line primitive, which draws pixels.

This is the step that changes how the game feels more than any other on this list. A lit corridor ahead and a remembered room behind is the difference between a map and a dungeon.

Part IV — things to carry

Step 12. Inventory

`i` to list, `d` to drop, and extend the existing `,` pickup. Carried items are a queue, and the machine has `INSQUE`/`REMQUE`/`SCANQUE` for exactly this — `obj.sda` already uses them for objects lying in the world.

Step 13. Item classes

A shared header — kind, glyph, name pointer, weight — with a per-class payload after it. **Weapons and armour first**, because `combat.sda` already exists and will show the difference immediately.

Step 14. Wield, wear, take off

`w`, `W`, `T`. Equipment slots feed the existing combat resolution: weapon damage, armour class.

Step 15. Consumables and an effects hook

Food, potions, scrolls, and a small effect dispatch table they trigger. NetWhack keeps 12 of each and

an `Effect` class; a dozen effects is plenty to start.

Part V — pressure

Step 16. Hunger

A food clock, and `e` to eat. This is what turns wandering into a game: it is the reason to go down rather than explore forever.

Step 17. Regeneration and death

HP returning slowly with time, resting to pass turns safely, and a proper death — tombstone, final score, and a return to the prompt rather than a hang.

Part VI — a world worth returning to

Step 18. A monster table

Replace hardcoded monsters with data: glyph, name, hit points, damage, speed, depth range, behaviour flags. NetWhack's `MobData` has 60 entries; a dozen would already transform the game. Spawn by depth so descending means something.

Step 19. Pathfinding

NetWhack's `PathMap.java` is a Dijkstra map — flood the distance-to-player across walkable tiles, and every monster moves downhill. It replaces “step toward the player” with something that goes around corners, and it costs one pass per turn rather than one search per monster.

Step 20. Save, restore, and scores

`S` to save and resume. The file services already exist behind `INT $15`. A high score table after that.

If only three steps get done

Steps 2, 7 and 11 — eight-way movement, stairs to a second level, and line of sight. That is the difference between a demo and a roguelike, and none of the three is large.