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 — DONE

`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  ; CL = dx, DL = 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 — DONE

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.

The three direction readers

Walking was not the only thing that asked for a direction. `open_door` and `do_talk` each had their own copy of the key list, so after Step 2 you could *walk* diagonally but not *open* or *talk* diagonally — the same bug in two more places, which is exactly what Step 1 was meant to prevent.

The fix is the Step 1 move again, one level up. Four routines now do the work for all three prompts:

```
get_dir_key   ; read a key, fold it to upper case
decode_dir    ; AL -> CL = dx, DL = dy, ELM -> the name, carry = valid
dir_target    ; player + CL/DL -> X, Y, carry = on the map
print_dir     ; "  OPEN " + "NORTHEAST" + newline, preserving C and D
```

`decode_dir` is now the only place in the game that knows what a movement key is. Adding a fifth prompt that wants a direction — fire, throw, kick — costs two calls.

Two details worth keeping in mind:

- `ELM` and `FLD` are *paired* registers (`EL:M`, `FL:D`), so

`FLD` cannot hold a pointer while `D` holds a delta. `print_dir`

```
  takes its prefix in ''GLK'' and consumes it before any call.
* ''dir_target'' bounds-checks, which ''open_door'' and ''do_talk'' never
  did. Reading a glyph one step off the edge of the map used to read
  whatever followed the map data.
```

“ `OPEN DOOR` ” plus “`NORTHEAST`” is 21 characters and the message window is 19 wide, so the prefix lost the word the prompt above it already supplies.

Step 3. A real turn loop — DONE

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.

The turn contract

`move_mon` was being called from inside `draw_map`, so monsters moved once per *redraw*, before the player's key had even been read. It is a call in the loop now, and the loop is the whole of the ordering:

```
main_loop:
    CALL @draw_map
    CALL @get_input         ; the player acts
    JNC @main_loop          ; ...or did not: redraw and ask again
    CALL @move_mon          ; now the world answers
    CALL @inc_game_time     ; and the clock moves
    JMP @main_loop
```

Every command routine answers the same question in the carry flag:

```
^ carry ^ meaning ^
| set   | the player spent a turn |
| clear | nothing happened |
```

`get_input` tail-jumps to those routines, so their carry *is* its carry and no dispatch code has to know which commands cost time. Walking into a wall, naming a direction with no door in it, and pressing an unknown key are all free — the world does not get to answer a non-move. `M` and `G` conjure things out of nothing, so they are free too; they are debug commands.

Three things this uncovered

Giving monsters a real turn meant there had to *be* monsters, and there never had been:

- `init_objects` said “`Write id = 0 for this record`” but the

instruction was `STB [ELM + I]`, and B was never zeroed. Every slot

```
  was born with a non-zero ID — that is, "in use" — so the ''SCANQUE'' in
  ''spawn_monster'' and ''create_gold'' never found a free one. **''M'' and
  ''G'' had been failing silently.** Whether this bit depended on what the
  title screen happened to leave in ''B''.
* ''MOD B, KL'' mixes a 16-bit destination with an 8-bit source. ''MOD''
  takes its width from the destination, so this read ''REGS[30]'' — past the
  end of the 16-bit register file — and ''B'' came back unreduced. Monsters
  and gold were being placed at coordinates like (56242, 15977), off the map
  and out of reach. Both spawners widen the dimensions into ''I'' and ''J''
  first now.
* ''move_mon'' still had a ''SED''/''CLD'' pair bracketing its commit —
  leftover CPU-trace instrumentation, harmless only for as long as no
  monster ever moved.
```

Machine note

The assembler rejects mismatched widths for `MOV` but accepts them for arithmetic, and the CPU then indexes the register file with the raw encoding. `MOD B, KL` is not a typo the tools will catch. Worth a width check on the arithmetic path.

Step 4. Wait — DONE

Numpad 5 and `.` both reach `move_wait`, which is now the one move that is nothing but time: it returns carry set without touching PX/PY, so the monsters get their turn and the clock advances. That makes it the way to watch monster behaviour without moving, which is what makes the next several steps testable.

Part II — depth

Step 5. Turn the map into a structure — DONE

`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.

What landed

Two bytes per square, row by row, at `$030000` in bank 3 – 6,400 bytes for 80×40:

```
tile(x, y) = TILE_BASE + (y * width + x) * 2
```

flag	value	meaning
TF_WALKABLE	$01	you can stand here
TF_OPAQUE	$02	you cannot see through it
TF_SEEN	$04	reserved for Step 10
TF_VISIBLE	$08	reserved for Step 10

SEEN and VISIBLE are defined now and set on every square, so the renderer behaves exactly as it did. Installing the machinery separately from switching it on is what keeps Step 10 small.

`load_level` expands the text into the structure on entry; `tile_flags_for` is the single place that decides what a glyph means, so the loader and `put_glyph` cannot disagree – which they would, the first time a door opened. `is_walkable` is a flag test now instead of a chain of glyph comparisons, and no longer walks the object list to get there.

The test for this step is that **nothing changes on screen**: the rendered display is byte-identical before and after, with the camera scrolled.

The assembler bug underneath it

`AND AL, @TF_WALKABLE` is the first 8-bit immediate in the tree naming a label defined in a *later* file, and forward-reference patching got that wrong: it wrote the low byte, then wrote a second byte unconditionally, zeroing the next instruction's opcode. Here that opcode was a JZ, so `is_walkable` fell through its own branch and nothing on the map could be stepped on – from an image that assembled with no errors and no warnings.

Worth remembering as a shape: a program that assembles cleanly and behaves absurdly is worth a look at the emitted bytes, and then at the trace.

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.