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 — DONE

`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 — DONE

< 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.

What landed

The descriptor, which is the old `map1_*` fields plus the two pointers they were missing:

offset	field	
0	`LV_ID`	1 byte
1	`LV_NAME`	9 bytes
10	`LV_DIM`	width, height
12	`LV_UP`	x, y of the staircase up
14	`LV_DOWN`	x, y of the staircase down
16	`LV_SRC`	→ the source text
19	`LV_TILES`	→ this level's tile array

Every level is built at startup and keeps its own tile array, so **levels persist** – a door you opened is still open when you come back. There is nothing to gain by discarding one: a 26-level dungeon is under 170 KB.

`enter_level` mirrors the dimensions and tile pointer into `level_dim` and `level_tiles`, because `tile_addr` and `draw_world` read them per square and per cell.

Both commands check you are standing on the staircase, switch the descriptor, and put you on the matching stair of the destination – down arrives at the level's *up* stair, and the reverse. Both honour the turn contract: a flight of stairs is a turn, refusing to move is not.

`place_stairs` stamps < and > onto each level as it is built. Those coordinates had been in the data since the game was written and nothing had ever drawn them.

The table has two rows. Level 2 borrows level 1's text as a placeholder – Step 8 replaces that single pointer. Its *tiles* are already separate, which is what made the persistence testable: open a door on level 1, and the same square on level 2 is still shut.

Step 8. A second level — DONE

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

The machinery is already in place and exercised – what is left is **content**. One pointer in `level_table` row two changes from `@map1_data` to `@map2_data`.

What landed

BrynnWell, the well maze from NetWhack's `src/netwhack/world/mapgen/BrynnWell.java` – 20 x 30. Exactly one pointer in the table changed, which is what Steps 6 and 7 were for.

Glyph for glyph: NetWhack's `.` is our `'' , ''#` and + carry over, and its six secret doors (`s`) are ordinary doors here. There is no search command yet, so a secret door would be a wall you could never get through; they can go back to being secret once something can find them.

It is a **20 x 30 level**, not a 20 x 30 maze padded out to match level 1. That needed two fixes in `draw_world`, which was the last place still assuming one map size:

- the camera clamp computed `(map_width - view_width)`, which underflows

when the map is the narrower of the two. A map that fits has nowhere to

```
  scroll to, so the answer is zero.
* the render loop drew 58 x 23 squares unconditionally, so past the right
  edge of a narrow map it carried on into the start of the next row.
  Squares outside the map draw as nothing now.
```

Maps *larger* than the view always worked – that is the scrolling case. Everything else had carried per-level dimensions since Step 6. Level 1 is unaffected and provably so: its rendered screen is byte-identical before and after, walked to the same square with the same door open.

Verified by rebuilding the level out of its tile array and diffing against the Java source: all 30 rows match, with < and > stamped where the descriptor says.

Step 9. DungeonMaker — DONE

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.

What landed

`programs/rogueima/dungeon.sda`. Fill the level with rock, mine one room in the middle, then repeatedly find a wall with **exactly one** floor square beside it and build on the far side, leaving a door in the wall you dug through.

That one-floor-neighbour rule is doing two jobs. It stops a level collapsing into a single cave, and it is also why every level comes out connected: each new piece is reachable through the door that made it. Connectivity is a property of the construction, not something checked afterwards.

Levels 3, 4 and 5 are generated, 78 x 22. LV_SRC of 0 in the level table means “there is no text to expand, build it”, and the generator writes LV_UP and LV_DOWN into the descriptor itself – a level that does not exist yet cannot say where its stairs are. Generated levels persist exactly like the hand-made ones.

The corridors NetWhack never built

`addfeature` in `DungeonMaker.java` reads:

```
switch (feature_type) {
    case 1:  success = makeshop(...); break;
    case 2:
    default: success = makeroom(...); break;
}
```

`case 2:` is an empty label falling straight into `default:`, so it builds a room like everything else. `TileData.CORRIDOR` exists and `readmapline` can produce one, but nothing ever generated one. The branch was never written rather than broken – which is why BDungeon levels are rooms hanging off rooms.

Writing it costs almost nothing, because **a corridor is a room with one dimension collapsed to zero**, and `makeroom`'s geometry already handles that: building EAST with height 0 gives `a.y = b.y = yloc`, a single row. So rooms and corridors are one routine called with different sizes, and the feature roll now actually branches.

Verification

Dump the tile array and analyse it rather than looking at it. Level 3 came out 310 floor squares with all 310 reachable by flood fill; level 4, 388 of 388. Solid border, 18 and 20 doors, both staircases placed, and the two levels different from each other. Corridor squares – floor with exactly two opposite floor neighbours – numbered 70 on level 3.

Part III — sight

Step 10. Per-tile seen and visible bits — DONE

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.

What landed

The flags have existed since Step 5, set on every square. The renderer reads them now:

state	drawn	colour
`TF_VISIBLE`	lit	`$07` light grey on black
`TF_SEEN` only	dim	`$08` dark grey on black

state	drawn	colour
neither	not at all	–

They are still set on every square, so **nothing changes on screen** – which is the point. Step 11 only has to start clearing TF_VISIBLE.

Colour arrives with this step, because “dim” needs one. Palette 5, CGA as an IBM 5153 showed it, and the indices are NetWhack's own: its ColorMap puts light grey at 7 and dark grey at 8 in the same order, so a colour there is a colour here. TileData also colours by *tile kind* – doors BROWN, staircases WHITE – which maps onto the same palette and is worth adding.

Two traps worth writing down:

- draw_borders clears the screen every redraw, and the clear repaints

the colour plane with the mode's default. The colours have to go back on

```
  after it, not once at startup.
* ''print_char'' takes its colour from ''VIDEO_CHAR_COLOR'', not from the
  plane. That is how the status panel is painted, and without setting it the
  panel comes out dark grey on brown under a CGA palette.
```

Verified by dumping both planes: glyphs unchanged, colour uniformly $07 across all 2000 cells. Clearing TF_VISIBLE on one square by hand drew it dim with its glyph intact; clearing TF_SEEN as well drew nothing, leaving a gap in the wall.

Step 11. Line of sight — DONE

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.

What landed

programs/rogueima/los.sda. NetWhack's visline() unchanged, but cast to a **radius of 8** rather than to the edge of the map.

makevis() there casts to every square on the map boundary – about 200 rays of up to 78 steps, which its own comment calls *“a terrible waste of resources”*. A radius is the same code with nearer endpoints: about 68 rays of at most 11 steps, some thirty times less work, and it gives a torch instead of sight to the far wall. LOS_RADIUS is one constant.

The radius is **circular**: a square is in range only if dx*dx + dy*dy ⇐ 64, so the corners of the box do not see further than the sides. Two MULs per square.

Bresenham is the error-accumulator form, which keeps every quantity non-negative – the signs live in LOS_SX/LOS_SY and are applied as 8-bit wraparound. No signed comparisons, which on this machine would have meant testing N against V by hand.

Switching it on was **one line**: TF_FLOOR and TF_WALL stop carrying SEEN and VISIBLE, so a square starts unknown. That is what Step 10 was for.

Three things came with it:

- put_glyph preserves SEEN and VISIBLE. Opening a door changes

what a square *is*, not whether you have been there.

- a known floor draws as ., because floor is stored as a space and a

blank square is what “never seen” looks like – lit floor was invisible.

- monsters and items are only drawn where TF_VISIBLE is set, or you see

them through walls.

Verification

Dump the glyph and colour planes together and separate lit from remembered by colour. In the open: a clean circle of lit floor spanning exactly px±8 and py±8, with a dim crescent trailing from where the player walked. In BrynnWell's maze, standing in a corridor:

1. #

+....@.+

1. #

the corridor, the walls above and below it, the closed doors at each end, and nothing through them.

Part IV — things to carry

Steps 12–14. Inventory, item classes, wield and wear — DONE

Planned as three steps and built as one, because none of them is any use alone: an item you cannot carry, a pack you cannot look in, or a suit you cannot put on. All of it lives in the new item.sda.

The item table

NetWhack knows what an item is from its **class**: Armor extends Item, carries an acmod, and takes its name and numbers from ArmorData[kind]. Here the class is the OBJ_TYPE tag and the per-class payload is OBJ_DATA1 — a tagged union, which is the same shape without the inheritance. What the tag does *not* give you is the data, and that is the table:

```
.equ IT_TYPE  0     ; which class this is
.equ IT_GLYPH 1
.equ IT_STAT  2     ; the class's one number
.equ IT_NAME  4     ; -> the name
.equ IT_SIZE  7
```

```
item_table:
    .bytes 3, ')',  3, 0, @str_dagger     ; WEAPON, 1d3
    .bytes 4, '[', 10, 0, @str_leather    ; ARMOR,  acmod 10
```

`IT_STAT` is the one number the class needs — a weapon's damage sides, a suit's ac modifier — exactly as `WeaponInfo` carries `dmg` and `ArmorInfo` carries `acmod`. **A new item is a row here, not a branch in `make_item`.**

The pack

A second queue over the *same* fixed node array the world uses. Picking something up is `REMQUE` from the world list and `INSQUE` into the pack; the node itself never moves and nothing is copied. That is what `INSQUE`/`REMQUE` are for, and it is why `draw_items` stops drawing something the moment you take it — it walks the world queue, and the node is no longer on it.

The screen

ADOM by way of NetWhack: a full page, grouped by category, one letter per item. Letters are the item's position in the pack, so an item keeps its letter whichever heading it appears under.

```
  INVENTORY
```

```
  Weapons
    a - dagger (wielded)
  Armour
    b - leather armor (worn)
```

1. - press any key –

`I` to look, `E` to wield, `W` to wear, `R` to take everything off, `D` to drop, and the existing `,` extended to pick items up. `X` is a debug command that drops one of each at your feet.

What it does to combat

`do_combat` used to land every time and take off exactly 1. Now it rolls **2d(attack) against 2d(defense)**, as NetWhack does, with ties to the defender; wielding anything at all is worth +10 attack, armour adds its `acmod`, and damage is the wielded weapon's die.

Verification

A miss proves nothing, so the rolls are called directly rather than played. `test_combat.sda` runs a thousand trials each way and leaves four totals in memory:

```
hit by a monster, unarmoured      458 / 1000
hit by a monster, in leather      148 / 1000
damage over 1000 swings, fist    1000        (a fist is always 1)
damage over 1000 swings, dagger  2006        (1d3, mean 2)
```

Armour makes you harder to hit and the weapon changes the damage, which were the two things this step set out to show.

The memory map, which moved

The step also flushed out a real bug. The game sat at `$02C000` with the object array 12 KB above it at `$02F000`. The code grew past that line, so `init_objects` cleared the node array **over the game's own instructions** — which presents as the player being unable to move, with the assembler reporting nothing.

The game now loads at `$020100`: `USER_ORIGIN`, the address the shell and `INT $20` already load programs to, so an assembled Rogueima is a program the shell can run by name like any other. Everything it allocates is a whole bank away, one bank per kind:

```
$020100   bank 2: code and data, growing up   (55 KB to the FS command
block)
$030000   bank 3: game data -- the objects
$040000   bank 4: the maps -- one tile array per level
```

The bases are labels and everything derives from them, so the second map bank a real dungeon will need is one line. `.equ` learned `@label + N` to make that possible.

Two smaller fixes fell out of testing: `draw_items` only ever drew gold, so a dropped dagger vanished (it now draws every object that is not a monster), and `inv_ask` compared item letters against lowercase a while `get_dir_key` folds keys to upper case, so no item could ever be selected.

Since then: the whole item table

Steps 12-14 shipped with two items written by hand. `itemtable.sda` is now NetWhack\'s five item arrays – `ArmorData`, `WeaponData`, `FoodData`, `PotionData`, `ScrollData`, **41 items** – generated by `tools/gen_itemtable.py`. Five arrays of five `Info` classes with five different constructor signatures come out as one table with a class tag and a per-class payload.

`IT_SLOTS` is the field that earned it. `ItemInfo.eqslots` is the *set* of equipment slots an item may occupy: gloves name `GLOVES` and nothing else, a dagger names `WEAPON` and `OFFHAND`. There was one `eq_weapon` and one `eq_armor`, and between them they cannot express a pair of gloves – putting gloves on took your plate off, and the only symptom was a defence number that went **down** when you added armour. There is an equipment rack now, one pointer per slot, and

`defense_rating` sums every worn piece.

```
defence 10 bare  ->  60 in field plate  ->  64 in plate and gloves
```

The equipment page

`Inventory.imEquipment()`. `w` opens a page listing all thirteen slots at once, lettered A to M. Press a slot: something in it comes off, an empty one offers the things that fit **that slot and nothing else**, which is `getitem_byslot()`. A single “wear what?” prompt has to guess where a thing goes and cannot show you what is empty, which is the question a player has.

Identification

Only potions and scrolls have a fake name – `ArmorInfo`, `WeaponInfo` and `FoodInfo` take no such argument, because a sword looks like a sword.

Two things about `PotionData.randomize()` that are easy to get wrong. It swaps the fakename **and the colour** together, so an appearance is a (name, colour) *pair* – get it wrong and you have a milky potion drawn in green. And `identified` is a field of the DATA row, not of the object: drink one potion of healing and every potion of healing is named for the rest of the game. Neither can live in the generated table, so both are a byte per kind in the data bank.

Names are therefore **built**, not stored: “potion of moonshine”, or “milky potion” when the appearance begins with a capital, as `Potion.name()` does.

Paging

`Inventory.getitem_step2/step3` – the one screen every “choose something” goes through, and, since the shops use it too, worth having once. Letters restart at a on every page, as NetWhack does: the letter means “the third thing I can see” and must not depend on how long the list is.

A bug worth writing down

Two commits of new variables were placed on top of existing ones in the hand-managed variable page. `inv_more` sat on `item_kind`, so the class filter was overwritten halfway through drawing the inventory and **the weapons and armour vanished from it**. Fourteen overlaps in all, no assembler warning, and the symptom in a routine that looked innocent.

`tools/check_vars.py` reads the `.equ` declarations and their “; N bytes” comments and reports overlaps. Run it after adding one.

Step 15. Consumables and an effects hook — HALF DONE

Food, potions, scrolls, and a small effect dispatch table they trigger.

The **things** all exist: importing NetWhack's item tables brought 7 foods, 5 potions and 5 scrolls with their real numbers, and with them the identification system (see below). e eats and q quaffs, and a potion gives you its nutrition – which for poison is *negative*.

What is left is the **effects**: healing, blindness, speed, teleport, identify, town portal. NetWhack has an `Effect` class with a trigger type, and `Food.event_eat` already calls `fxlist.transferByItemTrigger(this, Effect.T_FOOD)`. Until that exists, a potion of speed is a drink of water with a different name on it.

Part V — pressure

Step 16. Hunger, as an energy meter — DONE

A food clock, and e to eat. Taken ahead of Step 15 because it is the one that changes how the game is played: it is the reason to go down rather than explore forever.

Inverted

This is NetWhack's `Engine.pc_hunger()` with the sign flipped. There, `pc.hunger` counts **up** from zero and every test asks how big it has got:

```
pc.hunger++;
if (pc.hunger > 5000) { ... }
if (pc.hunger > 4000) { ... } else if (pc.hunger > 3200) { ... }
```

Here the same number counts **down** from EN_MAX and is called energy. They are the same meter — every threshold is 5000 minus NetWhack's — but the down-counting one can go on the status panel and be read without explaining. *Energy: 800* says you are nearly out of something. *Hunger: 4200* does not.

NetWhack	Rogueima	says	odds
hunger > 800	energy < 4200	You're feeling a might peckish.	1 in 500
hunger > 1600	energy < 3400	Your stomach is grumbling.	1 in 500
hunger > 2400	energy < 2600	You are feeling hungry.	1 in 500
hunger > 3200	energy < 1800	You are feeling very hungry.	1 in 400
hunger > 4000	energy < 1000	You are starving!	1 in 250
hunger > 5000	energy == 0	starving.. TO DEATH! (-1 hp)	1 in 50

Two consequences of the inversion, both small:

- eating **adds and clamps at the top** rather than subtracting toward zero,

so the bug to avoid is an unsigned overflow rather than an underflow.

```
  NetWhack lets hunger go negative on a big meal; a meter has a maximum.
* starving is energy **at** zero rather than hunger past 5000. NetWhack's
  hunger keeps climbing past its own limit and nothing reads it up there, so
```

```
  nothing is lost by parking at the bottom instead.
```

The clock

NetWhack calls `pc_hunger` when `(gametime % TICKS_PER_TURN) == 0`, because its `gametime` is fine-grained and ticks far faster than the player acts. Rogueima's `game_time` already counts player turns and only advances when one is spent, so the modulo is the identity here — `main_loop` just calls it once a turn.

At one point of drain per turn, a full meter is **five thousand turns**. That is deliberately a minor concern for now: it is there, it works, and the numbers are the thing to argue about when balance is the concern.

Eating

e eats. A ration is `FoodData[2]` — 500 nutrition, glyph `%` — and it is a **row in `item_table`**, not a branch anywhere, which is what Step 13 bought. NetWhack's `Food.event_eat` also heals a point when `Dice.roll(1, nutrition) > 200`, so a ration is a three-in-five chance of one hp back.

Disposal is the fiddly part, and it is not a REMQUE alone. `make_item` finds a free node by scanning the **world** queue for one with a zero id, so an item taken off the pack queue and put on no queue at all is leaked rather than freed. Eating therefore puts the node back on the world queue and only then zeroes the id — the same disposal `pick_up_gold` does, from the other end.

Verification

Five thousand turns of drain is not something you can sit through, and a complaint that fires one turn in five hundred is not something you can see by playing. `test_hunger.sda` calls `pc_hunger` directly with the meter **pinned** at a chosen level, which holds it inside one band of the chain, and asks whether that band says anything — by clearing the message buffer first and counting the non-zero bytes after.

```
drain past the bottom                  0        (floors, does not wrap)
hp lost at empty over 5000 turns       105      (1 in 50, so about 100)
messages at energy 4300                0 bytes  (above every band: silent)
messages at 4100/3300/2500/1700/900    247 bytes each
```

The silent row is the one that matters as much as the loud ones: it pins the EN_PECKISH boundary from above, so an inverted comparison could not pass.

And in the real loop, poked down to 30 and left to walk: energy floored at 0 and hp went 10 → 6 while starving.

A bug it turned up

`pick_up_items` listed the types that *could* be carried, so the ration was un-pickable the day it was

added. That is the same shape as `draw_items` only ever drawing gold, found in Step 12. Both now name the one type they **exclude** — a new kind of item should not need a line in either place.

Step 17. Regeneration and death — HALF DONE

Regeneration is in. `Mobile.per_turn()`\'s healing half: one turn in a hundred you mend a point, and NetWhack\'s `hunger++` beside it becomes a second point off the energy meter. That is the link that makes food matter for something other than eventually dying of it — **a wound is paid for in rations**.

One addition to NetWhack: it needs `EN_REGEN` (500) in the tank. Starving and mending at once is the single combination that lets a player wait out any wound for free, because the waiting is what heals them. Verified: **0** hit points healed over ten thousand turns below the line.

The cost is only taken when a point actually goes back. NetWhack rolls, calls `heal()` and does `hunger++` regardless — `heal()` clamps, so an unwounded player pays a point of food for nothing. That is an artifact of two statements sitting next to each other, not a design.

Healing needed something to stop at, so the player has an `hp_max` and every path that gives points back goes through one clamping routine.

What is left: **resting** to pass turns safely, and a proper **death** — tombstone, final score, and a return to the prompt. Starvation and monsters both drop into `kill_player`, which prints “You died.” and returns.

Part VI — a world worth returning to

Step 18. A monster table — DONE

There used to be `make_rat`, `make_snake` and `make_spider`: three routines, each with its own hit-point roll and its own hard-coded glyph, and a `random_monster` that rolled 1..3 and branched to one of them. **Adding a fourth monster meant writing a fourth routine.**

Now there is one routine and a table. `montable.sda` is all 28 rows of NetWhack's `MobData.mobdata[]`, with every field it carries.

Generated, not transcribed

The table is produced by `tools/gen_montable.py`, which reads NetWhack's Java directly — resolving `monsym` symbols to glyphs, `ColorMap` names to CGA indices, `Attack`/`Damage` constants to numbers, and dice strings like `“1d2-1”` to (n, sides, bonus). Twenty-eight rows of symbolic fields is exactly what gets copied wrong by hand, and NetWhack's table will change again. Re-run it; do not edit the output.

`ColorMap` turned out to be the standard CGA order already, so the colours came across as-is.

The record

```
MO_GLYPH  0   MO_AC    3   MO_MSPEED 6   MO_NATK 13
MO_COLOR  1   MO_PROB  4   MO_ASPEED 8   MO_ATK  14  (3 x 5 bytes)
MO_LEVEL  2   MO_ALIGN 5   MO_NAME  10   MO_SIZE 29
```

An instance keeps two things: `OBJ_DATA1` is current hit points, which change, and `OBJ_DATA2` is the **kind**, which does not. Everything else is read back out of the table from that kind, so an instance costs no more than it did.

What is wired

- **glyph, colour, name.** Colour is not decoration: 28 monsters share 6

glyphs, and without it a dog and a wolf are the same `d`.

- **level** — hit points are `level d8`, as `Mobile` rolls them, and

`10 x level` is the attack rating. `getAttackRating()` is

```
  ''10*xp_level + 3*DEX + STR + bonuses''; monsters here have no stat block,
  so the level term is what is left.
* **armour class**, added to the defence, as ''getDefenseRating()'' does.
* **probability**, and **the attacks**.
```

Every monster used to rate a flat 10 — which is exactly what `10 x level` comes to at level 1. So **the first floor plays as it always did** and the deeper ones get harder, which is what importing the levels was for.

Spawning follows `randomkind(plevel, zlevel)`: the window is `(plevel + zlevel) / 6` to `/ 2`, by rejection sampling as NetWhack does it. Floors 1 and 2 hold level-1 monsters only; by floor 9 it is anything up to level 5. There is no experience system yet, so the player's level is 1 and depth widens the window alone.

What was left behind

All of it is **imported**; these are the fields nothing reads yet:

- `MO_MSPEED` / `MO_ASPEED` — there is no speed system. A rock mole at

1500 and a phase rat at 400 currently move at the same rate. This is the

```
  big one, and it is a scheduler, not a data problem — which is the point of
  importing the numbers now.
* ''MO_ALIGN'' — no alignment, no peaceful monsters, no ''Really attack?''
* ''MOA_DMGT'' — no damage types, so a salamander's fire is ordinary.
* ''MOA_TYPE'' is stored and half-used: ''do_attack'' picks **one attack at
  random** from the list, which this does, so a wererat's two attacks and a
```

```
  salamander's three already work. What is unused is the claw/bite/spit
  distinction in the message.
```

Row 0 is the player. Its probability is 0 so it can never be picked; it is kept so an index here is an index into `MobData`.

Verification

A wrong offset in a 29-byte record reads as a monster that is subtly too strong, not as a crash, so `test_mon.sda` asks questions with known answers:

```
depth 1 level window              1 .. 1
depth 10 level window             1 .. 5
salamander hit points             14 .. 57   (8d8)
sewer rat damage                  0 .. 1     (1d2-1)
battle orc damage, highest        6          (2d3)
salamander attack / defence       80 / 81    (level 8, ac 1)
wererat attacks in the table      2
orc damage over 200 real swings   804        (through do_combat)
```

The last one goes through `do_combat` rather than calling `mon_damage` directly, because that is where a register clobbered across a call shows up. And one did: **`FLD is FL:D`**, so the pair aliases the `D` register, and using `D` as scratch while walking a row silently overwrote the low sixteen bits of the pointer. Every field then read as zero, the code took its “no attacks” early-out, and **every monster in the game did no damage at all** — with no crash and no assembler warning. `mon_damage` uses `C`.

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.

Part VII — saying it properly

Step 21. Item names in messages

Everything says *“You wield it.”* The item knows its name; the message does not ask. NetWhack has a small naming layer that every message goes through:

- `name()` — the real name, with `pre_name` and `post_name` when identified

- `aname()` — *“a dagger”*, *“an athame”*
- `tname()` — *“the dagger”*
- `bcstatus()` — *“blessed ”* / *“cursed ”* / *“uncursed ”*, when known
- `msg.You(s)` — prints *“You ”* + s

so `msg.You(“wield ” + tname())` is *“You wield the dagger.”* We already build names for potions and scrolls; this generalises that and routes the messages through it.

Do this first. It is small, it is the difference between a prototype and a game every single turn, and everything below it wants to say something.

The message window is 19 columns and wraps mid-word, so this also wants a word-wrapping `print_msg` — the hand-broken strings do not survive a name being spliced into the middle of them.

Part VIII — things that do something

Step 22. The effects hook

`effects/Effect.java` and `Effects.java`. An effect is a **type** and a **trigger**, held in a list on a mobile, and items hand theirs over when something happens to them:

```
E_MODSTAT  E_REGENHP  E_DEFENSE  E_P_SPEED  E_P_BLIND
E_S_TEMPORAL_SUSPENSION  E_W_PRE_ATP  E_F_EDNAS_PIES
```

```
T_PERTICK  T_PERTURN  T_PERSTEP  T_ONWEAR  T_ONBEAR
T_FOOD  T_QUAFF  T_READ  T_ZAP  T_ATTACK  T_DEFEND
```

`Food.event_eat` already calls `fxlist.transferByItemTrigger(this, T_FOOD)` — the call site exists in our port, with nothing behind it. The machinery is a fixed array of effect slots on the player, a dispatch on type, and `gc()` to expire them. Nothing here needs allocation.

Step 23. The speed system

`MobInfo` carries `mspeed` and `aspeed` and has since Step 18 — a rock mole is 1500, a phase rat 400, the player 1000 — and **nothing reads either**. Every mobile moves once per turn, so a phase rat is exactly as quick as a rock mole.

NetWhack's own comment says how it is meant to work: *“relative speed can be added by increasing a speed variable and only allowing movement when it reaches a certain value (then resetting it)”*. An energy counter per mobile, topped up each turn by its speed, and it acts while it can afford to. That is also what turns `gametime` into the fine-grained clock its `% TICKS_PER_TURN` assumes — ours counts player turns because nothing needed finer.

This is the one imported field that changes how the game *plays* rather than how it reads, and the potion of speed has nothing to do without it.

Step 24. Potions that do what they say

Healing, poison, blindness, speed. Until this lands a potion of speed is a drink of water with a different name on it. Blindness wants LOS_RADIUS temporarily set to 0, which is one variable; speed wants Step 23's clock.

Step 25. Scrolls, and ''r'' to read

`identify`, `teleport`, `town portal`, `temporal suspension`, and the crumpled note that starts identified. `Scroll.do_id` offers the pack through the chooser we already have and marks one kind known. Reading also identifies the scroll — which is the second half of Step 12's identification system finally paying for itself.

Step 26. Blessed, cursed, uncursed

Three bits per item and `bcstatus()` in front of the name. Cursed armour that will not come off is the first thing in the game that can go **wrong** in an interesting way.

Part IX — a world

Step 27. The rest of the tile kinds

`TileData` has 22: water, trees, flowers, road, bridge, bed, table, chair, altar, throne, shop floor and shop bar, on top of the six we use. Cheap — a table of glyph, colour and flags — and every one of them is a prerequisite for a village that looks like a village rather than a dungeon with the walls knocked out.

Step 28. A level script

Brynn is 84×30 of map text followed by directives:

```
"DENIZEN Farmer_Jim 19 17 The_apples_look_lovely_this_year.
The_farmin'_life_for_me! ..."
"SHOPKEEPER RANDOM 53 9"
"NOMONSTERS"
```

Underscores for spaces, because the parser splits on them. `LevelFactory` has a comment saying all of this *“needs to be put into some kind of script and attached to the level file, and not written here”* — and it is right. Our `level_table` already holds a source pointer; this is the format it points at.

Step 29. Branches, and stairs that know where they go

The dungeon is not one stack. A staircase carries a **destination branch**:

```
Brynn --(the well)--> BrynnWell --> LCave
                                --> BDungeon --(depth 9)--> Croky Castle
```

`LevelFactory.createlevel` takes a branch name and a depth and dispatches on it; `LevelLibrary` keeps the levels already built so going back finds them as you left them. Our `level_table` is a flat list of five and will need to become that.

Step 30. The village of Brynn

Depth 0, `popfreq` 0 — no wandering monsters, which is what a town *is*. The well in the middle is a staircase into `BrynnWell`. Needs Steps 27, 28 and 29 first, and then it is mostly data.

Step 31. NPCs and chat

`mobile/NPC.java`. A denizen has a name, a position and a list of things to say; `t` picks one at random. `Fortune` supplies a rumour when the script says `RANDOM`. Rufus the dog lives here too — the first mobile that is neither the player nor an enemy.

The four denizens of Brynn are how the sunsword quest is told: Farmer Jim, the Mayor, Edna and Father Monoly each know a piece of it.

Step 32. Shopkeepers

`engine/Shop.java`. Buy, sell, and a shopkeeper who objects when you leave with something you have not paid for. The paged list it uses is `item_pagedisplay` — **the screen we already built in Step 12's paging**, so this is the shop logic and not the shop interface. `IT_VALUE` has been sitting in the item table unread since the import, waiting for exactly this.

Part X — the quest

Step 33. Croky Castle

The goal level, and the only one with no stairs down. Reached from `BDungeon` depth 9, but only while the sunsword has not been found.

Step 34. The sunsword

`WeaponData[15]`, probability 0 so it is never generated at random — it is **placed**. Wielding it sets `GameFlags.has_sunsword`, says *“The sunsword seems to glow and gleam with an unearthly light!”*, and from then on it **attracts monsters**: `Engine.gametick` pops an extra one every hundred turns with *“You've got a bad feeling about this...”*

The first unique object in the game, and the first item whose being carried changes the rules.

Step 35. The endgame

Carry it back to the surface. `Engine` checks, every time you take a staircase, whether you have the sunsword and are standing in `Start` — and if so, +1000 and *"You have escaped the dungeons of doom!"* That is the win condition, and it is four lines. It needs Step 17's death screen to exist, because winning and dying print the same tombstone.

Part XI — depth, once it is a game

Step 36. Traps

Six: trapdoor, bear trap, teleport, dart, sleeping gas, rust. A tile flag, a kind, and a hidden bit.

Step 37. Search, and secret doors

`s`, and `TileData.SECDOOR`. BrynnWell was hand-converted with its secret doors turned into ordinary ones back in Step 8 *because there was no search command*; this is the step that lets them go back.

Step 38. Experience, and the stat block

`Stats`: STR, DEX, INT. The ratings are already the right shape and simply have the terms missing — `getAttackRating()` is `10*xp_level + 3*DEX + STR + bonuses`, and ours is `player_str` standing in for all of it. Killing things should raise a level, and the monster spawn window already reads `plevel` and has been given a hard-coded 1 since Step 18.

Step 39. LCave, and a third kind of level

The cave generator, skipped in Step 9. Gives `BrynnWell` somewhere to branch to that is not more of the same.

Step 40. Saving

S to save and resume, and `ObjSaver` for the shape of it. The file services already exist behind `INT $15`. The queues make this harder than it looks: what is saved is a graph of pointers into a fixed node array, so it saves as indices or not at all.

Step 41. The rest of the flavour

`Fortune` rumours, Edna's pies, the dogfood, the well, the altar and the throne. None of it is systems

work; all of it is what makes the place feel like somewhere rather than a grid.

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.

And if only three MORE get done

Steps 21, 22 and 29 — names in messages, the effects hook, and branching stairs. The first makes every turn read like a game; the second is what every consumable in the table is waiting for; the third is the shape the whole rest of the world hangs off.