No Man's Land:

A Custom SystemVerilog GPU and Game on the DE1-SoC

Rohit Biswas (rb3908) Kambinachi Obioha (kno2117) Nicola Paparella (np2953)

CSEE 4840 — Embedded System Design, Spring 2026
Instructor: Prof. Stephen Edwards
May 13, 2026

Abstract

No Man's Land is a top-down WWI horde-survival game implemented on the Terasic DE1-SoC. The project partitions the system between a custom SystemVerilog GPU peripheral (`nml_gpu`) running in the Cyclone V FPGA fabric and a C game on the ARM Cortex-A9 hard processor. The GPU drives a 640×480 @ 60 Hz VGA output with tile-mapped background, 32 hardware sprites with per-scanline priority sorting, a 256-entry palette, and an on-chip HUD overlay. The HPS writes the sprite table, palette, tile map, and player-state registers through a 16 KB Avalon-MM lightweight slave and signals a tear-free buffer swap at VBLANK. Two build flows exist: a fabric-only smoke test (`hw/quartus`) that confirms the rendering pipeline by initialising the memories from `$readmemh` hex files, and an HPS-integrated Platform Designer system (`nml_gpu_hw/`) that mounts the peripheral on the lightweight bridge and exposes it to a Linux userspace driver via `/dev/mem`. The smoke-test bitstream is verified on hardware. The HPS-integrated build boots Linux, exposes the peripheral on the lightweight bridge, runs the C game loop at 60 Hz, drives the VGA monitor, and reads SNES controller input through evdev. This report covers the architecture, register map, build flow, and verified results.

Contents

1 Introduction

No Man's Land is a WWI horde-survival game running on the Terasic DE1-SoC. The FPGA fabric carries a custom SystemVerilog GPU peripheral (`nml_gpu`) that drives a 640×480 @ 60 Hz VGA display. The on-board ARM Cortex-A9 runs the C game loop: wave logic, enemy AI, projectile physics, collision, auto-attack, and HUD updates, all inside a 16.67 ms frame. The two halves exchange sprite tables, palette entries, and control bits over the Avalon-MM lightweight bridge.

The player controls a soldier in a trench at the bottom of the screen. Waves of enemies spawn at the top and march down. The player fires bullets in four directions, picks up ammo drops, and at the end of each wave selects an auto-attack upgrade (mortar, mustard gas, or artillery). The goal is to survive as many waves as possible.

Input is a SNES-style USB gamepad (DragonRise generic, VID 0079:0011) read through Linux evdev. A deterministic stub (`input_fake.c`) stands in for the controller during off-line development.

2 System Architecture

The system is partitioned along the obvious line: anything that runs at pixel rate is in hardware; everything else is in software. The block diagram is shown in Figure 1.

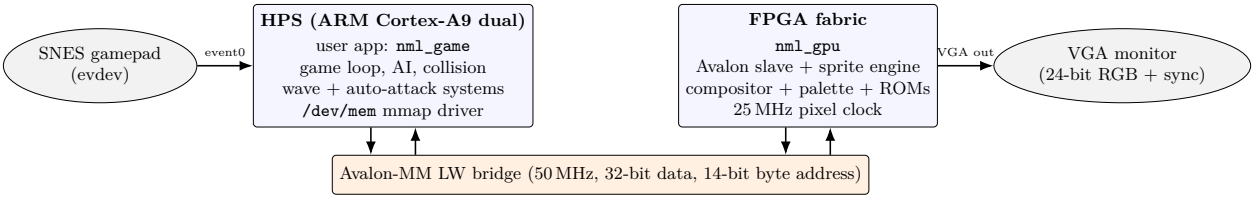

Figure 1: System block diagram. HPS runs the C game; FPGA fabric runs the GPU peripheral; the lightweight bridge carries register writes and reads between them.

The HPS reads input from the USB gamepad, ticks the game state at 60 Hz, writes the resulting sprite table, palette, and tile map into FPGA memory through the Avalon-MM lightweight slave, and signals a frame swap by writing the `CTRL.SWAP` bit. The FPGA latches the new state at the next VSYNC and continues to drive the monitor without ever stalling. A 50 MHz Avalon clock and a 25 MHz pixel clock are the only two clock domains in the design; both derive from the board's 50 MHz oscillator.

Frame budget at 60 Hz is 16.67 ms. Per VGA frame at 640×480 there are $800 \times 525 = 420{,}000$ pixel cycles at 25 MHz (16.8 ms wall-clock), of which $640 \times 480 = 307{,}200$ are visible. The 96-pixel horizontal blanking and multi-line vertical blanking give the sprite engine the time it needs to prepare each scanline.

3 Hardware Design

The hardware lives in `hw/` and consists of seven SystemVerilog modules plus one Verilog clock divider. A separate Platform Designer project under `nml_gpu_hw/` wraps the same RTL as a Qsys component for HPS integration. Figure 2 shows the module hierarchy.

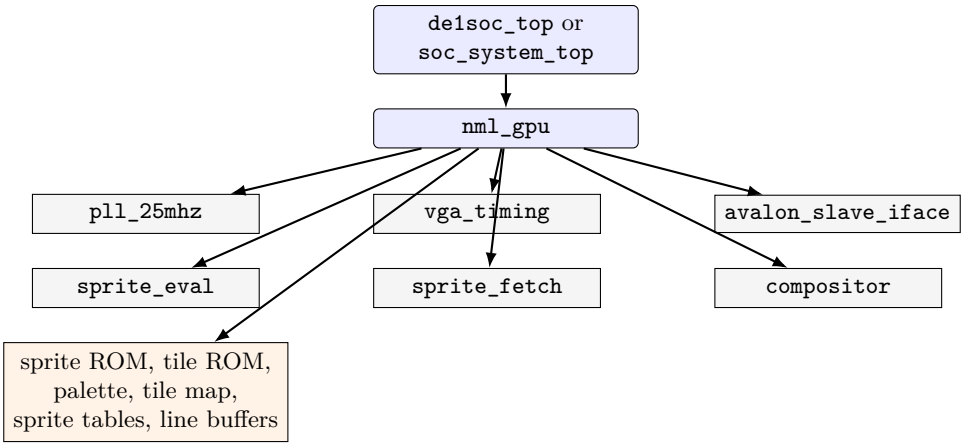

Figure 2: Module hierarchy of the GPU. The same `nml_gpu` is wrapped by `de1soc_top` (Phase 1) or by `soc_system_top` (Phase 2).

3.1 Pixel clock generation (`pll_25mhz.v`)

The VGA standard for 640×480 @ 60 Hz specifies a 25.175 MHz pixel clock. The Cyclone V on-chip PLL cannot produce that exact frequency from the board's 50 MHz reference, so we use 25 MHz, which monitors in the lab tolerate without issue. For the Phase 1 smoke test, the divider is a single D flip-flop that toggles on every 50 MHz rising edge (`hw/pll_25mhz.v:21-26`). Quartus promotes the divided register output onto a global clock network when it is used as a clock downstream, which is exactly how it is consumed in `hw/nml_gpu.sv:194` and elsewhere.

Listing 1: Divide-by-2 implementation. Replace with `altera_pll` IP in Phase 2.

```
module pll_25mhz (
    input  wire refclk,    // 50 MHz reference
    input  wire rst,       // active high reset
    output wire outclk_0   // 25 MHz pixel clock
);
    reg clk_div = 1'b0;
    always @(posedge refclk or posedge rst) begin
        if (rst) clk_div <= 1'b0;
        else     clk_div <= ~clk_div;
    end
    assign outclk_0 = clk_div;
endmodule
```

The module's port names (`refclk`, `rst`, `outclk_0`) match the `altera_pll` IP defaults, so the same instance signature applies if the divider is ever replaced with the hard PLL.

The board's VGA DAC (ADV7123) samples R/G/B on its own rising edge. We drive `vga_clk` as the inversion of the internal pixel clock (`hw/nml_gpu.sv:38`) so the DAC samples in the middle of the FPGA's hold window. This avoids hold-time violations into the DAC without adding a register stage.

3.2 VGA timing (`vga_timing.sv`)

`vga_timing` counts a horizontal axis from 0 to 799 and a vertical axis from 0 to 524, producing standard 640×480 @ 60 Hz parameters (`hw/vga_timing.sv:11-21`):

Axis	Active	Front porch	Sync	Back porch	Total
H	640	16	96	48	800
V	480	10	2	33	525

Table 1: VGA timing parameters for 640×480 @ 60 Hz.

`hsync` and `vsync` are active low, matching the VGA spec for this mode. `hblank` is high when `h_count` $\geq$ `640`; `vblank` is high when `v_count` $\geq$ `480`; `visible = !hblank && !vblank`.

3.3 Sprite engine

3.3.1 `sprite_eval.sv`

`sprite_eval` is a small FSM (`IDLE` → `FETCH` → `EVAL` → `DONE` → `IDLE`) that walks the 32-entry sprite table once per scanline and selects the top eight sprites by priority that intersect the *next* scanline ($y + 1$). The state machine starts when the top-level asserts `eval_strobe` at $x = 642$, two pixels into HBLANK (`hw/nml_gpu.sv:248`). Each sprite is fetched in `FETCH` and tested in `EVAL`; the test is

```
intersects = spr_active &&
             (next_scanline >= spr_y) &&
             (next_scanline <  (spr_y + 16));
```

(`hw/sprite_eval.sv:39-41`). All sprites are 16×16. The signed Y compare permits sprites to sit partly above the screen.

The insertion sort is fully unrolled (`hw/sprite_eval.sv:77-94`): each cycle, the incoming sprite is compared against eight registered slots; it lands in the first empty slot or the first slot with lower priority. Slots below the insertion point shift down by one; the lowest-priority slot is dropped if the array is full. Total walk takes 32 cycles for fetch + eval plus one cycle for `DONE`, well under the ~160-cycle HBLANK budget at 25 MHz.

3.3.2 `sprite_fetch.sv`

`sprite_fetch` is the data-mover. Its FSM (`IDLE` → `CLEAR_BUF` → `PROCESS_SPRITE` → `READ_PIXEL` → `WAIT_PIXEL` → `WRITE_PIXEL`) clears the inactive line buffer to zero (the transparent palette index), then walks the eight sorted sprites from lowest to highest priority, reading 16 palette-index bytes from the sprite ROM per sprite and writing each non-zero, on-screen pixel into the line buffer (`hw/sprite_fetch.sv:67-131`). The `READ_PIXEL` → `WAIT_PIXEL` → `WRITE_PIXEL` 3-state inner loop matches the 1-cycle read latency of the sprite ROM (`hw/nml_gpu.sv:194`).

A sprite is addressed by `sprrom_raddr = (spr_id` $\ll$ `8) + (pixel_v` $\ll$ `4) + actual_u`, where

`actual_u = spr_hflip ? (15 - pixel_u) : pixel_u` (`hw/sprite_fetch.sv:107-109`). Vertical flip is in the register layout but not wired in this revision — only `hflip` is honoured.

Walking from lowest to highest priority is a deliberate choice: each sprite's pixels overwrite whatever was written by a lower-priority sprite, so the highest-priority sprite ends up on top. The buffer-clear fills with zero (transparent) so the compositor can fall through to the tile layer where no sprite wrote.

3.3.3 Double line buffer

Two 640×8 line buffers (`linebuf_A`, `linebuf_B`, `hw/nml_gpu.sv:198-221`) form a ping-pong pair. While `sprite_fetch` fills buffer A during the HBLANK after scanline N, the compositor reads buffer B for scanline $N + 1$; on the next HBLANK, the selector toggles. The toggle fires when `hblank && x == 640` (`hw/nml_gpu.sv:209`), exactly at the transition from visible to blanking.

3.4 Compositor (`compositor.sv`)

The compositor is a 4-stage pipeline that delivers one RGB triplet per pixel. Figure 3 shows the stages.

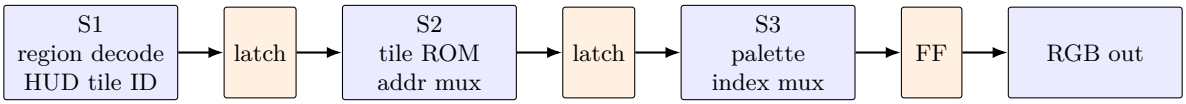

Figure 3: Compositor 4-stage pipeline. Tile ROM and palette RAM each contribute one register cycle.

The stages, with the latches between them, are (`hw/compositor.sv:75-311`):

1. **S1 (combinational):** decode the HUD region predicates (`in_hp_bar`, `in_ammo_label`, `in_ammo_digits`, …, `in_score_text`), compute the HUD label tile IDs from the local character index, and select the BCD digit nibble from `player_stats`, `hud_aux`, and `score_reg`. Compute the background tile map address from $(x[9:3], y[9:3])$.

2. **S1→S2 latch** (`hw/compositor.sv:256-268`): register the HUD tile ID, HUD tile X/Y, HP-bar fill flag, sprite pixel from the line buffer, and the background tile map data.

3. **S2 (combinational):** mux the tile ROM read address — either the background path (`tile_id, y[2:0], x[2:0]`) or the HUD glyph path (`hud_tile_id, hud_tile_y, hud_tile_x`).

4. **S2→S3 latch** (`hw/compositor.sv:280-286`): register the HP-bar and HUD-text flags so they line up with the tile ROM read data.

5. **S3 (combinational):** select the palette index — sprite pixel if non-zero, else the tile ROM byte (`hw/compositor.sv:289-292`).

6. **S3→S4 (registered output):** drive `rgb_out`. HUD layer wins in the top 16 pixels; HP bar fills $x \in [0, 191]$ green or dark red; HUD glyph pixels use white on dark grey; everywhere else, the palette RAM output drives the line.

Total latency from x, y to `rgb_out` is three to four cycles (120–160 ns at 25 MHz), well within `VGA_BLANK_N` slop.

3.5 Memories

All six FPGA-side memories are inferred. The relevant ports and depths are summarised in Table 2.

Memory	Storage	Words×Width	Bits	Used for
Sprite table A	`[63:0] [0:31]`	32 × 64	2,048	Active sprite list
Sprite table B	`[63:0] [0:31]`	32 × 64	2,048	Shadow written by HPS
Palette RAM	`[23:0] [0:255]`	256 × 24	6,144	RGB888 colour table
Tile map RAM	`[3:0][7:0] [0:1199]`	1200 × 32	38,400	80×60 tile IDs (packed)
Sprite ROM	`[7:0] [0:16383]`	16384 × 8	131,072	64 sprites × 256 B
Tile ROM	`[7:0] [0:4095]`	4096 × 8	32,768	64 tiles × 64 B
Line buffer A	`[7:0] [0:639]`	640 × 8	5,120	Sprite scanline (ping)
Line buffer B	`[7:0] [0:639]`	640 × 8	5,120	Sprite scanline (pong)

Table 2: FPGA-side memory inventory.

The sprite ROM, tile ROM, palette, and shadow sprite table are initialised from `.hex` files via `$readmemh` at synthesis (`hw/nml_gpu.sv:183-192`). The palette and sprite table are subsequently overwritten by the HPS at runtime; the ROMs stay frozen.

Tile-map storage is unusual: the register declaration is `logic [3:0][7:0] tilemap_ram [0:1199]` rather than the more obvious `logic [7:0] tilemap_ram [0:4799]`. The reason is byte-enable handling on the lightweight bridge. ARM emits `STRB` (single-byte stores) for the C driver's `nml_write_tile()` call, and the bridge translates them into a 32-bit write with `avs_byteenable` selecting one lane. With byte-organised storage, the bridge's implicit read-modify-write would see only one byte of the word and write back zeros for the other three, silently clobbering 75% of the tile map (`hw/nml_gpu.sv:138-149`). Packed four-lane storage matches Quartus's M10K byte-enable inference template; each `tilemap_ram[idx][i]` lane updates independently.

3.6 Avalon-MM slave interface (`avalon_slave_iface.sv`)

The peripheral is a 32-bit lightweight Avalon-MM slave with a 14-bit byte-address window (16 KB). The full register map is reproduced in Section 5.

Reads of the scalar registers (CTRL, STATUS, BG_SCROLL, IRQ_MASK, player state, HUD_AUX) return combinationally on the read cycle. Reads of the sprite-table, palette, and tile-map regions go through an `always_ff` register on `*_rdata_sw` lines and therefore arrive one cycle late. To make this work without breaking the master, the slave asserts `avs_waitrequest` for the first cycle of any RAM-region read and releases it on the second (`hw/avalon_slave_iface.sv:133-141`).

Writes to the CTRL register specifically handle `SWAP` by widening the single-cycle Avalon strobe into a 7-cycle counter (`swap_req_cnt`, `hw/avalon_slave_iface.sv:195-209`). The widening matters because the swap signal crosses from the 50 MHz Avalon clock into the 25 MHz pixel clock, where it is captured by a three-flop synchroniser (`hw/nml_gpu.sv:94-106`) and edge-detected to produce a

one-pulse event in the pixel-clock domain. A single 20 ns pulse on the Avalon side is too narrow for the 40 ns pixel clock to catch reliably; holding for 140 ns (seven 50 MHz cycles) makes the crossing deterministic.

3.7 Top-level integration

`nml_gpu.sv` is the integration shell. It instantiates the slave, the timing generator, sprite eval, sprite fetch, the compositor, and the PLL, and declares the inferred memories. The active and shadow sprite tables are kept inside this file rather than inside the slave because the swap logic lives here: when `swap_pending` is set and `vsync` is high, the 32-entry active table is bulk-assigned from the shadow on a single clock edge (`hw/nml_gpu.sv:109-118`).

`de1soc_top.sv` is the Phase 1 wrapper. It ties the Avalon inputs off (no master in the standalone build), wires reset to `KEY[0]`, mirrors `VGA_BLANK_N` to `LEDR[1]` as a heartbeat, and passes the VGA outputs through. The Phase 2 wrapper is `nml_gpu_hw/soc_system_top.sv`, which instantiates the Platform Designer system, exposes nml_gpu's slave on the HPS-to-FPGA lightweight bridge, and connects the same VGA pins.

4 Software Design

The software lives in `sw/` and is structured as nine translation units plus three headers. Three build variants share the same sources (Table 3).

Variant	Target	Renderer	Input	Output
`make` (default)	armhf	`render.c`	`input_real.c`	`nml_game`
`make fake-input`	armhf	`render.c`	`input_fake.c`	`nml_game_fake`
`make terminal`	host	`render_terminal.c`	`input_fake.c`	`nml_game_term`
`make native`	board native	`render.c`	`input_real.c`	`nml_game`

Table 3: Build variants (`sw/Makefile:42-72`).

The terminal build also defines `NML_TERMINAL_BUILD`, which compiles out the FPGA paths in `main.c`.

4.1 Game state machine and entity pool (`game.c`, `game.h`)

State is held in a single struct (`sw/game.h:93-133`). The active states are `STATE_PLAYING`, `STATE_LEVELUP`, and `STATE_GAMEOVER`. A fixed-size pool of 64 entities (`MAX_ENTITIES`, `sw/game.h:6`) holds the player, enemies, bullets, hazards, and ammo drops. Each entity tracks kind, active flag, position, velocity, hp, an animation phase, a TTL, a payload, and a fire cooldown (`sw/game.h:79-90`).

`game_tick()` (`sw/game.c:428-485`) dispatches on state:

- In `STATE_PLAYING`, it updates the player, runs the wave spawner, moves enemies, advances bullets, ages hazards, runs auto-attack timers, and resolves collisions. On wave clear it transitions to `STATE_LEVELUP`. On HP ≤ 0 it transitions to `STATE_GAMEOVER` and deactivates every entity including the player so the death screen has a clean tilemap to draw into.

- In `STATE_LEVELUP`, the player picks one of three offered weapons with the D-pad and confirms with the B button (`sw/game.c:416-426`). On confirm, the chosen weapon is upgraded and the next wave starts.
- In `STATE_GAMEOVER`, START restarts the game by re-calling `game_init()`.

Enemy motion (`sw/game.c:219-254`) is a per-enemy random walk in X combined with vertical pursuit of the player. The walk decision re-rolls every 12 frames (`ENEMY_DIR_FRAMES`) using a per-enemy seed (`phase`) so the cohort does not collapse into a single column where the player's bullet stream would mop them up trivially. Armed enemies also fire bullets back on a per-enemy cooldown jittered at spawn (`sw/game.c:21-23, 239-246`).

Collision resolution is AABB at 16×16 (`sw/game.c:298-312`). All kills go through `apply_damage()` (`sw/game.c:318-333`) so that score, kill counter, and ammo-drop probability are recorded uniformly across weapon paths. The drop probability is 30% per kill (`sw/game.h:144`, `AMMO_DROP_PERCENT`).

4.2 Wave system (`wave.c`, `wave.h`)

The wave table is a hard-coded array of five entries (`sw/wave.c:9-16`):

Wave	Spawn period (frames)	Total	Armed	Armed HP	Speed
1	60	6	0	1	1
2	50	8	1	2	1
3	45	10	2	2	2
4	35	12	4	2	2
5	25	16	6	3	3

After wave 5 the table clamps to its last entry, so the game can keep running indefinitely at maximum difficulty. `wave_advance()` top-ups ammo (+40, capped at 99), artillery (+2, capped at 5), and gas (+2, capped at 5) on every transition out of `STATE_LEVELUP`.

4.3 Auto-attack system (`autoatk.c`, `autoatk.h`)

Three auto-attack kinds exist (`sw/game.h:60-65`):

- **Mortar** is timer-driven. At level 1 it fires every 180 frames, decreasing by 30 frames per upgrade, floored at 60 frames (`sw/autoatk.c:10-20, 39-45`). Each shell launches upward from the player's column with random horizontal jitter and a TTL of 200 frames.
- **Artillery** is player-triggered with the L shoulder button. On fire it kills every enemy in a 16-pixel column above the player (`sw/autoatk.c:200-219`) and paints a vertical column of `TILE_BEAM` glyphs into the tile map for six frames as the visual effect.
- **Mustard gas** is player-triggered with the R shoulder. It spawns a single `ENT_HAZARD` at the player's position with a TTL of 90 frames. The cloud grows from 16×16 px to 48×32 px over the first 30 frames, holds, and shrinks over the last 15 (`sw/autoatk.c:160-178`).

Each weapon has a per-button input cooldown of 20 frames (`ABILITY_INPUT_COOLDOWN`, `sw/game.h:145`) to debounce noisy SNES shoulder buttons.

4.4 Input subsystem

The input API is a single function: `uint16_t input_read(int frame)` (`sw/input.h:6`). Either `input_real.c` (the DragonRise reader) or `input_fake.c` (the deterministic stub) is linked.

The real reader opens `/dev/input/event0` (the only evdev node that shows up for the gamepad on the DE1-SoC kernel — `joydev` is not built in, so there is no `/dev/input/js0`). It drains pending events on every call, updating an internal button bitmask and axis values (`sw/input_real.c:127-147`). The D-pad is reported as `ABS_X`/`ABS_Y`; axes are converted to four directional bits with a deadzone equal to a quarter of the reported range, queried via `EVIOCGABS` so the same code works whether the pad reports −1..1, 0..255, or signed 16-bit values.

4.5 Rendering pipeline (`render.c`, `render_terminal.c`)

`render_frame()` (`sw/render.c:342-421`) is called once per tick. It writes the entity pool into the FPGA sprite table:

- Slot 0 is always the player.
- Slots 1–31 hold active non-player entities in pool order.
- Gas hazards expand into up to four sprite slots (`sw/render.c:98-111`).
- Bullets get priority 2 (low), enemies and hazards priority 1, the player priority 0 (high).

Leftover slots are explicitly hidden by `nml_hide_sprite()` so prior frames do not ghost (`sw/render.c:407-409`). The HUD mailbox registers are updated every frame.

The terminal renderer (`sw/render_terminal.c`) mirrors the slot allocation logic exactly but prints each sprite slot to stdout as an ASCII glyph. This lets the game logic be exercised in CI and on a laptop without hardware.

4.6 Main loop (`main.c`)

`main()` (`sw/main.c:171-274`) is a fixed-timestep 60 Hz loop:

1. Read input (`input_read`).
2. Tick the game (`game_tick`).
3. Render (`render_frame` writes the sprite table).
4. Commit (`nml_commit_frame()` sets `CTRL.SWAP`, then polls `STATUS.SWAP_PENDING` until it clears or a 200,000-iteration timeout trips).
5. Sleep the remainder of the 16,666 μs frame budget.

On the host (terminal build), `nml_commit_frame()` is `#ifdef`-ed out and `nanosleep` is the sole pacing source. On the board, the SWAP poll synchronises the loop with VBLANK.

Palette initialisation happens once at startup (`init_palette_runtime()`, `sw/main.c:139-168`). The HPS writes sixteen entries covering player, enemies, projectiles, dirt, grass, and the white sprite border. This step shadows the `$readmemh` initialisation; once it runs, the software is the authoritative source of palette data.

5 Hardware–Software Interface

The peripheral occupies 16 KB starting at `0xFF200000` in HPS physical address space (the lightweight bridge base on the DE1-SoC). The userspace driver `mmap()`s the whole window via `/dev/mem` (`sw/nml_gpu.c:33-59`) and exposes typed writers.

5.1 Register map

Offset	Name	W	Acc.	Layout
`0x0000`	`CTRL`	32	R/W	`[0]=ENABLE`, `[1]=SWAP` (auto-clear), `[2]=HUD_ON`
`0x0004`	`STATUS`	32	R	`[0]=VBLANK`, `[1]=SWAP_PENDING`, `[15:8]=frame_ctr` (wired 0 at present)
`0x0008`	`BG_SCROLL`	32	R/W	`[15:0]=scroll_x` (signed), `[31:16]=scroll_y` (signed)
`0x000C`	`IRQ_MASK`	32	R/W	`[0]=VBLANK IRQ enable` (Phase 2; wired 0 here)
`0x0010`	`PLAYER_POS`	32	R/W	`[15:0]=px`, `[31:16]=py`
`0x0014`	`PLAYER_STATS`	32	R/W	`[7:0]=hp`, `[15:8]=wave` BCD (2 digits), `[31:16]=level`
`0x0018`	`SCORE`	32	R/W	`[23:0]=score` BCD (6 digits)
`0x001C`	`KILL_COUNT`	32	R/W	32-bit kill counter (HUD-only, optional)
`0x0020`	`HUD_AUX`	32	R/W	`[7:0]=ammo` BCD (2 digits), `[11:8]=art`, `[15:12]=gas`
`0x0100-0x01FF`	Sprite table	256 B	R/W	32 × 8 B entries
`0x0400-0x07FF`	Palette	1 KB	R/W	256 × 4 B; `[23:0]=RGB888`
`0x1000-0x22BF`	Tile map	4.7 KB	R/W	80×60 bytes; byte = tile ID

Table 4: Avalon-MM register map.

Each sprite-table entry is two 32-bit words:

Word	Bits	Field
W0	15:0	`x` (signed)
W0	31:16	`y` (signed)
W1	7:0	`sprite_id`
W1	15:8	`flags`
W1	23:16	`palette_off`
W1	31:24	reserved

The `flags` byte packs (`sw/nml_gpu.h:58-69`): bit 2 hflip, bit 3 vflip, bits 6:4 priority (0 = highest), bit 7 ACTIVE. The `ACTIVE` bit is the flag that `sprite_eval` actually checks at bit 47 of the 64-bit packed word; the original design used sentinel coordinates exclusively, but the hardware now also honours the explicit active bit (`sw/nml_gpu.h:12-16`, `hw/sprite_eval.sv:32-34`).

5.2 Frame-commit handshake

The frame-commit sequence is:

1. CPU writes the sprite table, palette, tile map, player state, and HUD aux fields. All writes land in shadow registers/RAMs immediately.
2. CPU writes `CTRL.SWAP = 1`. The slave widens the strobe to seven 50 MHz cycles (`hw/avalon_slave_iface.sv:199-209`), so the pixel clock can synchronise the request through three flip-flops and capture an edge (`hw/nml_gpu.sv:94-106`). `SWAP` auto-clears in the same cycle it was written.
3. At the next VSYNC, the 32-entry sprite table is bulk-assigned from shadow to active in one clock edge (`hw/nml_gpu.sv:113-118`). `swap_pending` clears.
4. CPU polls `STATUS.SWAP_PENDING` until it falls or the 200,000-iteration timeout trips (`sw/nml_gpu.c:179-190`).

The palette and tile map do not go through the swap handshake. Both have one logical bank that is written in the Avalon clock domain and read in the pixel clock domain. Visible artefacts from mid-frame writes are possible in theory; in practice palette and tile updates are rare, so we left it that way.

5.3 Byte-store semantics for the tile map

`nml_write_tile()` issues a single-byte `STRB` to the tile-map region (`sw/nml_gpu.c:92-99`). The lightweight bridge translates this into a 32-bit Avalon write with `avs_byteenable` set to one nibble. On the slave side, the packed-byte tile-map storage `logic [3:0][7:0] tilemap_ram` exposes four byte-enable lanes per M10K word and writes only the addressed lane (`hw/nml_gpu.sv:158-162`). Reads return the full word; the pixel side selects one of the four bytes per pixel based on `tilemap_raddr[1:0]` (`hw/nml_gpu.sv:169-173`).

This packed layout is non-obvious but is the difference between a correct tile map and one where only every fourth column receives the write.

6 Build and Toolchain

6.1 Hardware build

Quartus Prime Lite 21.1 is the synthesis and place-and-route tool. Two flows exist.

Phase 1 (`hw/quartus/`). A standalone bitstream that places `nml_gpu` inside `de1soc_top` with the Avalon slave tied off. The four `.hex` files generated by `hw/gen_rom.py` are picked up via `$readmemh` at synthesis time, courtesy of `SEARCH_PATH ".."` in the QSF (`hw/quartus/nml_gpu.qsf`). Compilation runs in 63 s on a lab machine and produces `output_files/nml_gpu.sof` (for JTAG) and `output_files/nml_gpu.rbf` (raw binary for SD-card boot).

Phase 2 (`nml_gpu_hw/`). A Platform Designer system with an HPS instance, a clock bridge, an AXI-to-Avalon adapter on the HPS-to-FPGA lightweight bridge, and the nml_gpu component as an Avalon slave. The component descriptor lives in `nml_gpu_hw.tcl` and sets the device-tree metadata (`compatible = "csee4840,nml_gpu-1.0"`) so the kernel exposes the peripheral at `/proc/device-tree/sopc@0/bridge@0xc0000000/vga@0x100000000/`.

6.2 ROM generation (`hw/gen_rom.py`)

`gen_rom.py` is a 777-line Python script that emits four hex files: `sprite_rom.hex` (16 384 bytes), `tile_rom.hex` (4 096 bytes), `palette.hex` (256 RGB888 entries), and `sprite_table.hex` (32 initial sprite slots).

6.3 Software build

The userspace binary cross-compiles with `arm-linux-gnueabihf-gcc` (`sw/Makefile:23`). It links statically (`-static`) so it runs on the board without a matching glibc, and uses standard POSIX APIs (`/dev/mem`, `mmap`, evdev, `nanosleep`, `clock_gettime`). No threading; everything is in one process loop.

6.4 Boot and SD-card image

The Phase 2 image follows the standard DE1-SoC Linux boot flow: preloader → U-Boot → kernel (Linux 4.19 from `altera-fpga/linux-socfpga`) → root filesystem on the ext4 partition. The FPGA RBF and DTB sit on the FAT boot partition. U-Boot loads the bitstream into the FPGA fabric before launching the kernel, so the smoke-test scene is already on the VGA monitor by the time userspace comes up. After login, the operator copies `nml_game` onto the board and runs it as root.

7 Testing and Verification

Three test rungs:

- **Phase 0 (host).** `cd sw && make terminal && ./nml_game_term` runs the entire game loop without hardware, printing every frame's sprite table to stdout. The deterministic input stub exercises every weapon path on a fixed cycle.
- **Phase 1 (FPGA only).** With the `.sof` loaded over JTAG, the monitor shows the smoke-test scene (dark battlefield, green player square, two red enemies, a yellow bullet) within one second. This proves the entire pixel pipeline.
- **Phase 2 (HPS-integrated).** With the RBF and DTB on the SD card, the kernel exposes the peripheral, and `./nml_game` mmaps it. The game loop runs at 60 Hz, the controller drives the player, the VGA monitor displays the rendered frame, and the terminal logs wave starts and clears.

8 Results

8.1 Resource utilisation

From `hw/quartus/output_files/nml_gpu.fit.summary` (Phase 1 build, Cyclone V SE 5CSEMA5F31C6):

Resource	Used	Available	Fraction
ALMs	205	32,070	$< 1\%$
Registers	365	—	—
Pins	54	457	12%
Block memory bits	86,144	4,065,280	2%
M10K blocks	14	397	4%
DSP blocks	0	87	0%
PLLs	0	6	0%

Table 5: Phase 1 resource utilisation.

Total compile time: 63 s (Analysis & Synthesis 13 s, Fitter 30 s, Assembler 12 s, Timing Analyzer 8 s).

8.2 Timing analysis

`hw/quartus/output_files/nml_gpu.sta.summary` reports negative slack on the slow-1100mV-85°C corner:

Corner	Clock	Setup slack	TNS
Slow 1100mV 85°C	`clk_div`	−5.111 ns	−1409.97 ns
Slow 1100mV 85°C	`CLOCK_50`	−3.506 ns	−3.506 ns
Fast 1100mV 85°C	`clk_div`	−2.654 ns	−725.39 ns

All hold checks pass. The dominant violation is a minimum-pulse-width report on the divided clock register. The pixel-clock domain runs at 25 MHz (40 ns period), but the timing analyser treats the divider register as if it were producing a much higher-frequency clock; the combination of `posedge` and `negedge` from the alternating toggle makes the analyser see a 20 ns half-period. On real silicon the register is promoted to the global clock network and the duty cycle is exact, so the divider works as intended — but the analyser cannot see that, and reports failure.

The cleaner alternative is to drive the pixel clock from an `altera_pll` IP instance configured for 25 MHz, which the analyser models correctly. The Phase 1 board passes timing visually at lab temperature on the divider, and the divider's port signature matches the IP defaults, so the swap remains a drop-in if the analyser numbers ever need to be clean.

8.3 Frame timing

In the deployed (Phase 2) configuration, `top -n 1 -b` while `nml_game` is running shows CPU usage under 5%. The loop spends most of its time in `nanosleep` or polling `STATUS.SWAP_PENDING` inside `nml_commit_frame()`.

9 Challenges and Resolutions

9.1 Tile-map byte-enable RMW

The original tile-map storage was `logic [7:0] tilemap_ram [0:4799]` — one byte per word. The Linux driver's `nml_write_tile()` issues an ARM `STRB` for each tile update. The HPS-to-FPGA lightweight bridge sets `avs_byteenable` to one nibble and the Avalon slave's effective behaviour was a read-modify-write: read the 32-bit word, mask in the new byte, write back. With byte-organised storage, the slave only returned one byte from each read, so the bridge masked three zero bytes into the other lanes and silently clobbered them. The visible symptom was that only every fourth tile column showed new content; the rest stayed at reset value 0.

The fix is the packed `logic [3:0][7:0] tilemap_ram [0:1199]` storage plus a per-lane write in `hw/nml_gpu.sv:158-162`. Each lane updates independently, and reads return all four bytes, so the bridge's read-modify-write preserves the untouched bytes.

9.2 Single-cycle `eval_done` against multi-cycle `CLEAR_BUF`

`sprite_eval` finishes its 32-sprite walk in roughly 33 cycles and asserts `eval_done` for one cycle. `sprite_fetch` is in `CLEAR_BUF` for 640 cycles (one write per pixel) and only checks `eval_done` once, at the end of the buffer clear. If the eval pulse fires while fetch is still mid-clear, the pulse is gone by the time fetch samples it.

The first fix attempted was to edge-detect a transition on `active_mask`. That worked for the smoke-test scene because the active set changes between non-sprite and sprite rows. It silently failed for the game's much busier scenes: 15 of every 16 scanlines inside a 16-pixel sprite see the *same* active set, so no transition fires.

The current fix is a sustain latch: `eval_done_r` is set when the pulse arrives and cleared at the next `eval_strobe`. `sprite_fetch` samples `eval_done_r` and is guaranteed to see it set after the first eval of a line, regardless of when fetch happens to check (`hw/nml_gpu.sv:264-270`).

9.3 CDC on `ctrl_swap_req`

`CTRL.SWAP` is written from the 50 MHz Avalon clock and consumed in the 25 MHz pixel clock. The straight single-cycle pulse from the slave was 20 ns wide; a two-flop synchroniser at 25 MHz samples every 40 ns, so it could easily miss the pulse entirely.

The slave now widens the request: a 3-bit counter loaded with 7 on each SWAP write holds `ctrl_swap_req` high for seven 50 MHz cycles (140 ns), which is enough that a three-flop chain on the pixel side captures the rising edge deterministically.

9.4 Pixel-clock duty cycle

The divide-by-2 register works on silicon but fails the analyser's minimum-pulse-width check. We documented the failure mode; the divider's port names match the `altera_pll` IP defaults, so any later replacement is a drop-in.

10 Conclusion

The Phase 1 smoke-test bitstream works on hardware: 640×480 @ 60 Hz, tile + sprite + palette + HUD pipeline, full VGA pin set. The Phase 2 HPS-integrated build boots Linux, exposes the peripheral on the lightweight bridge, runs the C game at 60 Hz, and reads SNES controller input via evdev. Player movement, firing, enemy spawning, wave progression, level-up, mortar timers, artillery, gas clouds, ammo drops, and the HUD numerics all function in the deployed configuration.

References

1. Terasic Inc., *DE1-SoC User Manual*, Revision 1.2.4.
2. Intel/Altera, *Cyclone V Device Handbook* (Volume 1).
3. Intel/Altera, *Avalon Interface Specifications*, MNL-AVABUSREF.
4. Analog Devices, *ADV7123 Triple High-Speed Video DAC Datasheet.*
5. VESA, *VGA 640×480 @ 60 Hz timing specification.*
6. Linux kernel evdev documentation: `Documentation/input/event-codes.rst`.
7. `lab3-reference/CSEE4840W-Lab3` — VGA ball lab providing the timing-generator skeleton.

Appendix: Source Listings

This appendix reproduces the complete source for the hardware and software components of the project, as built and run on the DE1-SoC.

Hardware (`hw/`)

`de1soc_top.sv`

```
// de1soc_top.sv -- Phase 1 top for the smoke-test bitstream. No HPS, no Qsys.
// nml_gpu boots with its memories pre-loaded from the .hex files via
// $readmemh, so the bitstream alone draws a VGA frame. Replaced by
// soc_system_top in Phase 2. VGA pins are in nml_gpu.qsf.
//
// KEY[0]    active-low reset
// LEDR[0]   tied 1 (FPGA powered)
// LEDR[1]   high during VGA visible region
// LEDR[9:2] SW[9:2] passthrough

`timescale 1ns / 1ps

module de1soc_top (
    input  logic        CLOCK_50,
    input  logic [3:0]  KEY,
    input  logic [9:0]  SW,
    output logic [9:0]  LEDR,

    // VGA (24-bit DAC + sync)
    output logic [7:0]  VGA_R,
    output logic [7:0]  VGA_G,
    output logic [7:0]  VGA_B,
```

```
    output logic        VGA_HS,
    output logic        VGA_VS,
    output logic        VGA_BLANK_N,
    output logic        VGA_SYNC_N,
    output logic        VGA_CLK
);

    // Active-low reset on KEY[0]; KEY is active-low on the DE1-SoC.
    logic reset_n;
    assign reset_n = KEY[0];

    // Tie the Avalon slave off (no master in standalone mode).
    logic [31:0] avs_readdata;
    /* verilator lint_off UNUSEDSIGNAL */
    logic        avs_waitrequest_unused;
    /* verilator lint_on UNUSEDSIGNAL */

    nml_gpu gpu_inst (
        .clk            (CLOCK_50),
        .reset_n        (reset_n),

        // No master is driving the Avalon slave in standalone mode; tie inputs.
        .avs_address    (14'd0),
        .avs_read       (1'b0),
        .avs_write      (1'b0),
        .avs_writedata  (32'd0),
        .avs_byteenable (4'b0000),
        .avs_readdata   (avs_readdata),
        .avs_waitrequest(avs_waitrequest_unused),

        .vga_r          (VGA_R),
        .vga_g          (VGA_G),
        .vga_b          (VGA_B),
        .vga_hs         (VGA_HS),
        .vga_vs         (VGA_VS),
        .vga_blank_n    (VGA_BLANK_N),
        .vga_sync_n     (VGA_SYNC_N),
        .vga_clk        (VGA_CLK)
    );

    // Heartbeat / status LEDs so the team can confirm the bitstream is live
    // even before plugging in a monitor.
    assign LEDR[0]   = 1'b1;
    assign LEDR[1]   = VGA_BLANK_N;     // ON while pixels are visible
    assign LEDR[9:2] = SW[9:2];

endmodule
```

nml_gpu.sv

```
`timescale 1ns / 1ps

module nml_gpu (
    // Avalon-MM slave (Lightweight bridge, 50 MHz)
    input  logic        clk,
    input  logic        reset_n,
    input  logic [13:0] avs_address,
```

```
    input  logic        avs_read,
    input  logic        avs_write,
    input  logic [31:0] avs_writedata,
    input  logic [3:0]  avs_byteenable,

    output logic [31:0] avs_readdata,
    output logic        avs_waitrequest, // Tied 0; single-cycle

    // VGA DAC Output
    output logic [7:0]  vga_r, vga_g, vga_b,
    output logic        vga_hs, vga_vs,
    output logic        vga_blank_n, vga_sync_n, vga_clk
);

    assign vga_sync_n = 1'b0; // Not used for our DAC

    // ============================================================================
    // 1. CLOCK GENERATION (50 MHz -> 25 MHz pixel clock)
    // ============================================================================
    logic pix_clk;

    pll_25mhz pll_inst (
        .refclk  (clk),
        .rst     (~reset_n),
        .outclk_0(pix_clk)
    );

    // Invert so the ADV7123 DAC samples R/G/B mid-cycle (on its rising edge,
    // which is now pix_clk's falling edge), well after the FPGA has updated
    // them on pix_clk's rising edge. Avoids hold-time violations at the DAC.
    assign vga_clk = ~pix_clk;

    // ============================================================================
    // 2. INTERNAL WIRES
    // ============================================================================
    // VGA Timing
    logic [9:0] x, y;
    logic visible, hsync, vsync, vblank, hblank;

    // Avalon / Register File
    logic ctrl_enable, ctrl_swap_req, ctrl_hud_on;
    logic [31:0] bg_scroll, player_pos, player_stats, score, kill_count;
    logic [31:0] hud_aux;
    logic [12:0] mem_waddr;
    logic sprtab_we, palette_we, tilemap_we;
    logic [31:0] mem_wdata;

    // Status tracking
    logic swap_pending;

    // Sprite Eval -> Sprite Fetch
    logic [4:0]  eval_sprtab_raddr;
    logic [63:0] eval_sprtab_rdata;
    logic [63:0] line_sprites [0:7];
    logic [7:0]  active_mask;
    logic        eval_strobe;
    logic        eval_done;

    // Line Buffer Control
```

```
    logic linebuf_sel; // 0 = A fills/B drains, 1 = B fills/A drains

    // ============================================================================
    // 3. MEMORIES (Inferred M10K RAMs)
    // ============================================================================
    // A. Sprite Table (Double Buffered internally for tear-free rendering)
    logic [63:0] sprite_table_active [0:31];
    logic [63:0] sprite_table_shadow [0:31];
    logic [31:0] sprtab_rdata_sw;

    always_ff @(posedge clk) begin
        // Avalon Write to Shadow
        if (sprtab_we) begin
            // 32-bit writes into 64-bit memory slots
            if (mem_waddr[0] == 1'b0) sprite_table_shadow[mem_waddr[5:1]][31:0]  <= mem_wdata;
            else                      sprite_table_shadow[mem_waddr[5:1]][63:32] <= mem_wdata;
        end
        // Avalon Read from Shadow
        sprtab_rdata_sw <= (mem_waddr[0] == 1'b0) ? sprite_table_shadow[mem_waddr[5:1]][31:0] :
                                                    sprite_table_shadow[mem_waddr[5:1]][63:32];
    end

    // Cross ctrl_swap_req from clk (50 MHz) into pix_clk (25 MHz). The Avalon
    // side now holds the request for several clk cycles, so a 2-FF synchroniser
    // here will reliably catch it; we then edge-detect to set swap_pending
    // exactly once per request rather than continuously while the source is
    // held high.
    logic ctrl_swap_req_sync_a, ctrl_swap_req_sync_b, ctrl_swap_req_sync_c;
    always_ff @(posedge pix_clk or negedge reset_n) begin
        if (!reset_n) begin
            ctrl_swap_req_sync_a <= 1'b0;
            ctrl_swap_req_sync_b <= 1'b0;
            ctrl_swap_req_sync_c <= 1'b0;
        end else begin
            ctrl_swap_req_sync_a <= ctrl_swap_req;
            ctrl_swap_req_sync_b <= ctrl_swap_req_sync_a;
            ctrl_swap_req_sync_c <= ctrl_swap_req_sync_b;
        end
    end
    wire ctrl_swap_req_pulse = ctrl_swap_req_sync_b && !ctrl_swap_req_sync_c;

    // Swap shadow to active on VSYNC if requested
    always_ff @(posedge pix_clk or negedge reset_n) begin
        if (!reset_n) swap_pending <= 1'b0;
        else begin
            if (ctrl_swap_req_pulse) swap_pending <= 1'b1;
            if (swap_pending && vsync) begin
                for (int i=0; i<32; i++) sprite_table_active[i] <= sprite_table_shadow[i];
                swap_pending <= 1'b0;
            end
        end
    end

    // Connect sprite_eval to the active sprite table. sprite_table_active is
    // written in this same pix_clk domain on the swap pulse above, so a
    // combinational read here is in-domain. Drives the 32-to-1 mux that
    // feeds sprite_eval's FETCH/EVAL pipeline.
    always_comb eval_sprtab_rdata = sprite_table_active[eval_sprtab_raddr];
```

```
// B. Palette RAM
logic [23:0] palette_ram [0:255];
logic [7:0]  palette_raddr;
logic [23:0] palette_rdata;
logic [31:0] palette_rdata_sw;

always_ff @(posedge clk) begin
    if (palette_we) palette_ram[mem_waddr[7:0]] <= mem_wdata[23:0];
    palette_rdata_sw <= {8'd0, palette_ram[mem_waddr[7:0]]};
end
always_ff @(posedge pix_clk) palette_rdata <= palette_ram[palette_raddr];

// C. Tile Map RAM -- packed [4][8] storage (1200 x 4 B = 4800 B).
// Background: the HPS-LW bridge implements ARM STRB (byte writes) as a
// {word-read, byte-modify, word-write} at the slave with byteenable=1111.
// With byte-organized storage we could only return 1 byte per word read,
// so the bridge saw {0,0,0,byte0} and wrote {0,0,0,byte0}+mod back,
// silently dropping bytes 1/2/3 at every word (display showed only every
// 4th column with new tile; others stayed at reset value 0 = brown).
//
// Packed [3:0][7:0] storage is Quartus's recommended template for
// M10K with byteena_a inference: each `ram[idx][i]` write maps to one
// byte lane; the full-word read returns all 4 bytes so bridge RMW
// preserves untouched bytes.
logic [3:0][7:0] tilemap_ram [0:1199];
logic [12:0]     tilemap_raddr;
logic [7:0]      tilemap_rdata;
logic [31:0]     tilemap_rdata_sw;
logic [3:0][7:0] tilemap_word_pix;
logic [1:0]      tilemap_raddr_lo_r;

always_ff @(posedge clk) begin
    if (tilemap_we) begin
        for (int i = 0; i < 4; i++) begin
            if (avs_byteenable[i])
                tilemap_ram[mem_waddr[12:2]][i] <= mem_wdata[i*8 +: 8];
        end
    end
    tilemap_rdata_sw <= tilemap_ram[mem_waddr[12:2]];
end

// Pix-clk read: register the word + low addr bits, then index-select the
// byte. Same 1-cycle raddr->rdata latency as the original byte RAM.
always_ff @(posedge pix_clk) begin
    tilemap_word_pix   <= tilemap_ram[tilemap_raddr[12:2]];
    tilemap_raddr_lo_r <= tilemap_raddr[1:0];
end
assign tilemap_rdata = tilemap_word_pix[tilemap_raddr_lo_r];

// D. Sprite ROM & Tile ROM ($readmemh initialization)
logic [7:0] sprite_rom [0:16383];
logic [7:0] tile_rom   [0:4095];
logic [13:0] sprrom_raddr;
logic [7:0]  sprrom_rdata;
logic [11:0] tilerom_raddr;
logic [7:0]  tilerom_rdata;

initial begin
    $readmemh("sprite_rom.hex",    sprite_rom);
```

```
        $readmemh("tile_rom.hex",      tile_rom);
        // Smoke-test pre-init: palette + initial sprite table.
        // Once the C driver on the HPS writes these regions, runtime values
        // override the init data on the next SWAP.
        $readmemh("palette.hex",       palette_ram);
        $readmemh("sprite_table.hex",  sprite_table_active);
        $readmemh("sprite_table.hex",  sprite_table_shadow);
    end

    always_ff @(posedge pix_clk) sprrom_rdata <= sprite_rom[sprrom_raddr];
    always_ff @(posedge pix_clk) tilerom_rdata <= tile_rom[tilerom_raddr];

    // E. Double Line Buffers (Ping-Pong)
    logic [7:0] linebuf_A [0:639];
    logic [7:0] linebuf_B [0:639];

    logic [9:0] fetch_waddr;
    logic [7:0] fetch_wdata;
    logic       fetch_we;
    logic [9:0] comp_raddr;
    logic [7:0] comp_rdata;

    always_ff @(posedge pix_clk or negedge reset_n) begin
        if (!reset_n) linebuf_sel <= 1'b0;
        else if (hblank && x == 640) linebuf_sel <= ~linebuf_sel; // Toggle at end of visible
    line
    end

    // Ping-Pong Routing
    always_ff @(posedge pix_clk) begin
        // Buffer A
        if (linebuf_sel == 1'b0 && fetch_we) linebuf_A[fetch_waddr] <= fetch_wdata;
        // Buffer B
        if (linebuf_sel == 1'b1 && fetch_we) linebuf_B[fetch_waddr] <= fetch_wdata;
    end

    // Compositor reads from the opposite buffer
    assign comp_rdata = (linebuf_sel == 1'b1) ? linebuf_A[comp_raddr] : linebuf_B[comp_raddr];

    // ============================================================================
    // 4. MODULE INSTANTIATIONS
    // ============================================================================
    avalon_slave_iface avalon_inst (
        .clk(clk), .reset_n(reset_n),
        .avs_address(avs_address), .avs_read(avs_read), .avs_write(avs_write),
        .avs_writedata(avs_writedata), .avs_byteenable(avs_byteenable),
        .avs_readdata(avs_readdata), .avs_waitrequest(avs_waitrequest),
        .vblank_in(vblank), .swap_pending_in(swap_pending), .frame_ctr_in(8'd0),
        .ctrl_enable(ctrl_enable), .ctrl_swap_req(ctrl_swap_req), .ctrl_hud_on(ctrl_hud_on),
        .bg_scroll(bg_scroll),
        .sprtab_we(sprtab_we), .palette_we(palette_we), .tilemap_we(tilemap_we),
        .mem_waddr(mem_waddr), .mem_wdata(mem_wdata),
        .sprtab_rdata_sw(sprtab_rdata_sw), .palette_rdata_sw(palette_rdata_sw), .
    tilemap_rdata_sw(tilemap_rdata_sw),
        .player_pos(player_pos), .player_stats(player_stats), .score_reg(score), .kill_count(
    kill_count),
        .hud_aux(hud_aux)
    );
```

```
    vga_timing timing_inst (
        .pix_clk(pix_clk), .reset_n(reset_n),
        .x(x), .y(y),
        .visible(visible), .hsync(hsync), .vsync(vsync), .vblank(vblank), .hblank(hblank)
    );

    // Strobe eval logic slightly after HBLANK starts to ensure next line is ready
    assign eval_strobe = (x == 642);

    logic eval_done_pulse;
    sprite_eval eval_inst (
        .pix_clk(pix_clk), .reset_n(reset_n),
        .next_scanline(y + 10'd1), .eval_strobe(eval_strobe),
        .sprtab_raddr(eval_sprtab_raddr), .sprtab_rdata(eval_sprtab_rdata),
        .line_sprites(line_sprites), .active_mask(active_mask),
        .eval_done(eval_done_pulse)
    );

    // Sustain the one-cycle DONE pulse from sprite_eval until the next
    // eval_strobe, so sprite_fetch reliably observes "eval finished" no
    // matter when in CLEAR_BUF it samples the signal. The previous version
    // edge-detected on active_mask transitions, which silently failed for
    // the 15-of-16 scanlines where the same set of sprites stays active.
    logic eval_done_r;
    always_ff @(posedge pix_clk or negedge reset_n) begin
        if (!reset_n)             eval_done_r <= 1'b0;
        else if (eval_strobe)     eval_done_r <= 1'b0;
        else if (eval_done_pulse) eval_done_r <= 1'b1;
    end
    assign eval_done = eval_done_r;

    sprite_fetch fetch_inst (
        .pix_clk(pix_clk), .reset_n(reset_n),
        .hblank(hblank), .next_scanline(y + 10'd1),
        .eval_done(eval_done), .active_mask(active_mask), .line_sprites(line_sprites),
        .sprrom_raddr(sprrom_raddr), .sprrom_rdata(sprrom_rdata),
        .linebuf_waddr(fetch_waddr), .linebuf_wdata(fetch_wdata), .linebuf_we(fetch_we)
    );

    logic [23:0] rgb_out;
    compositor comp_inst (
        .pix_clk(pix_clk), .reset_n(reset_n),
        .x(x), .y(y), .visible(visible),
        .scroll_x(bg_scroll[15:0]), .scroll_y(bg_scroll[31:16]),
        // HUD overlay routing
        .hud_on(ctrl_hud_on),
        .player_stats(player_stats),
        .score_reg(score),
        .hud_aux(hud_aux),
        .tilemap_raddr(tilemap_raddr), .tilemap_rdata(tilemap_rdata),
        .tilerom_raddr(tilerom_raddr), .tilerom_rdata(tilerom_rdata),
        .linebuf_raddr(comp_raddr), .linebuf_rdata(comp_rdata),
        .palette_raddr(palette_raddr), .palette_rdata(palette_rdata),
        .rgb_out(rgb_out)
    );

    // Final Output Routing
    assign vga_r = ctrl_enable ? rgb_out[23:16] : 8'd0;
    assign vga_g = ctrl_enable ? rgb_out[15:8]  : 8'd0;
```

```
    assign vga_b = ctrl_enable ? rgb_out[7:0]   : 8'd0;
    assign vga_hs = hsync;
    assign vga_vs = vsync;
    assign vga_blank_n = visible;

endmodule
```

avalon_slave_iface.sv

```
// avalon_slave_iface.sv
// Avalon-MM lightweight slave for nml_gpu.
// 16 KB address window (14-bit byte address).
//
// Register map (byte offsets):
//   0x0000  CTRL         R/W  bit0=ENABLE, bit1=SWAP (auto-clear), bit2=HUD_ON
//   0x0004  STATUS       R    bit0=VBLANK, bit1=SWAP_PENDING, [15:8]=frame_ctr
//   0x0008  BG_SCROLL    R/W  [15:0]=scroll_x signed, [31:16]=scroll_y signed
//   0x000C  IRQ_MASK     R/W  bit0=VBLANK IRQ enable (Phase 2; wired 0 here)
//   0x0010  PLAYER_POS   R/W  [15:0]=px, [31:16]=py
//   0x0014  PLAYER_STATS R/W  [7:0]=hp, [15:8]=wave, [31:16]=level
//   0x0018  SCORE        R/W  32-bit score
//   0x001C  KILL_COUNT   R/W  32-bit kills
//   0x0020  HUD_AUX      R/W  [7:0]=ammo BCD (2 digits), [11:8]=art charges
//                             (4-bit BCD), [15:12]=gas charges (4-bit BCD)
//                             0x0024-0x002F reserved (region_aux returns 0)
//   0x0100-0x01FF  Sprite table (32 x 8 B = 256 B); SW reads return shadow
//   0x0400-0x07FF  Palette     (256 x 4 B = 1 KB)
//   0x1000-0x22BF  Tile map    (4800 B)
//
// Writes land in shadow immediately.  Sprite table commits on next VSYNC
// after CTRL.SWAP=1. Palette and tile map use a per-region dirty flag.

module avalon_slave_iface (
    input  logic        clk,
    input  logic        reset_n,

    // Avalon-MM slave
    input  logic [13:0] avs_address,    // byte address within 16 KB window
    input  logic        avs_read,
    input  logic        avs_write,
    input  logic [31:0] avs_writedata,
    input  logic [3:0]  avs_byteenable,
    output logic [31:0] avs_readdata,
    output logic        avs_waitrequest,

    // Status inputs from rest of design
    input  logic        vblank_in,
    input  logic        swap_pending_in,
    input  logic [7:0]  frame_ctr_in,

    // Control outputs
    output logic        ctrl_enable,
    output logic        ctrl_swap_req,   // one-cycle pulse on SWAP write
    output logic        ctrl_hud_on,
    output logic [31:0] bg_scroll,

    // Memory write fan-out
```

```
    output logic        sprtab_we,
    output logic        palette_we,
    output logic        tilemap_we,
    output logic [12:0] mem_waddr,      // max of the three regions
    output logic [31:0] mem_wdata,

    // Read-back from shadow RAMs (for SW reads of sprite/palette/tilemap)
    input  logic [31:0] sprtab_rdata_sw,
    input  logic [31:0] palette_rdata_sw,
    input  logic [31:0] tilemap_rdata_sw,

    // Player state registers -> HUD overlay
    output logic [31:0] player_pos,
    output logic [31:0] player_stats,
    output logic [31:0] score_reg,
    output logic [31:0] kill_count,

    // HUD auxiliary register (ammo + ability charges) -> compositor
    output logic [31:0] hud_aux
);

    // ------------------------------------------------------------------------
    // Control registers
    // ------------------------------------------------------------------------
    logic [31:0] ctrl_reg;
    logic [31:0] bg_scroll_reg;
    logic [31:0] irq_mask_reg;
    logic [31:0] player_pos_reg;
    logic [31:0] player_stats_reg;
    logic [31:0] score_reg_int;
    logic [31:0] kill_count_reg;
    logic [31:0] hud_aux_reg;

    assign ctrl_enable = ctrl_reg[0];
    assign ctrl_hud_on = ctrl_reg[2];
    assign bg_scroll   = bg_scroll_reg;
    assign player_pos  = player_pos_reg;
    assign player_stats= player_stats_reg;
    assign score_reg   = score_reg_int;
    assign kill_count  = kill_count_reg;
    assign hud_aux     = hud_aux_reg;

    // ctrl_swap_req: SWAP request held for several clk cycles so the pix_clk
    // domain (running at clk/2) reliably samples it. A single-cycle pulse
    // (~20 ns at 50 MHz) is too narrow for the 40 ns pix_clk to catch through
    // a 2-FF synchronizer; widening it here removes the CDC race.
    logic [2:0] swap_req_cnt;
    assign ctrl_swap_req = (swap_req_cnt != 3'd0);

    // ------------------------------------------------------------------------
    // Address decode
    // ------------------------------------------------------------------------
    // Regions (byte addresses):
    //   [13:4] == 0 -> scalar registers (0x000-0x00F base)
    //   Specifically: 0x00-0x0F is ctrl/status, 0x10-0x1F is player state
    //   0x0100-0x01FF -> sprite table
    //   0x0400-0x07FF -> palette
    //   0x1000-0x22BF -> tile map
```

```
    logic region_ctrl;       // 0x0000-0x000F
    logic region_player;     // 0x0010-0x001F
    logic region_aux;        // 0x0020-0x002F (HUD_AUX + reserved siblings)
    logic region_sprtab;     // 0x0100-0x01FF
    logic region_palette;    // 0x0400-0x07FF
    logic region_tilemap;    // 0x1000-0x22BF

    assign region_ctrl    = (avs_address[13:4] == 10'h000);
    assign region_player  = (avs_address[13:4] == 10'h001);
    assign region_aux     = (avs_address[13:4] == 10'h002);
    assign region_sprtab  = (avs_address[13:8] == 6'h01);          // 0x100-0x1FF
    assign region_palette = (avs_address[13:10] == 4'h1) &&
                            (avs_address[9:8]  != 2'b00 ||
                             avs_address[13:10] == 4'h1);           // 0x400-0x7FF
    assign region_tilemap = (avs_address >= 14'h1000) &&
                            (avs_address <  14'h22C0);              // 0x1000-0x22BF

    // ------------------------------------------------------------------------
    // Read latency: RAM regions (sprtab/palette/tilemap) drive their
    // *_rdata_sw signals via an always_ff register, so the data is valid one
    // clk after avs_read goes high. Scalar registers (CTRL/STATUS/...) are
    // driven combinationally. Hold avs_waitrequest high for the first cycle
    // of any RAM-region read so the master captures the registered data on
    // the cycle it actually arrives, not the previous cycle's idle latch.
    // ------------------------------------------------------------------------
    logic ram_read_active;
    logic ram_read_seen;
    assign ram_read_active = avs_read && (region_sprtab || region_palette || region_tilemap);
    always_ff @(posedge clk or negedge reset_n) begin
        if (!reset_n)            ram_read_seen <= 1'b0;
        else if (!ram_read_active) ram_read_seen <= 1'b0;
        else                       ram_read_seen <= 1'b1;
    end
    assign avs_waitrequest = ram_read_active && !ram_read_seen;

    // Sprite table: 32 entries x 8B; SW address [7:2] = word index (0-63 words -> 32 entries x
     2 words)
    // Palette:      256 entries x 4B; SW address [9:2] = entry index
    // Tile map:     4800B; SW address [12:0] within region

    // Memory address and write-enable fan-out.
    //
    // The slave is configured with Address Units = SYMBOLS in Platform
    // Designer, so avs_address[13:0] is a byte address. The downstream
    // memory consumers in nml_gpu.sv expect a *word* index for sprtab and
    // palette, so we shift by 2 here. mem_waddr is computed whenever a
    // region matches (read or write) so sprtab_rdata_sw / palette_rdata_sw /
    // tilemap_rdata_sw return the correct location on devmem2 reads, not
    // just on writes.
    always_comb begin
        sprtab_we  = 1'b0;
        palette_we = 1'b0;
        tilemap_we = 1'b0;
        mem_waddr  = '0;
        mem_wdata  = avs_writedata;

        if (region_sprtab) begin
            mem_waddr = {7'b0, avs_address[7:2]};  // 6-bit word index (0..63 -> 32 sprites x 2
     words)
```

```
            if (avs_write) sprtab_we = 1'b1;
        end else if (region_palette) begin
            mem_waddr = {5'b0, avs_address[9:2]};  // 8-bit palette entry index
            if (avs_write) palette_we = 1'b1;
        end else if (region_tilemap) begin
            // Byte offset within the tile-map region (avs_address - 0x1000).
            // Packed as {avs_address[13], avs_address[11:0]}: for addresses
            // 0x1000-0x1FFF, bit 13=0 and bit 12=1 so this gives 0x000-0xFFF.
            // For 0x2000-0x22BF, bit 13=1 and bit 12=0 so this gives 0x1000-
            // 0x12BF. Truncating to [12:0] (the old behavior) aliased the
            // upper half on top of the lower half and clobbered rows 0..8.
            mem_waddr = {avs_address[13], avs_address[11:0]};
            if (avs_write) tilemap_we = 1'b1;
        end
    end

    // ------------------------------------------------------------------------
    // Write logic for scalar registers
    // ------------------------------------------------------------------------
    always_ff @(posedge clk or negedge reset_n) begin
        if (!reset_n) begin
            ctrl_reg         <= 32'h0000_0001;  // ENABLE=1 at reset
            bg_scroll_reg    <= '0;
            irq_mask_reg     <= '0;
            player_pos_reg   <= '0;
            player_stats_reg <= '0;
            score_reg_int    <= '0;
            kill_count_reg   <= '0;
            hud_aux_reg      <= '0;
            swap_req_cnt     <= 3'd0;
        end else begin
            // SWAP request: count down each cycle until 0. A new SWAP write
            // reloads the counter (held high for 7 clk cycles ~140 ns, well
            // over the 80 ns window needed by a 2-FF synchroniser at 25 MHz).
            if (swap_req_cnt != 3'd0) swap_req_cnt <= swap_req_cnt - 3'd1;

            if (avs_write) begin
                if (region_ctrl) begin
                    case (avs_address[3:2])
                        2'h0: begin
                            ctrl_reg <= avs_writedata & 32'hFFFFFF07;
                            if (avs_writedata[1]) begin
                                swap_req_cnt <= 3'd7;     // hold ctrl_swap_req
                                ctrl_reg[1]  <= 1'b0;     // SWAP auto-clears
                            end
                        end
                        2'h1: ; // STATUS is read-only
                        2'h2: bg_scroll_reg <= avs_writedata;
                        2'h3: irq_mask_reg  <= avs_writedata;
                    endcase
                end else if (region_player) begin
                    case (avs_address[3:2])
                        2'h0: player_pos_reg   <= avs_writedata;
                        2'h1: player_stats_reg <= avs_writedata;
                        2'h2: score_reg_int    <= avs_writedata;
                        2'h3: kill_count_reg   <= avs_writedata;
                    endcase
                end else if (region_aux) begin
                    case (avs_address[3:2])
```

```
                        2'h0: hud_aux_reg <= avs_writedata;
                        default: ; // 0x24-0x2F reserved, writes ignored
                    endcase
                end
            end
        end
    end

    // ------------------------------------------------------------------------
    // Read logic (combinational, single-cycle)
    // ------------------------------------------------------------------------
    always_comb begin
        avs_readdata = 32'h0000_0000;

        if (avs_read) begin
            if (region_ctrl) begin
                case (avs_address[3:2])
                    2'h0: avs_readdata = ctrl_reg;
                    2'h1: avs_readdata = {16'h0, frame_ctr_in, 6'h0, swap_pending_in, vblank_in
};
                    2'h2: avs_readdata = bg_scroll_reg;
                    2'h3: avs_readdata = irq_mask_reg;
                endcase
            end else if (region_player) begin
                case (avs_address[3:2])
                    2'h0: avs_readdata = player_pos_reg;
                    2'h1: avs_readdata = player_stats_reg;
                    2'h2: avs_readdata = score_reg_int;
                    2'h3: avs_readdata = kill_count_reg;
                endcase
            end else if (region_aux) begin
                case (avs_address[3:2])
                    2'h0:    avs_readdata = hud_aux_reg;
                    default: avs_readdata = 32'h0;
                endcase
            end else if (region_sprtab) begin
                avs_readdata = sprtab_rdata_sw;
            end else if (region_palette) begin
                avs_readdata = palette_rdata_sw;
            end else if (region_tilemap) begin
                avs_readdata = tilemap_rdata_sw;
            end
        end
    end

endmodule
```

vga_timing.sv

```
module vga_timing (
    input  logic       pix_clk, reset_n,
    output logic [9:0] x,       // 0..799 (640 visible + porches)
    output logic [9:0] y,       // 0..524 (480 visible + porches)
    output logic       visible,
    output logic       hsync, vsync,
    output logic       vblank, hblank
);
```

```
    // Standard 640x480 @ 60Hz parameters
    localparam H_ACTIVE      = 640;
    localparam H_FRONT_PORCH = 16;
    localparam H_SYNC_PULSE  = 96;
    localparam H_BACK_PORCH  = 48;
    localparam H_TOTAL       = 800;

    localparam V_ACTIVE      = 480;
    localparam V_FRONT_PORCH = 10;
    localparam V_SYNC_PULSE  = 2;
    localparam V_BACK_PORCH  = 33;
    localparam V_TOTAL       = 525;

    // Internal counters
    logic [9:0] h_count;
    logic [9:0] v_count;

    // Assign outputs directly from counters
    assign x = h_count;
    assign y = v_count;

    always_ff @(posedge pix_clk or negedge reset_n) begin
        if (!reset_n) begin
            h_count <= 10'd0;
            v_count <= 10'd0;
        end else begin
            if (h_count == H_TOTAL - 1) begin
                h_count <= 10'd0;
                if (v_count == V_TOTAL - 1) begin
                    v_count <= 10'd0;
                end else begin
                    v_count <= v_count + 10'd1;
                end
            end else begin
                h_count <= h_count + 10'd1;
            end
        end
    end

    // Sync generation (Active Low for standard 640x480 VGA)
    assign hsync = ~(h_count >= (H_ACTIVE + H_FRONT_PORCH) &&
                     h_count <  (H_ACTIVE + H_FRONT_PORCH + H_SYNC_PULSE));

    assign vsync = ~(v_count >= (V_ACTIVE + V_FRONT_PORCH) &&
                     v_count <  (V_ACTIVE + V_FRONT_PORCH + V_SYNC_PULSE));

    // Blanking logic
    assign hblank = (h_count >= H_ACTIVE);
    assign vblank = (v_count >= V_ACTIVE);

    // Visible region is when neither horizontal nor vertical blanking is active
    assign visible = !hblank && !vblank;

endmodule
```

sprite_eval.sv

```
module sprite_eval (
    input  logic        pix_clk, reset_n,
    input  logic [9:0]  next_scanline,
    input  logic        eval_strobe,

    // Sprite table read port
    output logic [4:0]  sprtab_raddr,
    input  logic [63:0] sprtab_rdata,

    // Output: up to 8 active sprites for the next line
    output logic [63:0] line_sprites [0:7],
    output logic [7:0]  active_mask,
    // High for one cycle when DONE finishes latching the sorted list.
    output logic        eval_done
);

    typedef enum logic [1:0] {IDLE, FETCH, EVAL, DONE} state_t;
    state_t state, next_state;

    logic [5:0] sprite_idx; // 0 to 32

    // Internal tracking for the top 8 sprites
    logic [63:0] top_sprites [0:7];
    logic [2:0]  top_prios   [0:7];
    logic        top_valid   [0:7];

    // Sprite data unpacking (based on struct layout)
    logic signed [15:0] spr_y;
    logic [2:0]         spr_prio;
    logic               spr_active;

    assign spr_y      = sprtab_rdata[31:16];
    assign spr_prio   = sprtab_rdata[46:44];
    assign spr_active = sprtab_rdata[47];

    // Intersection check: Y is signed, next_scanline is unsigned.
    // All sprites are 16x16 in baseline.
    logic intersects;
    assign intersects = (spr_active) &&
                        (next_scanline >= spr_y) &&
                        (next_scanline < (spr_y + 16));

    always_ff @(posedge pix_clk or negedge reset_n) begin
        if (!reset_n) begin
            state <= IDLE;
            sprite_idx <= 6'd0;
            active_mask <= 8'd0;
            eval_done <= 1'b0;
            for (int i = 0; i < 8; i++) begin
                top_valid[i] <= 1'b0;
                line_sprites[i] <= 64'd0;
            end
        end else begin
            eval_done <= 1'b0;
            case (state)
                IDLE: begin
```

```
                if (eval_strobe) begin
                    state <= FETCH;
                    sprite_idx <= 6'd0;
                    // Clear the tracking arrays for the new scanline
                    for (int i = 0; i < 8; i++) top_valid[i] <= 1'b0;
                end
            end

            FETCH: begin
                sprtab_raddr <= sprite_idx[4:0];
                state <= EVAL;
            end

            EVAL: begin
                if (intersects) begin
                    // Hardware Insertion Sort: Find where this sprite belongs based on
priority (0 is highest)
                    // This block checks all 8 slots simultaneously and shifts lower-
priority sprites down.
                    logic inserted;
                    inserted = 1'b0;

                    for (int i = 0; i < 8; i++) begin
                        if (!inserted) begin
                            if (!top_valid[i] || (spr_prio < top_prios[i])) begin
                                // Insert here
                                top_sprites[i] <= sprtab_rdata;
                                top_prios[i]   <= spr_prio;
                                top_valid[i]   <= 1'b1;
                                inserted       = 1'b1;

                                // Shift the rest down (drops the 8th item if array is full
)
                                for (int j = 7; j > i; j--) begin
                                    top_sprites[j] <= top_sprites[j-1];
                                    top_prios[j]   <= top_prios[j-1];
                                    top_valid[j]   <= top_valid[j-1];
                                end
                            end
                        end
                    end

                    sprite_idx <= sprite_idx + 1;
                    if (sprite_idx == 6'd31) begin
                        state <= DONE;
                    end else begin
                        state <= FETCH;
                    end
                end

                DONE: begin
                    // Latch the final sorted list into the output ports
                    for (int i = 0; i < 8; i++) begin
                        line_sprites[i] <= top_sprites[i];
                        active_mask[i]  <= top_valid[i];
                    end
                    eval_done <= 1'b1;
                    state    <= IDLE;
```

```
                end
            endcase
        end
    end
endmodule
```

sprite_fetch.sv

```
module sprite_fetch (
    input  logic        pix_clk, reset_n,
    input  logic        hblank,
    input  logic [9:0]  next_scanline,

    // Inputs from sprite_eval
    input  logic        eval_done,       // Pulses high when sprite_eval finishes
    input  logic [7:0]  active_mask,
    input  logic [63:0] line_sprites [0:7], // 0 is highest priority, 7 is lowest

    // Sprite ROM read port
    output logic [13:0] sprrom_raddr,
    input  logic [7:0]  sprrom_rdata,

    // Line Buffer write port (to the "Fill" buffer)
    output logic [9:0]  linebuf_waddr,
    output logic [7:0]  linebuf_wdata,
    output logic        linebuf_we
);

    typedef enum logic [2:0] {IDLE, CLEAR_BUF, PROCESS_SPRITE, READ_PIXEL, WAIT_PIXEL,
    WRITE_PIXEL} state_t;
    state_t state;

    logic [3:0] sprite_idx; // 0 to 7 (iterating backwards from 7 down to 0)
    logic [4:0] pixel_u;    // 0 to 15 (columns within the 16x16 sprite)
    logic [9:0] clear_idx;  // 0 to 639 for clearing the buffer

    // Unpacked sprite data for the CURRENT sprite being processed
    logic [63:0] current_sprite;
    logic signed [15:0] spr_x, spr_y;
    logic [5:0]  spr_id;
    logic        spr_hflip;

    assign current_sprite = line_sprites[sprite_idx[2:0]];
    assign spr_x     = current_sprite[15:0];
    assign spr_y     = current_sprite[31:16];
    assign spr_id    = current_sprite[37:32];
    assign spr_hflip = current_sprite[42];

    // Calculate which row (v) of the sprite we are drawing
    logic [3:0] pixel_v;
    assign pixel_v = next_scanline - spr_y;

    // Calculate target X screen coordinate
    logic signed [15:0] target_x;
    assign target_x = spr_x + pixel_u;

    always_ff @(posedge pix_clk or negedge reset_n) begin
```

```
    if (!reset_n) begin
        state <= IDLE;
        linebuf_we <= 1'b0;
        sprite_idx <= 4'd7;
        clear_idx <= 10'd0;
    end else begin
        // Default to not writing
        linebuf_we <= 1'b0;

        case (state)
            IDLE: begin
                // When HBLANK starts, first thing we do is clear the fill buffer
                if (hblank) begin
                    state <= CLEAR_BUF;
                    clear_idx <= 10'd0;
                end
            end

            CLEAR_BUF: begin
                // Sweep across the 640 pixels and write 0 (transparent)
                linebuf_waddr <= clear_idx;
                linebuf_wdata <= 8'd0;
                linebuf_we    <= 1'b1;

                if (clear_idx == 10'd639) begin
                    // Wait here until sprite_eval is done evaluating the next line
                    if (eval_done) begin
                        state <= PROCESS_SPRITE;
                        sprite_idx <= 4'd7; // Start with lowest priority (index 7)
                    end else begin
                        linebuf_we <= 1'b0;
                    end
                end else begin
                    clear_idx <= clear_idx + 10'd1;
                end
            end

            PROCESS_SPRITE: begin
                if (sprite_idx > 4'd7) begin // Underflowed past 0 (meaning we did 7 down
to 0)
                    state <= IDLE;           // We are done with all sprites!
                end else if (active_mask[sprite_idx[2:0]]) begin
                    // This sprite is active! Let's fetch its pixels.
                    pixel_u <= 5'd0;
                    state <= READ_PIXEL;
                end else begin
                    // Skip inactive sprites
                    sprite_idx <= sprite_idx - 4'd1;
                end
            end

            READ_PIXEL: begin
                if (pixel_u == 5'd16) begin
                    // Finished all 16 columns for this sprite, move to next highest
priority
                    sprite_idx <= sprite_idx - 4'd1;
                    state <= PROCESS_SPRITE;
                end else begin
                    // Address = (sprite_id * 256) + (v * 16) + u
```

```
                        logic [3:0] actual_u;
                        actual_u = spr_hflip ? (4'd15 - pixel_u[3:0]) : pixel_u[3:0];

                        sprrom_raddr <= ({spr_id, 8'd0}) + ({pixel_v, 4'd0}) + actual_u;
                        // sprrom_raddr update + ROM read register together cost 2 cycles,
                        // so insert a WAIT_PIXEL bubble before sampling sprrom_rdata.
                        state <= WAIT_PIXEL;
                    end
                end

                WAIT_PIXEL: begin
                    // ROM read in flight; sprrom_rdata becomes valid next cycle.
                    state <= WRITE_PIXEL;
                end

                WRITE_PIXEL: begin
                    // If pixel isn't transparent (0) and is on screen, write it!
                    if (sprrom_rdata != 8'd0 && target_x >= 0 && target_x < 640) begin
                        linebuf_waddr <= target_x[9:0];
                        linebuf_wdata <= sprrom_rdata;
                        linebuf_we    <= 1'b1;
                    end

                    pixel_u <= pixel_u + 5'd1;
                    state <= READ_PIXEL;
                end
            endcase
        end
    end
endmodule
```

compositor.sv

```
module compositor (
    input  logic        pix_clk, reset_n,
    input  logic [9:0]  x, y,
    input  logic        visible,
    input  logic [15:0] scroll_x, scroll_y,

    // HUD overlay inputs
    input  logic        hud_on,
    input  logic [31:0] player_stats,    // [7:0] hp, [15:8] wave BCD, [31:16] level
    input  logic [31:0] score_reg,       // [23:0] 6-digit BCD score
    input  logic [31:0] hud_aux,         // [7:0] ammo BCD, [11:8] art BCD, [15:12] gas BCD

    // Tile map read
    output logic [12:0] tilemap_raddr,
    input  logic [7:0]  tilemap_rdata,

    // Tile ROM read (shared between background tiles and HUD digit/letter
    // glyphs; never collide because the HUD-text region is a disjoint y-slice).
    output logic [11:0] tilerom_raddr,
    input  logic [7:0]  tilerom_rdata,

    // Sprite line buffer (already filled during HBLANK)
    output logic [9:0]  linebuf_raddr,
    input  logic [7:0]  linebuf_rdata,   // sprite palette index, 0 = transparent
```

```
    // Palette read
    output logic [7:0]  palette_raddr,
    input  logic [23:0] palette_rdata,

    output logic [23:0] rgb_out
);

    // ===== HUD layout =====
    // 16-px strip across the top. Text glyphs are 8x8 and live at y=4..11 so
    // they're vertically centered in the strip.
    localparam int HUD_H_END        = 16;
    localparam int HP_X_END         = 192;

    // AMMO counter: "AMMO" label (4 letters) + 2-digit value.
    localparam int AMMO_LBL_X       = 200;
    localparam int AMMO_DIG_X       = 232;
    localparam int AMMO_END_X       = 248;

    // WAVE counter: "WAVE" label (4 letters) + existing 2-digit value.
    localparam int WAVE_LBL_X       = 280;
    localparam int WAVE_X_START     = 312;
    localparam int WAVE_X_END       = 328;

    // ART (artillery charges): "ART" label (3 letters) + 1-digit value.
    localparam int ART_LBL_X        = 360;
    localparam int ART_DIG_X        = 384;
    localparam int ART_END_X        = 392;

    // GAS (gas charges): "GAS" label (3 letters) + 1-digit value.
    localparam int GAS_LBL_X        = 408;
    localparam int GAS_DIG_X        = 432;
    localparam int GAS_END_X        = 440;

    // SCORE: "SCORE" label (5 letters) + existing 6-digit value at 520..567.
    localparam int SCORE_LBL_X      = 472;
    localparam int SCORE_LBL_END_X  = 512;
    localparam int SCORE_X_START    = 520;
    localparam int SCORE_X_END      = 568;

    localparam int TEXT_Y_START     = 4;   // glyphs span y=4..11 (centered in 16-px strip)
    localparam int TEXT_Y_END       = 12;
    localparam int DIGIT_TILE_BASE  = 48;  // tile ROM slots 48..57 = digit glyphs 0..9
    localparam int LETTER_TILE_BASE = 16;  // tile ROM slots 16..41 = letters A..Z

    localparam logic [23:0] RGB_HP_FULL  = 24'h00FF00;  // green
    localparam logic [23:0] RGB_HP_EMPTY = 24'h800000;  // dark red
    localparam logic [23:0] RGB_HUD_FG   = 24'hFFFFFF;  // white digit/letter pixels
    localparam logic [23:0] RGB_HUD_BG   = 24'h202020;  // dark gray HUD fill

    // ===== Stage 1: combinational region + tile-id selection =====
    wire in_hud_strip   = hud_on && (y < HUD_H_END);
    wire in_hp_bar      = in_hud_strip && (x < HP_X_END);
    wire text_y_active  = in_hud_strip && (y >= TEXT_Y_START) && (y < TEXT_Y_END);

    wire in_ammo_label  = text_y_active && (x >= AMMO_LBL_X)   && (x < AMMO_DIG_X);
    wire in_ammo_digits = text_y_active && (x >= AMMO_DIG_X)   && (x < AMMO_END_X);
    wire in_wave_label  = text_y_active && (x >= WAVE_LBL_X)   && (x < WAVE_X_START);
    wire in_wave_text   = text_y_active && (x >= WAVE_X_START) && (x < WAVE_X_END);
```

```
    wire in_art_label   = text_y_active && (x >= ART_LBL_X)    && (x < ART_DIG_X);
    wire in_art_digit   = text_y_active && (x >= ART_DIG_X)    && (x < ART_END_X);
    wire in_gas_label   = text_y_active && (x >= GAS_LBL_X)    && (x < GAS_DIG_X);
    wire in_gas_digit   = text_y_active && (x >= GAS_DIG_X)    && (x < GAS_END_X);
    wire in_score_label = text_y_active && (x >= SCORE_LBL_X)  && (x < SCORE_LBL_END_X);
    wire in_score_text  = text_y_active && (x >= SCORE_X_START) && (x < SCORE_X_END);

    wire in_hud_text    = in_ammo_label  || in_ammo_digits
                        || in_wave_label  || in_wave_text
                        || in_art_label   || in_art_digit
                        || in_gas_label   || in_gas_digit
                        || in_score_label || in_score_text;
    wire in_hud_gap     = in_hud_strip && !in_hp_bar && !in_hud_text;

    // HP bar fill: hp*2 px clamped to 192 (hp in 0..100, bar width 192).
    wire [8:0] hp_x2          = {1'b0, player_stats[7:0]} << 1;
    wire [8:0] hp_bar_width   = (hp_x2 > 9'd192) ? 9'd192 : hp_x2;
    wire       hp_filled      = ({1'b0, x[8:0]} < hp_bar_width);

    // ----- Label tile-id lookup -----
    // For each label region, compute the char index within the label (0..N-1)
    // from (x - label_start) >> 3, then map to the letter tile via case.
    // Letter tile mapping: A=16, B=17, ..., Z=41 (LETTER_TILE_BASE + L-'A').

    wire [9:0] x_minus_ammo_lbl  = x - 10'(AMMO_LBL_X);
    wire [9:0] x_minus_wave_lbl  = x - 10'(WAVE_LBL_X);
    wire [9:0] x_minus_art_lbl   = x - 10'(ART_LBL_X);
    wire [9:0] x_minus_gas_lbl   = x - 10'(GAS_LBL_X);
    wire [9:0] x_minus_score_lbl = x - 10'(SCORE_LBL_X);

    wire [1:0] ammo_char_idx  = x_minus_ammo_lbl[4:3];     // 4 chars: 0..3
    wire [1:0] wave_char_idx  = x_minus_wave_lbl[4:3];     // 4 chars: 0..3
    wire [1:0] art_char_idx   = x_minus_art_lbl[4:3];      // 3 chars: 0..2
    wire [1:0] gas_char_idx   = x_minus_gas_lbl[4:3];      // 3 chars: 0..2
    wire [2:0] score_char_idx = x_minus_score_lbl[5:3];    // 5 chars: 0..4

    logic [5:0] ammo_label_tile;
    logic [5:0] wave_label_tile;
    logic [5:0] art_label_tile;
    logic [5:0] gas_label_tile;
    logic [5:0] score_label_tile;

    // "AMMO" -> A M M O -> tiles 16, 28, 28, 30
    always_comb begin
        case (ammo_char_idx)
            2'd0:    ammo_label_tile = 6'd16;
            2'd1:    ammo_label_tile = 6'd28;
            2'd2:    ammo_label_tile = 6'd28;
            2'd3:    ammo_label_tile = 6'd30;
            default: ammo_label_tile = 6'd0;
        endcase
    end

    // "WAVE" -> W A V E -> tiles 38, 16, 37, 20
    always_comb begin
        case (wave_char_idx)
            2'd0:    wave_label_tile = 6'd38;
            2'd1:    wave_label_tile = 6'd16;
            2'd2:    wave_label_tile = 6'd37;
```

```
            2'd3:    wave_label_tile = 6'd20;
            default: wave_label_tile = 6'd0;
        endcase
    end

    // "ART" -> A R T -> tiles 16, 33, 35
    always_comb begin
        case (art_char_idx)
            2'd0:    art_label_tile = 6'd16;
            2'd1:    art_label_tile = 6'd33;
            2'd2:    art_label_tile = 6'd35;
            default: art_label_tile = 6'd0;
        endcase
    end

    // "GAS" -> G A S -> tiles 22, 16, 34
    always_comb begin
        case (gas_char_idx)
            2'd0:    gas_label_tile = 6'd22;
            2'd1:    gas_label_tile = 6'd16;
            2'd2:    gas_label_tile = 6'd34;
            default: gas_label_tile = 6'd0;
        endcase
    end

    // "SCORE" -> S C O R E -> tiles 34, 18, 30, 33, 20
    always_comb begin
        case (score_char_idx)
            3'd0:    score_label_tile = 6'd34;
            3'd1:    score_label_tile = 6'd18;
            3'd2:    score_label_tile = 6'd30;
            3'd3:    score_label_tile = 6'd33;
            3'd4:    score_label_tile = 6'd20;
            default: score_label_tile = 6'd0;
        endcase
    end

    // ----- Digit BCD selection -----
    // AMMO digits (2 chars): leftmost = tens (hud_aux[7:4]), rightmost = ones.
    wire [9:0] x_minus_ammo_dig = x - 10'(AMMO_DIG_X);
    wire       ammo_d_idx       = x_minus_ammo_dig[3];   // 0 (tens) or 1 (ones)
    wire [3:0] ammo_digit_bcd   = ammo_d_idx ? hud_aux[3:0] : hud_aux[7:4];

    // WAVE digit (2 chars): bits [15:8] of player_stats hold 2 BCD digits.
    wire [9:0] x_minus_wave_dig = x - 10'(WAVE_X_START);
    wire       wave_d_idx     = x_minus_wave_dig[3];
    wire [3:0] wave_digit_bcd = wave_d_idx ? player_stats[11:8]
                                           : player_stats[15:12];

    // ART / GAS digits: single 4-bit BCD each.
    wire [3:0] art_digit_bcd = hud_aux[11:8];
    wire [3:0] gas_digit_bcd = hud_aux[15:12];

    // SCORE digits (6 chars): score_reg[23:0] holds 6 BCD nibbles (nibble 0
    // = ones, nibble 5 = highest digit; leftmost screen column shows highest).
    wire [9:0] x_minus_score_dig  = x - 10'(SCORE_X_START);
    wire [2:0] score_digit_idx    = x_minus_score_dig[5:3];   // 0..5
    wire [2:0] score_nibble_idx   = 3'd5 - score_digit_idx;
    logic [3:0] score_digit_bcd;
```

```
    always_comb begin
        case (score_nibble_idx)
            3'd0:    score_digit_bcd = score_reg[3:0];
            3'd1:    score_digit_bcd = score_reg[7:4];
            3'd2:    score_digit_bcd = score_reg[11:8];
            3'd3:    score_digit_bcd = score_reg[15:12];
            3'd4:    score_digit_bcd = score_reg[19:16];
            3'd5:    score_digit_bcd = score_reg[23:20];
            default: score_digit_bcd = 4'd0;
        endcase
    end

    // ----- Final hud_tile_id selection -----
    // Regions are mutually exclusive by x-range, so this priority chain just
    // picks whichever region the current pixel lies in.
    logic [5:0] hud_tile_id;
    always_comb begin
        hud_tile_id = 6'd0;
        if      (in_ammo_label)  hud_tile_id = ammo_label_tile;
        else if (in_ammo_digits) hud_tile_id = 6'd48 + {2'b00, ammo_digit_bcd};
        else if (in_wave_label)  hud_tile_id = wave_label_tile;
        else if (in_wave_text)   hud_tile_id = 6'd48 + {2'b00, wave_digit_bcd};
        else if (in_art_label)   hud_tile_id = art_label_tile;
        else if (in_art_digit)   hud_tile_id = 6'd48 + {2'b00, art_digit_bcd};
        else if (in_gas_label)   hud_tile_id = gas_label_tile;
        else if (in_gas_digit)   hud_tile_id = 6'd48 + {2'b00, gas_digit_bcd};
        else if (in_score_label) hud_tile_id = score_label_tile;
        else if (in_score_text)  hud_tile_id = 6'd48 + {2'b00, score_digit_bcd};
    end

    // Tile-local pixel coordinates. All HUD regions start on 8-px x-boundaries
    // so x[2:0] gives the column within the glyph. For y, we subtract
    // TEXT_Y_START so row 0 of each glyph corresponds to y=TEXT_Y_START.
    wire [2:0] hud_tile_x    = x[2:0];
    wire [3:0] hud_tile_y_4b = y[3:0] - 4'(TEXT_Y_START);
    wire [2:0] hud_tile_y    = hud_tile_y_4b[2:0];

    // ===== Background tile lookup (unchanged) =====
    logic [6:0] tile_col;
    logic [5:0] tile_row;
    assign tile_col = x[9:3];
    assign tile_row = y[9:3];
    assign tilemap_raddr = (tile_row * 80) + tile_col;
    assign linebuf_raddr = x;

    // ===== Stage 1 -> Stage 2 latches =====
    logic       s2_in_hp_bar, s2_hp_filled;
    logic       s2_in_hud_text, s2_in_hud_gap;
    logic [5:0] s2_hud_tile_id;
    logic [2:0] s2_hud_tile_x, s2_hud_tile_y;
    logic [2:0] x_delay, y_delay;
    logic [7:0] sprite_pixel_delay;
    logic       visible_delay1;

    always_ff @(posedge pix_clk) begin
        s2_in_hp_bar       <= in_hp_bar;
        s2_hp_filled       <= hp_filled;
        s2_in_hud_text     <= in_hud_text;
        s2_in_hud_gap      <= in_hud_gap;
```

```
        s2_hud_tile_id     <= hud_tile_id;
        s2_hud_tile_x      <= hud_tile_x;
        s2_hud_tile_y      <= hud_tile_y;
        x_delay            <= x[2:0];
        y_delay            <= y[2:0];
        sprite_pixel_delay <= linebuf_rdata;
        visible_delay1     <= visible;
    end

    // ===== Stage 2: tilerom_raddr mux (HUD text vs background) =====
    wire [11:0] tilerom_addr_bg  = ({tilemap_rdata, 6'd0}) + ({y_delay, 3'd0}) + x_delay;
    wire [11:0] tilerom_addr_hud = ({s2_hud_tile_id, 6'd0}) + ({s2_hud_tile_y, 3'd0}) +
    s2_hud_tile_x;
    assign tilerom_raddr = s2_in_hud_text ? tilerom_addr_hud : tilerom_addr_bg;

    // ===== Stage 2 -> Stage 3 latches =====
    logic s3_in_hp_bar, s3_hp_filled;
    logic s3_in_hud_text, s3_in_hud_gap;
    logic visible_delay2;

    always_ff @(posedge pix_clk) begin
        s3_in_hp_bar    <= s2_in_hp_bar;
        s3_hp_filled    <= s2_hp_filled;
        s3_in_hud_text  <= s2_in_hud_text;
        s3_in_hud_gap   <= s2_in_hud_gap;
        visible_delay2  <= visible_delay1;
    end

    // ===== Stage 3: palette mux for non-HUD pixels =====
    always_comb begin
        if (sprite_pixel_delay != 8'd0) palette_raddr = sprite_pixel_delay;
        else                            palette_raddr = tilerom_rdata;
    end

    // ===== Stage 4: final RGB =====
    // HUD layer always wins above sprites/tiles in the top strip.
    always_ff @(posedge pix_clk or negedge reset_n) begin
        if (!reset_n) begin
            rgb_out <= 24'd0;
        end else if (!visible_delay2) begin
            rgb_out <= 24'd0;
        end else if (s3_in_hp_bar) begin
            rgb_out <= s3_hp_filled ? RGB_HP_FULL : RGB_HP_EMPTY;
        end else if (s3_in_hud_text) begin
            // Glyph tile uses palette index 0xFF for lit pixels, 0x00 for blank.
            rgb_out <= (tilerom_rdata != 8'd0) ? RGB_HUD_FG : RGB_HUD_BG;
        end else if (s3_in_hud_gap) begin
            rgb_out <= RGB_HUD_BG;
        end else begin
            rgb_out <= palette_rdata;
        end
    end

endmodule
```

pll_25mhz.v

```
// pll_25mhz.v
// 50 MHz -> 25 MHz pixel clock for the VGA pipeline.
//
// Pragmatic divide-by-2 implementation suitable for the smoke-test bitstream.
// Quartus auto-promotes the divided register output onto a global clock
// network when it's used as a clock downstream, which is exactly how it's
// consumed inside nml_gpu (every always_ff @(posedge pix_clk) ...).
//
// PRODUCTION upgrade path (recommended after first working bitstream):
// replace this module with an `altera_pll` IP generated in Platform
// Designer with output_0 = 25 MHz. The instance port names below
// (refclk / rst / outclk_0) match the altera_pll defaults so swapping in
// the IP requires no edits in nml_gpu.sv.

module pll_25mhz (
    input  wire refclk,    // 50 MHz reference
    input  wire rst,       // active high reset
    output wire outclk_0   // 25 MHz pixel clock
);

    reg clk_div = 1'b0;

    always @(posedge refclk or posedge rst) begin
        if (rst) clk_div <= 1'b0;
        else     clk_div <= ~clk_div;
    end

    assign outclk_0 = clk_div;

endmodule
```

gen_rom.py

```
#!/usr/bin/env python3
"""
gen_rom.py -- generate sprite_rom.hex, tile_rom.hex, palette.hex, sprite_table.hex

Format: $readmemh-compatible. One byte per line, lowercase hex, no addresses.

Sprite ROM: 64 sprites x 256 bytes (16x16, palette indices). Sprite 0 reserved
            transparent. Sprites 1-10 are the live game art:
              1 = player           (green bordered)
              2 = armed enemy      (red bordered + white cross)
              3 = player bullet    (small yellow block)
              4 = unarmed enemy    (solid pink, no border)
              5 = mortar shell     (orange diamond)
              6 = (retired -- was barbed wire; AA_WIRE removed in this batch)
              7 = mustard gas      (large green-yellow circle, SW tiles 3+ to form cloud)
              8 = artillery flash  (white plus; for the stacked-sprite beam)
              9 = ammo drop        (green crate / box outline, white border)
             10 = enemy bullet     (red dot, small)

Tile ROM:   64 tiles x 64 bytes (8x8, palette indices).
              0      = solid background fill
              1      = background with corner accents
              2      = trench horizontal stripes
```

```
              3      = dirt scatter
              4..10  = battlefield mosaic (crater / sandbag / trench-edge /
                       barb-wire / mud-streak / shell-hole / reserved)
              11     = artillery beam (vertical white line) -- used by Batch C
              16..41 = letter glyphs A..Z   (tile_id = 16 + (letter - 'A'))
              42     = colon ':'
              43     = space ' '
              44..47 = reserved
              48..57 = digits 0..9
              58..63 = reserved

Palette indices used here are mirrored in sw/main.c init_palette_runtime():
    0x00 = transparent (sprite ROM only)
    0x10 = player color (green)
    0x11 = armed enemy red
    0x12 = bullet yellow
    0x13 = unarmed enemy pink
    0x14 = mortar orange
    0x15 = (retired -- barbed wire gray)
    0x16 = mustard gas yellow-green
    0x17 = artillery bright white-yellow
    0x18 = ammo drop green (new this batch)
    0x19 = enemy bullet red (new this batch)
    0x20 = tile background
    0x21 = tile accent
    0xFF = white (sprite borders)

Run: ``python3 gen_rom.py`` from the hw/ directory. Tested against the lab
Python (3.6) so the type hints below stick to syntax that predates PEP 585.
"""

from pathlib import Path

HERE = Path(__file__).resolve().parent

SPRITE_ROM_BYTES = 64 * 256          # 16384
TILE_ROM_BYTES   = 64 * 64           #  4096

PAL_TRANSPARENT  = 0x00
PAL_PLAYER       = 0x10
PAL_ENEMY_ARMED  = 0x11
PAL_BULLET       = 0x12
PAL_ENEMY_UNARMD = 0x13
PAL_MORTAR       = 0x14
# 0x15 retired (was PAL_WIRE)
PAL_GAS          = 0x16
PAL_ARTILLERY    = 0x17
PAL_AMMO_DROP    = 0x18
PAL_ENEMY_BULLET = 0x19
# Battlefield ground palette (top-down view, pixel-art style):
PAL_BG           = 0x20   # PAL_DIRT_MID    -- primary dirt color
PAL_BG_ACCENT    = 0x21   # PAL_DIRT_LIGHT  -- dirt highlight
PAL_DIRT_MID     = 0x20
PAL_DIRT_LIGHT   = 0x21
PAL_DIRT_DARK    = 0x22
PAL_GRASS_MID    = 0x23
PAL_GRASS_DARK   = 0x24
PAL_GRASS_LIGHT  = 0x25
PAL_MUD_DEEP     = 0x26
```

```
PAL_BORDER       = 0xFF


def make_sprite_rom() -> bytearray:
    rom = bytearray(SPRITE_ROM_BYTES)  # zero-init = all transparent

    def put(slot: int, u: int, v: int, pal: int) -> None:
        rom[slot * 256 + v * 16 + u] = pal

    def fill_solid(slot: int, pal: int) -> None:
        for v in range(16):
            for u in range(16):
                put(slot, u, v, pal)

    def fill_bordered(slot: int, fill: int, border: int) -> None:
        for v in range(16):
            for u in range(16):
                edge = (u == 0 or u == 15 or v == 0 or v == 15)
                put(slot, u, v, border if edge else fill)

    def fill_centered(slot: int, pal: int, half: int) -> None:
        # half=2 -> 4x4 centered block, etc.
        for v in range(16):
            for u in range(16):
                if abs(u - 8) < half and abs(v - 8) < half:
                    put(slot, u, v, pal)

    def fill_diamond(slot: int, pal: int, radius: int) -> None:
        # Manhattan-distance diamond centered at (8, 8). radius=7 fills 15px wide.
        for v in range(16):
            for u in range(16):
                if abs(u - 8) + abs(v - 8) <= radius:
                    put(slot, u, v, pal)

    def fill_circle(slot: int, pal: int, radius: int) -> None:
        # Euclidean disk centered at (8, 8).
        r2 = radius * radius
        for v in range(16):
            for u in range(16):
                dx = u - 8
                dy = v - 8
                if dx * dx + dy * dy <= r2:
                    put(slot, u, v, pal)

    def fill_x(slot: int, pal: int, thickness: int) -> None:
        # Two diagonals from corner to corner; thickness in pixels (1, 2, ...).
        for v in range(16):
            for u in range(16):
                if abs(u - v) < thickness or abs(u + v - 15) < thickness:
                    put(slot, u, v, pal)

    def fill_plus(slot: int, pal: int, arm_half: int, thickness: int) -> None:
        # A '+' centered at (8, 8). arm_half = arm length each side from center,
        # thickness = bar width.
        cx, cy = 7, 7   # so the bars span an even pair around the visual center
        for v in range(16):
            for u in range(16):
                in_h = (abs(v - cy) < thickness and abs(u - cx) <= arm_half)
                in_v = (abs(u - cx) < thickness and abs(v - cy) <= arm_half)
```

```
                if in_h or in_v:
                    put(slot, u, v, pal)

    def overlay_cross(slot: int, pal: int) -> None:
        # Small '+' marker centered, used to badge armed enemies.
        for d in range(-2, 3):
            put(slot, 7 + d, 7, pal)
            put(slot, 7, 7 + d, pal)
            put(slot, 8 + d, 8, pal)
            put(slot, 8, 8 + d, pal)

    def fill_ammo_crate(slot: int, fill: int, border: int) -> None:
        # Rounded "ammo box" outline with hollow interior, plus a faint cross
        # marker so it reads as a pickup. 12x12 inside a 16x16 sprite.
        for v in range(2, 14):
            for u in range(2, 14):
                on_edge = (u == 2 or u == 13 or v == 2 or v == 13)
                put(slot, u, v, border if on_edge else fill)
        # interior cross marker
        for d in range(-3, 4):
            put(slot, 7 + d, 7, border)
            put(slot, 7, 7 + d, border)

    def fill_dot(slot: int, pal: int, radius: int) -> None:
        # Small filled disk for the enemy bullet -- slightly bigger than the
        # player's 4x4 bullet so it reads as "incoming".
        r2 = radius * radius
        for v in range(16):
            for u in range(16):
                dx = u - 8
                dy = v - 8
                if dx * dx + dy * dy <= r2:
                    put(slot, u, v, pal)

    # slot 0: transparent (already)
    fill_bordered(1, PAL_PLAYER,       PAL_BORDER)   # player
    fill_bordered(2, PAL_ENEMY_ARMED,  PAL_BORDER)   # armed enemy
    overlay_cross(2, PAL_BORDER)                     # cross badge identifies "armed"
    fill_centered(3, PAL_BULLET, 2)                  # bullet: 4x4 centered
    fill_solid   (4, PAL_ENEMY_UNARMD)               # unarmed: solid pink
    fill_diamond (5, PAL_MORTAR, 7)                  # mortar shell: diamond
    # slot 6 retired (was barbed wire); leave transparent so SW won't render anything if reused
    fill_circle  (7, PAL_GAS, 7)                     # mustard gas: large disk
    fill_plus    (8, PAL_ARTILLERY, 7, 2)            # artillery flash: thick '+'
    fill_ammo_crate(9, PAL_AMMO_DROP, PAL_BORDER)    # ammo drop: green crate
    fill_dot     (10, PAL_ENEMY_BULLET, 3)           # enemy bullet: small red dot

    return rom


# 8x8 letter glyphs. Each cell is a tuple of 8 strings of 8 chars where '#'
# marks a lit pixel (PAL_BORDER white) and any other char leaves the pixel as
# the tile background (palette index 0 -> compositor draws RGB_HUD_BG).
# Designed in a 5-wide x 7-tall window with 1-pixel margins so adjacent
# glyphs don't run together when stitched at 8-px boundaries.
LETTER_GLYPHS = {
    'A': (".......",
          "..####..",
          ".##..##.",
```

```
        ".##..##.",
        ".######.",
        ".##..##.",
        ".##..##.",
        "........"),
    'B': ("........",
        ".#####..",
        ".##..##.",
        ".#####..",
        ".##..##.",
        ".##..##.",
        ".#####..",
        "........"),
    'C': ("........",
        "..####..",
        ".##..##.",
        ".##.....",
        ".##.....",
        ".##..##.",
        "..####..",
        "........"),
    'D': ("........",
        ".#####..",
        ".##..##.",
        ".##..##.",
        ".##..##.",
        ".##..##.",
        ".#####..",
        "........"),
    'E': ("........",
        ".######.",
        ".##.....",
        ".#####..",
        ".##.....",
        ".##.....",
        ".######.",
        "........"),
    'F': ("........",
        ".######.",
        ".##.....",
        ".#####..",
        ".##.....",
        ".##.....",
        ".##.....",
        "........"),
    'G': ("........",
        "..####..",
        ".##..##.",
        ".##.....",
        ".##.###.",
        ".##..##.",
        "..####..",
        "........"),
    'H': ("........",
        ".##..##.",
        ".##..##.",
        ".######.",
        ".##..##.",
        ".##..##.",
```

```
        ".##..##.",
        "........"),
    'I': ("........",
        ".######.",
        "...##...",
        "...##...",
        "...##...",
        "...##...",
        ".######.",
        "........"),
    'J': ("........",
        ".######.",
        ".....##.",
        ".....##.",
        ".....##.",
        ".##..##.",
        "..####..",
        "........"),
    'K': ("........",
        ".##..##.",
        ".##.##..",
        ".####...",
        ".####...",
        ".##.##..",
        ".##..##.",
        "........"),
    'L': ("........",
        ".##.....",
        ".##.....",
        ".##.....",
        ".##.....",
        ".##.....",
        ".######.",
        "........"),
    'M': ("........",
        ".##..##.",
        ".######.",
        ".######.",
        ".##..##.",
        ".##..##.",
        ".##..##.",
        "........"),
    'N': ("........",
        ".##..##.",
        ".###.##.",
        ".######.",
        ".##.###.",
        ".##..##.",
        ".##..##.",
        "........"),
    'O': ("........",
        "..####..",
        ".##..##.",
        ".##..##.",
        ".##..##.",
        ".##..##.",
        "..####..",
        "........"),
    'P': ("........",
```

```
        ".#####..",
        ".##..##.",
        ".#####..",
        ".##.....",
        ".##.....",
        ".##.....",
        "........"),
    'Q': ("........",
        "..####..",
        ".##..##.",
        ".##..##.",
        ".##.###.",
        ".##..##.",
        "..#####.",
        "........"),
    'R': ("........",
        ".#####..",
        ".##..##.",
        ".#####..",
        ".####...",
        ".##.##..",
        ".##..##.",
        "........"),
    'S': ("........",
        "..####..",
        ".##..##.",
        "..##....",
        "....##..",
        ".##..##.",
        "..####..",
        "........"),
    'T': ("........",
        ".######.",
        "...##...",
        "...##...",
        "...##...",
        "...##...",
        "...##...",
        "........"),
    'U': ("........",
        ".##..##.",
        ".##..##.",
        ".##..##.",
        ".##..##.",
        ".##..##.",
        "..####..",
        "........"),
    'V': ("........",
        ".##..##.",
        ".##..##.",
        ".##..##.",
        ".##..##.",
        "..####..",
        "...##...",
        "........"),
    'W': ("........",
        ".##..##.",
        ".##..##.",
        ".##..##.",
```

```
          ".######.",
          ".######.",
          ".##..##.",
          "........"),
    'X': ("........",
          ".##..##.",
          ".##..##.",
          "..####..",
          "..####..",
          ".##..##.",
          ".##..##.",
          "........"),
    'Y': ("........",
          ".##..##.",
          ".##..##.",
          "..####..",
          "...##...",
          "...##...",
          "...##...",
          "........"),
    'Z': ("........",
          ".######.",
          ".....##.",
          "....##..",
          "...##...",
          "..##....",
          ".######.",
          "........"),
    ':': ("........",
          "........",
          "...##...",
          "...##...",
          "........",
          "...##...",
          "...##...",
          "........"),
    ' ': ("........",
          "........",
          "........",
          "........",
          "........",
          "........",
          "........",
          "........"),
}
LETTER_TILE_BASE   = 16   # 'A' -> tile 16, 'B' -> 17, ..., 'Z' -> 41
COLON_TILE_ID      = 42
SPACE_TILE_ID      = 43
BEAM_TILE_ID       = 11   # artillery beam glyph (vertical white line)

# Battlefield ground tiles (top-down view, pixel-art style inspired by the
# brown-dirt + green-grass mosaic look of period RPG tilesets). Each cell is
# 8x8 and uses up to 6 palette colors. The legend maps each glyph char to a
# palette index:
#   'M' = PAL_DIRT_MID    (mid brown, primary dirt)
#   'D' = PAL_DIRT_DARK   (darker mud splotches)
#   'L' = PAL_DIRT_LIGHT  (sandy-tan dirt highlight)
#   'G' = PAL_GRASS_MID   (primary grass green)
#   'g' = PAL_GRASS_DARK  (shadowed grass / dense clump)
```

```
#   'h' = PAL_GRASS_LIGHT (highlight / new growth)
#   'm' = PAL_MUD_DEEP    (very dark mud puddle interior)
GROUND_LEGEND = {
    'M': PAL_DIRT_MID,
    'D': PAL_DIRT_DARK,
    'L': PAL_DIRT_LIGHT,
    'G': PAL_GRASS_MID,
    'g': PAL_GRASS_DARK,
    'h': PAL_GRASS_LIGHT,
    'm': PAL_MUD_DEEP,
}

BATTLEFIELD_GLYPHS = {
    4:  ("MMMDMMLM",   # dirt_a: mostly mid-brown with sparse dark/light specks
         "MDMMMMMM",
         "MMMMDMMM",
         "MLMMMMDM",
         "MMMDMMMM",
         "MMMMMLMM",
         "DMMMMMMM",
         "MMMMDMML"),
    5:  ("MMLMMMDM",   # dirt_b: same palette, different scatter to break tiling
         "DMMMDMML",
         "MMMMMMMM",
         "MMMMMLDM",
         "MLMMMMMM",
         "MMMDMMMM",
         "MMMMMMDM",
         "MDMMLMMM"),
    6:  ("GGgGGGGh",   # grass_a: mostly mid-green with shadow + highlight specks
         "GGGGgGGG",
         "GhGGGGGG",
         "GGGGGgGh",
         "GGgGGGGG",
         "GGGhGGGG",
         "gGGGGGGG",
         "GGGGgGhG"),
    7:  ("GGGGhGGG",   # grass_b: different scatter
         "GgGGGGGG",
         "GGGGGGgG",
         "hGGGGGGh",
         "GGGGgGGG",
         "GGgGGGhG",
         "GGGGGGGG",
         "GhGGgGGG"),
    8:  ("MMMMGGGG",   # transition: dirt on left, grass on right
         "MMMMGGGG",
         "MDMMGgGG",
         "MMMMGGGG",
         "MMLMGGgG",
         "MMMMhGGG",
         "MDMMGGGG",
         "MMMMGGGG"),
    9:  ("MMDDDDMM",   # mud puddle: dark wet crater
         "MDmmmDDM",
         "DmmmmmDM",
         "DmmDmmDD",
         "DmmmmmDD",
         "MDmmmDDM",
```

```
         "MDDDDDMM",
         "MMMMMMMM"),
   10:  ("ghGGgGhG",   # grass_dense: lush patch
         "GGghGGgG",
         "gGGGhGGG",
         "GgGGGgGh",
         "GhGgGGGG",
         "GGGGGhgG",
         "gGhGGGGg",
         "GGGgGhGG"),
}

# Vertical beam tile: 2-px-wide bright line spanning the full 8x8 cell, with
# faint edges so adjacent beam tiles read as continuous.
BEAM_GLYPH = ("...##...",
              "...##...",
              "...##...",
              "...##...",
              "...##...",
              "...##...",
              "...##...",
              "...##...")


DIGIT_GLYPHS = {
    0: (".......",
        ".######.",
        ".##..##.",
        ".##..##.",
        ".##..##.",
        ".##..##.",
        ".######.",
        "........"),
    1: (".......",
        "...##...",
        "..###...",
        "...##...",
        "...##...",
        "...##...",
        ".######.",
        "........"),
    2: (".......",
        ".######.",
        ".....##.",
        ".....##.",
        ".######.",
        ".##.....",
        ".######.",
        "........"),
    3: (".......",
        ".######.",
        ".....##.",
        ".....##.",
        ".######.",
        ".....##.",
        ".######.",
        "........"),
    4: (".......",
        "..#..#..",
```

```
        "..#..#..",
        "..####..",
        ".....#..",
        ".....#..",
        ".....#..",
        "........"),
    5: ("........",
        ".######.",
        ".##.....",
        ".######.",
        ".....##.",
        ".....##.",
        ".######.",
        "........"),
    6: ("........",
        ".######.",
        ".##.....",
        ".######.",
        ".##..##.",
        ".##..##.",
        ".######.",
        "........"),
    7: ("........",
        ".######.",
        ".....##.",
        ".....##.",
        "....##..",
        "...##...",
        "...##...",
        "........"),
    8: ("........",
        ".######.",
        ".##..##.",
        ".######.",
        ".##..##.",
        ".##..##.",
        ".######.",
        "........"),
    9: ("........",
        ".######.",
        ".##..##.",
        ".######.",
        ".....##.",
        ".....##.",
        ".######.",
        "........"),
}
DIGIT_TILE_BASE = 48   # tile slots 48..57


def make_tile_rom() -> bytearray:
    rom = bytearray(TILE_ROM_BYTES)

    def put(slot: int, u: int, v: int, pal: int) -> None:
        rom[slot * 64 + v * 8 + u] = pal

    def fill_solid(slot: int, pal: int) -> None:
        for v in range(8):
            for u in range(8):
```

```
            put(slot, u, v, pal)

def fill_glyph(slot: int, glyph: tuple, fg: int) -> None:
    for v in range(8):
        for u in range(8):
            if glyph[v][u] == '#':
                put(slot, u, v, fg)
            # else leave 0 (transparent / HUD background fill)

def fill_bg_glyph(slot: int, glyph: tuple, fg: int, bg: int) -> None:
    # Like fill_glyph but writes `bg` to non-lit cells too, so the result is
    # a full opaque background tile (no transparent pixels).
    for v in range(8):
        for u in range(8):
            put(slot, u, v, fg if glyph[v][u] == '#' else bg)

def fill_multicolor(slot: int, glyph: tuple, legend: dict) -> None:
    # Multi-color tile: each glyph char maps to a palette index via `legend`.
    # Any char missing from the legend is treated as PAL_BG so tiles can
    # safely include "fallback" pixels.
    for v in range(8):
        for u in range(8):
            ch = glyph[v][u]
            put(slot, u, v, legend.get(ch, PAL_BG))

# tile 0: solid background
fill_solid(0, PAL_BG)

# tile 1: background with accent dots in the corners
fill_solid(1, PAL_BG)
for v, u in ((0, 0), (0, 7), (7, 0), (7, 7)):
    put(1, u, v, PAL_BG_ACCENT)

# tile 2: trench horizontal stripes (alternating dark/light rows)
for v in range(8):
    for u in range(8):
        put(2, u, v, PAL_BG_ACCENT if (v & 1) else PAL_BG)

# tile 3: dirt scatter (a few accent dots inside the cell)
fill_solid(3, PAL_BG)
for v, u in ((1, 3), (3, 1), (3, 5), (5, 2), (6, 6)):
    put(3, u, v, PAL_BG_ACCENT)

# tiles 4..10 = battlefield ground mosaic (grass + dirt with mud accents).
# Multi-color tiles using GROUND_LEGEND for palette mapping.
for slot, glyph in BATTLEFIELD_GLYPHS.items():
    fill_multicolor(slot, glyph, GROUND_LEGEND)

# tile 11 = artillery beam (vertical white line). HUD-style: leave non-lit
# pixels transparent so the beam can be overlaid on existing background
# tiles when SW writes it into the tile map (Batch C usage).
fill_glyph(BEAM_TILE_ID, BEAM_GLYPH, PAL_BORDER)

# tiles 16..41 = letter glyphs A..Z. tile_id = LETTER_TILE_BASE + ord(L) - 'A'.
# FG pixel = PAL_BORDER (0xFF white); compositor treats any non-zero pixel
# as lit and emits RGB_HUD_FG.
for letter, glyph in LETTER_GLYPHS.items():
    if letter == ':':
        slot = COLON_TILE_ID
```

```
        elif letter == ' ':
            slot = SPACE_TILE_ID
        else:
            slot = LETTER_TILE_BASE + (ord(letter) - ord('A'))
        fill_glyph(slot, glyph, PAL_BORDER)

    # tiles 48..57 = digit glyphs for HUD font. FG pixel = PAL_BORDER (0xFF
    # white). compositor.sv treats any non-zero pixel as "lit" and emits
    # RGB_HUD_FG, so any non-zero palette index works.
    for d, glyph in DIGIT_GLYPHS.items():
        fill_glyph(DIGIT_TILE_BASE + d, glyph, PAL_BORDER)

    return rom


def write_hex(path: Path, data: bytes) -> None:
    with path.open("w") as f:
        for byte in data:
            f.write(f"{byte:02x}\n")
    print(f"wrote {path} ({len(data)} bytes)")


PALETTE_DEPTH = 256
SPRITE_TABLE_DEPTH = 32


def make_palette():
    """24-bit RGB888 entries; default black, key colors set explicitly. Must
       stay in sync with sw/main.c init_palette_runtime() since that overrides
       this table once the C driver opens the device."""
    pal = [0] * PALETTE_DEPTH
    pal[PAL_PLAYER]       = 0x00FF00   # green
    pal[PAL_ENEMY_ARMED]  = 0xFF0000   # red
    pal[PAL_BULLET]       = 0xFFFF00   # yellow
    pal[PAL_ENEMY_UNARMD] = 0xFF80A0   # pink
    pal[PAL_MORTAR]       = 0xFF8000   # orange
    # 0x15 (wire gray) retired
    pal[PAL_GAS]          = 0xC0E000   # yellow-green
    pal[PAL_ARTILLERY]    = 0xFFFFE0   # bright white
    pal[PAL_AMMO_DROP]    = 0x00C040   # ammo crate green
    pal[PAL_ENEMY_BULLET] = 0xC02020   # enemy bullet red
    # Battlefield ground palette: brown dirt (3 shades) + green grass (3 shades)
    # + deep mud. Inspired by retro pixel-art tilesets where saturated earth
    # tones read clearly against the player/enemy sprites.
    pal[PAL_DIRT_MID]     = 0x6B4423   # mid brown (primary dirt)
    pal[PAL_DIRT_LIGHT]   = 0x9B7142   # sandy tan highlight
    pal[PAL_DIRT_DARK]    = 0x3A2515   # dark wet earth
    pal[PAL_GRASS_MID]    = 0x5A8B2E   # mid grass green (primary grass)
    pal[PAL_GRASS_DARK]   = 0x3D5A1F   # shadowed grass
    pal[PAL_GRASS_LIGHT]  = 0x8FBC3E   # highlight / new growth
    pal[PAL_MUD_DEEP]     = 0x1A0F08   # mud puddle interior
    pal[PAL_BORDER]       = 0xFFFFFF   # white (sprite borders)
    return pal


def make_sprite_table():
    """
    32 entries, 64 bits each, packed as the HW expects.
      W0 [31:0]  = y [31:16] | x [15:0]
```

```
      W1 [31:0]  = reserved | palette_off | flags | sprite_id
        flags layout: bit15=ACTIVE (workaround for sprite_eval), bits14:12=prio,
                      bit11=vflip, bit10=hflip, bits9:8=reserved
    Packed as int64: (W1 << 32) | W0.
    """
    def pack(x: int, y: int, sprite_id: int, prio: int = 0) -> int:
        x &= 0xFFFF
        y &= 0xFFFF
        w0 = (y << 16) | x
        active = 1 << 15                        # bit 47 of 64-bit word
        prio_field = (prio & 0x7) << 12         # bits 14:12
        w1 = (sprite_id & 0xFF) | active | prio_field
        return (w1 << 32) | w0

    table = [0] * SPRITE_TABLE_DEPTH

    # slot 0 = player, centered-ish, sprite_id 1
    table[0] = pack(x=312, y=232, sprite_id=1, prio=0)
    # slot 1 = enemy, top-left quadrant, sprite_id 2
    table[1] = pack(x=120, y=80,  sprite_id=2, prio=1)
    # slot 2 = enemy, top-right quadrant, sprite_id 2
    table[2] = pack(x=480, y=80,  sprite_id=2, prio=1)
    # slot 3 = bullet, sprite_id 3
    table[3] = pack(x=320, y=300, sprite_id=3, prio=2)
    # slots 4..31 stay 0 (active=0, invisible)

    return table


def write_hex_words(path: Path, values, width_bits: int) -> None:
    nibbles = width_bits // 4
    with path.open("w") as f:
        for v in values:
            f.write(f"{v:0{nibbles}x}\n")
    print(f"wrote {path} ({len(values)} entries, {width_bits}b each)")


def main() -> None:
    write_hex(HERE / "sprite_rom.hex", make_sprite_rom())
    write_hex(HERE / "tile_rom.hex",   make_tile_rom())
    write_hex_words(HERE / "palette.hex",       make_palette(),       24)
    write_hex_words(HERE / "sprite_table.hex",  make_sprite_table(),  64)


if __name__ == "__main__":
    main()
```

Software (`sw/`)

`main.c`

```
/*
 * main.c -- top-level loop.
 *
 * Two build modes are supported via the Makefile:
 *   - Default (FPGA target): mmaps nml_gpu via /dev/mem and renders to VGA.
```

```
 *   - make terminal (host build): falls back to render_terminal.c so the
 *     game logic can be exercised on a laptop without hardware. The selection
 *     is purely a link-time choice between render.c and render_terminal.c;
 *     this file is unchanged either way.
 */

#define _POSIX_C_SOURCE 200809L

#include "game.h"
#include "input.h"
#include "render.h"
#include "wave.h"
#include "autoatk.h"

#include <signal.h>
#include <stdio.h>
#include <time.h>

#ifndef NML_TERMINAL_BUILD
#  include "nml_gpu.h"
#endif

static volatile sig_atomic_t g_running = 1;

static void on_sigint(int sig) {
    (void)sig;
    g_running = 0;
}

/* Wave-progression tracker. -1 means "not initialized yet"; on the first
   iteration (or after a restart) we print only the START banner without a
   spurious CLEAR line. */
typedef struct {
    int last_wave_index;
    int last_frame;
    int score_at_wave_start;
    int kills_armed_at_wave_start;
    int kills_unarmed_at_wave_start;
} wave_tracker_t;

static void wave_tracker_init(wave_tracker_t *t) {
    t->last_wave_index            = -1;
    t->last_frame                 =  0;
    t->score_at_wave_start        =  0;
    t->kills_armed_at_wave_start  =  0;
    t->kills_unarmed_at_wave_start =  0;
}

static void print_wave_start(const game_t *g) {
    const wave_def_t *w = wave_current(g);
    printf("=== Wave %d START | %d enemies (%d armed @ %d HP, %d unarmed) "
           "| speed %d px/frame | spawn every %d frames ===\n",
           g->wave_index + 1, w->total_enemies, w->armed_count,
           w->armed_hp, w->total_enemies - w->armed_count,
           w->enemy_speed, w->spawn_period_frames);
    printf("    starting bullets=%d  artillery=%d  gas=%d  hp=%d\n",
           g->ammo, g->artillery_charges, g->gas_charges, g->player_hp);
    fflush(stdout);
}
```

```c
static void print_levelup_menu(const game_t *g) {
    printf("\n=== LEVEL UP === (wave %d cleared, pick one)\n", g->wave_index + 1);
    for (int i = 0; i < LEVELUP_OPTIONS; i++) {
        int kind = g->levelup_options[i];
        if (kind < 0 || kind >= AA_COUNT) continue;
        int lvl = g->autoatks[kind].level;
        char marker = (i == g->levelup_cursor) ? '>' : ' ';
        if (lvl >= AA_LEVEL_CAP) {
            printf("  %c [%d] %-12s (LV %d MAX)\n",
                   marker, i + 1, autoatk_name((autoatk_kind_t)kind), lvl);
        } else {
            printf("  %c [%d] %-12s (LV %d -> %d)\n",
                   marker, i + 1, autoatk_name((autoatk_kind_t)kind), lvl, lvl + 1);
        }
    }
    printf("  Use LEFT/RIGHT to move cursor, B to confirm.\n");
    fflush(stdout);
}

static void print_levelup_cursor(const game_t *g) {
    int kind = g->levelup_options[g->levelup_cursor];
    if (kind < 0 || kind >= AA_COUNT) return;
    printf("  cursor -> [%d] %s\n",
           g->levelup_cursor + 1, autoatk_name((autoatk_kind_t)kind));
    fflush(stdout);
}

static void print_levelup_selection(const game_t *g, autoatk_kind_t k) {
    printf("=== SELECTED: %s (now LV %d) ===\n\n",
           autoatk_name(k), g->autoatks[k].level);
    fflush(stdout);
}

/* Prints the CLEAR delta vs the last wave-start snapshot. Caller is
   responsible for refreshing the tracker after start of the next wave. */
static void print_wave_clear(const game_t *g, const wave_tracker_t *t) {
    if (t->last_wave_index < 0) return;
    int dscore = g->score         - t->score_at_wave_start;
    int dka    = g->kills_armed   - t->kills_armed_at_wave_start;
    int dku    = g->kills_unarmed - t->kills_unarmed_at_wave_start;
    printf("=== Wave %d CLEAR | +%d score | +%dA / +%dU kills | "
           "total score %d ===\n",
           t->last_wave_index + 1, dscore, dka, dku, g->score);
    fflush(stdout);
}

static void snapshot_wave_start(const game_t *g, wave_tracker_t *t) {
    t->last_wave_index            = g->wave_index;
    t->score_at_wave_start        = g->score;
    t->kills_armed_at_wave_start  = g->kills_armed;
    t->kills_unarmed_at_wave_start = g->kills_unarmed;
}

/* Detects fresh wave starts (initial or after upgrade). Used every tick.
   No longer handles CLEAR -- that's printed explicitly on PLAYING->LEVELUP. */
static void track_wave_transitions(const game_t *g, wave_tracker_t *t) {
    /* Restart detected: game.frame jumped backward. Forget previous state. */
    if (g->frame < t->last_frame) {
```

```
        wave_tracker_init(t);
    }
    t->last_frame = g->frame;

    if (g->wave_index == t->last_wave_index) return;

    print_wave_start(g);
    snapshot_wave_start(g, t);
}

#ifndef NML_TERMINAL_BUILD
/* The battlefield-ground init lives in render.c so the same tile_for()
   helper drives both startup paint and the post-game-over restore. Call
   render_init_tilemap() instead of duplicating the layout logic here. */

static void init_palette_runtime(void) {
    /* Mirrors hw/gen_rom.py palette indices. We rewrite them here so the SW
     * remains the authoritative source of palette data once the C driver is
     * in charge -- the FPGA $readmemh init only matters before this runs.
     * Keep this table in sync with make_palette() in gen_rom.py. */
    nml_write_palette(0x10, 0x00, 0xFF, 0x00); /* player: green                */
    nml_write_palette(0x11, 0xFF, 0x00, 0x00); /* armed enemy: red             */
    nml_write_palette(0x12, 0xFF, 0xFF, 0x00); /* bullet: yellow               */
    nml_write_palette(0x13, 0xFF, 0x80, 0xA0); /* unarmed enemy: pink          */
    nml_write_palette(0x14, 0xFF, 0x80, 0x00); /* mortar: orange               */
    /* 0x15 (barbed wire gray) retired -- AA_WIRE removed from game. Leaving
       the palette slot blank since gen_rom.py still ships an X glyph there
       and Batch B may reclaim it. */
    nml_write_palette(0x16, 0xC0, 0xE0, 0x00); /* mustard gas: yellow-green    */
    nml_write_palette(0x17, 0xFF, 0xFF, 0xE0); /* artillery flash: bright white*/
    nml_write_palette(0x18, 0x00, 0xC0, 0x40); /* ammo drop placeholder: green */
    nml_write_palette(0x19, 0xC0, 0x20, 0x20); /* enemy bullet placeholder: red*/
    /* Battlefield ground palette: brown dirt (3 shades) + green grass (3
       shades) + deep mud. Saturated earth tones so the player and enemy
       sprites still read clearly on top. Keep in sync with the same
       palette block in hw/gen_rom.py make_palette(). */
    nml_write_palette(0x20, 0x6B, 0x44, 0x23); /* PAL_DIRT_MID:   mid brown    */
    nml_write_palette(0x21, 0x9B, 0x71, 0x42); /* PAL_DIRT_LIGHT: sandy tan    */
    nml_write_palette(0x22, 0x3A, 0x25, 0x15); /* PAL_DIRT_DARK:  dark earth   */
    nml_write_palette(0x23, 0x5A, 0x8B, 0x2E); /* PAL_GRASS_MID:  primary grass*/
    nml_write_palette(0x24, 0x3D, 0x5A, 0x1F); /* PAL_GRASS_DARK: shadowed grass*/
    nml_write_palette(0x25, 0x8F, 0xBC, 0x3E); /* PAL_GRASS_LIGHT:new growth   */
    nml_write_palette(0x26, 0x1A, 0x0F, 0x08); /* PAL_MUD_DEEP:   puddle dark  */
    nml_write_palette(0xFF, 0xFF, 0xFF, 0xFF); /* sprite border                */
}
#endif

int main(void) {
#ifndef NML_TERMINAL_BUILD
    if (nml_open() != 0) {
        fprintf(stderr,
                "nml_open() failed -- make sure the FPGA is loaded and "
                "you're running as root (mmap /dev/mem).\n");
        return 1;
    }
    init_palette_runtime();
    render_init_tilemap();
    nml_set_enable(1);
    nml_set_hud_on(1);
```

```
#endif

    /* Catch Ctrl-C so we can shut down cleanly and turn the video off. */
    signal(SIGINT,  on_sigint);
    signal(SIGTERM, on_sigint);

    game_t game;
    game_init(&game);

    wave_tracker_t tracker;
    wave_tracker_init(&tracker);

    while (g_running) {
        struct timespec t0;
        clock_gettime(CLOCK_MONOTONIC, &t0);

        uint16_t input = input_read(game.frame);
        game_tick(&game, input);

        /* Natural game-flow ordering:
           PLAYING -> LEVELUP: "Wave N CLEAR" then the menu.
           In LEVELUP        : cursor moves.
           LEVELUP -> PLAYING: "SELECTED: X" then "Wave N+1 START" (via tracker).
           PLAYING -> GAMEOVER: the final breakdown block. */

        if (game.prev_state == STATE_PLAYING && game.state == STATE_LEVELUP) {
            print_wave_clear(&game, &tracker);
            print_levelup_menu(&game);
        }

        if (game.state == STATE_LEVELUP &&
            game.levelup_cursor != game.levelup_prev_cursor) {
            print_levelup_cursor(&game);
        }

        if (game.prev_state == STATE_LEVELUP && game.state == STATE_PLAYING) {
            int kind = game.levelup_options[game.levelup_cursor];
            if (kind >= 0 && kind < AA_COUNT) {
                print_levelup_selection(&game, (autoatk_kind_t)kind);
            }
        }

        /* Wave-START printer: handles initial wave on game start, every
           wave_index change (LEVELUP->PLAYING), and game restart. */
        track_wave_transitions(&game, &tracker);

        if (game.prev_state == STATE_PLAYING && game.state == STATE_GAMEOVER) {
            printf("\n=== GAME OVER ===\n");
            printf("  score        : %d\n", game.score);
            printf("  wave reached : %d\n", game.wave_index + 1);
            printf("  kills armed  : %d\n", game.kills_armed);
            printf("  kills unarmd : %d\n", game.kills_unarmed);
            printf("  bullets left : %d\n", game.ammo);
            printf("  art charges  : %d\n", game.artillery_charges);
            printf("  gas charges  : %d\n", game.gas_charges);
            printf("  bullet drops : %d\n", game.drops_collected);
            printf("  press START to restart\n");
            printf("=================\n\n");
            fflush(stdout);
```

```
        }

        render_frame(&game);

#ifndef NML_TERMINAL_BUILD
        nml_commit_frame();
#endif

        /* Frame pacing: 60 Hz fixed timestep. nanosleep is a coarse cap; the
         * SWAP wait above is the real timing edge once the FPGA is driving
         * us. The terminal build needs nanosleep alone. */
        struct timespec t1;
        clock_gettime(CLOCK_MONOTONIC, &t1);
        long elapsed_us = (t1.tv_sec - t0.tv_sec) * 1000000L
                        + (t1.tv_nsec - t0.tv_nsec) / 1000L;
        long target_us = 16666;
        if (elapsed_us < target_us) {
            struct timespec rem = {
                .tv_sec  = 0,
                .tv_nsec = (target_us - elapsed_us) * 1000L,
            };
            nanosleep(&rem, NULL);
        }
    }

    printf("\nShutting down. Final score: %d\n", game.score);

#ifndef NML_TERMINAL_BUILD
    nml_set_enable(0);
    nml_close();
#endif
    return 0;
}
```

game.h

```
#ifndef GAME_H
#define GAME_H

#include <stdint.h>

#define MAX_ENTITIES 64

#define SCREEN_W 640
#define SCREEN_H 480
#define HUD_H 24

/* D-pad (movement) */
#define INPUT_LEFT        (1u << 0)
#define INPUT_RIGHT       (1u << 1)
#define INPUT_UP          (1u << 2)
#define INPUT_DOWN        (1u << 3)
/* Face buttons -- each fires a bullet in the direction the button "points".
   B = DOWN, Y = LEFT, X = UP, A = RIGHT. */
#define INPUT_FIRE_DOWN   (1u << 4)
#define INPUT_FIRE_LEFT   (1u << 5)
#define INPUT_FIRE_UP     (1u << 6)
```

```
#define INPUT_FIRE_RIGHT  (1u << 7)
/* Shoulder buttons -- player-triggered abilities. */
#define INPUT_ABIL_ART    (1u << 8)   /* L: artillery beam   */
#define INPUT_ABIL_GAS    (1u << 9)   /* R: gas cloud        */
/* Menu / utility. */
#define INPUT_START       (1u << 10)
#define INPUT_SELECT      (1u << 11)
/* Legacy alias: code that still says INPUT_FIRE (e.g. level-up "confirm")
   means "the B button" -- which now fires DOWN. */
#define INPUT_FIRE        INPUT_FIRE_DOWN

/*Entity kinds. ENT_ENEMY_ARMED and ENT_ENEMY_UNARMED replaced the old
 generic ENT_ENEMY in batch 2; armed enemies have higher HP and are worth
 more score, unarmed are single-shot fodder. ENT_AUTO_PROJ and ENT_HAZARD
 added in batch 3 for the auto-attack system (mortar shells / gas clouds).
 ENT_ENEMY_BULLET and ENT_AMMO_DROP added in this batch: armed enemies
 shoot back, and kills can drop ammo pickups the player walks over.*/
typedef enum {
  ENT_NONE = 0,
  ENT_PLAYER,
  ENT_ENEMY_ARMED,
  ENT_ENEMY_UNARMED,
  ENT_BULLET,
  ENT_AUTO_PROJ,
  ENT_HAZARD,
  ENT_ENEMY_BULLET,
  ENT_AMMO_DROP
} ent_kind_t;

typedef enum {
  STATE_PLAYING = 0,
  STATE_LEVELUP,
  STATE_GAMEOVER
} game_state_t;

/* Auto-attack catalogue. Kept here (not in autoatk.h) so game_t can embed
   the active state directly without an extra include cycle.
   AA_WIRE was retired this batch (barbed wire removed from the game). */
typedef enum {
  AA_MORTAR = 0,
  AA_GAS,
  AA_ARTILLERY,
  AA_COUNT
} autoatk_kind_t;

typedef struct {
  int level;       /* 0 = not owned, 1..AA_LEVEL_CAP = owned & upgraded */
  int cooldown;    /* frames until next fire */
  int period;      /* current period in frames (derived from level) */
} autoatk_t;

#define AA_LEVEL_CAP 5
#define LEVELUP_OPTIONS 3
/*Struct which contains the various attributes of the
 kinds of entities that exist in the game:
their kind, whether or not they are active, their position, velocity
and hp*/
typedef struct{
  ent_kind_t kind;
```

```
  int active;
  int x,y;
  int vx, vy;
  int hp;
  int phase;   /* per-spawn random seed; used by enemy AI for de-synced motion */
  int ttl;     /* >0 = frames until despawn; 0 = no ttl (player/bullet/enemy) */
  int ttl_max; /* original ttl at spawn; used for animation phase (e.g. gas growth) */
  int payload; /* aux: for ENT_AUTO_PROJ/ENT_HAZARD, the autoatk_kind_t that spawned us */
  int fire_cd; /* armed-enemy fire cooldown; counts down each frame */
} entity_t;
/*Struct which contains the various statistics and information that
 are crucial to the updating process of the game*/
typedef struct{
  entity_t ents[MAX_ENTITIES];
  int player_i;
  int frame;
  int score;
  int player_hp;
  game_state_t state;
  uint16_t prev_input;

  /* Wave system (batch 2) */
  int wave_index;                /* 0-based; clamped to last wave */
  int wave_enemies_spawned;      /* enemies emitted this wave so far */
  int wave_armed_remaining;      /* armed enemies still to spawn this wave */
  int wave_spawn_cooldown;       /* frames until next spawn */
  int kills_armed;
  int kills_unarmed;
  int fire_cooldown;             /* frames until player can fire again */
  game_state_t prev_state;       /* for one-shot game-over transition events */

  /* Auto-attack system (batch 3) */
  autoatk_t autoatks[AA_COUNT];
  int levelup_options[LEVELUP_OPTIONS];   /* autoatk_kind_t values; -1 = empty */
  int levelup_cursor;                     /* 0..LEVELUP_OPTIONS-1 */
  int levelup_prev_cursor;                /* for cursor-move edge events in main.c */

  /* Ammo + ability charges (this batch). Fully additive with caps:
     on wave_advance() each is bumped by its refill amount, capped at the max.
     Initial values come from game_init(). */
  int ammo;
  int artillery_charges;
  int gas_charges;
  int art_input_cd;       /* L-button post-fire lockout, frames */
  int gas_input_cd;       /* R-button post-fire lockout, frames */
  int drops_collected;    /* lifetime stat for the game-over log */

  /* Artillery beam visual state (Batch C). beam_ttl counts down each frame
     while the tile-map beam column is drawn; render.c saves/restores the
     underlying tiles based on this lifecycle. */
  int beam_ttl;
  int beam_col;           /* tile-map column (0..NML_TILEMAP_COLS) */
} game_t;

/* Ammo / charge tuning. */
#define AMMO_MAX             99
#define CHARGE_MAX           5
#define AMMO_INITIAL         40
#define ART_CHARGES_INITIAL  2
```

```
#define GAS_CHARGES_INITIAL  2
#define AMMO_REFILL_PER_WAVE 40
#define CHARGES_REFILL_PER_WAVE 2
#define AMMO_PICKUP_AMOUNT   10
#define AMMO_DROP_PERCENT    30
#define ABILITY_INPUT_COOLDOWN 20

void game_init(game_t *g);
void game_tick(game_t *g, uint16_t input);

/* Damage an enemy entity. On kill it scores, increments the kill counter,
   rolls for an ammo drop, and deactivates the entity. Exposed so autoatk.c
   can route its artillery instakill through the same bookkeeping path. */
void apply_damage(game_t *g, entity_t *e, int damage);

#endif
```

game.c

```
#include "game.h"
#include "wave.h"
#include "autoatk.h"
#include <stdlib.h>

/* Score weights -- tune here. */
#define SCORE_KILL_ARMED    200
#define SCORE_KILL_UNARMED  50

/* Player fire cooldown in frames. 60 fps -> 5 frames == 12 shots/sec. Shared
   across all 4 face buttons so the four-direction setup can't fire 4 bullets
   on the same tick (first-direction-found-in-the-priority-order wins). */
#define FIRE_COOLDOWN_FRAMES 5

/* Player bullet speed (always axis-aligned). */
#define BULLET_SPEED 8

/* Armed-enemy fire cadence + bullet speed. Generous gap so the player has
   time to dodge; cadence is randomized at spawn so the cohort doesn't
   coordinate. */
#define ENEMY_FIRE_BASE_COOLDOWN  120
#define ENEMY_FIRE_JITTER         60
#define ENEMY_BULLET_SPEED        3

/* Ammo drop entity lifetime in frames. */
#define AMMO_DROP_TTL  600   /* 10 seconds */

/*Function that spawns the entities on the map at coordinates x and y according to their kind*/

static int spawn_entity(game_t *g, ent_kind_t kind, int x, int y){
  for (int i = 0; i < MAX_ENTITIES; i++){
    if(!g->ents[i].active){
      g->ents[i].active = 1;
      g->ents[i].kind = kind;
      g->ents[i].x = x;
      g->ents[i].y = y;
      g->ents[i].vx = 0;
      g->ents[i].vy = 0;
```

```
      g->ents[i].hp = 1;
      g->ents[i].phase = 0;     /* player/bullet don't use phase; only enemies do */
      g->ents[i].ttl = 0;
      g->ents[i].ttl_max = 0;
      g->ents[i].payload = 0;
      g->ents[i].fire_cd = 0;
      return i;
    }
  }
  return -1;
}

static int is_enemy(const entity_t *e) {
    return e->kind == ENT_ENEMY_ARMED || e->kind == ENT_ENEMY_UNARMED;
}

static void maybe_spawn_ammo_drop(game_t *g, int x, int y) {
    if ((rand() % 100) >= AMMO_DROP_PERCENT) return;
    int slot = spawn_entity(g, ENT_AMMO_DROP, x, y);
    if (slot < 0) return;
    g->ents[slot].ttl     = AMMO_DROP_TTL;
    g->ents[slot].ttl_max = AMMO_DROP_TTL;
}

/*Initializes the game at frame 0 and with the player having 100 hp*/
void game_init(game_t *g) {
    for (int i = 0; i < MAX_ENTITIES; i++) {
        g->ents[i].active = 0;
    }

    g->frame = 0;
    g->score = 0;
    g->player_hp = 100;
    g->state = STATE_PLAYING;
    g->prev_state = STATE_PLAYING;
    g->prev_input = 0;
    g->fire_cooldown = 0;

    g->ammo              = AMMO_INITIAL;
    g->artillery_charges = ART_CHARGES_INITIAL;
    g->gas_charges       = GAS_CHARGES_INITIAL;
    g->art_input_cd      = 0;
    g->gas_input_cd      = 0;
    g->drops_collected   = 0;
    g->beam_ttl          = 0;
    g->beam_col          = 0;

    g->player_i = spawn_entity(g, ENT_PLAYER, SCREEN_W / 2, SCREEN_H - 60);

    wave_system_init(g);
    autoatk_init(g);
}

/* Edge-detect: was the bit unset last tick and set this tick? Static here
   because both update_player and tick_levelup want it; tick_levelup needs
   access via a local copy below. */
static int input_pressed(const game_t *g, uint16_t input, uint16_t bit) {
    return (input & bit) && !(g->prev_input & bit);
}
```

```
/* Try to fire one bullet in the requested direction. The cooldown is shared
   across all four directions: if any face button is held the player gets
   12 shots/sec, and if multiple are pressed the priority order (DOWN, LEFT,
   UP, RIGHT) decides which direction wins. Returns 1 if a bullet was fired. */
static int try_fire_bullet(game_t *g, int vx, int vy, int origin_dx, int origin_dy) {
    const entity_t *p = &g->ents[g->player_i];
    if (g->fire_cooldown != 0) return 0;
    if (g->ammo <= 0) return 0;
    int b = spawn_entity(g, ENT_BULLET, p->x + origin_dx, p->y + origin_dy);
    if (b < 0) return 0;
    g->ents[b].vx = vx;
    g->ents[b].vy = vy;
    g->fire_cooldown = FIRE_COOLDOWN_FRAMES;
    g->ammo--;
    return 1;
}

/* L-button: artillery beam. R-button: gas cloud. Both use edge-detect plus
   a 20-frame post-fire lockout to defend against bouncy SNES buttons. */
static void try_fire_abilities(game_t *g, uint16_t input) {
    if (g->art_input_cd == 0 &&
        input_pressed(g, input, INPUT_ABIL_ART) &&
        g->artillery_charges > 0) {
        if (autoatk_fire_artillery(g)) {
            g->artillery_charges--;
            g->art_input_cd = ABILITY_INPUT_COOLDOWN;
        }
    }
    if (g->gas_input_cd == 0 &&
        input_pressed(g, input, INPUT_ABIL_GAS) &&
        g->gas_charges > 0) {
        if (autoatk_fire_gas(g)) {
            g->gas_charges--;
            g->gas_input_cd = ABILITY_INPUT_COOLDOWN;
        }
    }
}

/*Updates the player's position and actions according to the received input*/
static void update_player(game_t *g, uint16_t input) {
    entity_t *p = &g->ents[g->player_i];
    int speed = 4;

    if (input & INPUT_LEFT)  p->x -= speed;
    if (input & INPUT_RIGHT) p->x += speed;
    if (input & INPUT_UP)    p->y -= speed;
    if (input & INPUT_DOWN)  p->y += speed;

    if (p->x < 0) p->x = 0;
    if (p->x > SCREEN_W - 16) p->x = SCREEN_W - 16;
    if (p->y < HUD_H) p->y = HUD_H;
    if (p->y > SCREEN_H - 16) p->y = SCREEN_H - 16;

    if (g->fire_cooldown > 0) g->fire_cooldown--;
    if (g->art_input_cd  > 0) g->art_input_cd--;
    if (g->gas_input_cd  > 0) g->gas_input_cd--;
    if (g->beam_ttl      > 0) g->beam_ttl--;
```

```
    /* Four-direction face-button fire. Bullets spawn just outside the player
       sprite on the firing side so they don't immediately self-collide. */
    if (input & INPUT_FIRE_DOWN)  try_fire_bullet(g,  0,  BULLET_SPEED,  8,  16);
    else if (input & INPUT_FIRE_LEFT)  try_fire_bullet(g, -BULLET_SPEED, 0, -8,   8);
    else if (input & INPUT_FIRE_UP)    try_fire_bullet(g,  0, -BULLET_SPEED, 8, -8);
    else if (input & INPUT_FIRE_RIGHT) try_fire_bullet(g,  BULLET_SPEED, 0, 16,   8);

    try_fire_abilities(g, input);
}

/*Step a coord toward a target by at most `speed` pixels, clamping the final
  step so we don't overshoot and oscillate.*/
static int step_toward(int from, int to, int speed) {
    int delta = to - from;
    if (delta >  speed) return from + speed;
    if (delta < -speed) return from - speed;
    return to;
}

/*Updates enemy motion. Vertical: enemies always close on the player's row at
  the wave's speed, so they reliably threaten the trench. Horizontal: a
  per-enemy random walk that re-decides direction every ENEMY_DIR_FRAMES.
  Phase desync prevents the whole cohort from synchronizing into a column,
  so the player's bullet stream no longer mops them up trivially.

  Decision distribution per re-roll (256 buckets):
    [   0,  64) -> strafe left   (25%)
    [  64, 128) -> strafe right  (25%)
    [ 128, 256) -> chase player  (50%) -- keeps a weak pull so enemies still
                                          gravitate toward the player overall.

  Armed enemies also fire bullets at the player on a per-enemy cooldown
  staggered at spawn. Bullet velocity locks on at fire time (no homing).*/
#define ENEMY_DIR_FRAMES 12u

static unsigned enemy_rng(int frame, int phase) {
    unsigned window = (unsigned)(frame + phase) / ENEMY_DIR_FRAMES;
    return (window * 1103515245u + (unsigned)phase * 12345u) >> 24;
}

/* Spawn an enemy bullet from `e` aimed at the player's current position.
   Manhattan-norm velocity so the bullet stays on a clean integer trajectory
   without floats. The bullet doesn't track the player after spawn. */
static void enemy_fire_bullet(game_t *g, const entity_t *e) {
    const entity_t *p = &g->ents[g->player_i];
    int dx = (p->x + 8) - (e->x + 8);
    int dy = (p->y + 8) - (e->y + 8);
    int absdx = dx < 0 ? -dx : dx;
    int absdy = dy < 0 ? -dy : dy;
    int norm = absdx + absdy;
    if (norm == 0) norm = 1;
    int vx = (dx * ENEMY_BULLET_SPEED) / norm;
    int vy = (dy * ENEMY_BULLET_SPEED) / norm;
    /* Avoid zero-velocity bullets if the player is exactly on top of the
       enemy: bias downward so it at least leaves the muzzle. */
    if (vx == 0 && vy == 0) vy = ENEMY_BULLET_SPEED;
    int b = spawn_entity(g, ENT_ENEMY_BULLET, e->x + 4, e->y + 4);
    if (b < 0) return;
    g->ents[b].vx = vx;
```

```
    g->ents[b].vy = vy;
}

static void update_enemies(game_t *g) {
    entity_t *p = &g->ents[g->player_i];
    int speed = wave_current(g)->enemy_speed;
    if (speed < 1) speed = 1;

    for (int i = 0; i < MAX_ENTITIES; i++) {
        entity_t *e = &g->ents[i];
        if (!e->active || !is_enemy(e)) continue;

        e->y = step_toward(e->y, p->y, speed);

        unsigned r = enemy_rng(g->frame, e->phase);
        if      (r <  64) e->x -= speed;
        else if (r < 128) e->x += speed;
        else              e->x = step_toward(e->x, p->x, speed);

        if (e->x < 0)              e->x = 0;
        if (e->x > SCREEN_W - 16)  e->x = SCREEN_W - 16;

        /* Armed enemies shoot back. */
        if (e->kind == ENT_ENEMY_ARMED) {
            if (e->fire_cd > 0) {
                e->fire_cd--;
            } else {
                enemy_fire_bullet(g, e);
                e->fire_cd = ENEMY_FIRE_BASE_COOLDOWN + (rand() % ENEMY_FIRE_JITTER);
            }
        }

        /*Makes the player lose hp if an enemy steps over the trench!*/
        if(e->y >= SCREEN_H - 20){
          e->active = 0;
          g->player_hp -= 10;
        }
    }
}

/*Updates player bullet position; bullets despawn when they leave the screen.*/
static void update_bullets(game_t *g) {
    for (int i = 0; i < MAX_ENTITIES; i++) {
        entity_t *e = &g->ents[i];
        if (!e->active || e->kind != ENT_BULLET) continue;

        e->x += e->vx;
        e->y += e->vy;

        if (e->y < 0 || e->y > SCREEN_H ||
            e->x < 0 || e->x > SCREEN_W) e->active = 0;
    }
}

/* Move enemy bullets each frame. Despawn off-screen. Collision with player
   is handled in handle_collisions(). */
static void update_enemy_bullets(game_t *g) {
    for (int i = 0; i < MAX_ENTITIES; i++) {
        entity_t *e = &g->ents[i];
```

```
        if (!e->active || e->kind != ENT_ENEMY_BULLET) continue;

        e->x += e->vx;
        e->y += e->vy;

        if (e->y < 0 || e->y > SCREEN_H ||
            e->x < 0 || e->x > SCREEN_W) e->active = 0;
    }
}

/* Tick ammo-drop ttl and despawn expired drops. Pickup collision is in
   handle_collisions(). Drops don't move. */
static void update_ammo_drops(game_t *g) {
    for (int i = 0; i < MAX_ENTITIES; i++) {
        entity_t *e = &g->ents[i];
        if (!e->active || e->kind != ENT_AMMO_DROP) continue;
        if (e->ttl > 0) {
            e->ttl--;
            if (e->ttl <= 0) e->active = 0;
        }
    }
}

/*Boolean function that returns 1 if entities a and b are making contact with each other*/
static int touching(entity_t *a, entity_t *b) {
    return a->x < b->x + 16 &&
           a->x + 16 > b->x &&
           a->y < b->y + 16 &&
           a->y + 16 > b->y;
}

/* Padded AABB overlap for hazards with wider effective hitboxes (e.g. gas). */
static int touching_pad(entity_t *a, entity_t *b, int pad) {
    return a->x - pad < b->x + 16 &&
           a->x + 16 + pad > b->x &&
           a->y - pad < b->y + 16 &&
           a->y + 16 + pad > b->y;
}

/* Apply `damage` to enemy `e`; if it kills, score + kills counters update,
   and we roll for an ammo drop at the kill site. Exposed via game.h so
   autoatk_fire_artillery() can route its instakill through here and get
   the drop probability uniformly. */
void apply_damage(game_t *g, entity_t *e, int damage) {
    if (damage <= 0) return;
    if (e->hp <= damage) {
        if (e->kind == ENT_ENEMY_ARMED) {
            g->score += SCORE_KILL_ARMED;
            g->kills_armed++;
        } else {
            g->score += SCORE_KILL_UNARMED;
            g->kills_unarmed++;
        }
        maybe_spawn_ammo_drop(g, e->x, e->y);
        e->active = 0;
    } else {
        e->hp -= damage;
    }
}
```

```c
/*Handles all collisions: enemy<->player touch, player bullets<->enemy,
  auto-projectile/hazard<->enemy, enemy-bullet<->player, ammo-drop<->player.
  Damage and score are funneled through apply_damage() so kill bookkeeping
  stays consistent across damage sources.*/
static void handle_collisions(game_t *g) {
    entity_t *p = &g->ents[g->player_i];

    for (int i = 0; i < MAX_ENTITIES; i++) {
        entity_t *e = &g->ents[i];
        if (!e->active) continue;

        /* Player vs ammo drop: pick it up. */
        if (e->kind == ENT_AMMO_DROP) {
            if (touching(p, e)) {
                g->ammo += AMMO_PICKUP_AMOUNT;
                if (g->ammo > AMMO_MAX) g->ammo = AMMO_MAX;
                g->drops_collected++;
                e->active = 0;
            }
            continue;
        }

        /* Player vs enemy bullet: HP damage, bullet despawns. */
        if (e->kind == ENT_ENEMY_BULLET) {
            if (touching(p, e)) {
                g->player_hp -= 10;
                e->active = 0;
            }
            continue;
        }

        if (!is_enemy(e)) continue;

        /* Enemy touches player: 10 HP off, enemy dies. No score for suicide
           kills since neither bullet nor auto-attack made the play. */
        if (touching(p, e)) {
            e->active = 0;
            g->player_hp -= 10;
            continue;
        }

        /* Enemy vs player bullets: 1 damage per bullet, bullet despawns. */
        for (int j = 0; j < MAX_ENTITIES; j++) {
            entity_t *b = &g->ents[j];
            if (!b->active || b->kind != ENT_BULLET) continue;
            if (!touching(b, e)) continue;
            b->active = 0;
            apply_damage(g, e, 1);
            if (!e->active) break;
        }
        if (!e->active) continue;

        /* Enemy vs auto-projectiles / hazards. Damage and despawn-on-hit
           policy come from autoatk.c per the payload weapon kind. Gas uses
           its own size-aware overlap so the expanding cloud's hitbox tracks
           the visible sprite. */
        for (int j = 0; j < MAX_ENTITIES; j++) {
            entity_t *q = &g->ents[j];
```

```
            if (!q->active) continue;
            if (q->kind != ENT_AUTO_PROJ && q->kind != ENT_HAZARD) continue;

            int hit;
            if (q->kind == ENT_HAZARD && q->payload == (int)AA_GAS) {
                hit = autoatk_gas_overlaps(q, e->x, e->y);
            } else {
                int pad = autoatk_hitbox_pad((autoatk_kind_t)q->payload);
                hit = touching_pad(q, e, pad);
            }
            if (!hit) continue;

            apply_damage(g, e, autoatk_damage((autoatk_kind_t)q->payload));
            if (autoatk_despawn_on_hit((autoatk_kind_t)q->payload)) {
                q->active = 0;
            }
            if (!e->active) break;
        }
    }
}

/* LEVELUP state input handling. Returns 1 if the player confirmed a choice
   this tick; the caller should then apply the upgrade and advance the wave. */
static int tick_levelup(game_t *g, uint16_t input) {
    g->levelup_prev_cursor = g->levelup_cursor;

    if (input_pressed(g, input, INPUT_LEFT)) {
        g->levelup_cursor = (g->levelup_cursor + LEVELUP_OPTIONS - 1) % LEVELUP_OPTIONS;
    }
    if (input_pressed(g, input, INPUT_RIGHT)) {
        g->levelup_cursor = (g->levelup_cursor + 1) % LEVELUP_OPTIONS;
    }
    return input_pressed(g, input, INPUT_FIRE);
}

void game_tick(game_t *g, uint16_t input) {
    g->prev_state = g->state;
    g->frame++;

    if (g->state == STATE_GAMEOVER) {
        if (input_pressed(g, input, INPUT_START)) {
            game_init(g);
            g->prev_input = input;
            return;
        }
        g->prev_input = input;
        return;
    }

    if (g->state == STATE_LEVELUP) {
        if (tick_levelup(g, input)) {
            int pick_idx = g->levelup_cursor;
            int kind = g->levelup_options[pick_idx];
            if (kind >= 0 && kind < AA_COUNT) {
                autoatk_upgrade(g, (autoatk_kind_t)kind);
            }
            wave_advance(g);
            g->state = STATE_PLAYING;
        }
```

```
        g->prev_input = input;
        return;
    }

    /* STATE_PLAYING */
    update_player(g, input);
    wave_tick(g);
    update_enemies(g);
    update_bullets(g);
    update_enemy_bullets(g);
    update_ammo_drops(g);
    autoatk_tick(g);
    autoatk_update_proj(g);
    handle_collisions(g);

    /* On wave clear, pause for level-up. The wave doesn't advance here --
       it advances when the player confirms an upgrade in STATE_LEVELUP. */
    if (wave_complete(g)) {
        g->state = STATE_LEVELUP;
        autoatk_pick_levelup_options(g);
    }

    if (g->player_hp <= 0) {
        g->state = STATE_GAMEOVER;
        /* Freeze the death frame: deactivate everything except the player so the
           game-over screen has clean slots to draw into. */
        for (int i = 0; i < MAX_ENTITIES; i++) {
            if (i != g->player_i) g->ents[i].active = 0;
        }
        g->ents[g->player_i].active = 0;
    }

    g->prev_input = input;
}
```

wave.h

```
#ifndef WAVE_H
#define WAVE_H

#include "game.h"

typedef struct {
    int spawn_period_frames;   /* frames between consecutive spawns */
    int total_enemies;         /* enemies emitted this wave */
    int armed_count;           /* of total, how many are ENT_ENEMY_ARMED */
    int armed_hp;              /* hp for armed enemies in this wave */
    int enemy_speed;           /* pixels/frame both axes; >=1 */
} wave_def_t;

extern const int        WAVE_COUNT;
extern const wave_def_t WAVES[];

/* Sets wave_index = 0 and seeds the per-wave counters. Call from game_init(). */
void                wave_system_init(game_t *g);

/* Returns the active wave def. After the final wave it clamps to the last
```

```
   entry rather than returning NULL, so the game can keep running. */
const wave_def_t *  wave_current(const game_t *g);

/* Runs once per game_tick during STATE_PLAYING. Spawns enemies according
   to the active wave's schedule. Armed enemies emit first (deterministic). */
void                wave_tick(game_t *g);

/* True when every enemy this wave was going to spawn has spawned, and no
   enemy entity (armed or unarmed) is still active. */
int                 wave_complete(const game_t *g);

/* Bumps wave_index (clamps at last wave) and resets per-wave counters. */
void                wave_advance(game_t *g);

#endif
```

wave.c

```
#include "wave.h"
#include <stdlib.h>

/*
 * Hardcoded wave table. To rebalance, edit this array and rebuild -- no
 * runtime file I/O. The narrative-arc shift toward more enemies and a higher
 * armed-fraction over time lives entirely in this table.
 */
const wave_def_t WAVES[] = {
    /* spawn_period, total, armed, armed_hp, enemy_speed */
    {  60,  6, 0, 1, 1 },   /* wave 1 -- warmup, all unarmed, slow */
    {  50,  8, 1, 2, 1 },   /* wave 2 -- first armed appears */
    {  45, 10, 2, 2, 2 },   /* wave 3 -- enemies move 2x faster */
    {  35, 12, 4, 2, 2 },   /* wave 4 */
    {  25, 16, 6, 3, 3 },   /* wave 5 -- loops; fastest + tankiest */
};
const int WAVE_COUNT = (int)(sizeof(WAVES) / sizeof(WAVES[0]));

/* spawn_entity is static inside game.c; re-implement the bits we need here.
   game.c retains ownership of the entity pool layout. */
static int alloc_entity(game_t *g) {
    for (int i = 0; i < MAX_ENTITIES; i++) {
        if (!g->ents[i].active) return i;
    }
    return -1;
}

static int active_enemy_count(const game_t *g) {
    int n = 0;
    for (int i = 0; i < MAX_ENTITIES; i++) {
        const entity_t *e = &g->ents[i];
        if (!e->active) continue;
        if (e->kind == ENT_ENEMY_ARMED || e->kind == ENT_ENEMY_UNARMED) n++;
    }
    return n;
}

const wave_def_t *wave_current(const game_t *g) {
    int idx = g->wave_index;
```

```
    if (idx < 0)              idx = 0;
    if (idx >= WAVE_COUNT)    idx = WAVE_COUNT - 1;
    return &WAVES[idx];
}

void wave_system_init(game_t *g) {
    g->wave_index           = 0;
    g->wave_enemies_spawned = 0;
    g->wave_armed_remaining = WAVES[0].armed_count;
    g->wave_spawn_cooldown  = 0;   /* first enemy spawns immediately */
    g->kills_armed          = 0;
    g->kills_unarmed        = 0;
}

void wave_tick(game_t *g) {
    const wave_def_t *w = wave_current(g);

    /* Already finished spawning this wave? wave_complete() will trip once
       the on-screen enemies are cleared and game_tick will advance us. */
    if (g->wave_enemies_spawned >= w->total_enemies) return;

    if (g->wave_spawn_cooldown > 0) {
        g->wave_spawn_cooldown--;
        return;
    }

    int slot = alloc_entity(g);
    if (slot < 0) {
        /* Pool full; try again next frame. Don't burn the cooldown. */
        return;
    }

    int spawn_armed = (g->wave_armed_remaining > 0);

    entity_t *e = &g->ents[slot];
    e->active = 1;
    e->kind   = spawn_armed ? ENT_ENEMY_ARMED : ENT_ENEMY_UNARMED;
    e->x      = rand() % (SCREEN_W - 16);
    e->y      = HUD_H;
    e->vx     = 0;
    e->vy     = 0;
    e->hp     = spawn_armed ? w->armed_hp : 1;
    e->phase  = rand() & 0xff;       /* desync horizontal random-walk across enemies */
    e->ttl    = 0;
    e->ttl_max = 0;
    e->payload = 0;
    /* Stagger initial fire cooldown for armed enemies so they don't all fire
       on the same frame; unarmed enemies never fire so the field is unused. */
    e->fire_cd = spawn_armed ? (60 + (rand() % 120)) : 0;

    g->wave_enemies_spawned++;
    if (spawn_armed) g->wave_armed_remaining--;
    g->wave_spawn_cooldown = w->spawn_period_frames;
}

int wave_complete(const game_t *g) {
    const wave_def_t *w = wave_current(g);
    return (g->wave_enemies_spawned >= w->total_enemies) &&
           (active_enemy_count(g) == 0);
```

```
}

void wave_advance(game_t *g) {
    if (g->wave_index < WAVE_COUNT - 1) {
        g->wave_index++;
    }
    const wave_def_t *w = wave_current(g);
    g->wave_enemies_spawned = 0;
    g->wave_armed_remaining = w->armed_count;
    g->wave_spawn_cooldown  = w->spawn_period_frames;  /* brief pause between waves */

    /* Fully additive refill (capped). Wave-start is the only top-up; mid-wave
       ammo comes from drops. Picking conservative caps so the player can't
       hoard infinitely. */
    g->ammo = g->ammo + AMMO_REFILL_PER_WAVE;
    if (g->ammo > AMMO_MAX) g->ammo = AMMO_MAX;
    g->artillery_charges = g->artillery_charges + CHARGES_REFILL_PER_WAVE;
    if (g->artillery_charges > CHARGE_MAX) g->artillery_charges = CHARGE_MAX;
    g->gas_charges = g->gas_charges + CHARGES_REFILL_PER_WAVE;
    if (g->gas_charges > CHARGE_MAX) g->gas_charges = CHARGE_MAX;
}
```

autoatk.h

```
#ifndef AUTOATK_H
#define AUTOATK_H

#include "game.h"

/* Resets all weapons to level 0 / cooldown 0. Call from game_init. */
void          autoatk_init(game_t *g);

/* Decrements the mortar cooldown and fires it when ready. Gas and Artillery
   are now player-triggered (see autoatk_try_fire_gas / artillery below) so
   they are NOT serviced here. Call once per PLAYING tick. */
void          autoatk_tick(game_t *g);

/* Moves auto-projectiles, ages hazards (decrements ttl, deactivates at 0).
   Call once per PLAYING tick after autoatk_tick. */
void          autoatk_update_proj(game_t *g);

/* Bumps the given weapon's level (capped at AA_LEVEL_CAP) and refreshes its
   period (mortar only; Gas/Artillery upgrades currently affect nothing
   game-side besides scoring "owned" semantics, but we still track levels so
   the level-up menu can show them). Picking a maxed weapon is a no-op. */
void          autoatk_upgrade(game_t *g, autoatk_kind_t k);

/* Damage a single hit from this weapon does. Used by handle_collisions. */
int           autoatk_damage(autoatk_kind_t k);

/* True iff the auto-projectile/hazard should despawn the moment it lands a hit.
   Mortar shells despawn; persistent hazards (gas) do not. */
int           autoatk_despawn_on_hit(autoatk_kind_t k);

/* Padded AABB hitbox helper for hazards. For AA_GAS the live hitbox grows
   with the cloud, so callers should pass the entity to get the right size. */
int           autoatk_hitbox_pad(autoatk_kind_t k);
```

```c
/* Returns 1 if `e` is a gas cloud entity (ENT_HAZARD with payload=AA_GAS)
   and the given target rect overlaps the gas cloud's current (size-aware)
   hitbox. (target rect is the standard 16x16 sprite at tx,ty.) */
int           autoatk_gas_overlaps(const entity_t *e, int tx, int ty);

/* Returns the current size of a gas cloud entity, expanding from a small
   puff (16x16) to full size (48x32) over the first ~30 frames of life,
   then holding, then shrinking. (out_w, out_h are full size in pixels.) */
void          autoatk_gas_size(const entity_t *e, int *out_w, int *out_h);

/* Player-triggered abilities. Each returns 1 if it actually fired (caller
   should then decrement the corresponding charge counter & set the input
   cooldown), 0 if blocked (no enemies in target column, slot pool full, etc.). */
int           autoatk_fire_artillery(game_t *g);
int           autoatk_fire_gas(game_t *g);

/* Fills g->levelup_options[] with up to LEVELUP_OPTIONS distinct random
   weapon kinds. Resets g->levelup_cursor to 0. */
void          autoatk_pick_levelup_options(game_t *g);

const char *  autoatk_name(autoatk_kind_t k);

#endif
```

autoatk.c

```c
#include "autoatk.h"
#include <stdlib.h>

/* Period at level 1, in frames. Higher level shortens by PERIOD_STEP per
   level, floored at PERIOD_MIN so a maxed weapon still has breathing room.
   Only AA_MORTAR is timer-driven now; AA_GAS and AA_ARTILLERY are
   player-triggered abilities with charges (see autoatk_fire_*). Their
   period entries are unused but kept so AA_COUNT-sized arrays stay aligned
   with the enum. */
static const int BASE_PERIOD[AA_COUNT] = {
    [AA_MORTAR]    = 180,
    [AA_GAS]       = 0,
    [AA_ARTILLERY] = 0,
};
static const int PERIOD_STEP[AA_COUNT] = {
    [AA_MORTAR]    = 30,
    [AA_GAS]       = 0,
    [AA_ARTILLERY] = 0,
};
#define PERIOD_MIN 60   /* 1 second */

/* Gas cloud animation phases. ttl_max == GAS_TTL_MAX; we map remaining ttl
   to one of three windows: grow, hold, shrink. Sizes are full-cloud pixels. */
#define GAS_TTL_MAX     90
#define GAS_GROW_FRAMES 30
#define GAS_SHRINK_FRAMES 15
#define GAS_W_MIN       16
#define GAS_W_MAX       48
#define GAS_H_MIN       16
#define GAS_H_MAX       32
```

```c
#define GAS_BASE_HITPAD 4    /* small additive pad on top of size-derived box */

/* Artillery beam: how many frames the tile-map beam stays visible. The
   visual itself is a column of tile-11 (vertical white line) glyphs written
   into the tile map by render.c when g->beam_ttl > 0. */
#define BEAM_VISUAL_TTL     6
#define ARTILLERY_KILL_DMG  999

static int period_for(autoatk_kind_t k, int level) {
    if (level <= 0) return 0;
    if (BASE_PERIOD[k] == 0) return 0;   /* non-timer weapons */
    int p = BASE_PERIOD[k] - PERIOD_STEP[k] * (level - 1);
    if (p < PERIOD_MIN) p = PERIOD_MIN;
    return p;
}

static int alloc_entity(game_t *g) {
    for (int i = 0; i < MAX_ENTITIES; i++) {
        if (!g->ents[i].active) return i;
    }
    return -1;
}

static void emit_proj(game_t *g, ent_kind_t kind, int x, int y, int vx, int vy,
                      int ttl, autoatk_kind_t payload) {
    int slot = alloc_entity(g);
    if (slot < 0) return;
    entity_t *e = &g->ents[slot];
    e->active  = 1;
    e->kind    = kind;
    e->x       = x;
    e->y       = y;
    e->vx      = vx;
    e->vy      = vy;
    e->hp      = 1;
    e->phase   = 0;
    e->ttl     = ttl;
    e->ttl_max = ttl;
    e->payload = (int)payload;
    e->fire_cd = 0;
}

/* Mortar (the only remaining auto-fire weapon) -- spawns near the player and
   launches upward so it visually reads as the player's lobbed attack rather
   than an enemy mortar dropping in. */
static void fire_mortar(game_t *g) {
    const entity_t *p = &g->ents[g->player_i];
    int jitter = (rand() % 65) - 32;        /* -32 .. +32 px */
    int x = p->x + jitter;
    if (x < 0) x = 0;
    if (x > SCREEN_W - 16) x = SCREEN_W - 16;
    int y = p->y - 8;
    if (y < HUD_H) y = HUD_H;
    emit_proj(g, ENT_AUTO_PROJ, x, y, 0, -6, /*ttl=*/200, AA_MORTAR);
}

void autoatk_init(game_t *g) {
    for (int i = 0; i < AA_COUNT; i++) {
        g->autoatks[i].level    = 0;
```

```
        g->autoatks[i].cooldown = 0;
        g->autoatks[i].period   = 0;
    }
    for (int i = 0; i < LEVELUP_OPTIONS; i++) {
        g->levelup_options[i] = -1;
    }
    g->levelup_cursor      = 0;
    g->levelup_prev_cursor = 0;
}

void autoatk_tick(game_t *g) {
    /* Only mortar runs on a timer now. */
    autoatk_t *a = &g->autoatks[AA_MORTAR];
    if (a->level <= 0) return;
    if (a->cooldown > 0) { a->cooldown--; return; }
    fire_mortar(g);
    a->cooldown = a->period;
}

void autoatk_update_proj(game_t *g) {
    for (int i = 0; i < MAX_ENTITIES; i++) {
        entity_t *e = &g->ents[i];
        if (!e->active) continue;
        if (e->kind != ENT_AUTO_PROJ && e->kind != ENT_HAZARD) continue;

        e->x += e->vx;
        e->y += e->vy;

        if (e->y > SCREEN_H || e->y < -16 || e->x < -16 || e->x > SCREEN_W) {
            e->active = 0;
            continue;
        }
        if (e->ttl > 0) {
            e->ttl--;
            if (e->ttl <= 0) e->active = 0;
        }
    }
}

void autoatk_upgrade(game_t *g, autoatk_kind_t k) {
    if ((int)k < 0 || (int)k >= AA_COUNT) return;
    autoatk_t *a = &g->autoatks[k];
    if (a->level >= AA_LEVEL_CAP) return;
    a->level++;
    a->period = period_for(k, a->level);
    /* Don't let a freshly-upgraded weapon fire on the same tick it was picked. */
    if (a->period > 0 && a->cooldown == 0) a->cooldown = a->period;
}

int autoatk_damage(autoatk_kind_t k) {
    switch (k) {
        case AA_MORTAR:    return 2;
        case AA_ARTILLERY: return ARTILLERY_KILL_DMG;
        case AA_GAS:       return 1;
        default:           return 1;
    }
}

int autoatk_despawn_on_hit(autoatk_kind_t k) {
```

```
    /* Mortar shells expend on first hit. Gas persists, artillery flash
       sprites self-despawn via ttl regardless. */
    return k == AA_MORTAR;
}

int autoatk_hitbox_pad(autoatk_kind_t k) {
    /* For gas the precise box is computed in autoatk_gas_overlaps; this
       fallback is used by the generic touching_pad path. Keep it small. */
    return (k == AA_GAS) ? GAS_BASE_HITPAD : 0;
}

void autoatk_gas_size(const entity_t *e, int *out_w, int *out_h) {
    int w = GAS_W_MAX, h = GAS_H_MAX;
    int elapsed = e->ttl_max - e->ttl;
    int hold_until = GAS_TTL_MAX - GAS_SHRINK_FRAMES;
    if (elapsed < GAS_GROW_FRAMES) {
        /* Linearly interpolate from MIN to MAX. */
        w = GAS_W_MIN + (GAS_W_MAX - GAS_W_MIN) * elapsed / GAS_GROW_FRAMES;
        h = GAS_H_MIN + (GAS_H_MAX - GAS_H_MIN) * elapsed / GAS_GROW_FRAMES;
    } else if (elapsed > hold_until) {
        int t = GAS_TTL_MAX - elapsed;  /* frames left in shrink window */
        if (t < 0) t = 0;
        w = GAS_W_MIN + (GAS_W_MAX - GAS_W_MIN) * t / GAS_SHRINK_FRAMES;
        h = GAS_H_MIN + (GAS_H_MAX - GAS_H_MIN) * t / GAS_SHRINK_FRAMES;
    }
    if (w < GAS_W_MIN) w = GAS_W_MIN;
    if (h < GAS_H_MIN) h = GAS_H_MIN;
    *out_w = w;
    *out_h = h;
}

int autoatk_gas_overlaps(const entity_t *e, int tx, int ty) {
    int w, h;
    autoatk_gas_size(e, &w, &h);
    /* Cloud is centered on the entity's stored origin (which is the player's
       position at fire time). Cloud rect: [cx - w/2, cy - h/2] .. [+w, +h]. */
    int cx = e->x + 8;
    int cy = e->y + 8;
    int gx0 = cx - w / 2 - GAS_BASE_HITPAD;
    int gy0 = cy - h / 2 - GAS_BASE_HITPAD;
    int gx1 = cx + w / 2 + GAS_BASE_HITPAD;
    int gy1 = cy + h / 2 + GAS_BASE_HITPAD;
    /* Target is a 16x16 sprite at (tx, ty). */
    return gx0 < tx + 16 && gx1 > tx && gy0 < ty + 16 && gy1 > ty;
}

/* Player-triggered artillery: instant column kill from the player upward,
   plus a tile-map beam visual that render.c paints into the tilemap for
   BEAM_VISUAL_TTL frames. Routes the kill through apply_damage() so
   ammo-drop probability + score bookkeeping stay consistent with bullet
   and mortar kills. */
int autoatk_fire_artillery(game_t *g) {
    const entity_t *p = &g->ents[g->player_i];
    int beam_x = p->x;   /* left edge of 16-px-wide beam column */

    for (int i = 0; i < MAX_ENTITIES; i++) {
        entity_t *e = &g->ents[i];
        if (!e->active) continue;
        if (e->kind != ENT_ENEMY_ARMED && e->kind != ENT_ENEMY_UNARMED) continue;
```

```
        if (e->x + 16 <= beam_x || e->x >= beam_x + 16) continue;
        if (e->y > p->y) continue;   /* beam shoots upward only */
        apply_damage(g, e, ARTILLERY_KILL_DMG);
    }

    /* Mark the beam visual. render.c owns the actual tile-map writes (and
       restores the underlying ground tiles when beam_ttl drops to 0). */
    g->beam_col = (p->x + 8) / 8;          /* tile column at player's center */
    g->beam_ttl = BEAM_VISUAL_TTL + 1;     /* +1 because game_tick decrements
                                              before render reads it          */
    return 1;
}

/* Player-triggered gas: spawns ONE ENT_HAZARD at player position. The hazard
   is the source-of-truth; render.c emits up to 4 sprite slots around it
   according to autoatk_gas_size(). */
int autoatk_fire_gas(game_t *g) {
    const entity_t *p = &g->ents[g->player_i];
    int slot = alloc_entity(g);
    if (slot < 0) return 0;
    entity_t *e = &g->ents[slot];
    e->active  = 1;
    e->kind    = ENT_HAZARD;
    e->x       = p->x;
    e->y       = p->y;
    e->vx      = 0;
    e->vy      = 0;
    e->hp      = 1;
    e->phase   = 0;
    e->ttl     = GAS_TTL_MAX;
    e->ttl_max = GAS_TTL_MAX;
    e->payload = (int)AA_GAS;
    e->fire_cd = 0;
    return 1;
}

void autoatk_pick_levelup_options(game_t *g) {
    /* AA_COUNT is small (3). With LEVELUP_OPTIONS == 3 we end up offering all
       three weapons -- the rejection-sampling loop still terminates quickly. */
    int picked[LEVELUP_OPTIONS];
    int n = 0;
    int safety = 0;
    while (n < LEVELUP_OPTIONS && safety < 256) {
        safety++;
        int k = rand() % AA_COUNT;
        int dup = 0;
        for (int j = 0; j < n; j++) if (picked[j] == k) { dup = 1; break; }
        if (dup) continue;
        picked[n++] = k;
    }
    /* Fill any tail with -1 if AA_COUNT < LEVELUP_OPTIONS. */
    for (int i = 0; i < LEVELUP_OPTIONS; i++) {
        g->levelup_options[i] = (i < n) ? picked[i] : -1;
    }
    g->levelup_cursor      = 0;
    g->levelup_prev_cursor = 0;
}

const char *autoatk_name(autoatk_kind_t k) {
```

```
    switch (k) {
        case AA_MORTAR:    return "Mortar";
        case AA_GAS:       return "Mustard Gas";
        case AA_ARTILLERY: return "Artillery";
        default:           return "?";
    }
}
```

input.h

```
#ifndef INPUT_H
#define INPUT_H

#include <stdint.h>

uint16_t input_read(int frame);

#endif
```

input_real.c

```
/*
 * input_real.c -- Linux evdev (/dev/input/event0) reader for the DragonRise
 * generic SNES-style USB gamepad (VID 0079:0011). Implements the same
 * input_read(frame) signature as input_fake.c so main.c can swap between
 * them at link time.
 *
 * Why evdev and not joydev: the DE1-SoC class kernel (4.19) is built
 * without joydev (modules.dep.bin missing, no js0 device), but evdev is
 * built-in and event0 enumerates the moment the gamepad is plugged in.
 *
 * DragonRise mapping verified against this controller (KIWITATA SNES USB):
 *   B button       -> BTN_THUMB2  (0x122)  -> INPUT_FIRE_DOWN  (shoot down)
 *   Y button       -> BTN_TOP     (0x123)  -> INPUT_FIRE_LEFT  (shoot left)
 *   X button       -> BTN_TRIGGER (0x120)  -> INPUT_FIRE_UP    (shoot up)
 *   A button       -> BTN_THUMB   (0x121)  -> INPUT_FIRE_RIGHT (shoot right)
 *   L shoulder     -> BTN_TOP2    (0x124)  -> INPUT_ABIL_ART   (artillery)
 *   R shoulder     -> BTN_PINKIE  (0x125)  -> INPUT_ABIL_GAS   (gas cloud)
 *   Start          -> BTN_BASE4   (0x129)  -> INPUT_START
 *   Select         -> BTN_BASE3   (0x128)  -> INPUT_SELECT
 *   D-pad X        -> ABS_X       (0x00)   -> INPUT_LEFT / INPUT_RIGHT
 *   D-pad Y        -> ABS_Y       (0x01)   -> INPUT_UP   / INPUT_DOWN
 *
 * If the mapping differs on this controller, set NML_INPUT_DEBUG=1 in the
 * environment and the first few events get dumped to stderr -- adjust the
 * BTN_CODE_* / ABS_CODE_* constants below and rebuild.
 *
 * Axis ranges are queried via EVIOCGABS at open time, so the deadzone math
 * works regardless of whether DragonRise reports 0..255, -1..1, or signed
 * 16-bit values.
 */

#include "input.h"
#include "game.h"
```

```c
#include <errno.h>
#include <fcntl.h>
#include <linux/input.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/ioctl.h>
#include <sys/types.h>
#include <unistd.h>

#define EV_DEVICE        "/dev/input/event0"

#define FD_UNINIT        (-1)
#define FD_DISABLED      (-2)

/* Button codes -- verified against the DragonRise dump from this controller. */
#define BTN_CODE_FIRE_DOWN   BTN_THUMB2    /* 0x122 -- B */
#define BTN_CODE_FIRE_LEFT   BTN_TOP       /* 0x123 -- Y */
#define BTN_CODE_FIRE_UP     BTN_TRIGGER   /* 0x120 -- X */
#define BTN_CODE_FIRE_RIGHT  BTN_THUMB     /* 0x121 -- A */
#define BTN_CODE_ABIL_ART    BTN_TOP2      /* 0x124 -- L shoulder */
#define BTN_CODE_ABIL_GAS    BTN_PINKIE    /* 0x125 -- R shoulder */
#define BTN_CODE_START       BTN_BASE4     /* 0x129 -- Start */
#define BTN_CODE_SELECT      BTN_BASE3     /* 0x128 -- Select */

/* D-pad axis codes -- edit if it's reported on ABS_HATOX/HATOY instead. */
#define ABS_CODE_X       ABS_X
#define ABS_CODE_Y       ABS_Y

static int      ev_fd          = FD_UNINIT;
static uint16_t button_state   = 0;
static int      axis_x_value   = 0;
static int      axis_x_min     = -1;
static int      axis_x_max     =  1;
static int      axis_x_center  =  0;
static int      axis_y_value   = 0;
static int      axis_y_min     = -1;
static int      axis_y_max     =  1;
static int      axis_y_center  =  0;
static int      debug_enabled  = 0;

static void query_axis(int code, int *min, int *max, int *center, int *initial) {
    struct input_absinfo info;
    if (ioctl(ev_fd, EVIOCGABS(code), &info) == 0) {
        *min     = info.minimum;
        *max     = info.maximum;
        *center  = info.minimum + (info.maximum - info.minimum) / 2;
        *initial = info.value;
    }
}

static void open_device(void) {
    debug_enabled = (getenv("NML_INPUT_DEBUG") != NULL);

    ev_fd = open(EV_DEVICE, O_RDONLY | O_NONBLOCK);
    if (ev_fd < 0) {
        fprintf(stderr,
                "input_real: open(%s) failed: %s -- "
```

```
                "no gamepad input until fixed.\n",
                EV_DEVICE, strerror(errno));
        ev_fd = FD_DISABLED;
        return;
    }

    query_axis(ABS_CODE_X, &axis_x_min, &axis_x_max, &axis_x_center, &axis_x_value);
    query_axis(ABS_CODE_Y, &axis_y_min, &axis_y_max, &axis_y_center, &axis_y_value);

    if (debug_enabled) {
        fprintf(stderr,
                "input_real: opened %s; "
                "X[min=%d max=%d center=%d initial=%d] "
                "Y[min=%d max=%d center=%d initial=%d]\n",
                EV_DEVICE,
                axis_x_min, axis_x_max, axis_x_center, axis_x_value,
                axis_y_min, axis_y_max, axis_y_center, axis_y_value);
    }
}

static uint16_t button_bit_for(uint16_t code) {
    switch (code) {
        case BTN_CODE_FIRE_DOWN:  return INPUT_FIRE_DOWN;
        case BTN_CODE_FIRE_LEFT:  return INPUT_FIRE_LEFT;
        case BTN_CODE_FIRE_UP:    return INPUT_FIRE_UP;
        case BTN_CODE_FIRE_RIGHT: return INPUT_FIRE_RIGHT;
        case BTN_CODE_ABIL_ART:   return INPUT_ABIL_ART;
        case BTN_CODE_ABIL_GAS:   return INPUT_ABIL_GAS;
        case BTN_CODE_START:      return INPUT_START;
        case BTN_CODE_SELECT:     return INPUT_SELECT;
        default:                  return 0;
    }
}

static void drain_events(void) {
    struct input_event ev;
    while (read(ev_fd, &ev, sizeof(ev)) == (ssize_t)sizeof(ev)) {
        if (debug_enabled && ev.type != EV_SYN) {
            fprintf(stderr, "ev: type=0x%02x code=0x%03x value=%d\n",
                    ev.type, ev.code, ev.value);
        }

        if (ev.type == EV_KEY) {
            uint16_t bit = button_bit_for(ev.code);
            if (!bit) continue;
            if (ev.value) button_state |=  bit;
            else          button_state &= ~bit;
        } else if (ev.type == EV_ABS) {
            if      (ev.code == ABS_CODE_X) axis_x_value = ev.value;
            else if (ev.code == ABS_CODE_Y) axis_y_value = ev.value;
        }
    }
    /* read() returning -1 with EAGAIN/EWOULDBLOCK means "drained"; not an
       error. Any other failure we silently swallow -- the game continues. */
}

static int deadzone_threshold(int min, int max) {
    int range = max - min;
    if (range <= 2) return 0;          /* digital pad on a -1/0/+1 axis */
```

```
    return range / 4;                   /* analog: 25% deadzone */
}

uint16_t input_read(int frame) {
    (void)frame;   /* unused; kept for ABI parity with input_fake.c */

    if (ev_fd == FD_UNINIT)   open_device();
    if (ev_fd == FD_DISABLED) return 0;

    drain_events();

    uint16_t dpad = 0;
    int dx = axis_x_value - axis_x_center;
    int dy = axis_y_value - axis_y_center;
    int tx = deadzone_threshold(axis_x_min, axis_x_max);
    int ty = deadzone_threshold(axis_y_min, axis_y_max);

    if (dx < -tx) dpad |= INPUT_LEFT;
    if (dx >  tx) dpad |= INPUT_RIGHT;
    if (dy < -ty) dpad |= INPUT_UP;
    if (dy >  ty) dpad |= INPUT_DOWN;

    return button_state | dpad;
}
```

input_fake.c

```
#include "input.h"
#include "game.h"

/* Fakes a SNES controller. Used by the terminal build and the fake-input
 * cross-build. The pattern exercises all four directional fires plus both
 * shoulder abilities so terminal traces can show the new code paths working.
 *
 * Cycle (mod 480 frames, ~8 sec at 60 fps):
 *   [0..80)   strafe right
 *   [80..160) strafe left
 *   [160..240) move up
 *   [240..320) move down
 *   [320..480) idle so the wave/levelup loop can play out
 *
 * Fire pattern (regardless of movement window):
 *   every 20 frames cycle through DOWN, LEFT, UP, RIGHT in turn.
 *   frame % 200 == 0 -> press L (artillery)
 *   frame % 240 == 0 -> press R (gas)
 *   frame % 600 == 0 -> press START (menu confirm / restart)
 */
uint16_t input_read(int frame) {
    uint16_t input = 0;

    int phase = frame % 480;
    if      (phase < 80)  input |= INPUT_RIGHT;
    else if (phase < 160) input |= INPUT_LEFT;
    else if (phase < 240) input |= INPUT_UP;
    else if (phase < 320) input |= INPUT_DOWN;

    if (frame % 20 == 0) {
```

```
        switch ((frame / 20) % 4) {
            case 0: input |= INPUT_FIRE_DOWN;  break;
            case 1: input |= INPUT_FIRE_LEFT;  break;
            case 2: input |= INPUT_FIRE_UP;    break;
            case 3: input |= INPUT_FIRE_RIGHT; break;
        }
    }

    if (frame % 200 == 0) input |= INPUT_ABIL_ART;
    if (frame % 240 == 0) input |= INPUT_ABIL_GAS;
    if (frame > 0 && frame % 600 == 0) input |= INPUT_START;

    return input;
}
```

render.h

```
#ifndef RENDER_H
#define RENDER_H

#include "game.h"

void render_frame(const game_t *g);

/* FPGA-only: paint the battlefield ground into the tile map. Called from
   main.c at startup and re-invoked after game-over -> restart to restore
   the playfield. No-op in terminal builds. */
#ifndef NML_TERMINAL_BUILD
void render_init_tilemap(void);
#endif

#endif
```

render.c

```
/*
 * render.c -- translates the entity pool into nml_gpu sprite-table writes.
 *
 * Conventions (matching render_terminal.c so behavior is identical between
 * builds): slot 0 is always the player, sprite_id 1. Subsequent slots are
 * filled in entity-array order with enemies, bullets, hazards, etc.; gas
 * clouds expand to up to 4 sprite slots each. Unused slots are hidden via
 * nml_hide_sprite() so leftover state from prior frames is gone.
 *
 * Sprite-id assignments must match the layout in hw/gen_rom.py.
 */

#include "render.h"
#include "autoatk.h"
#include "nml_gpu.h"

#include <stdio.h>
#include <string.h>
```

```c
/* Tile slots from hw/gen_rom.py BATTLEFIELD_GLYPHS + letter/digit tables. */
#define TILE_BLANK_BG       0
#define TILE_DIRT_A         4
#define TILE_DIRT_B         5
#define TILE_GRASS_A        6
#define TILE_GRASS_B        7
#define TILE_TRANSITION     8
#define TILE_MUD_PUDDLE     9
#define TILE_GRASS_DENSE    10
#define TILE_BEAM           11
#define TILE_LETTER_BASE    16   /* 'A' -> 16, 'Z' -> 41 */
#define TILE_COLON          42
#define TILE_SPACE          43
#define TILE_DIGIT_BASE     48   /* '0' -> 48, '9' -> 57 */

#define SPRITE_ID_PLAYER         1
#define SPRITE_ID_ENEMY_ARMED    2
#define SPRITE_ID_BULLET         3
#define SPRITE_ID_ENEMY_UNARMED  4
#define SPRITE_ID_MORTAR         5
/* Sprite ID 6 (barbed wire) retired -- AA_WIRE removed in Batch A. */
#define SPRITE_ID_GAS            7
#define SPRITE_ID_ARTILLERY      8
#define SPRITE_ID_AMMO           9   /* green ammo crate (Batch B sprite art) */
#define SPRITE_ID_ENEMY_BULLET   10  /* red dot for incoming enemy fire       */

static uint8_t autoatk_sprite_id(int payload) {
    switch (payload) {
        case AA_MORTAR:    return SPRITE_ID_MORTAR;
        case AA_GAS:       return SPRITE_ID_GAS;
        case AA_ARTILLERY: return SPRITE_ID_ARTILLERY;
        default:           return SPRITE_ID_BULLET;
    }
}

static uint8_t entity_to_sprite_id(const entity_t *e) {
    switch (e->kind) {
        case ENT_PLAYER:        return SPRITE_ID_PLAYER;
        case ENT_ENEMY_ARMED:   return SPRITE_ID_ENEMY_ARMED;
        case ENT_ENEMY_UNARMED: return SPRITE_ID_ENEMY_UNARMED;
        case ENT_BULLET:        return SPRITE_ID_BULLET;
        case ENT_ENEMY_BULLET:  return SPRITE_ID_ENEMY_BULLET;
        case ENT_AMMO_DROP:     return SPRITE_ID_AMMO;
        case ENT_AUTO_PROJ:
        case ENT_HAZARD:        return autoatk_sprite_id(e->payload);
        default:                return 0;
    }
}

static uint8_t entity_palette_off(const entity_t *e) {
    (void)e;
    /* All entities now have dedicated sprite art with native palette colors;
       palette-offset shifting from Batch A is retired. */
    return 0;
}

/* Emit a 16x16 sprite into slot `slot_inout` (advanced in-place). Returns 1
   if a slot was consumed, 0 if the table is already full (silent drop). */
static int emit_sprite(int *slot_inout, int x, int y, uint8_t sid,
```

```
                        uint8_t pal_off, int prio) {
    int slot = *slot_inout;
    if (slot >= NML_MAX_SPRITES) return 0;
    nml_sprite_t s = {
        .x = (int16_t)x, .y = (int16_t)y,
        .sprite_id   = sid,
        .flags       = NML_FLAGS(prio, /*hflip=*/0, /*vflip=*/0),
        .palette_off = pal_off,
        .reserved    = 0,
    };
    nml_write_sprite(slot, &s);
    *slot_inout = slot + 1;
    return 1;
}

/* Gas cloud rendering: emit a 16x16 sprite at the center plus up to 3 more
   forming a + pattern as the cloud expands toward its full 48x32 size.
   The visible-size threshold for emitting the side/top sprites grows with
   the cloud's current width/height. */
static void emit_gas_cluster(const entity_t *e, int *slot_inout) {
    int w, h;
    autoatk_gas_size(e, &w, &h);
    int cx = e->x;        /* origin is top-left of central 16x16 */
    int cy = e->y;
    emit_sprite(slot_inout, cx, cy, SPRITE_ID_GAS, 0, 1);   /* center */
    if (w >= 24) {
        emit_sprite(slot_inout, cx - 12, cy, SPRITE_ID_GAS, 0, 1);   /* left */
        emit_sprite(slot_inout, cx + 12, cy, SPRITE_ID_GAS, 0, 1);   /* right */
    }
    if (h >= 24) {
        emit_sprite(slot_inout, cx, cy - 10, SPRITE_ID_GAS, 0, 1);   /* top */
    }
}

/* ---------------------------------------------------------------------
 * Tile-map helpers: ground init, artillery beam, on-screen text.
 * Owned by render.c so the FPGA-only nml_write_tile() / nml_gpu.h surface
 * stays out of game.c and main.c.
 * ------------------------------------------------------------------- */

/* Same coarse-noise function used to populate the battlefield at startup.
   Exposed via tile_for(row, col) so the beam save/restore and game-over
   restore can recompute any cell without keeping a shadow copy. */
static uint8_t tile_for(int row, int col) {
    unsigned zone = ((unsigned)(row / 8) * 7u +
                     (unsigned)(col / 10) * 11u) % 5u;
    int favor_grass = (zone == 1 || zone == 3);

    unsigned r = ((unsigned)(row * 17 + col * 31) >> 1) & 31u;

    if (favor_grass) {
        if      (r < 16) return TILE_GRASS_A;
        else if (r < 22) return TILE_GRASS_B;
        else if (r < 26) return TILE_GRASS_DENSE;
        else if (r < 28) return TILE_DIRT_A;
        else if (r < 30) return TILE_TRANSITION;
        else             return TILE_MUD_PUDDLE;
    } else {
        if      (r < 16) return TILE_DIRT_A;
```

```
        else if (r < 22) return TILE_DIRT_B;
        else if (r < 26) return TILE_TRANSITION;
        else if (r < 28) return TILE_GRASS_A;
        else if (r < 30) return TILE_MUD_PUDDLE;
        else             return TILE_GRASS_DENSE;
    }
}

void render_init_tilemap(void) {
    for (int row = 0; row < NML_TILEMAP_ROWS; ++row) {
        for (int col = 0; col < NML_TILEMAP_COLS; ++col) {
            nml_write_tile(col, row, tile_for(row, col));
        }
    }
}

/* Artillery-beam tile-map state. The visual is a vertical column of
   TILE_BEAM glyphs written into the tile map for ~6 frames; the underlying
   ground tiles are recomputed (not stored) when the beam expires, so no
   per-cell shadow buffer is needed. */
static int s_beam_drawn      = 0;   /* 1 = beam column currently overwritten */
static int s_beam_drawn_col  = -1;  /* the column we overwrote               */

static void beam_paint_column(int col) {
    if (col < 0 || col >= NML_TILEMAP_COLS) return;
    /* Paint from just below the HUD strip (row 2 = y=16+) down to the row
       above the player's current tile. Painting all the way to the bottom
       reads better -- gives the player visual confirmation the beam reached
       deep into the field. */
    for (int row = 2; row < NML_TILEMAP_ROWS; ++row) {
        nml_write_tile(col, row, TILE_BEAM);
    }
}

static void beam_restore_column(int col) {
    if (col < 0 || col >= NML_TILEMAP_COLS) return;
    for (int row = 2; row < NML_TILEMAP_ROWS; ++row) {
        nml_write_tile(col, row, tile_for(row, col));
    }
}

/* Maintain beam_drawn state machine from the game's beam_ttl signal. */
static void beam_update(const game_t *g) {
    if (g->beam_ttl > 0 && !s_beam_drawn) {
        s_beam_drawn_col = g->beam_col;
        beam_paint_column(s_beam_drawn_col);
        s_beam_drawn = 1;
    } else if (g->beam_ttl == 0 && s_beam_drawn) {
        beam_restore_column(s_beam_drawn_col);
        s_beam_drawn = 0;
        s_beam_drawn_col = -1;
    }
}

/* Map an ASCII char to a tile slot. Unknown chars fall through to blank bg
   (tile 0) so the caller can include punctuation/whitespace freely. */
static uint8_t ascii_to_tile(char c) {
    if (c >= 'A' && c <= 'Z') return (uint8_t)(TILE_LETTER_BASE + (c - 'A'));
    if (c >= 'a' && c <= 'z') return (uint8_t)(TILE_LETTER_BASE + (c - 'a'));
```

```
    if (c >= '0' && c <= '9') return (uint8_t)(TILE_DIGIT_BASE  + (c - '0'));
    if (c == ':')             return TILE_COLON;
    if (c == ' ')             return TILE_SPACE;
    return TILE_BLANK_BG;
}

/* Write a string into the tile map, left-aligned at (row, col). Out-of-range
   cells are silently skipped. */
static void draw_text_at(int row, int col, const char *s) {
    int len = (int)strlen(s);
    for (int i = 0; i < len; ++i) {
        int c = col + i;
        if (c < 0) continue;
        if (c >= NML_TILEMAP_COLS) break;
        nml_write_tile(c, row, ascii_to_tile(s[i]));
    }
}

/* Center the string on `row` so it sits at the horizontal midpoint of the
   80-column tile map. */
static void draw_text_centered(int row, const char *s) {
    int len = (int)strlen(s);
    int col = (NML_TILEMAP_COLS - len) / 2;
    draw_text_at(row, col, s);
}

/*
 * Level-up scene: player frozen + three weapon-sprite tiles along the top
 * (each showing the actual sprite of the weapon on offer), with a cursor
 * sprite above the active option. The SSH terminal still carries the menu
 * text and level annotations; this gives the player on-screen "what am I
 * picking" feedback without leaving the VGA monitor.
 */
static void render_levelup(const game_t *g) {
    const entity_t *p = &g->ents[g->player_i];

    nml_sprite_t player = {
        .x = (int16_t)p->x, .y = (int16_t)p->y,
        .sprite_id = SPRITE_ID_PLAYER,
        .flags = NML_FLAGS(/*prio=*/0, /*hflip=*/0, /*vflip=*/0),
        .palette_off = 0, .reserved = 0,
    };
    nml_write_sprite(0, &player);

    const int xs[3]   = { 160, 320, 480 };
    const int y_opt   = 100;
    const int y_curs  = 80;

    for (int i = 0; i < LEVELUP_OPTIONS; ++i) {
        int kind = g->levelup_options[i];
        uint8_t sid = (kind >= 0 && kind < AA_COUNT)
            ? autoatk_sprite_id(kind)
            : SPRITE_ID_BULLET;
        nml_sprite_t s = {
            .x = (int16_t)xs[i], .y = (int16_t)y_opt,
            .sprite_id = sid,
            .flags = NML_FLAGS(/*prio=*/1, 0, 0),
            .palette_off = 0, .reserved = 0,
        };
```

```
        nml_write_sprite(1 + i, &s);
    }

    int cursor_x = xs[g->levelup_cursor < 0 ? 0 :
                      (g->levelup_cursor >= LEVELUP_OPTIONS
                       ? LEVELUP_OPTIONS - 1
                       : g->levelup_cursor)];
    nml_sprite_t cur = {
        .x = (int16_t)cursor_x, .y = (int16_t)y_curs,
        .sprite_id = SPRITE_ID_PLAYER,
        .flags = NML_FLAGS(/*prio=*/0, 0, 0),
        .palette_off = 0, .reserved = 0,
    };
    nml_write_sprite(4, &cur);

    for (int slot = 5; slot < NML_MAX_SPRITES; ++slot) {
        nml_hide_sprite(slot);
    }

    nml_set_player_state((int16_t)p->x, (int16_t)p->y,
                         (uint8_t)(g->player_hp > 0 ? g->player_hp : 0),
                         (uint8_t)(g->wave_index + 1),
                         /*level=*/0);
    nml_set_score((uint32_t)g->score, 0u);
    nml_set_hud_aux((uint8_t)(g->ammo > 99 ? 99 : g->ammo),
                    (uint8_t)g->artillery_charges,
                    (uint8_t)g->gas_charges);
}

/*
 * Game-over screen: draw a centered block of text directly into the tile
 * map. Letter glyphs live at tile slots 16..41 (A..Z), digits at 48..57,
 * colon at 42, blank at 43. The block lays out as:
 *
 *     GAME OVER
 *
 *     SCORE: NNNNNN
 *     WAVE:  NN
 *     KILLS A: NN
 *     KILLS U: NN
 *
 *     PRESS START
 *
 * Vertical center of the playfield is row ~30. We anchor the block around
 * row 24 so it sits comfortably above center.
 */
static void render_game_over(const game_t *g) {
    /* Cap displayable score at 999999 to fit the 6-digit slot consistently
       with the HUD score readout. */
    int score = g->score;
    if (score < 0)       score = 0;
    if (score > 999999)  score = 999999;
    int wave_n  = g->wave_index + 1;
    if (wave_n  > 99) wave_n = 99;
    int ka      = g->kills_armed;   if (ka > 99) ka = 99;
    int ku      = g->kills_unarmed; if (ku > 99) ku = 99;

    char buf[40];
```

```
    draw_text_centered(22, "GAME OVER");

    snprintf(buf, sizeof(buf), "SCORE: %06d", score);
    draw_text_centered(24, buf);

    snprintf(buf, sizeof(buf), "WAVE:  %02d", wave_n);
    draw_text_centered(25, buf);

    snprintf(buf, sizeof(buf), "KILLS A: %02d", ka);
    draw_text_centered(26, buf);

    snprintf(buf, sizeof(buf), "KILLS U: %02d", ku);
    draw_text_centered(27, buf);

    draw_text_centered(29, "PRESS START");

    /* Hide every sprite slot so nothing draws over the text. */
    for (int slot = 0; slot < NML_MAX_SPRITES; ++slot) {
        nml_hide_sprite(slot);
    }

    /* HUD still shows the final values (HP=0, ammo=0, etc.). */
    nml_set_player_state(0, 0, 0, /*wave=*/0, /*level=*/0);
    nml_set_score((uint32_t)g->score, /*kills=*/0u);
    nml_set_hud_aux(0, 0, 0);
}

void render_frame(const game_t *g) {
    /* Track state to drive one-shot tile-map restores. The game-over screen
       overwrites the battlefield with stat text; on the GAMEOVER -> PLAYING
       restart we re-init the whole tile map. The artillery beam follows the
       same save/restore pattern but is column-scoped. */
    static game_state_t s_prev_state = STATE_PLAYING;
    if (s_prev_state == STATE_GAMEOVER && g->state == STATE_PLAYING) {
        /* If the beam was mid-flash when the game ended, drop our shadow
           state so the next fire doesn't double-restore. */
        s_beam_drawn     = 0;
        s_beam_drawn_col = -1;
        render_init_tilemap();
    }
    s_prev_state = g->state;

    if (g->state == STATE_GAMEOVER) {
        render_game_over(g);
        return;
    }
    if (g->state == STATE_LEVELUP) {
        /* Pause the beam visual during LEVELUP so it doesn't ghost across
           the menu. STATE_LEVELUP can only fire after a wave clears, and
           beam_ttl is short enough that it usually expires before then. */
        beam_update(g);
        render_levelup(g);
        return;
    }

    /* Update the artillery-beam tile column. Must run BEFORE we draw sprites
       so the new tile state takes effect on the same frame. */
    beam_update(g);
```

```
    /* Slot 0 = player, always written. */
    const entity_t *p = &g->ents[g->player_i];

    nml_sprite_t player = {
        .x          = (int16_t)p->x,
        .y          = (int16_t)p->y,
        .sprite_id  = SPRITE_ID_PLAYER,
        .flags      = NML_FLAGS(/*prio=*/0, /*hflip=*/0, /*vflip=*/0),
        .palette_off = 0,
        .reserved   = 0,
    };
    nml_write_sprite(0, &player);

    /* Slots 1..31: active non-player entities. Gas hazards expand into
       multiple slots; other entities consume one slot each. */
    int slot = 1;
    for (int i = 0; i < MAX_ENTITIES && slot < NML_MAX_SPRITES; ++i) {
        const entity_t *e = &g->ents[i];
        if (!e->active || e->kind == ENT_PLAYER) continue;

        if (e->kind == ENT_HAZARD && e->payload == (int)AA_GAS) {
            emit_gas_cluster(e, &slot);
            continue;
        }

        int prio = (e->kind == ENT_BULLET || e->kind == ENT_ENEMY_BULLET) ? 2 : 1;
        emit_sprite(&slot, e->x, e->y,
                    entity_to_sprite_id(e),
                    entity_palette_off(e),
                    prio);
    }

    /* Hide leftover slots so old state doesn't ghost. */
    for (; slot < NML_MAX_SPRITES; ++slot) {
        nml_hide_sprite(slot);
    }

    /* Mailbox the player state for the HUD overlay. */
    nml_set_player_state((int16_t)p->x,
                         (int16_t)p->y,
                         (uint8_t)(g->player_hp > 0 ? g->player_hp : 0),
                         (uint8_t)(g->wave_index + 1),
                         /*level=*/0);
    nml_set_score((uint32_t)g->score, /*kills=*/0u);
    nml_set_hud_aux((uint8_t)(g->ammo > 99 ? 99 : g->ammo),
                    (uint8_t)g->artillery_charges,
                    (uint8_t)g->gas_charges);
}
```

render_terminal.c

```
#include "render.h"
#include "autoatk.h"
#include <stdio.h>

/* Subdivides the game sprites into 32 slots in order to accommodate the FPGA.
   This mirrors the production render.c so behavior is identical between
```

```
   builds; the only difference is that this file prints sprite-table writes
   to stdout instead of memory-mapping them. */

#define MAX_SPRITES 32

#define SPRITE_PLAYER         1
#define SPRITE_ENEMY_ARMED    2
#define SPRITE_BULLET         3
#define SPRITE_ENEMY_UNARMED  4
#define SPRITE_MORTAR         5
/* Sprite 6 (wire) retired this batch. */
#define SPRITE_GAS            7
#define SPRITE_ARTILLERY      8

typedef struct {
    int active;
    int x, y;
    int sprite_id;
    char glyph;    /* ASCII representation, for the terminal-build only */
} mock_sprite_t;

static int autoatk_sprite_id(int payload) {
    switch (payload) {
        case AA_MORTAR:    return SPRITE_MORTAR;
        case AA_GAS:       return SPRITE_GAS;
        case AA_ARTILLERY: return SPRITE_ARTILLERY;
        default:           return SPRITE_BULLET;
    }
}

static int entity_to_sprite_id(const entity_t *e) {
    switch (e->kind) {
        case ENT_PLAYER:        return SPRITE_PLAYER;
        case ENT_ENEMY_ARMED:   return SPRITE_ENEMY_ARMED;
        case ENT_ENEMY_UNARMED: return SPRITE_ENEMY_UNARMED;
        case ENT_BULLET:        return SPRITE_BULLET;
        case ENT_ENEMY_BULLET:  return SPRITE_BULLET;
        case ENT_AMMO_DROP:     return SPRITE_BULLET;
        case ENT_AUTO_PROJ:
        case ENT_HAZARD:        return autoatk_sprite_id(e->payload);
        default:                return 0;
    }
}

static char entity_glyph(const entity_t *e) {
    switch (e->kind) {
        case ENT_PLAYER:        return 'P';
        case ENT_ENEMY_ARMED:   return 'A';
        case ENT_ENEMY_UNARMED: return 'u';
        case ENT_BULLET:        return '*';
        case ENT_ENEMY_BULLET:  return '^';
        case ENT_AMMO_DROP:     return '+';
        case ENT_AUTO_PROJ: {
            switch (e->payload) {
                case AA_MORTAR:    return 'M';
                case AA_ARTILLERY: return '|';
                default:           return '?';
            }
        }
```

```
        case ENT_HAZARD: {
            if (e->payload == (int)AA_GAS) return '~';
            return '#';
        }
        default: return '.';
    }
}

static int alloc_slot(mock_sprite_t *sprites, int *slot_inout,
                      const entity_t *e) {
    if (*slot_inout >= MAX_SPRITES) return 0;
    int s = *slot_inout;
    sprites[s].active    = 1;
    sprites[s].x         = e->x;
    sprites[s].y         = e->y;
    sprites[s].sprite_id = entity_to_sprite_id(e);
    sprites[s].glyph     = entity_glyph(e);
    *slot_inout = s + 1;
    return 1;
}

static void emit_gas_cluster(mock_sprite_t *sprites, int *slot_inout,
                             const entity_t *e) {
    int w, h;
    autoatk_gas_size(e, &w, &h);
    /* Up to 4 sprite slots for the cluster -- center always; +sides when
       width crosses 24, +top when height crosses 24. */
    int cx = e->x, cy = e->y;
    entity_t synth = *e;

    synth.x = cx; synth.y = cy;
    alloc_slot(sprites, slot_inout, &synth);

    if (w >= 24) {
        synth.x = cx - 12; synth.y = cy;
        alloc_slot(sprites, slot_inout, &synth);
        synth.x = cx + 12; synth.y = cy;
        alloc_slot(sprites, slot_inout, &synth);
    }
    if (h >= 24) {
        synth.x = cx; synth.y = cy - 10;
        alloc_slot(sprites, slot_inout, &synth);
    }
}

void render_frame(const game_t *g) {
    mock_sprite_t sprites[MAX_SPRITES];

    for (int i = 0; i < MAX_SPRITES; i++) {
        sprites[i].active = 0;
        sprites[i].x = 0;
        sprites[i].y = 0;
        sprites[i].sprite_id = 0;
        sprites[i].glyph = ' ';
    }

    int slot = 0;

    /* Slot 0 is always the player. */
```

```
    const entity_t *p = &g->ents[g->player_i];
    sprites[slot].active = 1;
    sprites[slot].x = p->x;
    sprites[slot].y = p->y;
    sprites[slot].sprite_id = SPRITE_PLAYER;
    sprites[slot].glyph = 'P';
    slot++;

    /* Slots 1..31: active non-player entities. Gas hazards expand into up
       to 4 slots. */
    for (int i = 0; i < MAX_ENTITIES && slot < MAX_SPRITES; i++) {
        const entity_t *e = &g->ents[i];
        if (!e->active) continue;
        if (e->kind == ENT_PLAYER) continue;
        if (e->kind == ENT_HAZARD && e->payload == (int)AA_GAS) {
            emit_gas_cluster(sprites, &slot, e);
            continue;
        }
        sprites[slot].active = 1;
        sprites[slot].x = e->x;
        sprites[slot].y = e->y;
        sprites[slot].sprite_id = entity_to_sprite_id(e);
        sprites[slot].glyph = entity_glyph(e);
        slot++;
    }

    printf("frame=%d hp=%d bullets=%d art=%d gas=%d score=%d wave=%d "
           "kills(A/U)=%d/%d drops=%d sprite_count=%d state=%d\n",
           g->frame, g->player_hp, g->ammo,
           g->artillery_charges, g->gas_charges, g->score,
           g->wave_index + 1, g->kills_armed, g->kills_unarmed,
           g->drops_collected, slot, (int)g->state);

    for (int i = 0; i < MAX_SPRITES; i++) {
        if (!sprites[i].active) {
            printf("  slot %02d: inactive\n", i);
        } else {
            printf("  slot %02d: sprite_id=%d glyph=%c x=%d y=%d\n",
                   i,
                   sprites[i].sprite_id,
                   sprites[i].glyph,
                   sprites[i].x,
                   sprites[i].y);
        }
    }

    printf("\n");
}
```

nml_gpu.h

```
#ifndef NML_GPU_H
#define NML_GPU_H

/*
 * nml_gpu.h -- userspace driver header for the No Man's Land FPGA peripheral.
 *
```

```c
 * The driver maps the HPS-to-FPGA Lightweight bridge (16 KB at 0xFF200000 in
 * HPS physical address space) into the process via /dev/mem and exposes typed
 * writers for each register region. Register offsets and sprite layout are
 * defined below.
 *
 * Caveat: the SystemVerilog sprite_eval module reads bit 47 of the 64-bit
 * sprite-table entry as an "active" flag. The convenience writers below set
 * bit 7 of `flags` for visible sprites and clear it for hidden.
 */

#include <stdint.h>

/* HPS physical address of the Lightweight bridge base + window size. */
#define NML_LWFPGA_BASE       0xFF200000UL
#define NML_LWFPGA_SPAN       0x00004000UL    /* 16 KB */

/* Register offsets (bytes from peripheral base). */
#define NML_REG_CTRL          0x0000
#define NML_REG_STATUS        0x0004
#define NML_REG_BG_SCROLL     0x0008
#define NML_REG_IRQ_MASK      0x000C
#define NML_REG_PLAYER_POS    0x0010
#define NML_REG_PLAYER_STATS  0x0014
#define NML_REG_SCORE         0x0018
#define NML_REG_KILL_COUNT    0x001C
#define NML_REG_HUD_AUX       0x0020   /* [7:0]=ammo BCD, [11:8]=art, [15:12]=gas */
#define NML_SPRITE_TABLE_BASE 0x0100   /* 32 entries x 8B   = 256B  */
#define NML_PALETTE_BASE      0x0400   /* 256 entries x 4B  = 1KB   */
#define NML_TILEMAP_BASE      0x1000   /* 80 cols x 60 rows = 4800B */

/* CTRL bits */
#define NML_CTRL_ENABLE       (1u << 0)
#define NML_CTRL_SWAP         (1u << 1)
#define NML_CTRL_HUD_ON       (1u << 2)

/* STATUS bits */
#define NML_STATUS_VBLANK       (1u << 0)
#define NML_STATUS_SWAP_PENDING (1u << 1)

/* Off-screen sentinel (kept for future use; the current sprite_eval relies on
 * the flags ACTIVE bit instead, so nml_hide_sprite() also clears that bit). */
#define NML_SPRITE_HIDE       ((int16_t)-256)

/* Tile map dimensions */
#define NML_TILEMAP_COLS      80
#define NML_TILEMAP_ROWS      60

/* Number of sprite slots */
#define NML_MAX_SPRITES       32

/* Flags bit layout in the sprite struct (matches HW packing of W1[15:8]):
 *   bit 0..1  reserved (must be 0)
 *   bit 2     hflip
 *   bit 3     vflip
 *   bit 4..6  priority (0 = highest)
 *   bit 7     ACTIVE -- set for visible sprites (sprite_eval workaround)
 */
#define NML_FLAG_HFLIP        (1u << 2)
#define NML_FLAG_VFLIP        (1u << 3)
```

```
#define NML_FLAG_PRIO_SHIFT   4
#define NML_FLAG_PRIO_MASK    (0x7u << NML_FLAG_PRIO_SHIFT)
#define NML_FLAG_ACTIVE       (1u << 7)

#define NML_FLAGS(prio, hflip, vflip)                       \
    (((((prio) & 0x7u) << NML_FLAG_PRIO_SHIFT)) |           \
     ((hflip) ? NML_FLAG_HFLIP : 0u) |                      \
     ((vflip) ? NML_FLAG_VFLIP : 0u) |                      \
     NML_FLAG_ACTIVE)

/* Sprite descriptor matching the HW 8-byte slot layout exactly. */
typedef struct {
    int16_t  x;
    int16_t  y;
    uint8_t  sprite_id;
    uint8_t  flags;           /* use NML_FLAGS(...) */
    uint8_t  palette_off;
    uint8_t  reserved;
} nml_sprite_t;

/* Public API. */
int  nml_open(void);
void nml_close(void);
void nml_set_enable(int on);
void nml_set_hud_on(int on);
void nml_write_palette(int idx, uint8_t r, uint8_t g, uint8_t b);
void nml_write_tile(int col, int row, uint8_t tile_id);
void nml_write_sprite(int slot, const nml_sprite_t *s);
void nml_hide_sprite(int slot);
void nml_clear_sprites(void);
void nml_set_player_state(int16_t px, int16_t py,
                          uint8_t hp, uint8_t wave, uint16_t level);
void nml_set_score(uint32_t score, uint32_t kills);

/* Write the auxiliary HUD register (offset 0x20). ammo is 0..99, charges are
   each 0..9. SW BCD-packs ammo (2 nibbles); charges are written as raw
   nibbles since they fit in one digit. */
void nml_set_hud_aux(uint8_t ammo, uint8_t art_charges, uint8_t gas_charges);
void nml_commit_frame(void);              /* sets SWAP, returns when committed */
int  nml_in_vblank(void);

#endif /* NML_GPU_H */
```

nml_gpu.c

```
/*
 * nml_gpu.c -- userspace driver for the No Man's Land FPGA peripheral.
 * Maps the HPS-to-FPGA Lightweight bridge via /dev/mem and provides typed
 * writers for the register map (see nml_gpu.h).
 */

#define _GNU_SOURCE
#include "nml_gpu.h"

#include <fcntl.h>
#include <stdint.h>
#include <stdio.h>
```

```c
#include <string.h>
#include <sys/mman.h>
#include <sys/types.h>
#include <unistd.h>

static int             g_devmem_fd = -1;
static volatile void  *g_base      = NULL;

static volatile uint32_t *reg_ptr(unsigned offset) {
    return (volatile uint32_t *)((volatile uint8_t *)g_base + offset);
}

static void reg_write(unsigned offset, uint32_t value) {
    *reg_ptr(offset) = value;
}

static uint32_t reg_read(unsigned offset) {
    return *reg_ptr(offset);
}

int nml_open(void) {
    if (g_base != NULL) return 0;

    g_devmem_fd = open("/dev/mem", O_RDWR | O_SYNC);
    if (g_devmem_fd < 0) {
        perror("nml_open: /dev/mem");
        return -1;
    }

    g_base = mmap(NULL,
                  NML_LWFPGA_SPAN,
                  PROT_READ | PROT_WRITE,
                  MAP_SHARED,
                  g_devmem_fd,
                  (off_t)NML_LWFPGA_BASE);
    if (g_base == MAP_FAILED) {
        perror("nml_open: mmap");
        close(g_devmem_fd);
        g_devmem_fd = -1;
        g_base = NULL;
        return -1;
    }

    /* Hidden sentinel for unused slots. */
    nml_clear_sprites();
    return 0;
}

void nml_close(void) {
    if (g_base != NULL) {
        munmap((void *)g_base, NML_LWFPGA_SPAN);
        g_base = NULL;
    }
    if (g_devmem_fd >= 0) {
        close(g_devmem_fd);
        g_devmem_fd = -1;
    }
}
```

```c
void nml_set_enable(int on) {
    uint32_t ctrl = reg_read(NML_REG_CTRL);
    if (on) ctrl |=  NML_CTRL_ENABLE;
    else    ctrl &= ~NML_CTRL_ENABLE;
    reg_write(NML_REG_CTRL, ctrl);
}

void nml_set_hud_on(int on) {
    uint32_t ctrl = reg_read(NML_REG_CTRL);
    if (on) ctrl |=  NML_CTRL_HUD_ON;
    else    ctrl &= ~NML_CTRL_HUD_ON;
    reg_write(NML_REG_CTRL, ctrl);
}

void nml_write_palette(int idx, uint8_t r, uint8_t g, uint8_t b) {
    if (idx < 0 || idx >= 256) return;
    reg_write(NML_PALETTE_BASE + ((unsigned)idx * 4u),
              ((uint32_t)r << 16) | ((uint32_t)g << 8) | (uint32_t)b);
}

void nml_write_tile(int col, int row, uint8_t tile_id) {
    if (col < 0 || col >= NML_TILEMAP_COLS) return;
    if (row < 0 || row >= NML_TILEMAP_ROWS) return;
    unsigned offset = NML_TILEMAP_BASE + (unsigned)(row * NML_TILEMAP_COLS + col);
    /* Byte write: ARM emits STRB, the LW bridge sets avs_byteenable for the
     * targeted lane, and the HW selects that lane via avs_address[1:0]. */
    *((volatile uint8_t *)g_base + offset) = tile_id;
}

void nml_write_sprite(int slot, const nml_sprite_t *s) {
    if (slot < 0 || slot >= NML_MAX_SPRITES || s == NULL) return;
    unsigned base = NML_SPRITE_TABLE_BASE + (unsigned)slot * 8u;

    uint32_t w0 = ((uint32_t)(uint16_t)s->y << 16) | (uint32_t)(uint16_t)s->x;
    uint32_t w1 = (uint32_t)s->sprite_id        |
                  ((uint32_t)s->flags       <<  8) |
                  ((uint32_t)s->palette_off << 16) |
                  ((uint32_t)s->reserved    << 24);

    reg_write(base + 0, w0);
    reg_write(base + 4, w1);
}

void nml_hide_sprite(int slot) {
    nml_sprite_t hidden = {
        .x          = NML_SPRITE_HIDE,
        .y          = NML_SPRITE_HIDE,
        .sprite_id  = 0,
        .flags      = 0,                /* clears ACTIVE */
        .palette_off = 0,
        .reserved   = 0,
    };
    nml_write_sprite(slot, &hidden);
}

void nml_clear_sprites(void) {
    for (int i = 0; i < NML_MAX_SPRITES; ++i) nml_hide_sprite(i);
}
```

```
/* BCD-pack `n` into nibbles for the HW HUD digit lookup. nibble 0 = ones,
 * nibble 1 = tens, etc. Up to `nibbles` digits; overflow silently wraps. */
static uint32_t bcd_pack(uint32_t n, int nibbles) {
    uint32_t out = 0;
    for (int i = 0; i < nibbles; ++i) {
        out |= (n % 10u) << (i * 4);
        n /= 10u;
    }
    return out;
}

void nml_set_player_state(int16_t px, int16_t py,
                          uint8_t hp, uint8_t wave, uint16_t level) {
    reg_write(NML_REG_PLAYER_POS,
              ((uint32_t)(uint16_t)py << 16) | (uint32_t)(uint16_t)px);

    /* Wave is BCD-packed into bits [15:8] so the HW HUD can read 2 digit
     * nibbles without a binary divider. hp stays binary 0..100 because the
     * HP bar is a fill-fraction, not a digit string. */
    uint32_t wave_bcd = bcd_pack((uint32_t)wave, 2);
    reg_write(NML_REG_PLAYER_STATS,
              (uint32_t)hp                 |
              ((wave_bcd & 0xFFu)    <<  8) |
              ((uint32_t)level       << 16));
}

void nml_set_score(uint32_t score, uint32_t kills) {
    /* Score is BCD-packed into bits [23:0] (6 digits). Above 999,999 the
     * top digits silently wrap -- bullet stream caps score well below that. */
    uint32_t score_bcd = bcd_pack(score, 6);
    reg_write(NML_REG_SCORE,      score_bcd);
    reg_write(NML_REG_KILL_COUNT, kills);
}

void nml_set_hud_aux(uint8_t ammo, uint8_t art_charges, uint8_t gas_charges) {
    /* Ammo BCD-packed into bits [7:0]; charges occupy single 4-bit nibbles
     * at [11:8] (art) and [15:12] (gas). Clamp inputs so a runaway value
     * can't corrupt other bits. */
    uint32_t ammo_bcd = bcd_pack((uint32_t)ammo, 2) & 0xFFu;
    uint32_t art      = (uint32_t)(art_charges > 9 ? 9 : art_charges) & 0xFu;
    uint32_t gas      = (uint32_t)(gas_charges > 9 ? 9 : gas_charges) & 0xFu;
    reg_write(NML_REG_HUD_AUX, ammo_bcd | (art << 8) | (gas << 12));
}

int nml_in_vblank(void) {
    return (reg_read(NML_REG_STATUS) & NML_STATUS_VBLANK) ? 1 : 0;
}

void nml_commit_frame(void) {
    /* Set SWAP; HW auto-clears it. Wait until SWAP_PENDING falls (commit
     * happened on the next vsync). Cap the wait so a stuck pipeline doesn't
     * hang the game loop. */
    uint32_t ctrl = reg_read(NML_REG_CTRL);
    reg_write(NML_REG_CTRL, ctrl | NML_CTRL_SWAP);

    for (int i = 0; i < 200000; ++i) {
        if (!(reg_read(NML_REG_STATUS) & NML_STATUS_SWAP_PENDING)) return;
    }
    fprintf(stderr, "nml_commit_frame: timed out waiting for SWAP commit\n");
```

```
}
```

Makefile

```
# Makefile for No Man's Land userspace game binary.
#
# Targets:
#   make            -- cross-compile for the DE1-SoC ARM HPS (default).
#                      Links the real joydev input reader (input_real.c).
#                      Output: nml_game (ELF, armhf).
#   make native     -- compile for the host (laptop / lab machine) with the
#                      real driver + joydev reader. Runs only on Linux with
#                      /dev/mem accessible; useful from the board's own gcc.
#   make fake-input -- cross-compile for ARM but with input_fake.c instead
#                      of the joydev reader. Useful to smoke-test the board
#                      without a controller plugged in.
#                      Output: nml_game_fake.
#   make terminal   -- host build with the terminal renderer (no FPGA needed)
#                      and input_fake.c (no /dev/input/js0 on the host).
#                      Output: nml_game_term.
#   make clean      -- remove build artefacts.
#
# Cross-compile prerequisite: gcc-arm-linux-gnueabihf (Debian/Ubuntu pkg name).
# The Quartus SoC EDS toolchain ships an equivalent compiler if you prefer.

CC_HOST    ?= gcc
CC_ARM     ?= arm-linux-gnueabihf-gcc

CFLAGS_COMMON := -Wall -Wextra -O2 -std=c11 -D_POSIX_C_SOURCE=200809L
LDFLAGS       :=
TARGET_FPGA   := nml_game
TARGET_FAKE   := nml_game_fake
TARGET_TERM   := nml_game_term

SRC_GAME      := game.c main.c wave.c autoatk.c
SRC_INPUT     := input_real.c
SRC_INPUT_FAKE:= input_fake.c
SRC_RENDER    := render.c nml_gpu.c
SRC_TERMINAL  := render_terminal.c

ALL_FPGA      := $(SRC_GAME) $(SRC_INPUT)      $(SRC_RENDER)
ALL_FAKE      := $(SRC_GAME) $(SRC_INPUT_FAKE) $(SRC_RENDER)
ALL_TERMINAL  := $(SRC_GAME) $(SRC_INPUT_FAKE) $(SRC_TERMINAL)

# ---- default: cross-compile for ARM (DE1-SoC HPS) with joydev input ----
.PHONY: all
all: $(TARGET_FPGA)

$(TARGET_FPGA): $(ALL_FPGA)
        $(CC_ARM) $(CFLAGS_COMMON) -static -o $@ $(ALL_FPGA) $(LDFLAGS)
        @echo "Built $@ for armhf (DE1-SoC HPS). Copy to the board with scp."

# ---- native: compile on the board itself --------------------------------
.PHONY: native
native:
        $(CC_HOST) $(CFLAGS_COMMON) -o $(TARGET_FPGA) $(ALL_FPGA) $(LDFLAGS)
        @echo "Built $(TARGET_FPGA) for native arch. Useful only on the board."
```

```
# ---- fake-input: cross-compile with input_fake.c (no controller needed) -
.PHONY: fake-input
fake-input: $(TARGET_FAKE)

$(TARGET_FAKE): $(ALL_FAKE)
        $(CC_ARM) $(CFLAGS_COMMON) -static -o $@ $(ALL_FAKE) $(LDFLAGS)
        @echo "Built $@ for armhf with input_fake.c. Copy to the board with scp."

# ---- terminal: host build, no FPGA needed ------------------------------
.PHONY: terminal
terminal: $(TARGET_TERM)

$(TARGET_TERM): $(ALL_TERMINAL)
        $(CC_HOST) $(CFLAGS_COMMON) -DNML_TERMINAL_BUILD -o $@ $(ALL_TERMINAL) $(LDFLAGS)

.PHONY: clean
clean:
        rm -f $(TARGET_FPGA) $(TARGET_FAKE) $(TARGET_TERM) *.o
```