CSEE 4840 · EMBEDDED SYSTEM DESIGN · SPRING 2026

Plants vs Zombies.

Design document — a 4×8 tower-defense on the Terasic DE1-SoC, with an HPS game loop and an FPGA scanline renderer driving VGA at 60 Hz.

Hao Cai · hc3612

Chenhao Yang · cy2822

Zhenghang Zhao · zz3410

Yabkun Li · yl6022

DESIGN DOCUMENT · V2

APRIL 2026

CONTENTS

Overview.

A tower-defense game built on the Terasic DE1-SoC: HPS-side C game loop, FPGA-side scanline renderer, USB gamepad input.

We are building a simplified version of *Plants vs Zombies* on the Terasic DE1-SoC board. The game takes place on a 4×8 grid where the player places defensive plants to stop waves of zombies that walk in from the right edge of the screen. Surviving every wave wins the game; letting a single zombie reach the left edge loses it.

The work divides between hardware and software along the HPS–FPGA boundary. The FPGA drives a 640×480 VGA display at 60 Hz through a scanline-based pipeline with dual line buffers and a shape/sprite table. The HPS runs Linux and hosts the game loop in C: grid state management, plant and zombie logic, collision detection, the sun resource economy, and input polling from a USB gamepad via `libusb`. A kernel driver sits between the two, exposing memory-mapped Avalon registers as `ioctl` calls on `/dev/pvz`.

Audio goes through the on-board Wolfson WM8731 codec. We plan to stream an 8-bit PCM arrangement of the Plants vs Zombies theme as background music, with short sound effects (pea fire, zombie groan, plant placement) mixed on top.

The rest of this document covers the system architecture, algorithms on both sides of the HPS–FPGA boundary, resource budgets, and a bit-level specification of the control registers.

4×8
PLAYFIELD GRID

640×480
VGA @ 60 HZ

HPS + FPGA
HARDWARE / SOFTWARE SPLIT

Block diagram.

The full system — HPS runs Linux; the FPGA fabric does the scanline work; Avalon-MM crosses the lightweight bridge between them.

Figure 2.1 shows the full system. The left half runs on the ARM Cortex-A9 under Linux; the right half is synthesized into the FPGA fabric. Communication crosses a single lightweight AXI bridge that the Avalon-MM agent sits behind, mapped at physical address `0xFF200000` from the CPU's perspective.

Figure 2.1 is deliberately coarse: it answers "what talks to what, and how wide is the bus?" Section 2.2 opens up the `pvz_top` box in a separate diagram; the software side is expanded in Figure 6.2.

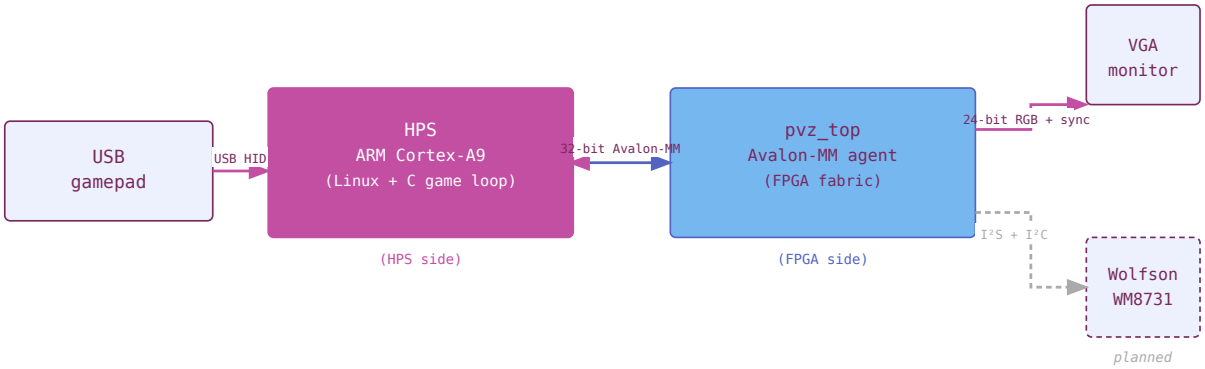

FIG 2.1 · TOP-LEVEL SYSTEM. THE HPS HOSTS THE C GAME LOOP AND ITS KERNEL DRIVER; THE `PVZ_TOP` PERIPHERAL LIVES IN THE FPGA FABRIC BEHIND THE LIGHTWEIGHT HPS-TO-FPGA BRIDGE (PHYSICAL BASE `0XFF20_0000`). ON THE BRIDGE, THE CPU ISSUES BYTE-ALIGNED `IOWRITE32`S WHILE `PVZ_TOP` SEES THE LOW-ORDER BITS AS A WORD-ALIGNED OFFSET. THE VGA PORT CARRIES 3×8-BIT R/G/B PLUS `HS`, `VS`, `BLANK_N`, `SYNC_N` AND `VGA_CLK`. THE DASHED PATH TO THE WOLFSON CODEC (I²S DATA + I²C CONFIG) IS NOT YET WIRED THROUGH: `SOC_SYSTEM_TOP.SV` ASSIGNS CODEC PINS TO SAFE STUBS PENDING AN `AUDIO_CONTROLLER` MODULE.

2.1 Module Summary

Table 2.1 lists every hardware module, its role, and key parameters.

TBL 2.1 · FPGA MODULE INVENTORY.

MODULE	TYPE	DESCRIPTION
`pvz_top`	Avalon-MM agent	Top-level peripheral. Decodes five 32-bit registers on the lightweight bridge and wires every sub-module together.
`vga_counters`	Timing gen.	Produces `hcount` (0–1599) and `vcount` (0–524) for 640×480 @ 60 Hz from the 50 MHz board clock.
`linebuffer`	Dual SRAM	Two `640×8-bit` banks. One is read for display while the other accepts writes from the renderer; roles swap every hsync.

MODULE	TYPE	DESCRIPTION
`bg_grid`	Grid LUT	Holds a per-cell palette index for the 4×8 game board. Shadow/active double-buffered at vsync.
`shape_table`	Entry store	48-entry table, each 48 bits. CPU writes go to a shadow copy that latches to the active copy at vsync.
`shape_renderer`	Scanline FSM	Fills the draw line buffer each scanline: background first, then overlays all visible shapes using a painter's-algorithm pass.
`color_palette`	LUT (comb.)	256-entry lookup from 8-bit index to 24-bit RGB. Purely combinational; no BRAM consumed.
`sprite_rom`	Block RAM	1,024×8-bit M10K block initialised from `peas_idx.mem`. Currently stores one 32×32 peashooter sprite; read at 1-cycle latency, `0xFF` means transparent.

2.2 Peripheral Internal Structure

Figure 2.2 expands the `pvz_top` box from Figure 2.1. The register decode on the west side accepts Avalon writes and routes them to the appropriate storage block. The renderer sits in the middle, pulling from the three memories (shape table, background grid, sprite ROM) and writing pixels into whichever line buffer is currently the "draw" buffer. The east side drives VGA pins through the palette LUT. Every labelled bit-width is the actual Verilog signal width, not a conceptual one.

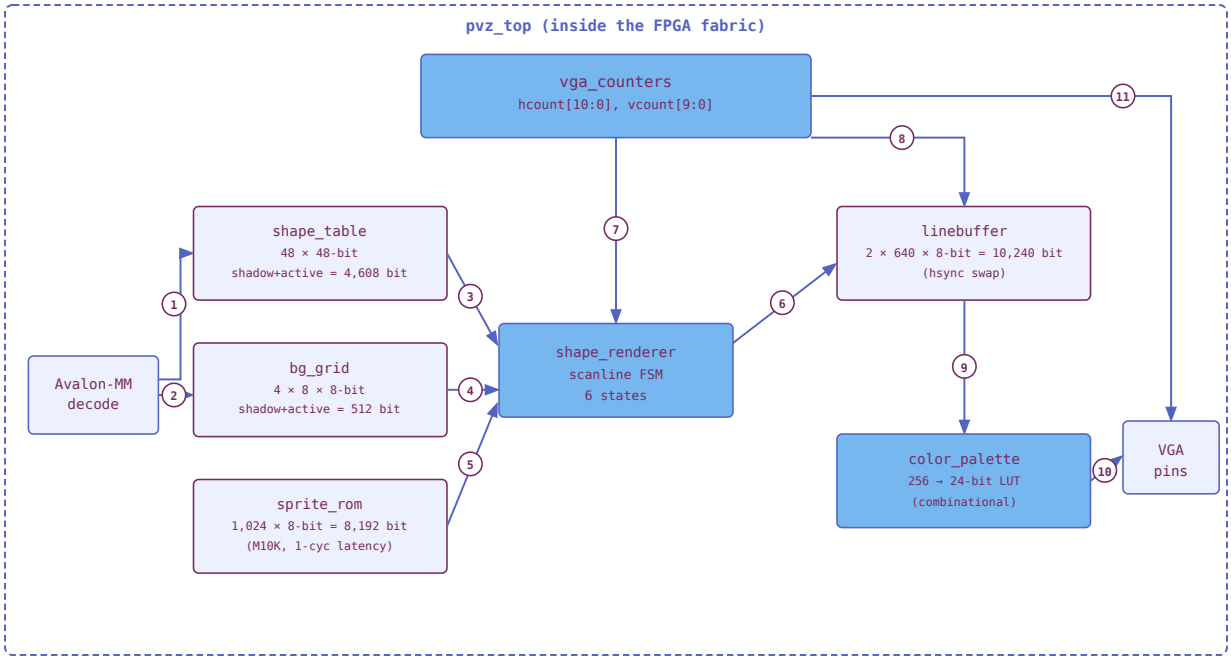

FIG 2.2 · INSIDE `PVZ_TOP`. MEMORIES CARRY A PLUM OUTLINE, FSM AND COMBINATIONAL LOGIC CARRY A ROYAL OUTLINE. THE NUMBERED EDGES ARE KEYED TO TABLE 2.2. THE SPRITE ROM CARRIES A 1-CYCLE READ LATENCY, SO `SHAPE_RENDERER` ISSUES THE ADDRESS ON ARROW 5 ONE CYCLE BEFORE IT EXPECTS THE PIXEL, KEEPING A 1-ENTRY PENDING BUFFER TO COMMIT EACH RETURNED BYTE (AND SUPPRESSING THE WRITE WHEN THE BYTE EQUALS THE TRANSPARENT SENTINEL `0XFF`).

TBL 2.2 · SIGNAL KEY FOR FIGURE 2.2.

#	FROM	TO	SIGNALS (WIDTH)
1	Avalon decode	`shape_table`	`addr_wr`, `data0/1_wr`, `commit_wr`; staged 32-bit `writedata`
2	Avalon decode	`bg_grid`	`wr_en`, `wr_row`[1:0], `wr_col`[2:0], `wr_color`[7:0]
3	`shape_table`	`shape_renderer`	48-bit entry unpacked to `rd_type`[1:0], `rd_visible`, `rd_x`[9:0], `rd_y/w/h`[8:0], `rd_color`[7:0]
4	`bg_grid`	`shape_renderer`	`color_out`[7:0] from (`bg_px`, `bg_py`) lookup
5	`sprite_rom`	`shape_renderer`	`sprite_rd_addr`[9:0] → `sprite_rd_pixel`[7:0] (1-cycle latency)
6	`shape_renderer`	`linebuffer`	`lb_wr_en`, `lb_wr_addr`[9:0], `lb_wr_data`[7:0]
7	`vga_counters`	`shape_renderer`	scanline (`vcount`[9:0]), `render_start` pulse
8	`vga_counters`	`linebuffer`	`rd_addr`[9:0] (pixel x), `swap` (hsync pulse)
9	`linebuffer`	`color_palette`	`rd_data`[7:0] palette index
10	`color_palette`	VGA pins	`R`[7:0], `G`[7:0], `B`[7:0]

#	FROM	TO	SIGNALS (WIDTH)
11	vga_counters	VGA pins	VGA_HS, VGA_VS, VGA_BLANK_n, VGA_SYNC_n, VGA_CLK

Game loop.

The HPS steps entities at 60 Hz and writes the shape table; the FPGA latches at vsync and blits two line buffers to VGA.

3.1 Game Loop (Software)

The main program runs a fixed-rate 60 Hz loop on the HPS. Each frame follows the sequence shown in Alg 3.1. Frame pacing uses `gettimeofday()` to measure elapsed time and sleeps for the remainder of the 16.67 ms budget.

```
function main_loop():
    game_init()
    while state == PLAYING do
        t0 = current_time()

        // 1. Input
        key = input_poll()            // non-blocking USB/keyboard read
        handle_input(key)             // move cursor, place/remove plant

        // 2. Update game state
        update_sun()                  // +25 sun every 10 s
        update_spawning()             // wave-table-driven zombie spawns
        update_plant_firing()         // peashooters fire every 2 s
(PLANT_FIRE_COOLDOWN)
        update_projectiles()          // peas move right at 2 px/frame
        update_zombies()              // zombies move left; eat plants
        check_collisions()            // pea-zombie, zombie-plant
        check_win_lose()              // all waves cleared? zombie at x=0?

        // 3. Render
        render_background()           // ioctl: BG_CELL for each grid cell
        render_plants()               // ioctl: shape/sprite per plant
        render_zombies()              // ioctl: shape/sprite per zombie
        render_projectiles()          // ioctl: shape/sprite per pea
        render_hud()                  // sun counter, wave indicator
        commit_shapes()               // ioctl: SHAPE_COMMIT

        // 4. Pace
        dt = current_time() - t0
        sleep(16667 us - dt)
    end
```

Alg 3.1 · Main game loop (60 Hz).

3.1.1 State-Aware Rendering and Shape-Index Allocation

`render_frame` in `sw/render.c` is the sole writer of shape-table entries. It always refreshes the background grid first. When `gs→state == STATE_PLAYING` it then emits the plants, zombies, projectiles, cursor, and HUD; when the game is in `STATE_WIN` or `STATE_LOSE` it

instead hides every entity slot (writing `visible = 0`) and draws a single coloured rectangle as a win/lose banner. The renderer still walks all 48 slots every scanline — visibility is gated in software, not hardware.

The shape index determines z-order, since the renderer applies a painter's-algorithm pass in index order and later writes overwrite earlier ones. Software therefore allocates indices by stacking layer:

TBL 3.1 · SHAPE-TABLE INDEX ALLOCATION.

INDICES	SLOT	ROLE
0–31	Plants	One slot per grid cell (4 rows × 8 cols).
32–36	Zombies	`MAX_ZOMBIES` = 5 active at once.
37–42	Projectiles	Up to 6 visible peas; extras are dropped this frame.
43–46	HUD digits	Up to 4 7-segment digits for the sun counter.
47	Cursor	Yellow cell highlight; drawn last so it never hides behind a plant.

The layout is the fix landed in commit `dc297d4`: an earlier version placed the cursor at a low index and discovered that plants drew on top of it. Pushing the cursor to slot 47 makes the top-of-stack guarantee explicit, and symmetrically ensures HUD digits always render above the game entities.

3.1.2 Collision Detection

Collision checks run in software every frame. Two kinds of overlap matter:

1. **Pea–Zombie.** A pea and a zombie collide when they occupy the same row and their bounding boxes overlap horizontally: $\text{pea}_x + \text{pea}_w \geq \text{zombie}_x \wedge \text{pea}_x \leq \text{zombie}_x + \text{zombie}_w$. On collision the pea is removed and the zombie loses 1 HP.
2. **Zombie–Plant.** A zombie's bounding box overlaps a grid cell when the zombie's left edge enters the cell's pixel region. The zombie then stops walking and instead deals damage over time to the plant until it is destroyed.

3.1.3 Sun Economy

The player starts with 150 sun. A background timer adds 25 sun every 10 seconds. Sunflower plants accelerate income by producing 25 sun every 10 seconds each. Placing a plant deducts its cost; the game refuses the placement when the player cannot afford it.

3.1.4 Wave System

Zombie spawns follow a table stored in software. Each entry specifies a time offset (in frames) and a zombie type. The table grows harder over successive waves by introducing

Conehead and Buckethead zombies and by decreasing the gap between spawns.

3.2 VGA Rendering Pipeline (Hardware)

The display engine produces pixels one scanline at a time using dual line buffers. While the VGA DAC reads from buffer A, the renderer fills buffer B with the next line's pixel data. The two swap roles at every horizontal sync pulse. This avoids the memory cost of a full frame buffer (307 KB at 8-bit color) and instead requires only 2 × 640 = 1,280 bytes.

3.2.1 Timing

The `vga_counters` module divides the 50 MHz board clock by two to produce a 25 MHz pixel clock. Table 3.1 lists the timing parameters.

TBL 3.1 · VGA 640×480 @ 60 HZ TIMING PARAMETERS.

PARAMETER	PIXELS/LINES	CLOCKS (50 MHZ)	NOTES
H Active	640 px	1280	Visible pixels
H Front Porch	16 px	32	
H Sync Pulse	96 px	192	Active low
H Back Porch	48 px	96	
H Total	**800 px**	**1600**	
V Active	480 lines	—	Visible lines
V Front Porch	10 lines	—	
V Sync Pulse	2 lines	—	Active low
V Back Porch	33 lines	—	
V Total	**525 lines**	—	
Frame Rate	50M / (1600 × 525) ≈ 59.5 Hz		

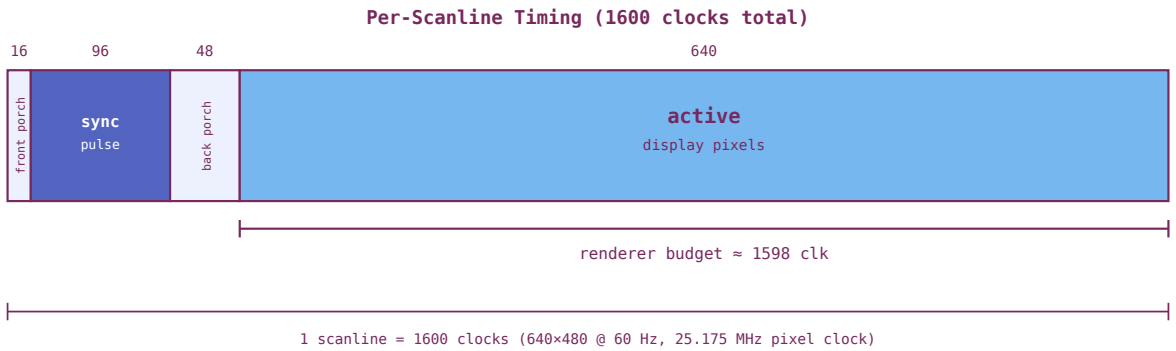

FIG 3.2 · PER-SCANLINE VGA TIMING.

3.2.2 Scanline Rendering Sequence

Figure 3.2 illustrates the per-scanline timing. Three control signals derived from `hcount` and `vcount` govern the pipeline:

- `lb_swap` fires at `hcount = 0` for each active line, toggling which buffer the DAC reads from.
- `render_start` fires at `hcount = 2`, kicking off the renderer's FSM for the *current* scanline.
- `vsync_latch` fires once per frame at `vcount = 480, hcount = 0`, copying all shadow registers to their active counterparts.

The renderer has roughly 1,598 clock cycles to fill the draw buffer before the next swap.

3.2.3 Renderer FSM

The `shape_renderer` walks through six states each scanline:

1. **S_IDLE** — Waits for `render_start`.
2. **S_BG_FILL** — Writes 640 pixels of background color from `bg_grid` (one clock per pixel).
3. **S_SHAPE_SETUP** — Drives `shape_index` into the table's read port; one clock later the decoded geometry (`type`, `visible`, `x`, `y`, `w`, `h`, `color`) is latched into local registers.
4. **S_SHAPE_CHECK** — Tests visibility and whether the shape's vertical extent intersects the current scanline. If not, advances `cur_shape` and loops back to S_SHAPE_SETUP.
5. **S_SHAPE_DRAW** — Walks `draw_x` across the shape's horizontal span. Behaviour branches on `type`:
    - **Types 0/1/2** (rectangle, circle, 7-seg digit): compute the hit predicate and, when it holds, write `s_color` into the line buffer at column `draw_x`.
    - **Type 3** (sprite): issue a sprite-ROM read at the downscaled address `{sprite_local_y[5:1], sprite_local_x[5:1]}`, set the `sprite_pending`

flag, and on the following clock commit the returned pixel into the line buffer *unless* it equals `0xFF` (transparent).

Later shapes overwrite earlier ones, giving painter's-algorithm z-ordering by table index.

6. **S_DONE** — Signals completion; idles until the next `render_start`.

States 3–5 repeat for all 48 shape entries, regardless of visibility. The sprite branch absorbs the ROM's 1-cycle read latency with a single-entry pending-pixel register, and spends one additional clock after `draw_x` passes `s_x + s_w` to flush the final fetch before advancing.

3.3 Shape Types

The renderer supports four shape primitives, selected by a 2-bit type field:

Type 0 — Filled Rectangle

Every pixel within the bounding box is a hit. Used for zombies, the cursor highlight, HUD backgrounds, and the win/lose banner.

Type 1 — Filled Circle

The hit test computes the squared distance from the pixel to the shape's center and compares it to the squared radius: $(x - c_x)^2 + (y - c_y)^2 \le (w/2)^2$, where $c_x = s_x + w/2$ and $c_y = s_y + h/2$. This is used for pea projectiles.

Type 2 — 7-Segment Digit

A hardcoded 20 × 30 pixel font with 3-pixel-thick segments. The digit value (0–9) is stored in bits [3:0] of the width field. A combinational decoder maps each value to a 7-bit segment mask, and the hit test checks whether the local (x, y) coordinate falls inside any active segment. This is used for the sun counter display.

Type 3 — Sprite

Reads pixel data from `sprite_rom.sv` instead of computing a geometric hit test. Currently one 32×32 sprite (the peashooter) lives in ROM and is up-scaled 2× to 64×64 on screen. The renderer suppresses the line-buffer write whenever the sampled byte equals `0xFF`, so irregular sprite silhouettes compose cleanly over the background. See Section 3.4 for the read-latency pipeline and Section 6.3 for how `w` / `h` are interpreted when `type = 3`.

3.4 Sprite System

Bitmap sprites are now rendered alongside the shape primitives. The V2Dino milestone added `sprite_rom.sv`, a 1,024-byte M10K ROM holding a single 32×32 peashooter sprite, plus a type-3 branch in `shape_renderer.sv` that renders the ROM contents on screen. Software

selects the sprite path by writing `type = 3` (`SHAPE_SPRITE`) in the shape-table entry; `w` and `h` still encode the on-screen footprint.

ROM layout. Pixels are stored row-major as 1,024 bytes (32 rows × 32 columns, 1 byte per pixel). Each byte is either a palette index that matches `color_palette.sv` (0–12 today) or the sentinel value `0xFF`, which the renderer treats as transparent. Quartus initialises the ROM at power-up from `peas_idx.mem` via `$readmemh`, which is listed as a `QUARTUS_SYNTH` file in `pvz_top_hw.tcl`.

2× up-scaling. The renderer addresses the ROM with `{sprite_local_y[5:1], sprite_local_x[5:1]}`, i.e. the local (*x*, *y*) within the on-screen footprint shifted right by one. Every 2×2 pixel block on screen reads the same source pixel, so a 32×32 sprite renders as 64×64 without any interpolation. Software currently emits peashooters with `w = h = 64` so the on-screen footprint exactly matches the scaled sprite.

Read-latency pipeline. The ROM is an inferred M10K block with a 1-clock registered output, so the pixel for address *A* appears on the data port in the *next* cycle. The renderer's sprite branch carries a 1-entry pending queue (`sprite_pending`, `sprite_pending_addr`): each clock it issues the fetch for column `draw_x`, increments `draw_x`, and (one cycle later) commits the returned byte to the line buffer at `sprite_pending_addr`. When the sampled byte is `0xFF` the write is suppressed, so the background shows through transparent regions. After `draw_x` passes `s_x + s_w` the FSM spends one extra cycle draining the final fetch before advancing to the next shape.

Limits of the current implementation. Only a single sprite lives in ROM, so the shape-table entry's `color` field is ignored for type 3 (there is no sprite-ID field yet). A larger ROM, multi-frame animation, and per-entry sprite IDs are future work tracked under the proposal's non-MVP features; the register map and shape-table layout leave room to add these without breaking the current binding.

3.5 Audio Playback (Planned)

The Wolfson WM8731 codec on the DE1-SoC accepts I²S serial audio data and provides analog output through the 3.5 mm jack.

Configuration. At boot, software writes a sequence of register values over the I²C bus to set the codec's sample rate (8 kHz), word length (16-bit), and enable the DAC output path.

Playback. A hardware audio controller reads PCM samples from an on-chip wave ROM and feeds them to the codec at the configured sample rate. The wave ROM holds a looping background track and a handful of short sound-effect clips. Software selects which clip to play by writing a start address and length to audio control registers; the controller then

streams from that region, mixing it with the background track by clamping the sum of the two sample values.

16-bit
PCM SAMPLE WIDTH

8 kHz
CODEC SAMPLE RATE

I²S + I²C
DATA & CONFIG BUSES

Plants & zombies.

Six plant archetypes and five zombie archetypes. Each entity is a bag of ints — HP, cost, damage, speed — plus a shape-table slot.

4.1 Plants

Three plant types are planned. Table 4.1 lists their properties.

TBL 4.1 · PLANT TYPES AND PROPERTIES.

PLANT	SUN COST	HP (RELATIVE)	BEHAVIOR
Peashooter	50	3 HP	Fires a pea down its row every 2 s
Sunflower	50	1×	Produces 25 sun every 10 s
Wall-nut	50	4×	Absorbs damage; no attack

4.2 Zombies

Three zombie types differ mainly in durability. All move leftward at the same speed and eat any plant they reach.

TBL 4.2 · ZOMBIE TYPES AND PROPERTIES.

ZOMBIE	HP (RELATIVE)	NOTES
Basic	1×	Goes down quickly
Conehead	2×	Traffic cone absorbs extra hits
Buckethead	4×	Highest durability

4.3 Projectiles

Peashooters fire pea projectiles that travel rightward at approximately 120 px/s (2 pixels per frame at 60 fps). Each pea deals 1 HP of damage on contact with a zombie and is then removed.

4.4 Sun Economy

- Starting sun: 150
- Passive income: +25 sun every 10 s
- Sunflower bonus: additional +25 sun per Sunflower per 10 s
- Maximum sun: capped (exact value TBD; bounds sprite table usage)

What fits in-chip.

On-chip SRAM is the main constraint. Graphics, audio, and framebuffer budgets are sized to live inside the Cyclone V's block RAM.

The DE1-SoC's Cyclone V 5CSEMA5F31C6 provides 4,450 Kbits of embedded memory (M10K blocks), 32,070 ALMs, and 87 DSP blocks. This section estimates our consumption of each.

5.1 Graphics Memory

Table 5.1 breaks down sprite storage. All sprites use 8-bit indexed color.

TBL 5.1 · GRAPHICS MEMORY BUDGET (8-BIT INDEXED COLOR).

CATEGORY	SIZE (PX)	FRAMES	BITS/SPRITE	TOTAL (BITS)
Peashooter	48×48	4	18,432	73,728
Sunflower	48×48	4	18,432	73,728
Wall-nut	48×48	2	18,432	36,864
Basic Zombie	48×64	4	24,576	98,304
Conehead Zombie	48×64	4	24,576	98,304
Buckethead Zombie	48×64	4	24,576	98,304
Pea projectile	12×12	1	1,152	1,152
Sun drop	24×24	1	4,608	4,608
Cursor highlight	72×80	1	46,080	46,080
Plant cards (UI)	40×50	3	16,000	48,000
Background tile	72×80	2	46,080	92,160
Digit font (0–9)	20×30	10	4,800	48,000
			Graphics total	**719,232**

5.2 Audio Memory

TBL 5.2 · AUDIO MEMORY BUDGET (16-BIT PCM, 8 KHZ).

CLIP	DURATION (S)	F_S (KHZ)	SAMPLES	BITS
Background music	15.0	8	120,000	1,920,000
Pea fire	0.3	8	2,400	38,400
Zombie groan	0.5	8	4,000	64,000
Plant placement	0.3	8	2,400	38,400
Game over	3.0	8	24,000	384,000

CLIP	DURATION (S)	F_S (KHZ)	SAMPLES	BITS
Victory	3.0	8	24,000	384,000
			Audio total	**2,828,800**

5.3 On-Chip SRAM Summary

TBL 5.3 · TOTAL ON-CHIP MEMORY USAGE ESTIMATE. "ACTUAL" IS WHAT COMMIT `A6EB347` SYNTHESISES; "PROJECTED" ADDS THE NOT-YET-SHIPPED SPRITES AND AUDIO PATH (TABLES 5.1, 5.2).

COMPONENT	ACTUAL (B)	PROJECTED (B)	NOTES
Line buffers (2×640×8)	10,240	10,240	Dual scanline buffers
Shape table (48 entries × 48)	4,608	4,608	Shadow + active combined
Background grid (4×8×8)	512	512	Shadow + active = 1,024
Color palette (256×24)	6,144	6,144	Combinational LUT (no BRAM)
Sprite ROMs	8,192	719,232	32×32 peashooter only (Actual)
Audio wave ROM	0	2,828,800	Not implemented yet
Total	**29,696**	**3,569,536**	
Available (M10K)		**4,450,000**	
Utilization	**0.67%**	**80.2%**	

The current image is far under budget; roughly 880 Kbits of headroom remain even after all planned sprites and the audio wave ROM are added. If audio memory proves too tight we can reduce the background music loop duration or drop the sample rate to 4 kHz, which would halve the audio footprint.

5.4 Logic Utilization

The shape renderer is the most logic-intensive module due to the signed multipliers for the circle distance check (11 × 11 bit multiply, two per pixel). Quartus typically maps these onto DSP blocks rather than fabric ALMs. We estimate:

- VGA counters + control: ~150 ALMs
- Shape renderer FSM + multipliers: ~800 ALMs + 4 DSP blocks
- Shape table + background grid: ~200 ALMs
- Color palette LUT: ~100 ALMs

- Avalon decode: ~50 ALMs
- Audio controller (planned): ~300 ALMs

Total estimated: ~1,600 ALMs out of 32,070 available (5%). Logic is not a constraint for this design.

~1,600	32,070	5%
ALMS USED	ALMS AVAILABLE	FABRIC UTILIZATION

Crossing the bridge.

How the HPS talks to the FPGA: a handful of Avalon-MM registers, a shape-table shadow, and a kernel ioctl layer in between.

6.1 System Architecture

The custom `pvz_top` peripheral connects to the HPS through Platform Designer's lightweight HPS-to-FPGA bridge. The CPU sees the peripheral's base address at `0xFF200000`. The Avalon bus uses 32-bit words internally, so `pvz_top`'s `address` port carries word offsets (0–4). The kernel driver accesses registers using byte offsets: `iowrite32(val, base + 4*N)` for word N.

All rendering-visible registers are **shadow/active double-buffered**. Writes from the CPU land in shadow copies. At the start of the vertical blanking interval (`vcount = 480, hcount = 0`), every shadow register latches into its active counterpart. This ensures the display never shows a partially updated frame.

6.2 Register Map

`pvz_top` exposes five 32-bit words at byte offsets `0x00`–`0x10` from the peripheral's base address (`0xFF20 0000` on the board). All five are write-only from the CPU side: the Avalon slave declared in `pvz_top_hw.tcl` and instantiated in `pvz_top.sv` has no `readdata` port, so a read transaction takes the bus default (the fabric returns `0`).

Table 6.1 summarises what each address does; the per-register subsections below break out the exact field layout and when each field becomes visible to the renderer.

TBL 6.1 · AVALON-MM REGISTER MAP (BYTE OFFSETS FROM BASE). "LATENCY" IS THE DELAY BETWEEN A SUCCESSFUL CPU WRITE AND THE CHANGE TAKING EFFECT FOR THE DISPLAY.

OFFSET	NAME	R/W	WRITE CAUSES	READ RETURNS	LATENCY
`0x00`	`BG_CELL`	W	Writes `color` into the shadow `bg_grid` at (`row`, `col`).	`0` (no readback).	Next vsync
`0x04`	`SHAPE_ADDR`	W	Latches the shape-table staging index `cur_addr` ← `writedata[5:0]`.	`0` (no readback).	See below
`0x08`	`SHAPE_DATA0`	W	Latches `staged_data0` ← `writedata` (type, visible, x, y).	`0` (no readback).	At commit
`0x0C`	`SHAPE_DATA1`	W	Latches `staged_data1` ← `writedata` (w, h, color).	`0` (no readback).	At commit
`0x10`	`SHAPE_COMMIT`	W	If `writedata` $\neq$ `0`, packs `{staged_data1, staged_data0}` into `shadow[cur_addr]`.	`0` (no readback).	Next vsync

6.2.1 BG_CELL — Background grid cell (offset 0x00)

TBL 6.2 · BG_CELL BIT-FIELDS.

BITS	NAME	DESCRIPTION
[2:0]	col	Grid column (0–7).
[4:3]	row	Grid row (0–3).
[15:8]	color	Palette index written into the cell.

Write causes. One entry of the shadow bg_grid is updated on the next clock: shadow[row][col] ⩽ color.

Read returns. The peripheral does not drive a readdata line, so reads return the fabric default (0).

Latching. The written cell is invisible until the next vsync_latch pulse (at vcount = 480, hcount = 0), at which point all 32 cells snap from shadow to active simultaneously.

6.2.2 SHAPE_ADDR — Shape-table staging address (offset 0x04)

TBL 6.3 · SHAPE_ADDR BIT-FIELDS.

BITS	NAME	DESCRIPTION
[5:0]	index	Shape slot selected for the next SHAPE_COMMIT (0–47).

Write causes. The internal cur_addr register captures writedata[5:0] on the next clock.

Read returns. 0.

Latching. No display-visible effect by itself. cur_addr is used only to pick which shadow slot SHAPE_COMMIT writes into.

6.2.3 SHAPE_DATA0 — Shape fields: type/visible/x/y (offset 0x08)

TBL 6.4 · SHAPE_DATA0 BIT-FIELDS.

BITS	NAME	DESCRIPTION
[1:0]	type	0=rect, 1=circle, 2=7-seg digit, 3=sprite.
[2]	visible	1 = draw, 0 = skipped in *S_SHAPE_CHECK*.
[12:3]	x	Pixel x (0–1023, but only 0–639 is on-screen).
[21:13]	y	Pixel y (0–511, on-screen 0–479).
[31:22]	–	Ignored (wired to 0 by the driver).

Write causes. `staged_data0` captures `writedata` on the next clock. Nothing else changes yet.

Read returns. `0`.

Latching. The staged word does not affect rendering until `SHAPE_COMMIT` packs it into `shadow[cur_addr]`, and `shadow` does not move into `active` until `vsync_latch`. Net effect: at most one frame of latency.

6.2.4 `SHAPE_DATA1` — Shape fields: w/h/color (offset `0x0C`)

TBL 6.5 · `SHAPE_DATA1` BIT-FIELDS.

BITS	NAME	DESCRIPTION
[8:0]	w	Width; for `type = 2`, `w[3:0]` encodes the digit value (see §6.3).
[17:9]	h	Height in pixels.
[25:18]	color	Palette index (ignored for `type = 3`).
[31:26]	–	Ignored.

Write causes. `staged_data1` captures `writedata` on the next clock.

Read returns. `0`.

Latching. Same two-stage latency as `SHAPE_DATA0`.

6.2.5 `SHAPE_COMMIT` — Commit staged shape (offset `0x10`)

TBL 6.6 · `SHAPE_COMMIT` BIT-FIELDS.

BITS	NAME	DESCRIPTION
[31:0]	commit	Any nonzero word triggers commit; zero is a no-op.

Write causes. When `writedata ≠ 0`, the hardware packs the staging registers into `shadow[cur_addr]` on the next clock, using this field order:

```
shadow[cur_addr] ⩽ {
    staged_data1[25:18], // color → bits [47:40]
    staged_data1[17:9],  // h     → bits [39:31]
    staged_data1[8:0],   // w     → bits [30:22]
    staged_data0[21:13], // y     → bits [21:13]
    staged_data0[12:3],  // x     → bits [12:3]
    staged_data0[2],     // visible → bit  [2]
    staged_data0[1:0]    // type  → bits [1:0]
};
```

Read returns. `0`.

Latching. `shadow[cur_addr]` becomes the value the renderer sees at the next `vsync_latch` pulse; until then the old active entry is still drawn.

6.2.6 Write Sequence

To update a shape entry, the CPU performs four writes in order:

1. Write `SHAPE_ADDR` with the entry index (0–47).
2. Write `SHAPE_DATA0` with type, visible, *x*, and *y*.
3. Write `SHAPE_DATA1` with *w*, *h*, and color.
4. Write `SHAPE_COMMIT` with any nonzero value.

The `PVZ_WRITE_SHAPE` ioctl emits exactly this four-write burst (see Appendix 10). Background cell writes (`BG_CELL`) need only a single write — no commit pulse is required because `bg_grid` has no per-entry staging register, only a single shadow array.

6.3 Shape Table Entry Format

Each entry in the 48-deep shape table is stored as 48 bits, packed as shown in Figure 6.1. The same field layout serves all four `type` values; the interpretation of `w` , `h` , and `color` depends on `type` :

- `type = 0` **(rectangle):** `w` and `h` are the footprint in pixels; `color` is the palette index written into the line buffer on every hit.
- `type = 1` **(circle):** `w` doubles as the diameter; the circle is centred at (`x` + `w` /2, `y` + `h` /2) with radius `w` /2. `color` fills the disk.
- `type = 2` **(7-segment digit):** `w` 's low four bits (`w[3:0]`) carry the digit value 0–9; software must still pass a `w` ≥ the glyph width (20 px) so the draw loop covers the full footprint (hence the common idiom `w = 32 + digit`). `h` should be ≥ 30. `color` is the stroke color.
- `type = 3` **(sprite):** `w` and `h` give the on-screen footprint, set to 64×64 today to match the 2× up-scaled peashooter. `color` is ignored; the pixel indices come from `sprite_rom.sv` and `0xFF` is the transparent sentinel. A future expansion will repurpose part of the `color` field as a sprite-ID when the ROM holds multiple sprites.

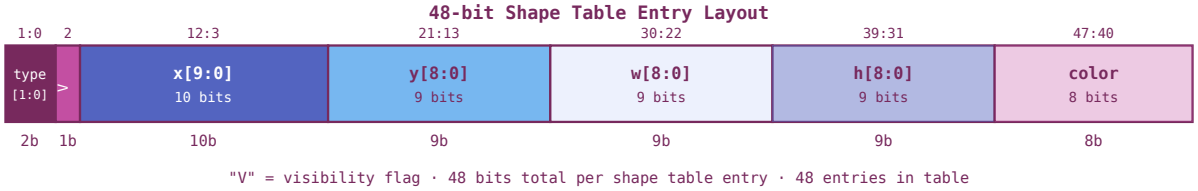

FIG 6.1 · 48-BIT SHAPE-TABLE ENTRY LAYOUT. "V" IS THE VISIBILITY FLAG.

6.4 Planned Register Extensions

Three groups of registers are not yet implemented. Sprite rendering currently keys entirely off `type = 3` in `SHAPE_DATA0`, so once the ROM holds more than one sprite the shape-table entry must also carry a sprite ID and an animation frame — the most natural place is to repurpose part of the currently unused upper bits of `SHAPE_DATA1` rather than add a new register. Audio and a frame-sync status read are genuinely new:

TBL 6.7 · PLANNED ADDITIONAL REGISTERS (NOT PRESENT IN COMMIT `A6EB347`).

OFFSET	NAME	PURPOSE
0x14	AUDIO_CTRL	Start/stop BGM, trigger sound effect
0x18	AUDIO_ADDR	Wave ROM start address for sound effect
0x1C	AUDIO_LEN	Sample count for sound effect clip
0x20	STATUS	[0]: vsync flag (read-only), for frame sync

6.5 Kernel Driver Interface

The Linux kernel module (`pvz_driver.ko`) registers a misc device at `/dev/pvz`. It matches the device tree compatible string `csee4840,pvz_gpu-1.0` generated by Platform Designer's `_hw.tcl` component descriptor. Figure 6.2 shows the full software path from the game loop to a peripheral register write.

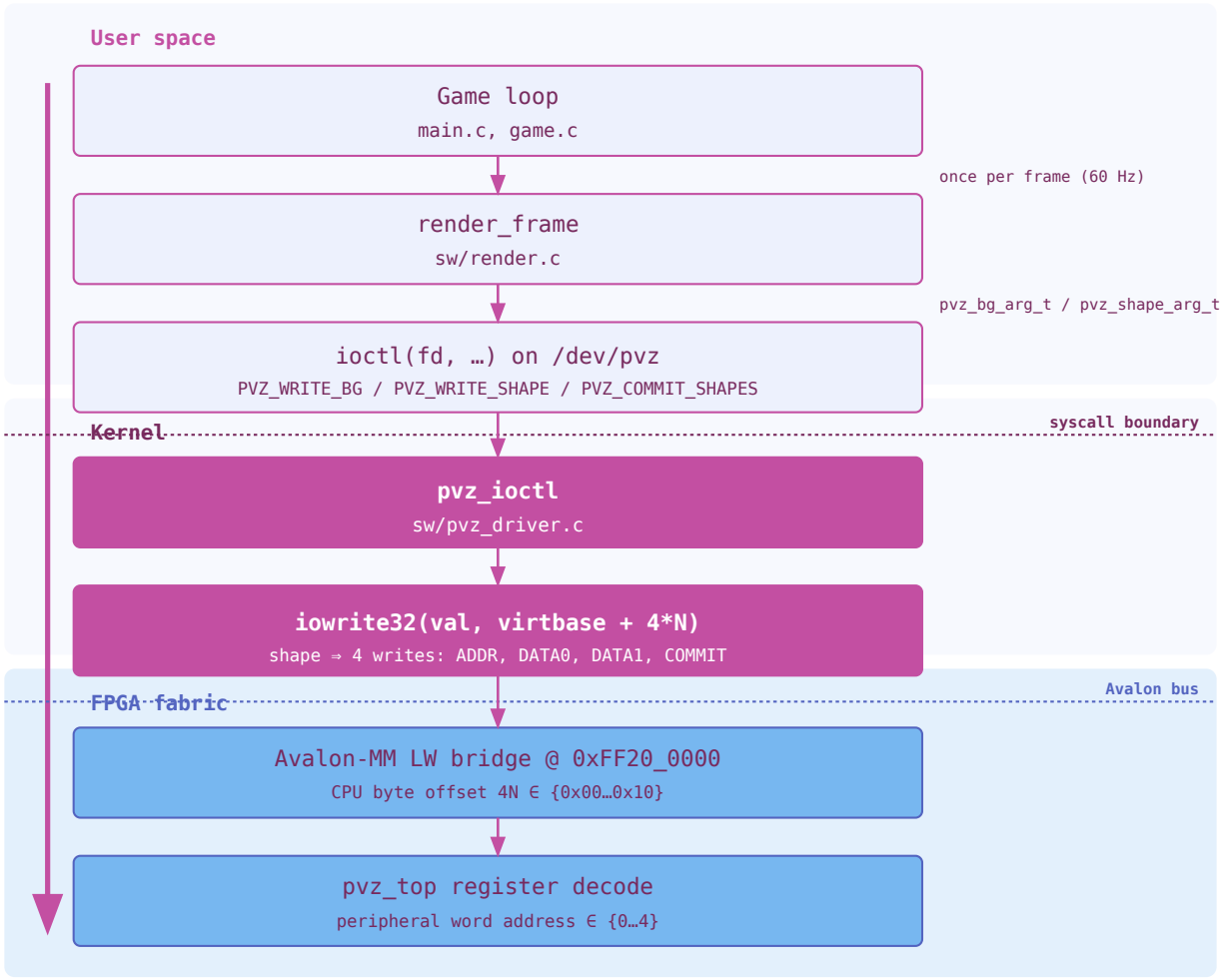

FIG 6.2 · SOFTWARE STACK. EACH FRAME, `RENDER_FRAME` CALLS INTO THE KERNEL DRIVER VIA `IOCTL`; THE DRIVER TRANSLATES IOCTL COMMANDS INTO MEMORY-MAPPED WRITES ON THE AVALON BRIDGE TO `PVZ_TOP`.

TBL 6.8 · KERNEL DRIVER IOCTL COMMANDS.

IOCTL	ARGUMENT	ACTION
PVZ_WRITE_BG	pvz_bg_arg_t	Pack col, row, color into BG_CELL
PVZ_WRITE_SHAPE	pvz_shape_arg_t	Sequence: ADDR → DATA0 → DATA1 → COMMIT
PVZ_COMMIT_SHAPES	(none)	Reserved for batch-commit optimization

The argument structures are:

```
typedef struct {
    unsigned char col;    // 0-8
    unsigned char row;    // 0-4
    unsigned char color;  // palette index
} pvz_bg_arg_t;

typedef struct {
    unsigned char index;   // 0-47
    unsigned char type;    // 0=rect, 1=circle, 2=7seg
    unsigned char visible; // 0 or 1
    unsigned short x, y;   // pixel position
    unsigned short w, h;   // dimensions (or digit value for type 2)
    unsigned char color;   // palette index
} pvz_shape_arg_t;
```

Lst 6.1 · Driver argument structures.

6.6 Platform Designer Integration

Three artifacts must stay in sync whenever the peripheral changes:

1. `pvz_top.sv` — Verilog module ports and register decode logic.
2. `pvz_top_hw.tcl` — Platform Designer component descriptor. Sets the compatible string to `"csee4840,pvz_gpu-1.0"` so the generated device tree includes the right node.
3. `pvz_driver.c` — Kernel module whose `of_device_id` table matches that compatible string.

If any one of these three diverges, the peripheral silently fails to appear at boot. This kind of mismatch is easy to introduce and hard to debug because nothing errors out—the device node just never shows up.

6.7 Color Palette

The MVP palette is a small, hardcoded set of 13 colors (Table 6.9). The final version will expand this to cover all sprite artwork.

TBL 6.9 · MVP COLOR PALETTE (8-BIT INDEX → 24-BIT RGB).

INDEX	NAME	RGB (HEX)	SWATCH
0x00	Black	#000000	
0x01	Dark Green	#1B5E20	

INDEX	NAME	RGB (HEX)	SWATCH
0x02	Light Green	#2D8B2D	
0x03	Brown	#8B4513	
0x04	Yellow	#FFD700	
0x05	Red	#FF0000	
0x06	Dark Red	#8B0000	
0x07	Green	#008000	
0x08	Dark Green 2	#006400	
0x09	Bright Green	#00FF00	
0x0A	White	#FFFFFF	
0x0B	Gray	#808080	
0x0C	Orange	#FFA500	

Gamepad.

Standard USB HID gamepad polled every frame via libusb. A small event queue carries decoded button presses to the game loop.

The player uses a USB gamepad connected to the HPS. The game program reads it through `libusb`, parsing USB HID reports for button and D-pad state. A keyboard fallback reads from `/dev/input/eventX` using non-blocking `read()` calls with `O_NONBLOCK`.

TBL 7.1 · CONTROLLER MAPPING.

BUTTON	KEYBOARD FALLBACK	ACTION
D-pad Up	Arrow Up	Move cursor up one row
D-pad Down	Arrow Down	Move cursor down one row
D-pad Left	Arrow Left	Move cursor left one column
D-pad Right	Arrow Right	Move cursor right one column
A	Space	Place selected plant at cursor
B	D	Remove plant / cancel selection
L/R	(planned)	Cycle plant type selection
Start	Escape	Quit game

Input is polled once per frame (60 Hz). Only key-down events are processed; key repeats and releases are filtered out.

On-screen real estate.

640×480 splits into a 60-pixel HUD strip up top and a 360-pixel 4×8 play grid below. Each cell is exactly 80×90 pixels.

Figure 8.1 shows how the 640×480 screen is divided.

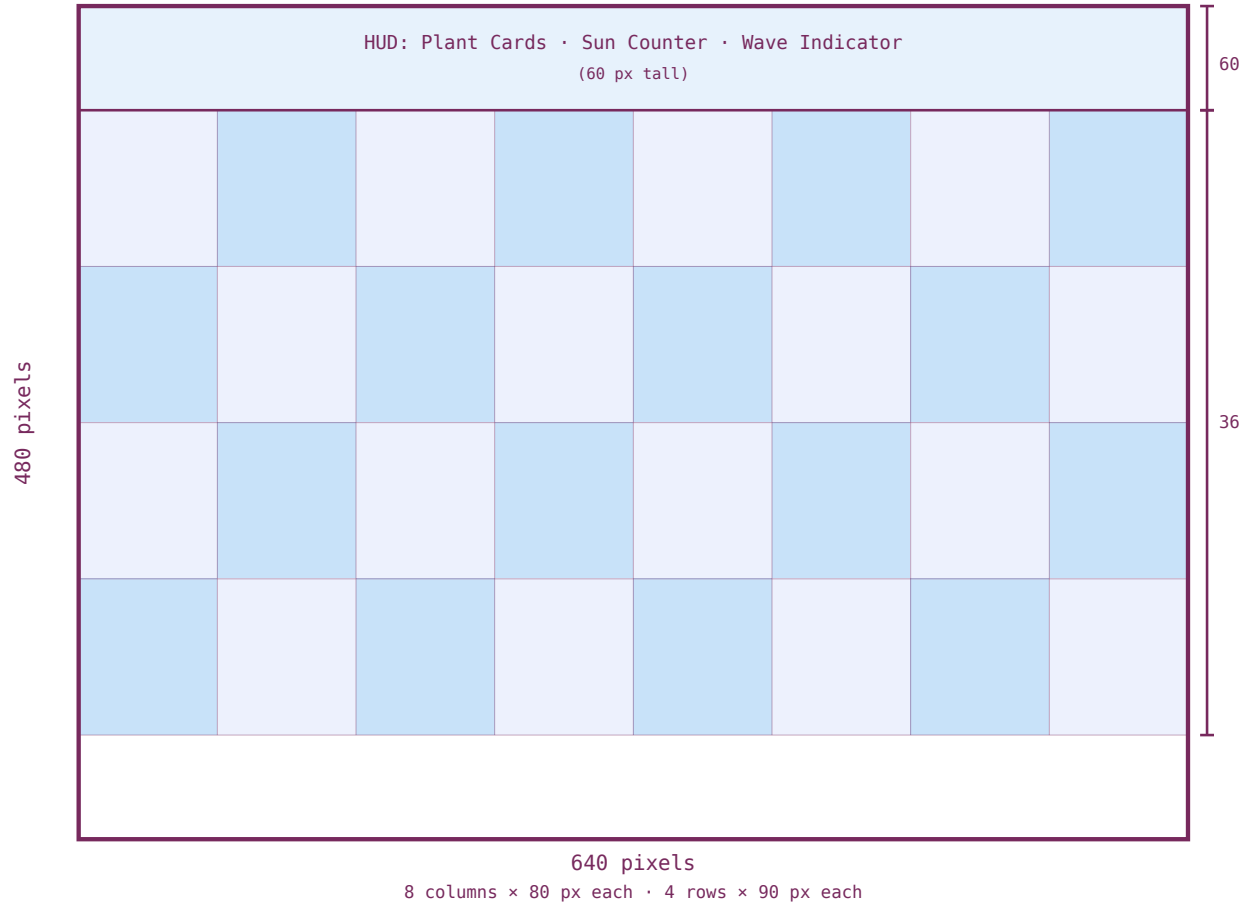

FIG 8.1 · SCREEN LAYOUT. TOP 60 PX HOLDS THE HUD; REMAINING 360 PX HOLDS THE 4×8 GRID (CELLS = 80×90 PX).

Coordinate mapping from grid cell (*r*, *c*) to pixel origin:

$$x = c \times 80$$
$$y = 60 + r \times 90$$

How we know it works.

Three layers of verification: SystemVerilog testbenches for hardware, C unit tests for game logic, and board-level smoke tests on the DE1-SoC.

9.1 Hardware Testbenches

- **VGA timing** — Verify `hcount` / `vcount` ranges, sync pulse widths and polarities, and blanking intervals against the 640×480 standard.
- **Linebuffer swap** — Confirm that buffer roles toggle correctly on the `lb_swap` pulse and that read and write ports never alias the same bank simultaneously.
- **Shape rendering** — Feed known shape entries and compare the linebuffer contents against a golden reference computed in the testbench.
- **End-to-end** — Simulate a full frame using Verilator, dump the pixel output to a PPM image, and visually inspect.

9.2 Software Unit Tests

A host-compilable test harness (`test/test_game.c`) exercises the game logic without any hardware dependency:

- Initial state validation (grid clear, sun = 150)
- Plant placement and sun deduction
- Sun auto-increment timing
- Pea–zombie collision detection
- Zombie destruction at 0 HP
- Lose condition (zombie at $x = 0$)
- Win condition (all zombies spawned and defeated)

9.3 On-Board Integration

- `test_shapes` — Writes a checkerboard background and several sample shapes via ioctl, confirming that the driver–hardware path works end to end.
- `test_input` — Polls `/dev/input/eventX` at 60 Hz and prints recognized keys, verifying that the gamepad/keyboard input pipeline works.

Software header reference.

The shared contract between game code, the kernel driver, and the FPGA — bitfield macros, ioctl IDs, kernel device state, and game-side function prototypes.

The declarations below are quoted verbatim from the source tree as of commit `a6eb347`. Static internals are omitted; only the identifiers a user-space or kernel-module reader needs are included.

3
IOCTL COMMANDS

a6eb347
SOURCE COMMIT

/dev/pvz
MISC DEVICE NODE

10.1 sw/pvz.h — shared ioctl and bitfield contract

```
#ifndef _PVZ_H
#define _PVZ_H

#include <linux/ioctl.h>

/* Register byte offsets (from driver base) */
#define PVZ_BG_CELL       0x00
#define PVZ_SHAPE_ADDR    0x04
#define PVZ_SHAPE_DATA0   0x08
#define PVZ_SHAPE_DATA1   0x0C
#define PVZ_SHAPE_COMMIT  0x10

/* Shape types (matches hw/shape_renderer.sv) */
#define SHAPE_RECT    0
#define SHAPE_CIRCLE  1
#define SHAPE_DIGIT   2
#define SHAPE_SPRITE  3  /* 32x32 sprite ROM, 2x → 64x64 on screen */

/* Background cell write argument */
typedef struct {
    unsigned char row;    /* 0-3 */
    unsigned char col;    /* 0-7 */
    unsigned char color;  /* palette index 0-255 */
} pvz_bg_arg_t;

/* Shape write argument */
typedef struct {
    unsigned char  index;   /* shape table index 0-47 */
    unsigned char  type;    /* SHAPE_RECT, SHAPE_CIRCLE, SHAPE_DIGIT, SHAPE_SPRITE
*/
    unsigned char  visible; /* 1=visible, 0=hidden */
    unsigned short x;       /* x position 0-639 */
    unsigned short y;       /* y position 0-479 */
    unsigned short w;       /* width (or digit value in low 4 bits for SHAPE_DIGIT)
*/
    unsigned short h;       /* height */
    unsigned char  color;   /* palette color index */
} pvz_shape_arg_t;

#define PVZ_MAGIC         'p'
#define PVZ_WRITE_BG      _IOW(PVZ_MAGIC, 1, pvz_bg_arg_t)
#define PVZ_WRITE_SHAPE   _IOW(PVZ_MAGIC, 2, pvz_shape_arg_t)
#define PVZ_COMMIT_SHAPES _IO (PVZ_MAGIC, 3)

#endif /* _PVZ_H */
```

Lst 10.1 · `sw/pvz.h` — register offsets, shape-type constants, ioctl argument structs, and ioctl IDs.

10.2 `sw/pvz_driver.h` — kernel-side device state

```
#ifndef _PVZ_DRIVER_H
#define _PVZ_DRIVER_H

#include <linux/miscdevice.h>
#include <linux/platform_device.h>

#define DRIVER_NAME "pvz"

struct pvz_dev {
    struct resource res;
    void __iomem   *virtbase;
};

#endif /* _PVZ_DRIVER_H */
```

Lst 10.2 · `sw/pvz_driver.h` — kernel module device-state struct.

10.3 Game-side function prototypes

`sw/game.h` (elided: constants and struct bodies — see the source for the full type definitions; constants match Table 5.1 and the timing numbers in §7):

```
/* Grid + window constants (subset) */
#define GRID_ROWS     4
#define GRID_COLS     8
#define GAME_AREA_Y   60
#define CELL_WIDTH    80
#define CELL_HEIGHT   90

/* Public entry points */
void game_init  (game_state_t *gs);
void game_update(game_state_t *gs);
int  game_place_plant (game_state_t *gs);
int  game_remove_plant(game_state_t *gs);
```

Lst 10.3 · `sw/game.h` — grid/window constants and public game-logic entry points.

The renderer's public surface from `sw/render.h` (all other functions in `render.c` are `static`):

```
int  render_init (int fpga_fd);  /* stash /dev/pvz fd */
void render_frame(const game_state_t *gs);  /* one-shot per frame */
```

Lst 10.4 · `sw/render.h` — renderer public API.

60 Hz
FRAME RATE

48
SHAPE-TABLE SLOTS

4×8
BACKGROUND GRID

Verilog module reference.

Hierarchy of Verilog modules and their port summaries — the hardware contract for the FPGA side.

11.1 Module-instantiation hierarchy

```
soc_system_top                (hw/soc_system_top.sv)
  soc_system (generated by Platform Designer)
    hps_0        -- HPS hard IP (ARM Cortex-A9, DDR3, peripherals)
    pvz_top_0    -- our peripheral:
      vga_counters   (1 instance)
      linebuffer     (1 instance; holds two 640x8 banks internally)
      bg_grid        (1 instance)
      shape_table    (1 instance)
      sprite_rom     (1 instance)
      shape_renderer (1 instance)
      color_palette  (1 instance; combinational, no state)
```

Lst 11.1 · Module-instantiation hierarchy showing where our peripheral sits within the Platform Designer system.

`soc_system` is generated by `hw/soc_system.qsys` and is not checked in as hand-written Verilog. The `pvz_top_0` instance is registered through `hw/pvz_top_hw.tcl` (§12.2).

11.2 Per-module port summaries

Each listing below reproduces the module's `module name( ... )` header (comments trimmed). Non-obvious ports carry a one-line annotation.

`pvz_top` — Avalon-MM agent, top of the peripheral tree.

```
module pvz_top(
    input  logic        clk,
    input  logic        reset,
    // Avalon-MM slave (agent); readdata deliberately absent (write-only).
    input  logic [2:0]  address,       // word offset 0..4; byte offsets are 4x
    input  logic [31:0] writedata,
    input  logic        write,
    input  logic        chipselect,
    // VGA conduit to board pins (see soc_system_top)
    output logic [7:0]  VGA_R, VGA_G, VGA_B,
    output logic        VGA_CLK, VGA_HS, VGA_VS,
    output logic        VGA_BLANK_n,
    output logic        VGA_SYNC_n
);
```

Lst 11.2 · `pvz_top` ports.

`vga_counters` — 640×480@60 Hz timing generator (extracted from lab3, Stephen A. Edwards).

```
module vga_counters(
    input  logic        clk50,         // 50 MHz board clock
    input  logic        reset,
    output logic [10:0] hcount,        // 0..1599 (2 x pixel column)
    output logic [9:0]  vcount,        // 0..524  (scanline + blanking)
    output logic        VGA_CLK,       // 25 MHz pixel clock (hcount[0])
    output logic        VGA_HS, VGA_VS,
    output logic        VGA_BLANK_n,
    output logic        VGA_SYNC_n
);
```

Lst 11.3 · `vga_counters` ports.

`linebuffer` — dual 640×8-bit banks with hsync swap.

```
module linebuffer(
    input  logic        clk,
    input  logic        reset,
    // Write port: shape_renderer drives the "draw" bank
    input  logic        wr_en,
    input  logic [9:0]  wr_addr,
    input  logic [7:0]  wr_data,
    // Read port: pvz_top drives pixel_x as rd_addr; registered output
    input  logic [9:0]  rd_addr,
    output logic [7:0]  rd_data,
    input  logic        swap           // pulse at hcount=0 flips sel flop
);
```

Lst 11.4 · `linebuffer` ports.

`bg_grid` — 4×8 shadow+active grid with coordinate lookup.

```
module bg_grid(
    input  logic        clk,
    input  logic        reset,
    // CPU write port (targets shadow only)
    input  logic        wr_en,
    input  logic [2:0]  wr_col,        // 0..7
    input  logic [1:0]  wr_row,        // 0..3
    input  logic [7:0]  wr_color,
    input  logic        vsync_latch,   // shadow → active on this pulse
    // Pixel-coordinate query: returns cell color or 0 outside game area
    input  logic [9:0]  px, py,
    output logic [7:0]  color_out
);
```

Lst 11.5 · `bg_grid` ports.

`shape_table` — 48-entry shadow+active table, write-staging with commit pulse.

```
module shape_table(
    input  logic        clk,
    input  logic        reset,
    // CPU write interface: staging regs then explicit commit
    input  logic        addr_wr,     input  logic [5:0]  addr_data,
    input  logic        data0_wr,    input  logic [31:0] data0,
    input  logic        data1_wr,    input  logic [31:0] data1,
    input  logic        commit_wr,   // shadow[cur_addr] ⩽ packed staged data
    input  logic        vsync_latch, // shadow → active each vsync
    // Read port (renderer-facing, unpacked from active)
    input  logic [5:0]  rd_index,
    output logic [1:0]  rd_type,
    output logic        rd_visible,
    output logic [9:0]  rd_x,
    output logic [8:0]  rd_y, rd_w, rd_h,
    output logic [7:0]  rd_color
);
```

Lst 11.6 · `shape_table` ports.

`sprite_rom` — 1024×8-bit M10K ROM loaded from `peas_idx.mem`; 1-cycle registered output.

```
module sprite_rom(
    input  logic        clk,
    input  logic [9:0]  addr,          // y*32 + x
    output logic [7:0]  pixel          // registered: valid one cycle after addr
);
```

Lst 11.7 · `sprite_rom` ports.

`shape_renderer` — scanline FSM; ports include a ready/valid-style handshake with the sprite ROM.

```
module shape_renderer(
    input  logic        clk, reset,
    input  logic        render_start,   // pulse at start of each scanline
    input  logic [9:0]  scanline,
    output logic        render_done,
    // bg_grid query
    output logic [9:0]  bg_px, bg_py,
    input  logic [7:0]  bg_color,
    // shape_table read port (1-cycle latency through shape_table internals)
    output logic [5:0]  shape_index,
    input  logic [1:0]  shape_type,
    input  logic        shape_visible,
    input  logic [9:0]  shape_x,
    input  logic [8:0]  shape_y, shape_w, shape_h,
    input  logic [7:0]  shape_color,
    // sprite_rom read port (address comb, pixel returns NEXT cycle)
    output logic [9:0]  sprite_rd_addr,
    input  logic [7:0]  sprite_rd_pixel,
    // linebuffer write port
    output logic        lb_wr_en,
    output logic [9:0]  lb_wr_addr,
    output logic [7:0]  lb_wr_data
);
```

Lst 11.8 · `shape_renderer` ports.

`color_palette` — combinational 8-bit → 24-bit LUT.

```
module color_palette(
    input  logic [7:0]  index,
    output logic [7:0]  r, g, b        // hardcoded 13 entries; rest black
);
```

Lst 11.9 · `color_palette` ports.

`soc_system_top` — DE1-SoC board top-level. Ports are all DE1-SoC-mandated I/O pins (VGA, HPS DDR3, HPS USB/Ethernet/SD, HEX, KEY, SW, LEDR, GPIO, etc.). The full list is too long to repeat here; see `hw/soc_system_top.sv` lines 9–155. Internally this module only instantiates `soc_system` (the Platform Designer-generated wrapper) and wires safe stubs to the unused board pins.

File formats.

On-disk formats and build artifacts: the sprite ROM hex file, device-tree fragments for Platform Designer, and what the build produces.

12.1 `peas_idx.mem` — sprite ROM initialiser

The file is the ASCII-hex format accepted by Verilog's `$readmemh`. `sprite_rom.sv` reads exactly 1024 bytes in row-major order (row 0 first, 32 columns per row, so byte $N = y \times 32 + x$). One byte per pixel, storing a palette index:

- `0x00`–`0x0C` — palette entries 0..12 as defined by `hw/color_palette.sv`.
- `0xFF` — transparent sentinel; the renderer suppresses the line-buffer write when the sampled byte equals `0xFF`.
- Any other value — currently treated as black by `color_palette` (its `default` case returns `8'h00`); software avoids emitting such values.

The file is generated out-of-band from the peashooter PNG artwork and must be kept in sync with the palette. It is consumed at synthesis time by Quartus and at simulation time by the testbench.

12.2 Device-tree fragment

`hw/pvz_top_hw.tcl` declares the Platform Designer metadata that ends up in the generated `.dtb`:

```
set_module_assignment embeddedsw.dts.compatible "csee4840,pvz_gpu-1.0"
set_module_assignment embeddedsw.dts.group      "pvz_gpu"
set_module_assignment embeddedsw.dts.vendor     "csee4840"
```

Lst 12.1 · `hw/pvz_top_hw.tcl` — Platform Designer metadata assignments that drive device-tree generation.

After `sopc2dts` + `dtc`, the node in `soc_system.dtb` looks like:

```
pvz_gpu_0: pvz_gpu@ff200000 {
    compatible = "csee4840,pvz_gpu-1.0";
    reg        = <0xff200000 0x00000020>;
};
```

Lst 12.2 · Generated device-tree node for the PvZ GPU peripheral.

The compatible string must also appear verbatim in the kernel module's `of_device_id` table (`sw/pvz_driver.c:pvz_of_match`) for probe to succeed.

12.3 Other inputs and outputs

The project currently reads or writes no other external-format files beyond the two above and the Quartus toolchain outputs. No save-game file, config file, or on-board persistent store exists: the game state is ephemeral and the player's only input device is the USB gamepad (parsed live by `sw/input.c` via libusb). Test fixtures in `sw/test/` link directly against the game library and need no external data.

.rbf
FPGA BITSTREAM

.dtb
DEVICE TREE BLOB

.mem
SPRITE ROM INIT