Hardware-Accelerated Block Game

A voxel renderer for the DE1-SoC

Wesley Maa (wm2505)
Mihir Joshi (mnj2122)
Josh Bernheisel (jcb2301)

CSEE 4840 – Embedded System Design
Spring 2026

Contents

1 Introduction

We made a block game. But it's not just any block game, it's a hardware-accelerated block game, run on a HPS and FPGA. The ARM cores under Linux handle the irregular control-heavy components of the game, which are the keyboard and mouse input, player physics, terrain generation, chunk streaming, lighting, fluids, redstone, inventory and crafting, chat commands, and per-face clipping and projection. The FPGA handles the regular pixel-parallel rendering work through a custom `voxel_gpu` peripheral that consumes packed quad descriptors, evaluates edge functions in parallel, looks up texture samples, blends color and depth, and drives a 640×480 VGA display out of SDRAM.

Our initial proposal described "a hardware-accelerated 3D rendering engine on the DE1-SoC that produces a real-time, interactive block game on a VGA display." This final system extends that original proposal in two major ways: On the hardware side, the rasterizer outgrew the original on-chip framebuffer plan and became a banded SDRAM-backed renderer with its own background flush engine and three line buffers feeding the VGA pins. On the software side, the "small procedurally generated block world" evolved into a complete game stack with chunk persistence, async meshing, four palette light banks, falling sand and gravel, water and lava flow, redstone circuits, a survival inventory, crafting menus, a chat console with slash commands, three game modes, and a home menu for managing saves.

This report covers what we built, why we made the architectural choices we did, and what we would do differently. Section 2 gives the system-level view and revised proposal. Section 3 walks through the rendering algorithms. Section 4 covers the FPGA peripheral and Section 5 covers the software stack. Section 6 documents the hardware/software interface in detail. Section 7 reports verification and Quartus implementation results. Section 8 summarizes team contributions, lessons learned, and recommendations for future work. Finally, Section 9 enumerates the actual software and RTL written.

2 Overview

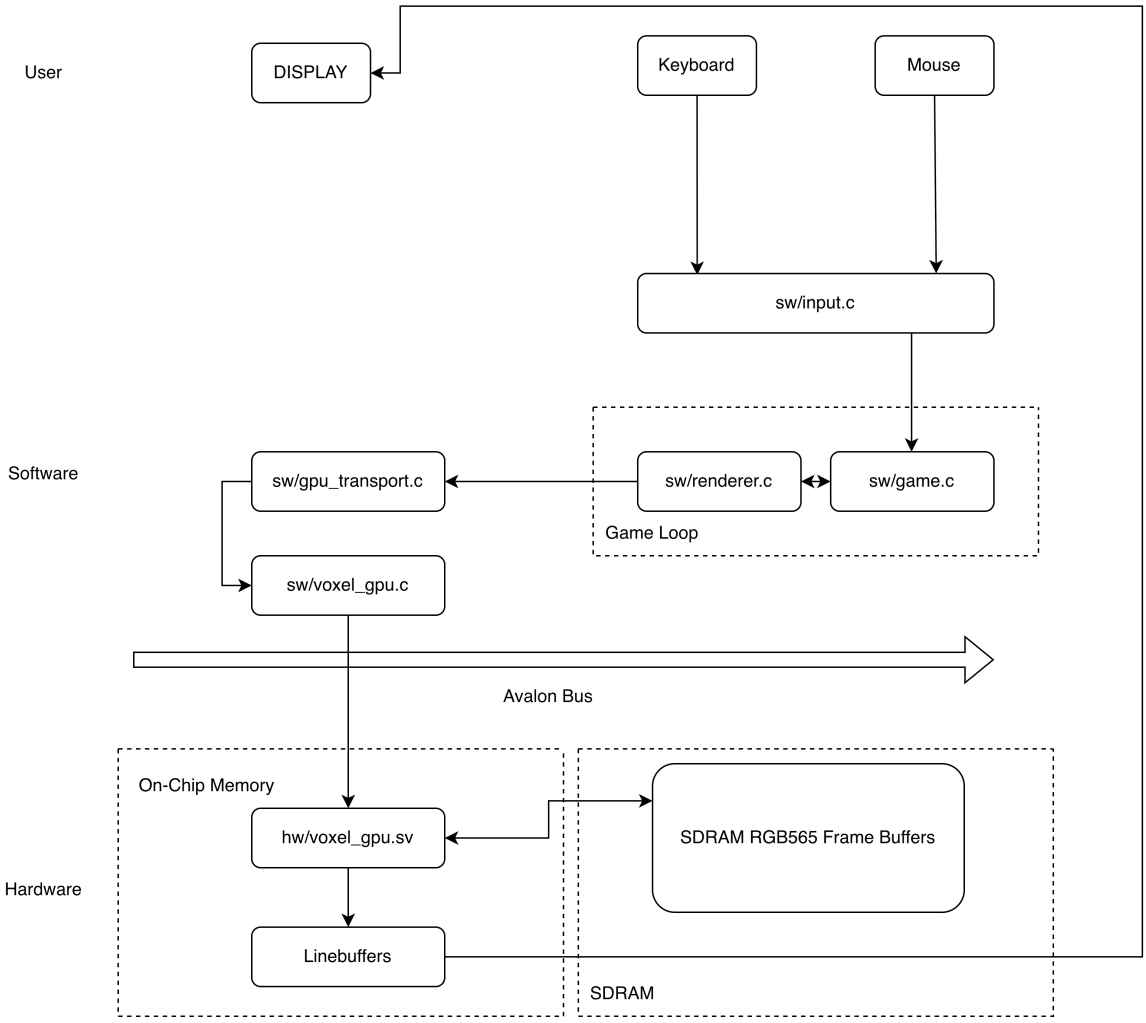

Figure 1: High-level system block diagram. The dashed regions separate the user, software, on-chip hardware, and SDRAM portions of the system. Game logic, world simulation, the renderer, and the kernel driver all live on the HPS. The FPGA receives a packed descriptor stream over the lightweight HPS-to-FPGA bridge via the Avalon Bus and produces VGA output from frame buffers held in board SDRAM.

2.1 What the player sees

When the board boots, our game starts at a small home menu where the player can pick a world from disk or create a new one.

Figure 2: Block Game Main Menu

Inside a world, the player has a first-person camera with mouse-look and WASD movement. Double pressing W will let the user sprint, while pressing space jumps or, in creative mode, flies up. The hotbar at the bottom of the screen shows nine inventory slots, selected either with the number keys or the scroll wheel. Left-click destroys the block under the crosshair and (in survival) drops the appropriate item while right-click places a held block. Pressing `E` opens the inventory with a 2×2 crafting grid and opening a crafting table or furnace shows the corresponding UI.

Pressing `T` or `/` opens a chat console. The chat can run slash commands: `/gamemode survival`, `/time set day`, `/setblock 10 5 0 stone`, `/fill ... lamp`, `/give diamond_pickaxe`, `/physics off`, and additional debugging and world-editing commands.

The world also has a day/night cycle. Sky color is generated as a 24-step gradient palette that walks through sunrise, noon, sunset, and night colors. In addition to skylight, lava, torches, and lamps emit block light that the renderer turns into one of four palette light banks per face. Water and lava flow with source/flow levels and react when they touch (generating cobblestone, stone, or obsidian, depending on the lava and water types involved). Sand and gravel fall as smooth entities before snapping back to the block grid. Redstone wires, torches, repeaters with 1–4 tick delays, comparators, levers, buttons, doors, pressure plates, and lamps all settle each tick in a bounded number of passes. The combinational parts of a circuit reach a stable state within that tick, while repeaters and other delayed components keep their own state across ticks.

Figure 3: Some redstone during night cycle

2.2 The hardware/software split

The architectural decision that mattered most was where to put the boundary between HPS code and FPGA logic. We decided to place that boundary at the descriptor interface. That way, the FPGA does not need know about chunks, blocks, the camera, or any other game state. It only sees a stream of 32-bit words that decodes into screen-space quadrilateral descriptors plus a handful of control registers. Everything else happens on the ARM in C. The renderer in `sw/renderer.c` is responsible for converting visible block faces into packed descriptors with Q24.8 edge equations, Q1.15 depth gradients, and (for textured faces) Q16.16 perspective-correct UV planes. The FPGA consumes that descriptor stream and performs rasterization, depth testing, color lookup, blending, and scanout.

2.3 Memory architecture

The single biggest change from our initial proposal is the framebuffer. We originally planned to keep a reduced 320x240x8 bit framebuffer on chip. However, our new frame requirements are 640×480 in RGB565, resulting in 614 400 bytes. Our 16-bit Z buffer adds another 614 400. On Cyclone V we have 397 M10K blocks, or 4 065 280 raw block-memory bits (508 160 bytes) before packing overhead. A full color and Z framebuffer would already exceed all available M10K storage before we drew any descriptors.

Instead, we render in bands and store full frames in external SDRAM. The screen is divided into eight 60-row bands. At any given moment, the FPGA holds two color band caches and two matching Z band caches (each 640×60×16 bits), totalling 307 200 bytes of logical on-chip cache, or roughly 60% of the raw M10K bit capacity. Software bins each descriptor into the one or more bands it touches. When a band finishes, a background flush controller copies the resulting RGB565 rows out to one of two full frames sitting in board SDRAM. A separate scanout path reads from the

other SDRAM frame through three 640-word line buffers and drives the VGA pins. When the frame is done, we flip which SDRAM frame is the front.

The on-chip storage budget is laid out in Table 1.

Resource	Dimensions	Size
SDRAM front color frame	640×480 RGB565	614 400 B
SDRAM back color frame	640×480 RGB565	614 400 B
Color band cache A	640×60 RGB565	76 800 B
Color band cache B	640×60 RGB565	76 800 B
Z band cache A	640×60×16 bit	76 800 B
Z band cache B	640×60×16 bit	76 800 B
VGA scanout line buffers	3×640×16 bit	3 840 B
Descriptor FIFO	1024 words	4 096 B
Texture atlas ROMs	2×128 tiles×16×16 indices	65 536 B physical
Reciprocal LUT	1025×32 bit	4 100 B
Palette RAM	256×24 bit	6 144 bit
Sky palette RAM	24×24 bit	576 bit

Table 1: Memory budget. Frame storage lives in SDRAM; everything else is on chip.

2.4 What is in the source tree

The codebase is split into four major directories. `sw/` contains user-space game code, the renderer, the GPU transport layer, the Linux kernel driver, and unit tests. `hw/` contains SystemVerilog RTL, the Platform Designer system file, asset-generation scripts, and Quartus project files. `virtual_hw/` is a Python package that implements the same descriptor protocol over a Unix socket, with a numpy-backed software rasterizer that we used as a reference model. `worlds/` contains a generator script and chunk files for a redstone demo lab. A full file inventory appears in Section 9.

3 Algorithms

3.1 Quadrilateral rasterization

Block faces are rectangles in world space. Under perspective projection they become convex quadrilaterals on screen. Instead of triangulating every face and shipping two triangles to the FPGA, we ship one quad and let hardware evaluate four edge functions, which simplifies rendering on the software side.

For each quad, software computes four signed edge coefficients A_i, B_i, C_i such that the edge function

$$E_i(x, y) = A_i x + B_i y + C_i$$

is non-negative on the inside of edge i. A pixel at (x, y) is inside the quad if and only if all four edge functions are non-negative. We store the coefficients in signed Q24.8 fixed point. Edge values are evaluated incrementally on the FPGA: stepping right adds A_i, stepping down adds B_i. That way we avoid per-pixel multiplication.

We also need to handle the case where adjacent quads share an edge. Without care, two faces along a shared edge will either both draw the boundary pixel (visible double-shading on translucent geometry, wasted Z writes on opaque) or both leave it blank (a one-pixel crack). We evaluate edges at pixel centers by folding the $+0.5(A_i + B_i)$ offset into C_i, then apply the standard top-left fill rule by subtracting one Q24.8 least-significant unit from exclusive edges before packing the descriptor.

3.2 Perspective-correct texturing

A linear interpolation of (u, v) across a quad's screen-space bounding box is only correct when the quad is axis-aligned with the camera. For anything else (the side of a block seen at an angle, the floor under the player's feet), linear UV interpolation makes the texture swim and shear. The fix is to interpolate u/w, v/w, and $1/w$ linearly across the screen, then recover u and v per pixel by multiplying by w.

Software solves three planes in Q16.16 for each textured quad:

$$(u/w)(x, y) = (u/w)_0 + (u/w)_x \cdot \Delta x + (u/w)_y \cdot \Delta y$$
$$(v/w)(x, y) = (v/w)_0 + (v/w)_x \cdot \Delta x + (v/w)_y \cdot \Delta y$$
$$(1/w)(x, y) = (1/w)_0 + (1/w)_x \cdot \Delta x + (1/w)_y \cdot \Delta y$$

The deltas are computed relative to the quad's bounding-box origin. Software fits these planes from a deterministic non-degenerate vertex triple in `sw/renderer.c`; the choice of triple matters because two of the four vertices might be collinear with the third after clipping.

On the FPGA, the math utils module (`hw/voxel_gpu/rtl/voxel_math_utils.sv`) handles the per-pixel recovery: normalize $1/w$ into a fixed range, index a 1025-entry reciprocal LUT, linearly interpolate between adjacent table entries, denormalize back to get w, then multiply w by u/w and v/w to recover the texture coordinates. The LUT is precomputed at build time by `hw/voxel_gpu/scripts/generate_recip_lut.py` and loaded into on-chip memory at synthesis.

3.3 Z-buffering

Depth is interpolated linearly in screen space the same way edge functions are, but in unsigned Q1.15. Per quad we store z_0 at the bounding-box origin plus dz/dx and dz/dy gradients. The renderer stores near fragments as smaller values and far fragments as larger values, so the hardware keeps a fragment when its z is less than the cached z.

The Z cache is 640×60×16 bit per band, ping-ponged in A/B banks that mirror the color caches. Each band pass starts in `ST_CACHE_INIT`, which writes a sentinel value across the configured resident Z-cache window (normally the full 60-row band) so that nothing has been written yet in the rows being refreshed. The hardware also uses the sentinel to short-circuit the destination color read. That is, if the Z cache still holds the sentinel for a pixel, the destination color is the configured clear color rather than the stale RGB565 inside the color cache.

3.4 Banded rendering

The screen is divided into eight bands of 60 rows. Software bins every descriptor into the band or bands it touches. A descriptor that lies inside one band gets appended directly to that band's buffer. A descriptor that crosses a band boundary gets cloned into each affected band, with the bounding box clipped to that band's row range and the depth/UV plane origins shifted so each clone is locally correct.

3.5 Lighting, fog, and alpha

Lighting is handled entirely on the software side and encoded into descriptors using a 2-bit light-bank field. Each block face is shaded by the brighter of its block-light and sky-light neighbors, quantized into four levels, and packed into bits [5:4] of the descriptor flags. On the hardware side, those two bits select one of four 64-color palette banks. The texture index goes through a small remap that

picks the appropriately-lit version of the same colors. This way we get four-level shading without sending different RGB values for every face, and the dimmest level approaches near-black without software having to upload four full palettes.

Fog blends the rasterized color toward a configurable fog palette index as a function of approximate radial camera distance. The hardware starts with the per-pixel recovered w value and scales it by a small screen-center radius correction using the uploaded `inv_proj_sq` term. Fog is opt-in per descriptor. The fog state – start distance, end distance, color index, enable – lives in two CSRs that software updates through `VOXEL_IOC_SET_FOG`.

Alpha is also descriptor-level: two bits select opaque, 75%, 50%, or 25% source alpha. There is also a 1-bit alpha key flag for textured faces with cutout transparency (leaves, doors, cross/torch/flat models, and sprite holes such as stars or the moon). Translucent geometry is the only thing the renderer sorts on the CPU. Opaque quads run first in any order, then translucent quads are run ordered back-to-front by camera distance.

3.6 Chunk streaming and meshing

The world is divided into chunks of 16×16 blocks and 32 blocks tall. Each chunk stores block IDs, sky and block light, water flow level, and per-cell redstone metadata in flat arrays. A chunk also holds a pointer to its most recent immutable `ChunkMesh`, which is the list of visible block faces.

When the player crosses a chunk boundary or when a block is changed, the affected chunks are marked dirty and queued to the mesh worker thread. The worker takes a snapshot of the chunk plus its four cardinal neighbor chunks, builds a new mesh outside the world lock, and atomically publishes the new mesh pointer only if the chunk generation count still matches. The renderer reads mesh pointers atomically and never touches the mutable arrays. This is what lets us run lighting updates and mesh rebuilds in parallel with rendering without locking either thread for long.

The terrain generator (`sw/world_gen.c`) is driven by deterministic random seed. It picks one of five biomes per chunk – plains, ocean, desert, hills, mountains – using a hash of chunk coordinates plus the world seed. Within each biome it lays out heightmaps, ores, trees, sugar cane, cactus, flowers, mushrooms, and lava pools. Sea level is also fixed at $y = 14$.

Figure 4: Randomly generated world terrain

4 Hardware Design

4.1 The voxel_gpu peripheral

The FPGA side of the project is an Avalon-MM slave called `voxel_gpu`, packaged as a Platform Designer IP component (`hw/voxel_gpu_hw.tcl`) and instantiated under `hw/soc_system.qsys`. It connects to the HPS through the lightweight HPS-to-FPGA bridge, exports a VGA conduit to top-level pins, and owns a board SDR SDRAM controller through a second conduit. The top-level wrapper (`hw/soc_system_top.sv`) wires pads and clocks and ties off unused board peripherals.

The peripheral's address space is 8 KB. The bottom 256 bytes are control and status registers; the top 4 KB are a FIFO window where software streams descriptor words. Section 6 has the full register map.

4.2 Main FSM

The descriptor consumer is a state machine with seven states. Figure 5 shows the layout.

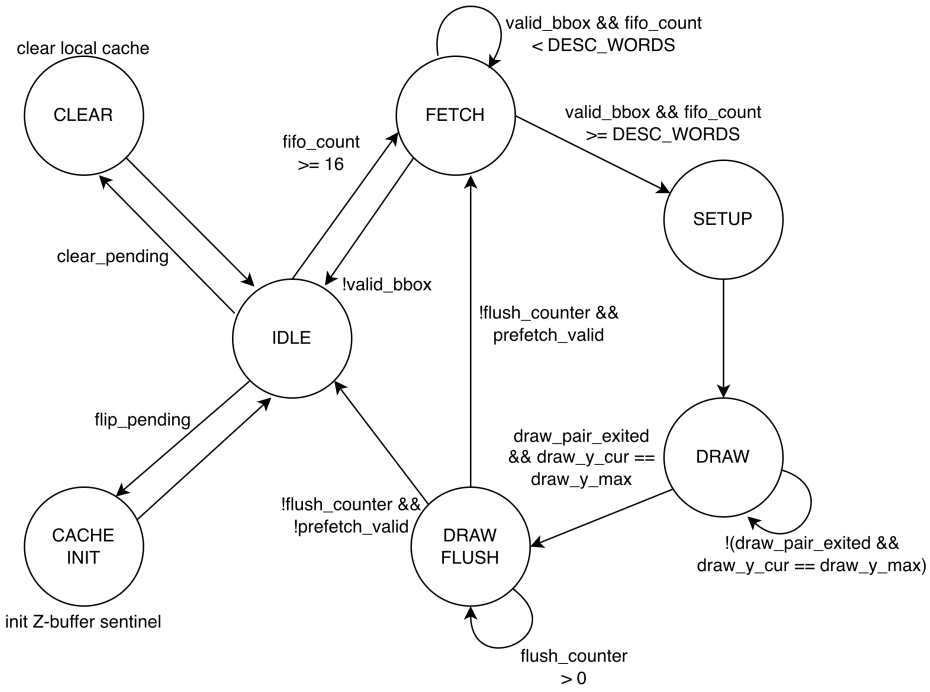

Figure 5: The voxel_gpu main FSM. `ST_IDLE` arbitrates among descriptor work, band maintenance, and clear/restart requests. `ST_FETCH` pulls 16 base words and (if the texture flag is set) 9 UV words from the FIFO into the descriptor scratch. `ST_SETUP` computes the starting edge, depth, and UV plane values. `ST_DRAW` steps two pixels at a time across the bounding box. `ST_DRAW_FLUSH` drains the pipeline once the geometry is done. `ST_CACHE_INIT` clears the resident Z band when a new band begins. `ST_CLEAR` handles `CLEAR_FRAME` bookkeeping such as invalidating local cache state, dropping queued FIFO work, and choosing the next SDRAM write target.

`ST_IDLE` watches for any of: a pending clear/restart request, a pending `BEGIN_BAND`, a pending `END_BAND` flush, prefetched descriptor words ready to consume, or new FIFO data. `ST_FETCH` pulls a 16-word base descriptor out of the FIFO into a scratch array of size `MAX_DESC_WORDS = 25`, and if the texture flag is set in the flag byte it pulls the additional 9 UV words. `ST_SETUP` reads the scratch array and produces the initial values that `ST_DRAW` will increment: edge values at the starting pixel pair, z at the starting pair, perspective UV planes at the starting pair. This work all happens in `voxel_math_utils.sv`, which are combinational helpers consumed by the FSM.

`ST_DRAW` is where almost all of the cycles go. Each clock advances two horizontally adjacent pixel lanes through the pipeline. At the end of a row, the FSM either advances down a row (adding the B coefficients to the edges and dz/dy to depth) or, if we've reached $y_{\max} + 1$, declares the descriptor done and moves to `ST_DRAW_FLUSH` to drain in-flight pixels before going back to `ST_IDLE`.

4.3 Two-lane pixel pipeline

We originally implemented a one lane pipeline, but at 50 MHz a single lane caps us at 50 M pixel evaluations per second. While 640×480 screen only has 307 200 pixels per frame, so even 60 FPS only needs 18.4 M useful evaluations per second, that ignores overdraw, descriptor switching costs, band initialization, and SDRAM flushing. We observe in-flight quad numbers in the low thousands, which suggests significant overdraw. Two lanes leaves enough breathing room for fast rasterization while keeping the addressing simple (adjacent pixels go to even-and-odd banks of the same RAM, no

arbitration between lanes). We note that the final two-lane design is likely the most efficient to use since our M10K utilization is already 100%.

Each lane carries a valid bit, an inside-quad bit (from the four edge tests), the interpolated z, the texture sample (or palette index for flat-color quads), the alpha level, the fog flag, the light-bank index, the destination address into the band cache, and the destination color read from that cache.

4.4 Band caches

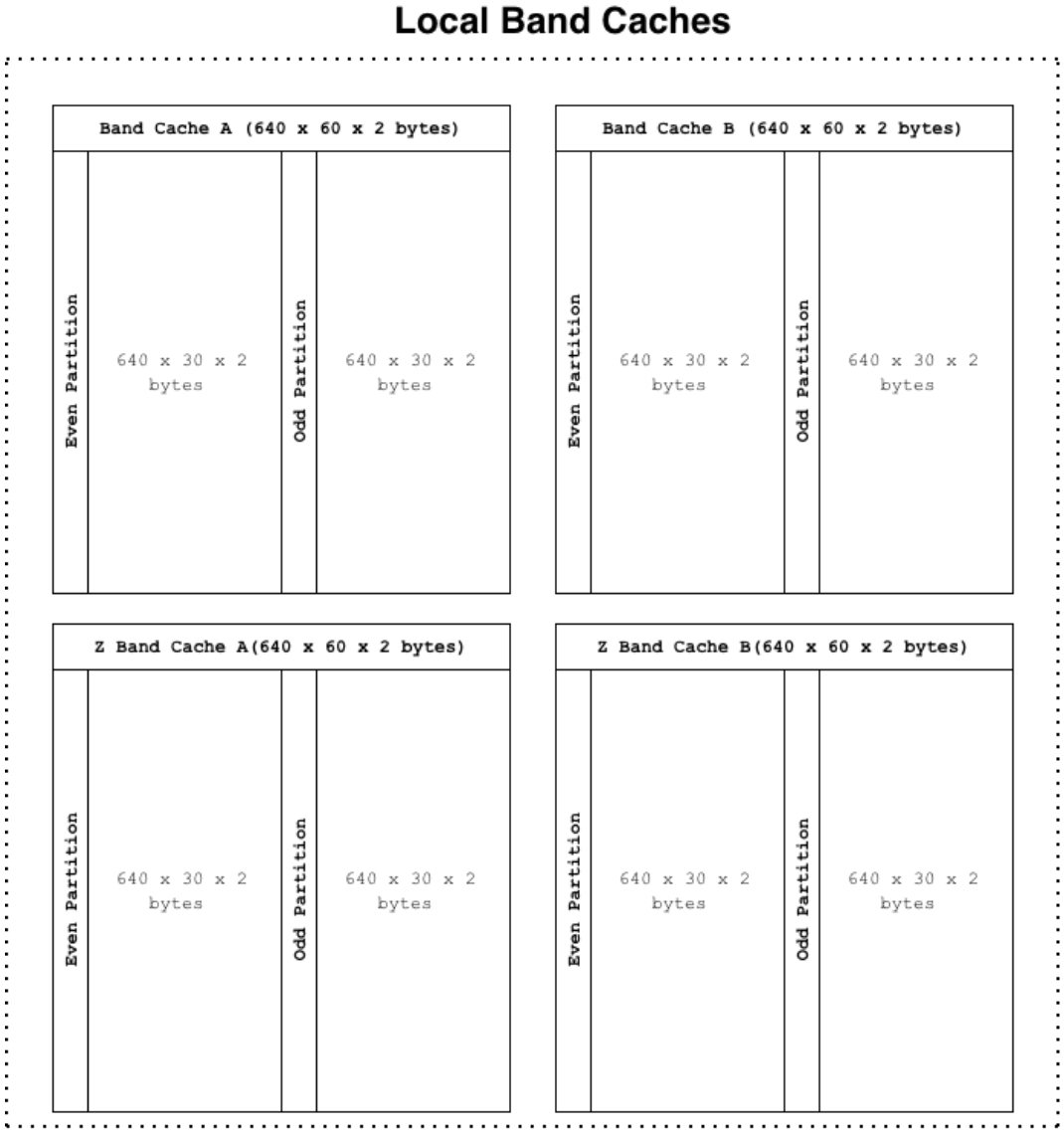

Figure 6: Local band caches. Two color caches (A/B) and two Z caches (A/B) each hold one 640×60 working set. The draw path writes to the active caches; the background flush controller reads from the inactive ones and pushes RGB565 words to SDRAM. Each cache is implemented as a banked simple dual-port RAM whose even/odd sides are selected by linear address bit 0, so the two-pixel draw step gets two simultaneous accesses.

The band caches are the bulk of the on-chip memory architecture. WE have four of them, each 640×60×16 bit, totalling 307 200 bytes. They are instantiated through `voxel_banked_sdp_ram.sv`, a thin wrapper that splits the address space into even and odd banks of `voxel_sdp_ram.sv`. The wrapper is what lets the two-lane draw step run without arbitration. Lane 0 always goes to one bank, lane 1 always goes to the other, and the hardware rounds the descriptor's starting x down to an even pixel pair and masks off any lane that falls outside the true bounding box.

We use A/B cache ping-ponging to simultaneously flush the previous band while rendering the next. When software calls `BEGIN_BAND`, the FSM swaps which set of caches is "active" for drawing and schedules `ST_CACHE_INIT` on the new active Z cache. When software later calls `END_BAND`, the background flush controller writes that completed color cache to SDRAM while the other cache set is available for the next band. The two operations share neither read nor write ports, so they can overlap.

4.5 Background flush

Flushing a 60-row band to SDRAM is 38 400 RGB565 words. The flush controller runs as a separate sub-FSM that:

1. Loads the active band base address from `EXTMEM_BACK_BASE` plus the band offset.
2. Walks rows y_0 through y_1 (set via `BAND_WINDOW`; default is the whole band).
3. For each row, walks 640 pixels, reading from the inactive color cache and writing 16-bit words to the SDRAM write FIFO.
4. Drains the write FIFO before signalling completion.

We implement one key render optimization in hardware here. When software enables `VOXEL_EXTMEM_CTRL_SKY_GR` and a band's color cache was never written by a draw, the flush controller can substitute a generated sky-gradient palette read instead of reading the cache. This means sky-only bands can be produced by the flush path instead of by reading color-cache pixels for a redundant sky clear. The transport cooperates by recognizing redundant generated-sky clear descriptors when the hardware optimization is enabled. Without that cooperation, the hardware can still patch those sky-clear pixels, but the extra descriptors will cost bandwidth.

4.6 VGA scanout

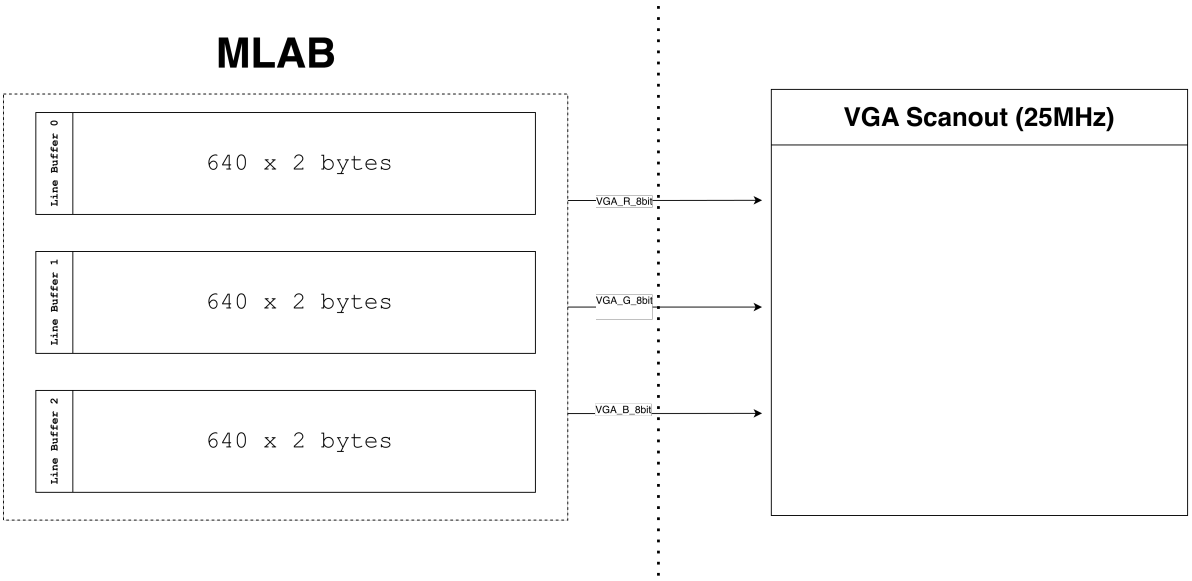

Figure 7: VGA scanout path. The 640×480 timing comes from the `voxel_vga_counters.sv` module, which divides the 50 MHz logic clock by 2 to get the 25 MHz VGA pixel clock (`VGA_CLK = ~hcount[0]`). Three line buffers triple-buffer the scanout: one is being displayed, one is being filled from SDRAM, one is on deck. Each buffer holds 640 RGB565 words. SDRAM reads pre-fetch the row that will be displayed several lines ahead.

The scanout side is independent of the draw side. The VGA counter module produces `hcount` and `vcount` from the 50 MHz clock with the standard 640×480 timing parameters. A separate state machine watches `vcount`, decides which physical SDRAM row needs to be in a line buffer N rows ahead of the current scan line, issues SDRAM reads through the read FIFO, fills the line buffer, and tags it with its row number. When the VGA pixel counter for a given line reaches the active region, the correct line buffer is selected, RGB565 is expanded to RGB888 for the VGA DAC, and the pixels stream out.

The line-buffer count was a key design constraint. Two would have been enough if SDRAM read latency were always less than one full VGA line (about 32 μs), but SDRAM contention with the band flush controller can push us close to that edge. Three line buffers give the scanout state machine an extra prefetched row of headroom and reduce the chance of dropping pixels during read/write contention.

4.7 Texture ROM and palette

The texture atlas holds 128 tiles of 16×16 palette indices, packed into one 32 768-byte atlas. Because the draw step runs two lanes at the same time, the atlas is instantiated twice as single-port ROMs initialized from the same MIF file (`voxel_texture_rom.sv`). This doubles the M10K usage for textures but lets both lanes read the same or different texels in the same cycle.

The palette is a 256-entry RGB888 RAM written through `PALETTE_ADDR`/`PALETTE_DATA`. The sky palette is a separate 24-entry RAM that holds the day/night gradient and is written through the `SKY_PALETTE_*` registers. The sky palette is used both by the sky-gradient flush optimization and by regular flat-color draws whose palette index is in the generated sky-gradient range.

4.8 Performance counters

`voxel_perf_counters.sv` keeps ten 32-bit counters that increment every cycle a particular condition is true. They are reset on each frame flip. The categories are: `draw_active` (a lane committed a pixel), `draw_idle` (in a draw state but no commit, usually descriptor starvation or pipeline bubbles), `flush_active` (the flush controller pushed a word), `flush_stall` (flush blocked), `init` (cache init cycles), `load` (legacy/reserved, currently zero), and four flush-wait breakdown counters: `flush_wait_load` (waiting to launch the SDRAM write stream), `flush_wait_fifo` (write FIFO full), `flush_wait_data` (waiting on the cache to deliver a word), `flush_wait_drain` (final drain).

Software reads these through `VOXEL_IOC_GET_PERF` or `VOXEL_IOC_GET_PERF2`. These counters were used throughout renderer tuning. In an early version of the band path, a high `flush_stall` count revealed an SDRAM controller misconfiguration that was not otherwise obvious from visual inspection.

5 Software Design

5.1 Linux kernel driver

The driver (`sw/voxel_gpu.c`) is a Linux platform device plus a misc device that exposes `/dev/voxel_gpu`. On probe it allocates the write bounce buffer, registers the misc device, resolves and reserves the FPGA peripheral's memory resource, and maps it with `of_iomap`.

`write()`: user space writes a buffer of descriptor bytes to the device file, and the driver streams those bytes into the FIFO window at `0x1000..0x1FFF` as 32-bit MMIO writes. We use an 8 KB bounce buffer because `copy_from_user` can fault, and we never want to fault inside an MMIO loop. The write call `iowrite32_rep`s the contents of the bounce buffer to the FIFO base address. The FIFO is one window, so we write the same address every word and the hardware enqueues each into its descriptor FIFO.

The other paths are ioctls. `BEGIN_BAND` and `END_BAND` bracket each band, `FLIP`, `FLIP_ASYNC`, and `WAIT_FLIP` bracket each frame, and the rest upload palette state, fog state, the SDRAM configuration, the sky palette, and read back status or performance counters. The synchronizing ioctls use `STATUS` polling around the hardware pulses. `CLEAR_FRAME` pulses `CLR` and then waits for the engine to return idle. `FLIP`, `BEGIN_BAND`, and `END_BAND` first wait for the required FIFO/engine condition before pulsing the relevant `CONTROL` or `BAND_CTRL` bit. Palette, fog, sky-palette, and SDRAM-configuration ioctls are direct CSR writes under the driver lock. This keeps the timing rules easy to reason about: when `BEGIN_BAND` returns, the band-index and window CSR writes have been ordered ahead of any later FIFO writes.

The driver deliberately does not parse descriptors. That keeps the kernel boundary narrow and means the shared header `sw/voxel_gpu.h` is the single source of truth for descriptor layout.

5.2 Renderer

`sw/renderer.c` is the largest graphics-specific software file at about 4400 lines and does most of the work of turning a frame's worth of world state into packed descriptors. The frame begins with a camera setup: yaw and pitch get baked into a 3×3 rotation, the near plane is fixed at 0.1, and we compute the inverse of the projection scale so that perspective division can be done once per vertex rather than per pixel.

After that, the renderer walks visible chunks. Chunk-level frustum culling skips entire chunks whose axis-aligned bounding boxes (AABB) don't intersect the camera frustum. For each surviving chunk, we read its `mesh` pointer atomically. The mesh is a list of visible block faces in three

pools: opaque cubes, opaque non-cubes (slabs, stairs, redstone devices, plants), and translucent faces. Farther chunks can be greedily meshed so runs of coplanar, same-material faces become one larger face. The renderer normally emits those merged faces as repeat-UV quads using `QUAD_TEX_REPEAT_UV`; setting `VOXEL_MERGE_FAR_QUADS=0` expands them back to unit quads for visual A/B testing.

Each face is transformed to camera space, clipped against the near plane, projected to screen coordinates, and clipped against the viewport. The viewport clip can turn a quad into a triangle or a pentagon depending on which edges crossed which screen sides; the renderer handles up to 12-vertex output polygons and triangulates as quads where possible. Screen-space vertices are then snapped to the Q24.8 grid the hardware expects, the pixel-center offset of 0.5 is folded into the edge C values, and the top-left fill rule is applied.

For textured geometry, we further split quads by hardware band boundaries. For Z-tested textured world geometry, we can also split by a configurable vertical texture-slice height. The reason for the extra texture slice is numerical: as y grows, the accumulated error in Q16.16 UV interpolation grows too, and forcing a re-solve every few dozen rows keeps the worst-case error bounded. This is controlled by the environment variable `VOXEL_TEXTURE_SLICE_PX`; the current default is 30 pixels, with `0` disabling the extra slicing. For distant block faces, `VOXEL_TEXTURE_LOD` is enabled by default and swaps in the mip tile ids declared in `sw/texture_tiles.def`.

The same frame then adds falling block entities, item drops, furnace animation, the block-break overlay when active, hand and hotbar UI, inventory or furnace panels if open, the chat overlay, pause menu, crosshair, and any debug HUD elements. All of these go through the same quad descriptor packing path.

5.3 GPU transport

`sw/gpu_transport.c` sits between the renderer and the kernel driver. Its job is to bin descriptors into per-band buffers, manage palette and fog state diffs, run band reuse caching, and (optionally) drive the submit and flip from a background thread.

There are three backends. The hardware backend talks to `/dev/voxel_gpu`. The socket backend writes the same descriptor protocol over a Unix socket to the Python virtual GPU. The tee backend writes the same command stream to both hardware and the socket backend, which lets us inspect hardware output against the virtual GPU using one run. The backend is selected at startup by the `VOXEL_GPU_BACKEND` environment variable.

For hardware mode, `VOXEL_PIPELINE_FRAMES` defaults on. The transport double-buffers the per-band bins so a submit worker can own one set while the renderer fills the other for the next frame; socket and tee mode keep submission synchronous.

The band binning code is the most subtle part. For each emitted descriptor, the transport computes which bands its bounding box touches. If only one band, it appends to that band's buffer. If two or more, it clones once per band, recomputes the clipped bounding box, evaluates the depth and UV planes at the new $y_{\min}$, and updates the depth/UV origin words in the cloned descriptor. This makes the fast path roughly a memcpy while the slow path is a quad recompute.

Each band currently prepends a hidden one-pixel clear-color primer descriptor. This began as a bring-up guard for first-descriptor pipeline hazards after a fresh `BEGIN_BAND` and was retained as a low-cost synchronization safeguard after the underlying setup-path issue was fixed.

The transport also caches per-band content using FNV-1a of the descriptor bytes plus a render epoch and a sky epoch. If the hash, render epoch, and sky epoch all match the previous frame for a given band, we can skip both the redraw and the flush for that band – the SDRAM frame still holds the right pixels from last time. In stationary or mostly static views, this removes repeated hardware work for bands whose descriptor stream and relevant sky state are identical to the previous copy of

the same SDRAM target buffer.

When only part of a band changed (e.g. a single block was placed), we narrow the flush row window via `BAND_WINDOW` so we only push the affected rows to SDRAM.

5.4 World, lighting, fluids, redstone

`sw/world.c` contains all simulation logic. Chunks live in a hashmap keyed by chunk coordinates. The world has a moving window of loaded chunks around the player; the window radius is the render distance plus one (so that meshing has a border of neighbors to consult). Default render distance is 3; the maximum is 9.

The mesh worker (`sw/mesh_worker.c`) is a bounded-queue background thread. It takes dirty-chunk jobs, snapshots the chunk and its four cardinal neighbors, builds a face list outside the world lock, and publishes it atomically. Publishing checks the chunk's generation counter. If a player edit happened during the build, the generation has advanced and the worker drops the result. Most of the time meshing keeps up. If it falls behind, the renderer can still draw stale meshes for a frame or two without correctness problems because mesh pointers are atomic.

The generation worker (`sw/gen_worker.c`) does the analogous work for terrain. When the player walks into a new chunk, the worker generates terrain, ores, decoration, and initial sky lighting off the main thread.

Lighting has two passes. Sky light is column-based: each column remains at level 15 until a light-blocking cell, then stays dark below it. Block light is a decaying BFS from emitters, using the per-block emission levels in the block registry (for example, torches at 14 and lamps/lava at 15). Block edits update lighting incrementally: sky light recomputes the affected column, and block light uses remove/add BFS queues seeded from the edited cell and its neighbors. Full lighting rebuilds are reserved for stream-integration cases, such as initial loading or deferred consistency after chunk generation.

Fluids tick on the environment timer, which defaults to one step every 750 ms. Water and lava distinguish source cells from flow cells. Sources carry level 0 and flow cells carry levels 1–7, where larger levels are thinner. Each fluid tick snapshots the currently active source/flow cells, spreads one ring from that snapshot, increments the parent flow level for lateral spread, and resets downward falling flow to level 1. Unsupported flow blocks evaporate. Lava/water contact is resolved around the lava cell: a lava source touching a water source becomes obsidian, flowing lava resting on water becomes stone, and lava touching flowing water becomes cobblestone. Sand and gravel above empty cells become smooth falling entities; the entity falls under gravity until it lands, then snaps back to the grid and updates the chunk.

Redstone is the most complex simulation. On each redstone tick, the world advances button pulses and repeater delay state. When anything is dirty, it builds a temporary settle cache from loaded chunks that contain redstone components, runs bounded settle passes over wires and components, and updates dependent blocks such as lamps and doors. Pressure plates are checked separately from player and item-entity trigger boxes, then feed back into the same redstone-settle path. Repeaters with 1–4 tick delays maintain their own delay state. The settle loop is bounded because, in principle, a redstone circuit could oscillate. Our implementation caps each settle attempt at 16 passes.

The redstone lab world (`worlds/redstone_lab/`) contains a hand-built counter circuit and 7 segment display decoder that exercises wires, torches, repeaters, comparators, and lamps We used it as an illustrative demo

5.5 Virtual hardware

The Python virtual GPU lives in `virtual_hw/` and was one of the most useful pieces of project infrastructure. It implements the same descriptor protocol over a Unix socket. The server (`server.py`) accepts a Unix-domain socket connection from the GPU transport, deserializes the protocol (`protocol.py`), and feeds descriptors into the reference rasterizer (`raster.py`). The rasterizer is a numpy implementation of the same edge-test plus perspective-correct UV plus Z-test plus palette plus fog plus alpha pipeline that lives in the FPGA. With `numba` installed it JIT-compiles the hot loop and allows for reasonable FPS in a virtual machine.

The virtual GPU was useful for three things. First, it let us iterate on renderer code on a laptop without rebuilding the FPGA (saving us a bitstream rebuild of 25–30 minutes).

Second, when a visual bug appeared on hardware, we could replay the same descriptor stream through the virtual GPU and see whether the bug was in the descriptors (in which case both backends would produce the same wrong output) or in the FPGA (in which case the virtual GPU would be correct and the FPGA wrong). Third, the tee backend drives both in the same run, which makes direct hardware/virtual comparison practical. This substantially reduced the time required to isolate descriptor-generation bugs from RTL bugs.

The Python tests (`virtual_hw/tests/test_raster.py`) double as regression tests for the reference rasterizer and descriptor protocol assumptions: they cover texture clamp and repeat-UV, fog blending, bounding-box clamping, the opaque flat fast path, alpha-keyed transparent texels, and the generated sky palette path.

5.6 Game loop and input

The main loop in `sw/game.c` runs at a target of 30 FPS (adjustable via `VOXEL_TARGET_FPS`). Each frame: read input, update physics, tick world simulation, rebuild any meshes that finished on the worker, prepare the camera, render, and then either submit/flip synchronously or queue that work to the hardware pipeline worker. Input comes from evdev (`sw/input.c`): one fd for the keyboard, one for the mouse, with a fallback to absolute-pointer touchscreen events if the mouse can't be opened. A pointer grab pins the cursor so mouse-look works.

Survival, creative, and spectator are three different physics modes in `sw/player_physics.c`. Survival has gravity, collision with the world, swimming in water with current drag, jumping, sprinting, and falling-damage. Creative adds double-tap flight and immunity, and spectator removes collision entirely.

6 Hardware/Software Interface

The shared header `sw/voxel_gpu.h` is our single source of truth. It defines every register offset, status bit, ioctl number, packed struct, and descriptor flag. The driver and user-space transport include this header directly while the Python virtual GPU and SystemVerilog rasterizer mirror the same ABI constants and are held consistent by the protocol and rasterizer tests. We place `_Static_assert`s at the bottom of the header pin the packed struct sizes so a refactor that breaks the ABI fails to compile rather than producing wrong pixels.

6.1 Register map

Offset	Register	Access	Contents
0x0000	CONTROL	R/W	[0]=EN [1]=FLP [2]=IEN [3]=CLR
0x0004	STATUS	R	[0]=BSY [1]=FFL [2]=FEM [3]=VSY [19:4]=FIFO_COUNT
0x0008	FRAME_COUNT	R	32-bit free-running frame counter
0x000C	PALETTE_ADDR	W	8-bit palette index
0x0010	PALETTE_DATA	W	[23:16]=R [15:8]=G [7:0]=B
0x0014	FOG_RANGE	W	[15:0]=start_dist [31:16]=end_dist, Q8.8
0x0018	FOG_CTRL	W	[31:16]=inv_proj_sq [8]=EN [7:0]=fog palette idx
0x001C	EXTMEM_CTRL	R/W	SDRAM display path enable bits
0x0020	EXTMEM_FRONT_BASE	R/W	Byte address of SDRAM frame 0
0x0024	EXTMEM_BACK_BASE	R/W	Byte address of SDRAM frame 1
0x0028	EXTMEM_STRIDE	R/W	Bytes per scanline; default 1280
0x002C	EXTMEM_TILE	R/W	Reserved tile cache config
0x0030	EXTMEM_STAT	R	SDRAM copy/scanout status
0x0034	BAND_INDEX	R/W	Active 60-line band, 0..7
0x0038	BAND_CTRL	W	[0]=BEGIN [1]=FLUSH pulses
0x003C	BAND_WINDOW	R/W	[5:0]=y0 [13:8]=y1 flush row window
0x0040..0x0064	PERF_*	R	Performance counters, 50 MHz cycle counts
0x0068	SKY_PALETTE_ADDR	W	Sky gradient entry, 0..23
0x006C	SKY_PALETTE_DATA	W	[23:16]=R [15:8]=G [7:0]=B
0x1000..0x1FFF	FIFO_WINDOW	W	Streaming write into descriptor FIFO

Table 2: voxel_gpu Avalon-MM register map.

6.2 IOCTLs

No.	Name	Payload	Purpose
1	CLEAR_FRAME	none	Pulse the clear/restart bit, wait until engine idle.
2	FLIP	none	Drain FIFO, wait idle, pulse flip, wait for vsync.
3	SET_PALETTE	voxel_palette_entry	Write one RGB888 palette entry.
4	GET_STATUS	voxel_status	Read raw + decoded status.
5	GET_FRAME_COUNT	u32	Read frame counter.
6	SET_FOG	voxel_fog_state	Set fog range, color, enable, inverse projection-depth squared.
7	SET_EXTMEM	voxel_extmem_state	Configure SDRAM display path.
8	GET_EXTMEM	voxel_extmem_state	Read SDRAM display config + DMA status.
9	BEGIN_BAND	voxel_band_state	Set band index + flush window, pulse band begin.
10	END_BAND	none	Wait idle, pulse band flush.
11	FLIP_ASYNC	none	Drain + flip without waiting for vsync.

No.	Name	Payload	Purpose
12	WAIT_FLIP	none	Wait for vsync after async flip.
13	GET_PERF	voxel_perf_counters	Base perf counters.
14	GET_PERF2	voxel_perf_counters_v2	Base + flush-wait breakdown.
15	SET_SKY_PALETTE	voxel_sky_palette_entry	Write one sky gradient entry.

Table 3: IOCTL command set on /dev/voxel_gpu.

6.3 Descriptor layout

Every drawing command is a packed quadrilateral descriptor. The base descriptor is 64 bytes (16 32-bit words) and is always present. A textured descriptor adds a 36-byte UV extension (9 words), for 100 bytes and 25 words total. `MAX_DESC_WORDS` in the RTL is 25, matching `BASE_QUAD_WORDS = 16` plus `UV_QUAD_WORDS = 9`.

Word	Fields	Meaning
0	[15:0]=x_min,[31:16]=y_min	Bounding box origin, signed 16-bit
1	[15:0]=x_max,[31:16]=y_max	Bounding box end, inclusive
2–4	edge0.{A,B,C}	Edge 0, signed Q24.8
5–7	edge1.{A,B,C}	Edge 1, signed Q24.8
8–10	edge2.{A,B,C}	Edge 2, signed Q24.8
11–13	edge3.{A,B,C}	Edge 3, signed Q24.8
14	[15:0]=z0,[31:16]=dz/dx	Depth origin and x gradient, Q1.15
15	[15:0]=dz/dy,[23:16]=tex_or_color,[31:24]=flags	Depth y gradient, palette/tile byte, flag byte
16–18	u/w plane	Q16.16, present if QUAD_FLAG_TEX set
19–21	v/w plane	Q16.16, present if QUAD_FLAG_TEX set
22–24	1/w plane	Q16.16, present if QUAD_FLAG_TEX set

Table 4: Packed descriptor word layout. Base descriptor is words 0–15; UV extension is words 16–24.

The flag byte in word 15 encodes: `bit 0` = textured (UV extension follows); `bit 1` = Z test and write enable; `bit 2` = alpha-key (skip palette index 0 in textured samples); `bit 3` = fog enable; `bits [5:4]` = light bank 0–3; `bits [7:6]` = alpha level (opaque, 75%, 50%, 25%). The `tex_or_color` byte is a tile id (with bit 7 selecting repeat-UV) when the texture flag is set, or a palette index otherwise.

All numerical formats: edge coefficients are signed Q24.8, depth is unsigned Q1.15 with z_0 and dz/dx as 16-bit fields stored together (depth gradient sign comes from the field reinterpretation in the FSM), UV planes are signed Q16.16, fog range is Q8.8, palette colors are RGB888, on-screen pixels and SDRAM frames are RGB565.

7 Implementation Results

7.1 Resource utilization

Our resource utilization is dominated by on-chip storage. The design uses every single M10K block on the Cyclone V part (397/397). Most M10K pressure comes from the band caches (four 76 800-byte arrays), the two texture ROM copies, the descriptor FIFO, and the SDRAM FIFOs. The scanout line buffers and reciprocal LUT add additional on-chip storage/logic pressure, but they are not the

dominant M10K users. The fit also uses all 3 207 LABs, while the post-fit ALM requirement remains at 89% of the available ALMs. DSP usage is moderate at 44 out of 87 blocks.

Metric	Reported value	Source
Fitter status	Successful, 2026-05-12 17:20:38	`soc_system.fit.summary`
Flow status	Successful, 2026-05-12 17:22:27	`soc_system.flow.rpt`
Logic utilization in ALMs	28 687 / 32 070 (89%)	`fit.summary`
Total LABs partially or completely used	3 207 / 3 207 (100%)	`fit.rpt`
Combinational ALUT usage for logic	31 792	`fit.rpt`
Total registers	46 525	`fit.summary`
Pins	362 / 457 (79%)	`fit.summary`
Block memory bits	3 079 296 / 4 065 280 (76%)	`fit.summary`
M10K blocks	397 / 397 (100%)	`fit.rpt`
DSP blocks	44 / 87 (51%)	`fit.summary`
PLLs and DLLs	2 / 6 PLLs, 1 / 4 DLLs	`fit.summary`

Table 5: Post-fit resource utilization from the Quartus fitter reports.

7.2 Timing

The timing evidence separates functional fit success from full timing closure. The flow and fitter are successful, and the TimeQuest Fmax summary reports same-clock Fmax values above the intended 50 MHz system clock and 100 MHz local SDRAM clock. However, TimeQuest explicitly warns that Fmax summaries are not sign-off analysis for all generated-clock and cross-clock relationships. The `sta.summary` file reports negative setup, hold, and recovery slack in the local `sdram_ctrl` PLL-derived clock domain, so we describe the final build as fitting successfully but not fully timing-clean in every reported domain.

Timing item	Reported value	Interpretation
System PLL same-clock Fmax	62.07–63.17 MHz	Above the 50 MHz input clock target in the Fmax summary.
Voxel SDRAM PLL same-clock Fmax	122.34–126.18 MHz	Above the 100 MHz local SDRAM target in the Fmax summary.
Slow 1100 mV 85°C setup, voxel SDRAM PLL domain	Slack -2.269 ns, TNS -114.181 ns	Negative sign-off setup slack in the local SDRAM-controller clock domain.
Slow 1100 mV 85°C hold, voxel SDRAM PLL domain	Slack -0.590 ns, TNS -0.590 ns	Negative hold slack in the same local domain.
Slow 1100 mV 85°C recovery, voxel SDRAM PLL domain	Slack -2.353 ns, TNS -152.014 ns	Negative recovery slack in the same local domain.
Slow 1100 mV 85°C setup, HPS DDR write clock	Slack 1.730 ns, TNS 0.000 ns	Positive setup margin for the HPS DDR write-clock entry in the summary.
Slow 1100 mV 85°C setup, system PLL domain	Slack 4.169 ns, TNS 0.000 ns	Positive setup margin for the main system PLL entry in the summary.

Table 6: Selected TimeQuest timing results from `soc_system.sta.summary` and `soc_system.sta.rpt`.

The same negative local SDRAM-domain pattern also appears at the slow 1100 mV 0°C corner:

setup slack is -2.164 ns, hold slack is -0.650 ns, and recovery slack is -2.192 ns. Fast-corner setup and recovery entries for that domain are positive, but fast-corner hold slack remains negative.

7.3 Verification

There are four layers of tests.

ABI and unit tests live in `sw/tests/` and build on the host with `make -C sw tests`. The pure software unit tests run directly on a host, while the FPGA smoke tests require the board-facing device files or physical address windows. `sw/voxel_gpu.h` pins descriptor and ioctl payload sizes with `_Static_asserts`, while `voxel_test.c` is a hardware flat-color and Z-buffer smoke test for `/dev/voxel_gpu`. `command_parser_test.c`, `inventory_test.c`, and `player_physics_test.c` cover the corresponding subsystems. `world_chunk_test.c` is 2500 lines and exercises chunk persistence, lighting, fluid flow, falling blocks, and redstone state transitions.

Renderer smoke tests (`renderer_quad_test.c`, `renderer_scene_test.c`, `renderer_edge_test.c`, `renderer_fog_test.c`, `renderer_static_test.c`) build the renderer and submit fixed or procedural scenes through the selected GPU backend. They print expected visual outcomes for screen-space quads, shared-edge coverage, static block scenes, and radial fog rather than asserting descriptor words directly.

7.4 Performance

At a render distance of 3 with a 30 FPS target, the system remained interactively playable on the DE1-SoC during generated-world traversal and redstone-lab testing. We did not collect a complete frame-time trace, so we report this as functional validation rather than a quantitative FPS benchmark. The band-reuse cache means that when a band's descriptor bytes, render epoch, and sky epoch are unchanged, the transport can skip that band's redraw and flush. In stationary scenes this removes most of the repeated hardware work for unchanged world and sky-gradient-only bands. When the camera moves, the debug timing output usually points at CPU-side renderer work rather than the FPGA datapath: descriptor generation, particularly for textured world quads with band cloning, is the expensive phase we saw most often.

When SDRAM contention is high, the flush-wait counters are mostly in `flush_wait_data` (waiting on the band cache to deliver a word). The `flush_wait_fifo` counter rarely peaks unless we're also reading SDRAM aggressively for scanout, which suggests our SDRAM controller is mostly limited by read/write switching overhead at this clock.

8 Who Did What, Lessons Learned, Advice

8.1 Division of work

Wesley led the FPGA datapath and the hardware-facing software interface. That included the `voxel_gpu` peripheral itself (`hw/voxel_gpu/rtl/voxel_gpu.sv` and its helper modules), the shared `sw/voxel_gpu.h` ABI, the Linux kernel driver, the redstone simulation, several software/hardware pipeline debugging passes, and the Python virtual GPU used as a reference model.

Mihir focused on gameplay systems and validation. His main contributions were exercising the world systems through gameplay and the redstone lab, Quartus project configuration, board pin assignments, and helping validate the final game flow once the renderer and game loop were running together.

Josh focused on the renderer stack and board-level integration. He worked on the renderer, the GPU transport with band binning and band reuse, hardware bring-up, SDRAM controller integration under `hw/sdram_local_test/`, and debugging the path from the HPS-side renderer through SDRAM-backed VGA output.

8.2 Lessons learned

Memory architecture comes first. Our biggest pivot was abandoning the on-chip framebuffer in favor of SDRAM-backed frames with band caches. This had ripple effects everywhere: software learned to bin descriptors into bands, the hardware grew `BEGIN_BAND`/`END_BAND` and a background flush controller, and the scanout side had to be decoupled from drawing with line buffers. An earlier memory-capacity analysis would have led us to the SDRAM-backed banded architecture sooner.

Make the ABI explicit and testable. `sw/voxel_gpu.h` ended up the most important file in the project. The `_Static_assert` sizes at the bottom of the file caught a half-dozen accidental layout changes that would have produced silently wrong pixels.

Build a reference model. The Python virtual GPU was one of the most useful pieces of project infrastructure. A bitstream rebuild takes 25–30 minutes; a Python restart takes seconds. Whenever the FPGA showed a visual bug, we re-played the same descriptor stream through the virtual GPU and immediately knew whether the bug was in our descriptors or in the RTL. The tee backend made this even easier by running both at once.

Pipeline carefully but keep counters. The performance counters were not in the proposal. We added them halfway through the project after a session where we could not tell whether the bottleneck was descriptor generation, FIFO starvation, cache initialization, or SDRAM flush. The counters made bottleneck diagnosis directly observable through a single ioctl. If we had added them earlier we would have saved a lot of guessing.

8.3 Advice for future teams

If you are doing a graphics-oriented project on this board, decide where your framebuffer lives in the first week. Our project would have benefited from committing to a single path earlier.

Build in milestones. For example, start with a single quadrilateral, then a moving single quadrilateral, then two, then one behind the other. Debugging is much simpler when there are fewer objects to see.

Use AI carefully. AI tools are capable of building well-organized and extensible code, but understanding that code is essential to implementing system design optimally, and especially for debugging when the time arises. We found success using AI tools to implement much of our code because we built a strong case for all of our design descisions before doing so. Whether or not you use AI tools to enhance your workflow, having a deep understanding of all of the logic and hierarchy of your code is essential to creating a successful project.

9 File Listings

9.1 Source inventory

The inventory below covers the buildable implementation sources: the authored C, SystemVerilog/Verilog, Tcl, Python, test, and build/configuration files that define the project. Non-build artifacts such as prose drafts, manual diagram sources, generated Quartus outputs, asset MIFs, world chunk files, and LaTeX intermediates are excluded.

HPS user-space game and renderer (`sw/`)

File	Lines	Role
`block_types.c`	818	Block registry, texture ids, helper predicates
`block_types.h`	185	Block ids, render models, texture tile enum
`chat.c`	858	Chat state, text rendering, history/input
`chat.h`	72	Chat API
`command_parser.c`	1392	Slash command parser and completion
`command_parser.h`	119	Command AST and status types
`env_util.h`	179	Environment variable helpers
`game.c`	5653	Main loop, UI, interaction, orchestration
`game_home.c`	775	Home/world selection menu
`game_home.h`	24	Home menu API
`game_items.c`	339	Item entities, drops, pickup/toss
`game_items.h`	59	Item entity API
`gen_worker.c`	299	Background chunk generation worker
`gen_worker.h`	36	Generation worker API
`gpu_transport.c`	2080	Backend selection, band binning, caching
`gpu_transport.h`	36	GPU transport API
`input.c`	888	evdev keyboard/pointer/text input
`input.h`	114	Input state and API
`inventory.c`	1130	Item stacks, recipes, crafting, furnace
`inventory.h`	132	Inventory definitions and API
`mesh_worker.c`	375	Background mesh rebuild worker
`mesh_worker.h`	41	Mesh worker API
`pause_menu.c`	245	Pause menu and runtime settings
`pause_menu.h`	52	Pause menu API
`player_physics.c`	357	Movement, collision, water, flight
`player_physics.h`	79	Physics constants and API
`renderer.c`	4414	Projection, clipping, descriptor packing
`renderer.h`	110	Renderer API and structs
`texture_tiles.def`	107	Texture tile id definitions
`thread_affinity.c`	104	Thread affinity helpers
`thread_affinity.h`	20	Thread affinity API
`virtual_gpu_protocol.h`	34	Socket protocol shared with virtual hw
`voxel_gpu.c`	753	Linux kernel platform/misc driver
`voxel_gpu.h`	285	Shared FPGA ABI header
`world.c`	6181	Chunk storage, lighting, fluids, redstone
`world.h`	387	World/chunk/mesh structures
`world_gen.c`	800	Procedural terrain generation
`world_gen.h`	14	World generation API
`Makefile`	127	Software build/test/module targets

Software tests (`sw/tests/`)

File	Lines	Role
command_parser_test.c	314	Command parser unit tests
fpga_band_offset_test.c	178	Visual band-offset diagnostic
fpga_sdram_test.c	135	FPGA SDRAM smoke test
inventory_test.c	301	Inventory/crafting tests
player_physics_test.c	86	Player physics tests
renderer_edge_test.c	89	Shared-edge renderer smoke test
renderer_fog_test.c	103	Fog and HUD renderer smoke test
renderer_quad_test.c	64	Screen-space quad smoke test
renderer_scene_test.c	68	Minimal block-scene smoke test
renderer_static_test.c	86	Static shared-edge scene smoke test
voxel_test.c	212	Hardware flat-color/Z-buffer smoke test
world_chunk_test.c	2511	World, chunk, fluid, redstone tests

FPGA RTL and hardware integration (hw/)

File	Lines	Role
voxel_gpu/rtl/voxel_gpu.sv	4189	Main FPGA peripheral and datapath
voxel_gpu/rtl/voxel_math_utils.sv	301	Raster setup, draw step, reciprocal, fog
voxel_gpu/rtl/voxel_perf_counters.sv	76	Performance counter module
voxel_gpu/rtl/voxel_sdp_ram.sv	73	Dual-port M10K RAM wrapper
voxel_gpu/rtl/voxel_banked_sdp_ram.sv	48	Even/odd banked RAM wrapper
voxel_gpu/rtl/voxel_texture_rom.sv	111	Dual-copy texture ROM wrapper
voxel_gpu/rtl/voxel_vga_counters.sv	76	640×480 VGA timing
voxel_gpu/rtl/voxel_color_helpers.svh	64	Color conversion, alpha, light banks
voxel_gpu/rtl/voxel_raster_helpers.svh	215	Raster clamp, band, sky, texture helpers
voxel_gpu_hw.tcl	186	Platform Designer custom IP definition
soc_system_top.sv	334	Top-level wrapper, pin hookup
soc_system.qsys	742	Platform Designer system
soc_system.tcl	402	Project/pin assignment generator
soc_system.qsf	857	Quartus project settings
soc_system.qpf	31	Quartus project file
soc_system.sdc	15	Timing constraints
soc_system_board_info.xml	235	Board metadata
Makefile	374	Hardware build and install targets

SDRAM support Verilog (hw/sdram_local_test/)

File	Lines	Role
RW_Test.v	186	SDRAM local test support
Sdram_Control.v	436	SDRAM controller used by voxel_gpu
Sdram_Params.h	64	SDRAM parameter include
Sdram_RD_FIFO.v	196	SDRAM read FIFO

`Sdram_WR_FIFO.v`	196	SDRAM write FIFO
`command.v`	447	SDRAM command generator
`control_interface.v`	198	SDRAM control interface
`sdr_data_path.v`	34	SDRAM data path
`sdram_pll0.v`	255	SDRAM PLL wrapper

Asset scripts and virtual hardware

File	Lines	Role
`hw/voxel_gpu/scripts/generate_textures.py`	992	Texture atlas generator
`hw/voxel_gpu/scripts/generate_recip_lut.py`	35	Reciprocal LUT generator
`virtual_hw/pyproject.toml`	26	Python package metadata
`virtual_hw/virtualhw/__init__.py`	3	Package init
`virtual_hw/virtualhw/__main__.py`	5	Module entry point
`virtual_hw/virtualhw/protocol.py`	171	Socket protocol parser/packer
`virtual_hw/virtualhw/raster.py`	723	Reference software rasterizer
`virtual_hw/virtualhw/server.py`	381	Virtual GPU server
`virtual_hw/tests/test_raster.py`	257	Virtual raster tests
`worlds/redstone_lab/generate_redstone_lab.py`	716	Redstone demo world generator
`worlds/redstone_lab/settle_redstone_lab.c`	75	Redstone lab settle helper

9.2 Selected listings

Because the full implementation spans roughly 47 500 lines, the appendix lists just the final interface for hardware/software, reusable FPGA modules, and core game logic. Other logic can be seen within our tar submission or GitHub Repository

Complete `voxel_gpu.c` Software Interface

Listing 1: Complete `voxel_gpu.c` interface.

```c
/*
 * voxel_gpu.c - MVP device driver for the FPGA voxel GPU peripheral.
 *
 * Responsibilities (per the MVP spec):
 *   (A) Map the FPGA registers exposed through the lightweight HPS bridge.
 *   (B) Expose /dev/voxel_gpu via the misc subsystem.
 *   (C) Stream user-space command bytes into the on-chip FIFO via write().
 *   (D) Provide the control ioctls: CLEAR_FRAME, FLIP, SET_PALETTE,
 *       GET_STATUS, GET_FRAME_COUNT, SET_FOG, SET_EXTMEM, GET_EXTMEM,
 *       BEGIN_BAND, END_BAND.
 *   (E) Use polling on STATUS for synchronization (no interrupts).
 *
 * The driver is intentionally thin: it does not parse, validate, or
 * schedule descriptors. It is a pass-through pipe between user space
 * and the FIFO.
 *
 * Bind via the device tree compatible string "csee4840,voxel_gpu-1.0".
 */

#include <linux/module.h>
```

```c
#include <linux/init.h>
#include <linux/errno.h>
#include <linux/kernel.h>
#include <linux/platform_device.h>
#include <linux/miscdevice.h>
#include <linux/io.h>
#include <linux/of.h>
#include <linux/of_address.h>
#include <linux/fs.h>
#include <linux/uaccess.h>
#include <linux/slab.h>
#include <linux/mutex.h>
#include <linux/delay.h>
#include <linux/jiffies.h>
#include <linux/ktime.h>
#include <linux/math64.h>
#include <linux/string.h>
#include <linux/version.h>

#include "voxel_gpu.h"

#define DRIVER_NAME "voxel_gpu"

/* write() staging buffer; FIFO writes still back-pressure on STATUS.FIFO_COUNT. */
#define VOXEL_BOUNCE_WORDS  2048                    /* 8 KB per chunk */
#define VOXEL_BOUNCE_BYTES  (VOXEL_BOUNCE_WORDS * 4)

/* Generous polling timeout for a 60 Hz display path. */
#define VOXEL_POLL_TIMEOUT_MS   250
#define VOXEL_POLL_DELAY_US     1
#define VOXEL_FIFO_MIN_BURST_WORDS 64

static bool voxel_diag_upload;
module_param_named(diag_upload, voxel_diag_upload, bool, 0644);
MODULE_PARM_DESC(diag_upload,
                 "Print descriptor upload timing split: FIFO-space wait vs MMIO writes");

struct voxel_gpu_dev {
        struct resource res;
        void __iomem   *base;
        struct mutex    lock;       /* serializes register access */
        u32            *bounce;     /* reusable write() staging buffer */
};

static struct voxel_gpu_dev voxdev;

struct voxel_upload_diag {
        u64 bytes;
        u64 calls;
        u64 bursts;
        u64 wait_ns;
        u64 mmio_ns;
};

static struct voxel_upload_diag upload_diag_accum;
static unsigned long upload_diag_next_report;

/* ---------- low-level register helpers ---------- */

```

```
static inline u32 voxel_rd(u32 off)
{
        return ioread32(voxdev.base + off);
}

static inline void voxel_wr(u32 off, u32 val)
{
        iowrite32(val, voxdev.base + off);
}

static inline u32 voxel_status(void)
{
        return voxel_rd(VOXEL_REG_STATUS);
}

static inline u32 voxel_fifo_count(void)
{
        return (voxel_status() >> VOXEL_STAT_FIFO_SHIFT) &
               VOXEL_STAT_FIFO_MASK;
}

/*
 * Wait until (status & mask) == expect, polling every VOXEL_POLL_DELAY_US.
 * Returns 0 on success, -ETIMEDOUT if the condition never holds.
 */
static int voxel_poll_status(u32 mask, u32 expect, unsigned int timeout_ms)
{
        unsigned long deadline = jiffies + msecs_to_jiffies(timeout_ms);

        while (time_before(jiffies, deadline)) {
                if ((voxel_status() & mask) == expect)
                        return 0;
                udelay(VOXEL_POLL_DELAY_US);
                cond_resched();
        }
        return ((voxel_status() & mask) == expect) ? 0 : -ETIMEDOUT;
}

/* ---------- FIFO streaming ---------- */

static int voxel_fifo_wait_space(size_t *space_words, size_t min_req)
{
        unsigned long deadline = jiffies + msecs_to_jiffies(VOXEL_POLL_TIMEOUT_MS);
        unsigned int short_waits = 0;

        for (;;) {
                u32 used = voxel_fifo_count();
                u32 space = used < VOXEL_FIFO_WORDS ? VOXEL_FIFO_WORDS - used : 0;

                if (space >= min_req) {
                        *space_words = space;
                        return 0;
                }
                if (space > 0 && ++short_waits >= 8) {
                        *space_words = space;
                        return 0;
                }
                if (time_after(jiffies, deadline))
                        return -ETIMEDOUT;
```

```
                usleep_range(2, 5);
        }
}

static void voxel_diag_record_upload(size_t bytes, u64 wait_ns, u64 mmio_ns,
                                     u64 bursts)
{
        u64 total_ns;
        u64 total_kib_s;
        u64 mmio_kib_s;
        u64 wait_pct;
        u64 mmio_pct;

        if (!voxel_diag_upload)
                return;

        upload_diag_accum.bytes += bytes;
        upload_diag_accum.calls++;
        upload_diag_accum.bursts += bursts;
        upload_diag_accum.wait_ns += wait_ns;
        upload_diag_accum.mmio_ns += mmio_ns;

        if (!upload_diag_next_report ||
            time_after_eq(jiffies, upload_diag_next_report)) {
                total_ns = upload_diag_accum.wait_ns + upload_diag_accum.mmio_ns;
                total_kib_s = total_ns ?
                        div64_u64(upload_diag_accum.bytes * 1000000000ULL,
                                  total_ns * 1024ULL) : 0;
                mmio_kib_s = upload_diag_accum.mmio_ns ?
                        div64_u64(upload_diag_accum.bytes * 1000000000ULL,
                                  upload_diag_accum.mmio_ns * 1024ULL) : 0;
                wait_pct = total_ns ?
                        div64_u64(upload_diag_accum.wait_ns * 100ULL,
                                  total_ns) : 0;
                mmio_pct = total_ns ?
                        div64_u64(upload_diag_accum.mmio_ns * 100ULL,
                                  total_ns) : 0;

                pr_info(DRIVER_NAME
                        ": upload diag: calls=%llu bytes=%llu bursts=%llu "
                        "wait=%lluus(%llu%%) mmio=%lluus(%llu%%) "
                        "rate_total=%lluKiB/s rate_mmio=%lluKiB/s\n",
                        upload_diag_accum.calls,
                        upload_diag_accum.bytes,
                        upload_diag_accum.bursts,
                        div64_u64(upload_diag_accum.wait_ns, 1000),
                        wait_pct,
                        div64_u64(upload_diag_accum.mmio_ns, 1000),
                        mmio_pct,
                        total_kib_s,
                        mmio_kib_s);

                memset(&upload_diag_accum, 0, sizeof(upload_diag_accum));
                upload_diag_next_report = jiffies + msecs_to_jiffies(1000);
        }
}

/* ---------- file ops ---------- */

```

```
static int voxel_open(struct inode *inode, struct file *filp)
{
        filp->private_data = &voxdev;
        return 0;
}

static int voxel_release(struct inode *inode, struct file *filp)
{
        return 0;
}

/*
 * write(): treat the byte stream as a packed sequence of 32-bit FIFO
 * words. The driver does not interpret descriptor boundaries; user
 * space is responsible for handing us properly-aligned, multiple-of-4
 * payloads (typically whole quads = 16 words = 64 bytes).
 */
static ssize_t voxel_write(struct file *filp, const char __user *buf,
                           size_t count, loff_t *ppos)
{
        u32 *bounce;
        size_t total = 0;
        int ret = 0;

        if (count == 0)
                return 0;
        if (count & 0x3)
                return -EINVAL;        /* must be 32-bit aligned */

        bounce = voxdev.bounce;
        if (!bounce)
                return -ENODEV;

        if (mutex_lock_interruptible(&voxdev.lock)) {
                return -ERESTARTSYS;
        }

        while (total < count) {
                size_t chunk = min_t(size_t, count - total, VOXEL_BOUNCE_BYTES);
                size_t words;
                size_t written_words = 0;
                u64 write_wait_ns = 0;
                u64 write_mmio_ns = 0;
                u64 write_bursts = 0;

                if (copy_from_user(bounce, buf + total, chunk)) {
                        ret = -EFAULT;
                        break;
                }

                words = chunk / 4;
                while (written_words < words) {
                        size_t space_words;
                        size_t burst_words;
                        size_t rem_words = words - written_words;
                        size_t min_req = min_t(size_t, VOXEL_FIFO_MIN_BURST_WORDS,
                                               rem_words);
                        u64 t0;

```

```
                        t0 = ktime_get_ns();
                        ret = voxel_fifo_wait_space(&space_words, min_req);
                        write_wait_ns += ktime_get_ns() - t0;
                        if (ret)
                                goto out;

                        burst_words = min(space_words, rem_words);
                        t0 = ktime_get_ns();
                        iowrite32_rep(voxdev.base + VOXEL_FIFO_BASE,
                                      &bounce[written_words],
                                      burst_words);
                        write_mmio_ns += ktime_get_ns() - t0;
                        write_bursts++;

                        written_words += burst_words;
                }
                total += chunk;
                voxel_diag_record_upload(chunk, write_wait_ns, write_mmio_ns,
                                         write_bursts);
        }

out:
        mutex_unlock(&voxdev.lock);

        if (total > 0)
                return total;
        return ret;
}

/* ---------- ioctl handlers ---------- */

static long voxel_ioc_clear(void)
{
        u32 ctrl;
        int ret;

        mutex_lock(&voxdev.lock);

        /* Pulse CLR with EN held high; hardware self-clears CLR. */
        ctrl = voxel_rd(VOXEL_REG_CONTROL) | VOXEL_CTRL_EN | VOXEL_CTRL_CLR;
        voxel_wr(VOXEL_REG_CONTROL, ctrl);

        ret = voxel_poll_status(VOXEL_STAT_BSY, 0, VOXEL_POLL_TIMEOUT_MS);

        mutex_unlock(&voxdev.lock);
        return ret;
}

static long voxel_ioc_flip_common(bool wait_for_vsync)
{
        u32 ctrl;
        int ret;

        mutex_lock(&voxdev.lock);

        /*
         * Drain the FIFO before requesting a flip -- the rasterizer must
         * have consumed all pending quads before we swap buffers.
         */
```

```
        ret = voxel_poll_status(VOXEL_STAT_FEM, VOXEL_STAT_FEM,
                                VOXEL_POLL_TIMEOUT_MS);
        if (ret)
                goto out;

        ret = voxel_poll_status(VOXEL_STAT_BSY, 0, VOXEL_POLL_TIMEOUT_MS);
        if (ret)
                goto out;

        ctrl = voxel_rd(VOXEL_REG_CONTROL) | VOXEL_CTRL_EN | VOXEL_CTRL_FLP;
        voxel_wr(VOXEL_REG_CONTROL, ctrl);

        if (!wait_for_vsync)
                goto out;

        /* Block until the next vsync pulse latches. */
        ret = voxel_poll_status(VOXEL_STAT_VSY, VOXEL_STAT_VSY,
                                VOXEL_POLL_TIMEOUT_MS);

out:
        mutex_unlock(&voxdev.lock);
        return ret;
}

static long voxel_ioc_flip(void)
{
        return voxel_ioc_flip_common(true);
}

static long voxel_ioc_flip_async(void)
{
        return voxel_ioc_flip_common(false);
}

static long voxel_ioc_wait_flip(void)
{
        int ret;

        mutex_lock(&voxdev.lock);
        ret = voxel_poll_status(VOXEL_STAT_VSY, VOXEL_STAT_VSY,
                                VOXEL_POLL_TIMEOUT_MS);
        mutex_unlock(&voxdev.lock);
        return ret;
}

static long voxel_ioc_begin_band(void __user *uarg)
{
        struct voxel_band_state band;
        int ret;

        if (copy_from_user(&band, uarg, sizeof(band)))
                return -EFAULT;
        if (band.band_index > 7)
                return -EINVAL;
        if (band.flush_y_min >= VOXEL_BAND_CACHE_HEIGHT ||
            band.flush_y_max >= VOXEL_BAND_CACHE_HEIGHT ||
            band.flush_y_min > band.flush_y_max)
                return -EINVAL;

```

```
        mutex_lock(&voxdev.lock);

        ret = voxel_poll_status(VOXEL_STAT_FEM, VOXEL_STAT_FEM,
                                VOXEL_POLL_TIMEOUT_MS);
        if (ret)
                goto out;
        ret = voxel_poll_status(VOXEL_STAT_BSY, 0, VOXEL_POLL_TIMEOUT_MS);
        if (ret)
                goto out;

        voxel_wr(VOXEL_REG_BAND_INDEX, band.band_index);
        voxel_wr(VOXEL_REG_BAND_WINDOW,
                 (band.flush_y_min & 0x3fu) |
                 ((band.flush_y_max & 0x3fu) << 8));
        voxel_wr(VOXEL_REG_BAND_CTRL, VOXEL_BAND_CTRL_BEGIN);
        /* Keep BEGIN_BAND pipelined; the readback orders CSR writes before write(). */
        voxel_rd(VOXEL_REG_BAND_CTRL);

out:
        mutex_unlock(&voxdev.lock);
        return ret;
}

static long voxel_ioc_end_band(void)
{
        int ret;

        mutex_lock(&voxdev.lock);

        ret = voxel_poll_status(VOXEL_STAT_FEM, VOXEL_STAT_FEM,
                                VOXEL_POLL_TIMEOUT_MS);
        if (ret)
                goto out;
        ret = voxel_poll_status(VOXEL_STAT_BSY, 0, VOXEL_POLL_TIMEOUT_MS);
        if (ret)
                goto out;

        voxel_wr(VOXEL_REG_BAND_CTRL, VOXEL_BAND_CTRL_FLUSH);
        /* The RTL orders this background flush before the next band begins. */

out:
        mutex_unlock(&voxdev.lock);
        return ret;
}

static long voxel_ioc_set_palette(void __user *uarg)
{
        struct voxel_palette_entry e;
        u32 data;

        if (copy_from_user(&e, uarg, sizeof(e)))
                return -EFAULT;

        data = ((u32)e.r << 16) | ((u32)e.g << 8) | (u32)e.b;

        mutex_lock(&voxdev.lock);
        voxel_wr(VOXEL_REG_PALETTE_ADDR, e.index);
        voxel_wr(VOXEL_REG_PALETTE_DATA, data);
        mutex_unlock(&voxdev.lock);
```

```
434
435         return 0;
436 }
437
438 static long voxel_ioc_set_sky_palette(void __user *uarg)
439 {
440         struct voxel_sky_palette_entry e;
441         u32 data;
442
443         if (copy_from_user(&e, uarg, sizeof(e)))
444                 return -EFAULT;
445
446         data = ((u32)e.r << 16) | ((u32)e.g << 8) | (u32)e.b;
447
448         mutex_lock(&voxdev.lock);
449         voxel_wr(VOXEL_REG_SKY_PALETTE_ADDR, e.index);
450         voxel_wr(VOXEL_REG_SKY_PALETTE_DATA, data);
451         mutex_unlock(&voxdev.lock);
452
453         return 0;
454 }
455
456 static long voxel_ioc_get_status(void __user *uarg)
457 {
458         struct voxel_status s;
459         u32 raw = voxel_status();
460
461         s.raw        = raw;
462         s.fifo_count = (raw >> VOXEL_STAT_FIFO_SHIFT) & VOXEL_STAT_FIFO_MASK;
463         s.busy       = !!(raw & VOXEL_STAT_BSY);
464         s.fifo_full  = !!(raw & VOXEL_STAT_FFL);
465         s.fifo_empty = !!(raw & VOXEL_STAT_FEM);
466         s.vsync      = !!(raw & VOXEL_STAT_VSY);
467
468         if (copy_to_user(uarg, &s, sizeof(s)))
469                 return -EFAULT;
470         return 0;
471 }
472
473 static long voxel_ioc_get_frame_count(void __user *uarg)
474 {
475         u32 frames = voxel_rd(VOXEL_REG_FRAME_COUNT);
476
477         if (copy_to_user(uarg, &frames, sizeof(frames)))
478                 return -EFAULT;
479         return 0;
480 }
481
482 static long voxel_ioc_set_fog(void __user *uarg)
483 {
484         struct voxel_fog_state fog;
485         u32 range_word;
486         u32 ctrl_word;
487
488         if (copy_from_user(&fog, uarg, sizeof(fog)))
489                 return -EFAULT;
490
491         range_word = ((u32)fog.end_dist << 16) | (u32)fog.start_dist;
492         ctrl_word = ((u32)fog.inv_proj_sq << 16) |
```

```
                    ((fog.enabled ? 1u : 0u) << 8) |
                    (u32)fog.color_index;

        mutex_lock(&voxdev.lock);
        voxel_wr(VOXEL_REG_FOG_RANGE, range_word);
        voxel_wr(VOXEL_REG_FOG_CTRL, ctrl_word);
        mutex_unlock(&voxdev.lock);

        return 0;
}

static long voxel_ioc_set_extmem(void __user *uarg)
{
        struct voxel_extmem_state ext;

        if (copy_from_user(&ext, uarg, sizeof(ext)))
                return -EFAULT;

        mutex_lock(&voxdev.lock);
        voxel_wr(VOXEL_REG_EXTMEM_CTRL, ext.ctrl);
        voxel_wr(VOXEL_REG_EXTMEM_FRONT, ext.front_base);
        voxel_wr(VOXEL_REG_EXTMEM_BACK, ext.back_base);
        voxel_wr(VOXEL_REG_EXTMEM_STRIDE, ext.stride_bytes);
        voxel_wr(VOXEL_REG_EXTMEM_TILE, ext.tile_cfg);
        mutex_unlock(&voxdev.lock);

        return 0;
}

static long voxel_ioc_get_extmem(void __user *uarg)
{
        struct voxel_extmem_state ext;

        mutex_lock(&voxdev.lock);
        ext.ctrl = voxel_rd(VOXEL_REG_EXTMEM_CTRL);
        ext.front_base = voxel_rd(VOXEL_REG_EXTMEM_FRONT);
        ext.back_base = voxel_rd(VOXEL_REG_EXTMEM_BACK);
        ext.stride_bytes = voxel_rd(VOXEL_REG_EXTMEM_STRIDE);
        ext.tile_cfg = voxel_rd(VOXEL_REG_EXTMEM_TILE);
        ext.dma_status = voxel_rd(VOXEL_REG_EXTMEM_STAT);
        mutex_unlock(&voxdev.lock);

        if (copy_to_user(uarg, &ext, sizeof(ext)))
                return -EFAULT;
        return 0;
}

static long voxel_ioc_get_perf(void __user *uarg)
{
        struct voxel_perf_counters p;

        mutex_lock(&voxdev.lock);
        p.draw_active  = voxel_rd(VOXEL_REG_PERF_DRAW_ACT);
        p.draw_idle    = voxel_rd(VOXEL_REG_PERF_DRAW_IDLE);
        p.flush_active = voxel_rd(VOXEL_REG_PERF_FLUSH_ACT);
        p.flush_stall  = voxel_rd(VOXEL_REG_PERF_FLUSH_STL);
        p.init         = voxel_rd(VOXEL_REG_PERF_INIT);
        p.load         = voxel_rd(VOXEL_REG_PERF_LOAD);
        mutex_unlock(&voxdev.lock);
```

```
552
553         if (copy_to_user(uarg, &p, sizeof(p)))
554                 return -EFAULT;
555         return 0;
556 }
557
558 static long voxel_ioc_get_perf2(void __user *uarg)
559 {
560         struct voxel_perf_counters_v2 p;
561
562         mutex_lock(&voxdev.lock);
563         p.base.draw_active  = voxel_rd(VOXEL_REG_PERF_DRAW_ACT);
564         p.base.draw_idle    = voxel_rd(VOXEL_REG_PERF_DRAW_IDLE);
565         p.base.flush_active = voxel_rd(VOXEL_REG_PERF_FLUSH_ACT);
566         p.base.flush_stall  = voxel_rd(VOXEL_REG_PERF_FLUSH_STL);
567         p.base.init         = voxel_rd(VOXEL_REG_PERF_INIT);
568         p.base.load         = voxel_rd(VOXEL_REG_PERF_LOAD);
569         p.flush_wait_load   = voxel_rd(VOXEL_REG_PERF_FLUSH_LOAD);
570         p.flush_wait_fifo   = voxel_rd(VOXEL_REG_PERF_FLUSH_FIFO);
571         p.flush_wait_data   = voxel_rd(VOXEL_REG_PERF_FLUSH_DATA);
572         p.flush_wait_drain  = voxel_rd(VOXEL_REG_PERF_FLUSH_DRAIN);
573         mutex_unlock(&voxdev.lock);
574
575         if (copy_to_user(uarg, &p, sizeof(p)))
576                 return -EFAULT;
577         return 0;
578 }
579
580 static long voxel_ioctl(struct file *filp, unsigned int cmd, unsigned long arg)
581 {
582         if (_IOC_TYPE(cmd) != VOXEL_IOC_MAGIC)
583                 return -ENOTTY;
584         if (_IOC_NR(cmd) == 0 || _IOC_NR(cmd) > VOXEL_IOC_MAXNR)
585                 return -ENOTTY;
586
587         switch (cmd) {
588         case VOXEL_IOC_CLEAR_FRAME:
589                 return voxel_ioc_clear();
590         case VOXEL_IOC_FLIP:
591                 return voxel_ioc_flip();
592         case VOXEL_IOC_FLIP_ASYNC:
593                 return voxel_ioc_flip_async();
594         case VOXEL_IOC_WAIT_FLIP:
595                 return voxel_ioc_wait_flip();
596         case VOXEL_IOC_SET_PALETTE:
597                 return voxel_ioc_set_palette((void __user *)arg);
598         case VOXEL_IOC_SET_SKY_PALETTE:
599                 return voxel_ioc_set_sky_palette((void __user *)arg);
600         case VOXEL_IOC_GET_STATUS:
601                 return voxel_ioc_get_status((void __user *)arg);
602         case VOXEL_IOC_GET_FRAME_COUNT:
603                 return voxel_ioc_get_frame_count((void __user *)arg);
604         case VOXEL_IOC_SET_FOG:
605                 return voxel_ioc_set_fog((void __user *)arg);
606         case VOXEL_IOC_SET_EXTMEM:
607                 return voxel_ioc_set_extmem((void __user *)arg);
608         case VOXEL_IOC_GET_EXTMEM:
609                 return voxel_ioc_get_extmem((void __user *)arg);
610         case VOXEL_IOC_BEGIN_BAND:
```

```
                return voxel_ioc_begin_band((void __user *)arg);
        case VOXEL_IOC_END_BAND:
                return voxel_ioc_end_band();
        case VOXEL_IOC_GET_PERF:
                return voxel_ioc_get_perf((void __user *)arg);
        case VOXEL_IOC_GET_PERF2:
                return voxel_ioc_get_perf2((void __user *)arg);
        default:
                return -ENOTTY;
        }
}

static const struct file_operations voxel_fops = {
        .owner          = THIS_MODULE,
        .open           = voxel_open,
        .release        = voxel_release,
        .write          = voxel_write,
        .unlocked_ioctl = voxel_ioctl,
        .llseek         = noop_llseek,
};

static struct miscdevice voxel_miscdev = {
        .minor = MISC_DYNAMIC_MINOR,
        .name  = DRIVER_NAME,
        .fops  = &voxel_fops,
        .mode  = 0666,
};

/* ---------- platform driver glue ---------- */

static int voxel_probe(struct platform_device *pdev)
{
        int ret;

        mutex_init(&voxdev.lock);
        voxdev.bounce = kmalloc(VOXEL_BOUNCE_BYTES, GFP_KERNEL);
        if (!voxdev.bounce)
                return -ENOMEM;

        ret = misc_register(&voxel_miscdev);
        if (ret) {
                dev_err(&pdev->dev, "misc_register failed: %d\n", ret);
                goto err_bounce;
        }

        ret = of_address_to_resource(pdev->dev.of_node, 0, &voxdev.res);
        if (ret) {
                dev_err(&pdev->dev, "of_address_to_resource failed\n");
                ret = -ENOENT;
                goto err_misc;
        }

        if (!request_mem_region(voxdev.res.start, resource_size(&voxdev.res),
                                DRIVER_NAME)) {
                dev_err(&pdev->dev, "request_mem_region failed\n");
                ret = -EBUSY;
                goto err_misc;
        }

```

```
        voxdev.base = of_iomap(pdev->dev.of_node, 0);
        if (!voxdev.base) {
                dev_err(&pdev->dev, "of_iomap failed\n");
                ret = -ENOMEM;
                goto err_release;
        }

        /* Bring the engine up: enable, no interrupts, no pending pulses. */
        voxel_wr(VOXEL_REG_CONTROL, VOXEL_CTRL_EN);

        dev_info(&pdev->dev,
                 "voxel_gpu probed @ %pa, %llu bytes; status=0x%08x\n",
                 &voxdev.res.start,
                 (unsigned long long)resource_size(&voxdev.res),
                 voxel_status());
        return 0;

err_release:
        release_mem_region(voxdev.res.start, resource_size(&voxdev.res));
err_misc:
        misc_deregister(&voxel_miscdev);
err_bounce:
        kfree(voxdev.bounce);
        voxdev.bounce = NULL;
        return ret;
}

#if LINUX_VERSION_CODE >= KERNEL_VERSION(6, 11, 0)
static void voxel_remove(struct platform_device *pdev)
#else
static int voxel_remove(struct platform_device *pdev)
#endif
{
        (void)pdev;

        /* Disable the engine before tearing down. */
        voxel_wr(VOXEL_REG_CONTROL, 0);

        iounmap(voxdev.base);
        release_mem_region(voxdev.res.start, resource_size(&voxdev.res));
        misc_deregister(&voxel_miscdev);
        kfree(voxdev.bounce);
        voxdev.bounce = NULL;

#if LINUX_VERSION_CODE < KERNEL_VERSION(6, 11, 0)
        return 0;
#endif
}

#ifdef CONFIG_OF
static const struct of_device_id voxel_of_match[] = {
        { .compatible = "csee4840,voxel_gpu-1.0" },
        {},
};
MODULE_DEVICE_TABLE(of, voxel_of_match);
#endif

static struct platform_driver voxel_driver = {
        .driver = {
```

```
                .name           = DRIVER_NAME,
                .owner          = THIS_MODULE,
                .of_match_table = of_match_ptr(voxel_of_match),
        },
        .remove = __exit_p(voxel_remove),
};

static int __init voxel_init(void)
{
        pr_info(DRIVER_NAME ": init\n");
        return platform_driver_probe(&voxel_driver, voxel_probe);
}

static void __exit voxel_exit(void)
{
        platform_driver_unregister(&voxel_driver);
        pr_info(DRIVER_NAME ": exit\n");
}

module_init(voxel_init);
module_exit(voxel_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("the gang and twin");
MODULE_DESCRIPTION("Voxel GPU MVP driver -- FIFO streaming + control ioctls");
```

Complete `renderer.c` Implementation

Listing 2: Complete `renderer.c` Implementation.

```
#include "renderer.h"
#include "world.h"
#include "gpu_transport.h"
#include "voxel_gpu.h"
#include "env_util.h"
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <string.h>

#define NEAR_PLANE 0.1f
/*
 * Projected screen coordinates live on pixel-box edges: the visible viewport
 * spans [0, SCREEN_WIDTH] x [0, SCREEN_HEIGHT], while raster samples sit at
 * pixel centers (x + 0.5, y + 0.5). Keep clipping in edge space, then bake the
 * 0.5 center offset into edge/depth setup before descriptors hit the FPGA.
 */
#define VIEW_X_MIN 0.0f
#define VIEW_Y_MIN 0.0f
#define VIEW_X_MAX SCREEN_WIDTH
#define VIEW_Y_MAX SCREEN_HEIGHT
#define MAX_VIEW_CLIP_VERTS 12
#define PI_F 3.14159265358979323846f
#define SKY_DAY_LENGTH_SECONDS 180.0f
#define SKY_DOME_DISTANCE 512.0f
#define SKY_GRADIENT_BANDS 24
#define SKY_STAR_COUNT 144
```

```
#define OCCLUSION_TILE_SIZE 4
#define OCCLUSION_TILES_X ((VOXEL_RENDER_WIDTH + OCCLUSION_TILE_SIZE - 1u) / \
                           OCCLUSION_TILE_SIZE)
#define OCCLUSION_TILES_Y ((VOXEL_RENDER_HEIGHT + OCCLUSION_TILE_SIZE - 1u) / \
                           OCCLUSION_TILE_SIZE)
#define OCCLUSION_TILE_COUNT (OCCLUSION_TILES_X * OCCLUSION_TILES_Y)
#define OCCLUSION_DEPTH_EPSILON (2.0f / 32768.0f)
#define REDSTONE_DEVICE_PLATE_INSET 0.09f
#define REDSTONE_DEVICE_PLATE_HALF (0.5f - REDSTONE_DEVICE_PLATE_INSET)
#define REDSTONE_DEVICE_PLATE_TEXEL (REDSTONE_DEVICE_PLATE_HALF / 8.0f)
#define REDSTONE_DEVICE_POST_HALF 0.060f

/* Quantize sky-gradient palette updates so tiny time deltas do not churn the
 * hardware band-reuse epoch every frame. */
#define DEFAULT_SKY_PALETTE_TIME_STEP_SECONDS 1.0f
#define MIN_SKY_PALETTE_TIME_STEP_SECONDS 0.1f
#define MAX_SKY_PALETTE_TIME_STEP_SECONDS 10.0f

enum {
    PAL_SKY_HIGH = 25,
    PAL_SKY_MID = 26,
    PAL_SKY_HORIZON = 27,
    PAL_CLOUD = 28,
    PAL_CLOUD_SHADOW = 29,
    PAL_SUN_CORE = 30,
    PAL_SUN_GLOW = 31,
    PAL_MOON = 32,
    PAL_MOON_SHADOW = 33,
    PAL_STAR = 34,
    PAL_GLASS = 35,
    PAL_GLASS_EDGE = 36,
    PAL_GLASS_HIGHLIGHT = 37,
    PAL_LAMP_GLOW = 38,
    PAL_LAMP_FRAME = 39,
    PAL_SKY_GRADIENT_BASE = 40,
};

typedef struct {
    int x, y, z;
    uint8_t type;
    int occupied;
} LookupEntry;

typedef struct {
    LookupEntry *entries;
    int capacity;
    int mask;
} BlockLookup;

typedef struct {
    uint8_t r, g, b;
} RGB24;

typedef struct {
    RGB24 zenith;
    RGB24 high;
    RGB24 mid;
    RGB24 horizon;
    RGB24 cloud;
```

```
    RGB24 cloud_shadow;
    RGB24 sun_core;
    RGB24 sun_glow;
    RGB24 moon;
    RGB24 moon_shadow;
    RGB24 star;
} SkyPalette;

static Vec3 sun_direction_for_time(float time_seconds);

static void upload_default_palette(GPUTransport *transport)
{
    static const struct voxel_palette_entry entries[] = {
        {  0, 0x10, 0x10, 0x18 }, /* background */
        {  1, 0x6b, 0xa4, 0x3a }, /* grass top */
        {  2, 0x8b, 0x63, 0x41 }, /* dirt side */
        {  3, 0x6f, 0x57, 0x37 }, /* wood side */
        {  4, 0x7c, 0x7c, 0x7c }, /* stone side */
        {  5, 0xff, 0xff, 0xff }, /* debug white */
        {  6, 0xff, 0x40, 0x40 }, /* debug red */
        {  7, 0x40, 0xa0, 0xff }, /* debug blue */
        {  8, 0xff, 0xd0, 0x40 }, /* debug yellow */
        {  9, 0x5c, 0x86, 0x34 }, /* grass side */
        { 10, 0x6a, 0x4a, 0x2c }, /* grass bottom / dark dirt */
        { 11, 0x9d, 0x7b, 0x4d }, /* wood top */
        { 12, 0x53, 0x38, 0x23 }, /* wood bottom */
        { 13, 0x98, 0x98, 0x98 }, /* stone top */
        { 14, 0x5c, 0x5c, 0x5c }, /* stone bottom */
        { 15, 0xa7, 0x79, 0x52 }, /* dirt top */
        { 16, 0x59, 0x41, 0x2a }, /* dirt bottom */
        { 17, 0x4f, 0x78, 0x2d }, /* grass dark */
        { 18, 0x84, 0xba, 0x57 }, /* grass highlight */
        { 19, 0x6f, 0x4f, 0x32 }, /* dirt dark */
        { 20, 0xaa, 0x81, 0x5a }, /* dirt light */
        { 21, 0x88, 0x6a, 0x44 }, /* wood grain */
        { 22, 0x50, 0x3b, 0x24 }, /* wood bark dark */
        { 23, 0x63, 0x63, 0x63 }, /* stone dark */
        { 24, 0x9a, 0x9a, 0x9a }, /* stone light */
        { 25, 0x78, 0xb4, 0xf0 }, /* sky high */
        { 26, 0xb0, 0xd8, 0xff }, /* sky mid */
        { 27, 0xe2, 0xef, 0xff }, /* sky horizon */
        { 28, 0xf5, 0xfa, 0xff }, /* cloud */
        { 29, 0xcb, 0xd8, 0xec }, /* cloud shadow */
        { 30, 0xff, 0xee, 0xaa }, /* sun core */
        { 31, 0xff, 0xbb, 0x55 }, /* sun glow */
        { 32, 0xe7, 0xeb, 0xf8 }, /* moon */
        { 33, 0x9c, 0xa4, 0xc0 }, /* moon shadow */
        { 34, 0xff, 0xff, 0xff }, /* stars */
        { 35, 0xb8, 0xe4, 0xff }, /* glass body */
        { 36, 0x5e, 0x7c, 0x98 }, /* glass edge / frame */
        { 37, 0xff, 0xff, 0xff }, /* glass highlight */
        { 38, 0xff, 0xd7, 0x79 }, /* lamp glow */
        { 39, 0x6d, 0x53, 0x30 }, /* lamp frame */
        { 40, 0x2a, 0x52, 0x9c }, /* banked water deep */
        { 41, 0x3a, 0x6c, 0xc4 }, /* banked water mid */
        { 42, 0x6f, 0x9d, 0xe4 }, /* banked water highlight */
        { 43, 0x7a, 0x20, 0x10 }, /* lava dark */
        { 44, 0xe8, 0x5c, 0x18 }, /* lava orange */
        { 45, 0xff, 0xd8, 0x5a }, /* lava hot */
```

```
        { 46, 0xd8, 0xc0, 0x74 }, /* sand */
        { 47, 0xa8, 0x90, 0x50 }, /* sand dark */
        { 48, 0xb8, 0x4d, 0x3f }, /* brick */
        { 49, 0x6e, 0x2a, 0x27 }, /* brick dark */
        { 50, 0x20, 0x1b, 0x2c }, /* obsidian */
        { 51, 0x43, 0x36, 0x58 }, /* obsidian edge */
        { 52, 0x72, 0x82, 0x91 }, /* clay */
        { 53, 0xed, 0xdc, 0x9c }, /* sand light */
        { 54, 0xc3, 0xac, 0x67 }, /* sand shadow */
        { 55, 0xd0, 0x68, 0x55 }, /* brick light */
        { 56, 0x8b, 0x56, 0x4e }, /* brick mortar */
    };

    for (size_t i = 0; i < sizeof(entries) / sizeof(entries[0]); i++) {
        if (gpu_transport_set_palette(transport, &entries[i]) < 0)
            break;
    }
}

typedef struct {
    const Chunk *chunk;
    const ChunkFace *face;
    float view_z;
    uint32_t stable_order;
} TranslucentFaceRef;

typedef enum {
    OCCLUSION_PASS_DISABLED = 0,
    OCCLUSION_PASS_OPAQUE_WORLD,
} OcclusionPass;

struct RenderContext {
    GPUTransport *transport;
    Camera current_camera;
    Vec3 light_dir;
    float world_daylight;

    float cos_yaw, sin_yaw;
    float cos_pitch, sin_pitch;

    uint8_t *submit_buffer;
    size_t submit_capacity;
    size_t submit_bytes;
    LookupEntry *lookup_entries;
    int lookup_capacity;
    uint16_t palette_cache[256];
    uint8_t palette_valid[256];
    struct voxel_palette_entry palette_pending[256];
    uint8_t palette_dirty[256];
    uint16_t sky_palette_cache[SKY_GRADIENT_BANDS];
    uint8_t sky_palette_valid[SKY_GRADIENT_BANDS];
    struct voxel_sky_palette_entry sky_palette_pending[SKY_GRADIENT_BANDS];
    uint8_t sky_palette_dirty[SKY_GRADIENT_BANDS];
    struct voxel_fog_state fog_cache;
    struct voxel_fog_state fog_pending;
    uint8_t fog_valid;
    uint8_t fog_dirty;
    uint8_t light_palette_key;
    uint8_t light_palette_key_valid;
```

```
    uint8_t face_light_flags[NUM_FACES];
    Vec3 last_sun_dir;
    uint8_t last_sun_dir_valid;
    int n_quads;
    uint8_t occlusion_enabled;
    uint8_t occlusion_diag;
    OcclusionPass occlusion_pass;
    uint8_t occlusion_covered[OCCLUSION_TILE_COUNT];
    float occlusion_depth[OCCLUSION_TILE_COUNT];
    uint32_t occlusion_tested;
    uint32_t occlusion_culled;
    uint32_t occlusion_tiles_written;
    uint32_t occlusion_frame_count;

    /* Scratch buffer reused across frames to back-to-front sort translucent
     * faces. Grown on demand; memory only leaves if the scene peaks higher
     * than it has before. */
    TranslucentFaceRef *translucent_scratch;
    int translucent_scratch_capacity;

};

static bool stage_prepared_quad(RenderContext *ctx, RenderQuad quad);
static bool stage_projected_quad_no_clip(RenderContext *ctx,
                                         const RenderQuad *quad);
static bool stage_projected_polygon(RenderContext *ctx,
                                    const RenderQuad *quad,
                                    const Vertex2D *verts,
                                    int count);

typedef struct {
    float x, y, z;
    float u, v;
} CameraVertex;

static RGB24 rgb24(uint8_t r, uint8_t g, uint8_t b)
{
    RGB24 color = { r, g, b };
    return color;
}

static RGB24 default_palette_color(uint8_t index)
{
    switch (index) {
    case 0:  return rgb24(0x10, 0x10, 0x18);
    case 1:  return rgb24(0x6b, 0xa4, 0x3a);
    case 2:  return rgb24(0x8b, 0x63, 0x41);
    case 3:  return rgb24(0x6f, 0x57, 0x37);
    case 4:  return rgb24(0x7c, 0x7c, 0x7c);
    case 5:  return rgb24(0xff, 0xff, 0xff);
    case 6:  return rgb24(0xff, 0x40, 0x40);
    case 7:  return rgb24(0x40, 0xa0, 0xff);
    case 8:  return rgb24(0xff, 0xd0, 0x40);
    case 9:  return rgb24(0x5c, 0x86, 0x34);
    case 10: return rgb24(0x6a, 0x4a, 0x2c);
    case 11: return rgb24(0x9d, 0x7b, 0x4d);
    case 12: return rgb24(0x53, 0x38, 0x23);
    case 13: return rgb24(0x98, 0x98, 0x98);
    case 14: return rgb24(0x5c, 0x5c, 0x5c);
```

```
    case 15: return rgb24(0xa7, 0x79, 0x52);
    case 16: return rgb24(0x59, 0x41, 0x2a);
    case 17: return rgb24(0x4f, 0x78, 0x2d);
    case 18: return rgb24(0x84, 0xba, 0x57);
    case 19: return rgb24(0x6f, 0x4f, 0x32);
    case 20: return rgb24(0xaa, 0x81, 0x5a);
    case 21: return rgb24(0x88, 0x6a, 0x44);
    case 22: return rgb24(0x50, 0x3b, 0x24);
    case 23: return rgb24(0x63, 0x63, 0x63);
    case 24: return rgb24(0x9a, 0x9a, 0x9a);
    case 25: return rgb24(0x78, 0xb4, 0xf0);
    case 26: return rgb24(0xb0, 0xd8, 0xff);
    case 27: return rgb24(0xe2, 0xef, 0xff);
    case 28: return rgb24(0xf5, 0xfa, 0xff);
    case 29: return rgb24(0xcb, 0xd8, 0xec);
    case 30: return rgb24(0xff, 0xee, 0xaa);
    case 31: return rgb24(0xff, 0xbb, 0x55);
    case 32: return rgb24(0xe7, 0xeb, 0xf8);
    case 33: return rgb24(0x9c, 0xa4, 0xc0);
    case 34: return rgb24(0xff, 0xff, 0xff);
    case 35: return rgb24(0xb8, 0xe4, 0xff);
    case 36: return rgb24(0x5e, 0x7c, 0x98);
    case 37: return rgb24(0xff, 0xff, 0xff);
    case 38: return rgb24(0xff, 0xd7, 0x79);
    case 39: return rgb24(0x6d, 0x53, 0x30);
    case 40: return rgb24(0x2a, 0x52, 0x9c);
    case 41: return rgb24(0x3a, 0x6c, 0xc4);
    case 42: return rgb24(0x6f, 0x9d, 0xe4);
    case 43: return rgb24(0x7a, 0x20, 0x10);
    case 44: return rgb24(0xe8, 0x5c, 0x18);
    case 45: return rgb24(0xff, 0xd8, 0x5a);
    case 46: return rgb24(0xd8, 0xc0, 0x74);
    case 47: return rgb24(0xa8, 0x90, 0x50);
    case 48: return rgb24(0xb8, 0x4d, 0x3f);
    case 49: return rgb24(0x6e, 0x2a, 0x27);
    case 50: return rgb24(0x20, 0x1b, 0x2c);
    case 51: return rgb24(0x43, 0x36, 0x58);
    case 52: return rgb24(0x72, 0x82, 0x91);
    case 53: return rgb24(0xed, 0xdc, 0x9c);
    case 54: return rgb24(0xc3, 0xac, 0x67);
    case 55: return rgb24(0xd0, 0x68, 0x55);
    case 56: return rgb24(0x8b, 0x56, 0x4e);
    default: return rgb24(0x00, 0x00, 0x00);
    }
}

static float clamp01(float value)
{
    if (value < 0.0f)
        return 0.0f;
    if (value > 1.0f)
        return 1.0f;
    return value;
}

static float smoothstepf(float edge0, float edge1, float x)
{
    if (edge0 == edge1)
        return x >= edge1 ? 1.0f : 0.0f;
```

```
323
324     x = clamp01((x - edge0) / (edge1 - edge0));
325     return x * x * (3.0f - 2.0f * x);
326 }
327
328 static RGB24 lerp_rgb24(RGB24 a, RGB24 b, float t)
329 {
330     RGB24 out;
331
332     t = clamp01(t);
333     out.r = (uint8_t)lroundf(a.r + (b.r - a.r) * t);
334     out.g = (uint8_t)lroundf(a.g + (b.g - a.g) * t);
335     out.b = (uint8_t)lroundf(a.b + (b.b - a.b) * t);
336     return out;
337 }
338
339 static RGB24 mix_rgb24(RGB24 base, RGB24 tint, float amount)
340 {
341     return lerp_rgb24(base, tint, amount);
342 }
343
344 static RGB24 scale_rgb24(RGB24 color, float factor)
345 {
346     return rgb24((uint8_t)lroundf(color.r * factor),
347                  (uint8_t)lroundf(color.g * factor),
348                  (uint8_t)lroundf(color.b * factor));
349 }
350
351 static uint16_t rgb24_to_rgb565(RGB24 color)
352 {
353     return (uint16_t)(((uint16_t)(color.r & 0xF8) << 8) |
354                       ((uint16_t)(color.g & 0xFC) << 3) |
355                       ((uint16_t)color.b >> 3));
356 }
357
358 static bool renderer_set_palette_rgb(RenderContext *ctx, uint8_t index, RGB24 color)
359 {
360     uint16_t packed = rgb24_to_rgb565(color);
361
362     if (ctx->palette_valid[index] && ctx->palette_cache[index] == packed)
363         return true;
364
365     ctx->palette_valid[index] = 1;
366     ctx->palette_cache[index] = packed;
367     ctx->palette_pending[index] = (struct voxel_palette_entry){
368         .index = index,
369         .r = color.r,
370         .g = color.g,
371         .b = color.b,
372     };
373     ctx->palette_dirty[index] = 1;
374     return true;
375 }
376
377 static bool renderer_set_sky_palette_rgb(RenderContext *ctx, uint8_t index, RGB24 color)
378 {
379     uint16_t packed;
380
381     if (!ctx || index >= SKY_GRADIENT_BANDS)
```

```
        return false;

    packed = rgb24_to_rgb565(color);
    if (ctx->sky_palette_valid[index] &&
        ctx->sky_palette_cache[index] == packed)
        return true;

    ctx->sky_palette_valid[index] = 1;
    ctx->sky_palette_cache[index] = packed;
    ctx->sky_palette_pending[index] = (struct voxel_sky_palette_entry){
        .index = index,
        .r = color.r,
        .g = color.g,
        .b = color.b,
    };
    ctx->sky_palette_dirty[index] = 1;
    return true;
}

static bool same_fog_state(const struct voxel_fog_state *a,
                           const struct voxel_fog_state *b)
{
    return a->start_dist == b->start_dist &&
           a->end_dist == b->end_dist &&
           a->color_index == b->color_index &&
           a->enabled == b->enabled &&
           a->inv_proj_sq == b->inv_proj_sq;
}

static bool renderer_set_fog_state(RenderContext *ctx,
                                   const struct voxel_fog_state *fog)
{
    if (ctx->fog_valid && same_fog_state(&ctx->fog_cache, fog))
        return true;

    ctx->fog_cache = *fog;
    ctx->fog_pending = *fog;
    ctx->fog_valid = 1;
    ctx->fog_dirty = 1;
    return true;
}

static bool renderer_flush_gpu_state(RenderContext *ctx)
{
    for (int i = 0; i < SKY_GRADIENT_BANDS; i++) {
        if (!ctx->sky_palette_dirty[i])
            continue;
        if (gpu_transport_set_sky_palette(ctx->transport,
                                          &ctx->sky_palette_pending[i]) < 0)
            return false;
        ctx->sky_palette_dirty[i] = 0;
    }

    for (int i = 0; i < 256; i++) {
        if (!ctx->palette_dirty[i])
            continue;
        if (gpu_transport_set_palette(ctx->transport,
                                      &ctx->palette_pending[i]) < 0)
            return false;
```

```
        ctx->palette_dirty[i] = 0;
    }

    if (ctx->fog_dirty) {
        if (gpu_transport_set_fog(ctx->transport, &ctx->fog_pending) < 0)
            return false;
        ctx->fog_dirty = 0;
    }

    return true;
}

static void upload_light_palette_banks(RenderContext *ctx, float daylight)
{
    static const float bank_scale_day[4] = { 1.00f, 0.82f, 0.64f, 0.50f };
    static const float bank_scale_night[4] = { 1.00f, 0.34f, 0.20f, 0.10f };
    int daylight_key;

    if (!ctx)
        return;

    daylight = clamp01(daylight);
    daylight_key = (int)lroundf(daylight * 31.0f);
    if (ctx->light_palette_key_valid &&
        ctx->light_palette_key == (uint8_t)daylight_key)
        return;

    daylight = (float)daylight_key / 31.0f;

    for (int bank = 1; bank < 4; bank++) {
        float bank_scale = bank_scale_night[bank] +
                           (bank_scale_day[bank] - bank_scale_night[bank]) *
                           daylight;

        for (int index = 1; index < 64; index++) {
            RGB24 color = scale_rgb24(default_palette_color((uint8_t)index),
                                      bank_scale);

            if (!renderer_set_palette_rgb(ctx,
                                          (uint8_t)(bank * 64 + index),
                                          color))
                return;
        }
    }

    ctx->light_palette_key = (uint8_t)daylight_key;
    ctx->light_palette_key_valid = 1;
}

static Vec3 normalize_vec3(Vec3 v)
{
    float len_sq = v.x * v.x + v.y * v.y + v.z * v.z;

    if (len_sq <= 0.0f)
        return (Vec3){ 0.0f, 0.0f, 1.0f };

    float inv_len = 1.0f / sqrtf(len_sq);
    return (Vec3){ v.x * inv_len, v.y * inv_len, v.z * inv_len };
}
```

```
500
501 static Vec3 direction_from_azimuth_elevation(float azimuth, float elevation)
502 {
503     float cos_elevation = cosf(elevation);
504
505     return (Vec3){
506         cos_elevation * sinf(azimuth),
507         sinf(elevation),
508         cos_elevation * cosf(azimuth),
509     };
510 }
511
512 static const Vec3 face_normals[NUM_FACES] = {
513     { 0, 1, 0 },
514     { 0, -1, 0 },
515     { -1, 0, 0 },
516     { 1, 0, 0 },
517     { 0, 0, -1 },
518     { 0, 0, 1 },
519 };
520
521 static uint8_t light_flags_for_normal(Vec3 light_dir, Vec3 normal)
522 {
523     float ndotl = normal.x * light_dir.x +
524                   normal.y * light_dir.y +
525                   normal.z * light_dir.z;
526     unsigned bank;
527
528     if (ndotl >= 0.65f)
529         bank = 0;
530     else if (ndotl >= 0.15f)
531         bank = 1;
532     else if (ndotl >= -0.25f)
533         bank = 2;
534     else
535         bank = 3;
536
537     return QUAD_LIGHT_LEVEL(bank);
538 }
539
540 static void update_face_light_flags(RenderContext *ctx)
541 {
542     for (int face = 0; face < NUM_FACES; face++)
543         ctx->face_light_flags[face] =
544             light_flags_for_normal(ctx->light_dir, face_normals[face]);
545 }
546
547 static uint8_t light_bank_for_level(uint8_t light_level)
548 {
549     if (light_level >= 12)
550         return 0;
551     if (light_level >= 8)
552         return 1;
553     if (light_level >= 4)
554         return 2;
555     return 3;
556 }
557
558 static uint8_t sky_light_level_for_face(const RenderContext *ctx,
```

```
                                        BlockFace face,
                                        uint8_t sky_light)
{
    float ndotl;
    float facing;
    float intensity;
    int level;

    if (!ctx || face < 0 || face >= NUM_FACES || sky_light == 0)
        return 0;

    ndotl = face_normals[face].x * ctx->light_dir.x +
            face_normals[face].y * ctx->light_dir.y +
            face_normals[face].z * ctx->light_dir.z;
    facing = clamp01(0.5f + 0.5f * ndotl);
    intensity = 0.12f + ctx->world_daylight * (0.25f + 0.63f * facing);
    level = (int)lroundf((float)sky_light * clamp01(intensity));
    if (level < 0)
        level = 0;
    if (level > 15)
        level = 15;
    return (uint8_t)level;
}

static uint8_t choose_chunk_face_light_flags(const RenderContext *ctx,
                                             BlockID type, BlockFace face,
                                             uint8_t sky_light,
                                             uint8_t block_light)
{
    uint8_t effective_level = block_light;
    uint8_t sky_level = sky_light_level_for_face(ctx, face, sky_light);

    if (block_is_self_lit(type))
        effective_level = 15;
    if (sky_level > effective_level)
        effective_level = sky_level;

    return QUAD_LIGHT_LEVEL(light_bank_for_level(effective_level));
}

static float daylight_strength_for_sun_direction(Vec3 sun_dir)
{
    return smoothstepf(-0.18f, 0.08f, sun_dir.y);
}

static Vec3 terrain_light_direction_for_sun(Vec3 sun_dir)
{
    if (sun_dir.y < 0.18f)
        sun_dir.y = 0.18f;
    return normalize_vec3(sun_dir);
}

static void update_world_light_from_sun(RenderContext *ctx, Vec3 sun_dir)
{
    if (!ctx)
        return;

    if (ctx->last_sun_dir_valid &&
        ctx->last_sun_dir.x == sun_dir.x &&
```

```
        ctx->last_sun_dir.y == sun_dir.y &&
        ctx->last_sun_dir.z == sun_dir.z)
        return;

    ctx->world_daylight = daylight_strength_for_sun_direction(sun_dir);
    ctx->light_dir = terrain_light_direction_for_sun(sun_dir);
    update_face_light_flags(ctx);
    upload_light_palette_banks(ctx, ctx->world_daylight);
    ctx->last_sun_dir = sun_dir;
    ctx->last_sun_dir_valid = 1;
}

static float palette_time_for(float time_seconds)
{
    static int initialized;
    static float step_seconds = DEFAULT_SKY_PALETTE_TIME_STEP_SECONDS;

    if (!initialized) {
        step_seconds =
            env_float_or_default("VOXEL_SKY_PALETTE_STEP_SECONDS",
                                 DEFAULT_SKY_PALETTE_TIME_STEP_SECONDS,
                                 MIN_SKY_PALETTE_TIME_STEP_SECONDS,
                                 MAX_SKY_PALETTE_TIME_STEP_SECONDS);
        initialized = 1;
    }

    return floorf(time_seconds / step_seconds) * step_seconds;
}

static void update_world_light_state(RenderContext *ctx, float time_seconds)
{
    update_world_light_from_sun(ctx,
                                sun_direction_for_time(palette_time_for(time_seconds)));
}

static void world_to_camera(RenderContext *ctx, Vec3 world, CameraVertex *out)
{
    float dx = world.x - ctx->current_camera.position.x;
    float dy = world.y - ctx->current_camera.position.y;
    float dz = world.z - ctx->current_camera.position.z;

    float x_yaw =  dx * ctx->cos_yaw - dz * ctx->sin_yaw;
    float z_yaw =  dx * ctx->sin_yaw + dz * ctx->cos_yaw;
    float y_yaw =  dy;

    out->x = x_yaw;
    out->y =  y_yaw * ctx->cos_pitch + z_yaw * ctx->sin_pitch;
    out->z = -y_yaw * ctx->sin_pitch + z_yaw * ctx->cos_pitch;
}

/* Project a camera-space point into screen space.
 * Returns false if the point is behind the near plane. */
static bool project_camera_vertex(RenderContext *ctx, const CameraVertex *in, Vertex2D *out)
{
    if (in->z < NEAR_PLANE)
        return false;

    /*
     * Use an inverse-depth mapping so the per-quad depth plane stays affine
```

```
     * in screen space and fits inside the descriptor's Q1.15 range.
     * Smaller values remain closer, matching the hardware z compare.
     */
    out->z = 1.0f - (NEAR_PLANE / in->z);

    float depth = ctx->current_camera.depth;
    float inv_w = 1.0f / in->z;
    out->x = in->x * inv_w * depth + SCREEN_WIDTH  / 2.0f;
    out->y = SCREEN_HEIGHT / 2.0f - in->y * inv_w * depth;
    /*
     * Store u/w, v/w, 1/w instead of u, v. These three quantities are linear
     * in screen space for any planar world-space quad, so viewport clipping
     * can lerp them correctly and the HW can recover true u, v by dividing
     * per pixel. This eliminates the affine-texture swim under perspective.
     */
    out->u_over_w   = in->u * inv_w;
    out->v_over_w   = in->v * inv_w;
    out->one_over_w = inv_w;
    return true;
}

static bool camera_quad_outside_view(RenderContext *ctx, const CameraVertex v[4])
{
    float x_slope = (SCREEN_WIDTH * 0.5f) / ctx->current_camera.depth;
    float y_slope = (SCREEN_HEIGHT * 0.5f) / ctx->current_camera.depth;
    bool outside_near = true;
    bool outside_left = true;
    bool outside_right = true;
    bool outside_top = true;
    bool outside_bottom = true;

    /*
     * The side-plane tests (x + x_slope*z < 0, etc.) are valid only for
     * vertices in front of the near plane. For z <= 0, x_slope*z is non-positive,
     * so the test can flip sign and false-positive -- a vertex behind the camera
     * with any x looks "to the left of left" by this math, which used to cull
     * any face that had a vertex behind the player. Gate side-plane votes on
     * z >= NEAR_PLANE; a partially-behind face only gets culled if every vertex
     * is behind the near plane.
     */
    for (int i = 0; i < 4; i++) {
        bool z_in_front = (v[i].z >= NEAR_PLANE);
        outside_near &= !z_in_front;
        outside_left &= z_in_front && (v[i].x + x_slope * v[i].z < 0.0f);
        outside_right &= z_in_front && (-v[i].x + x_slope * v[i].z < 0.0f);
        outside_top &= z_in_front && (y_slope * v[i].z - v[i].y < 0.0f);
        outside_bottom &= z_in_front && (v[i].y + y_slope * v[i].z < 0.0f);
    }

    return outside_near || outside_left || outside_right ||
           outside_top || outside_bottom;
}

/* Dot-product backface cull: skip face if normal points away from camera. */
static bool is_face_visible(Vec3 block_pos, Vec3 normal, Vec3 cam_pos)
{
    Vec3 v = {
        (block_pos.x + 0.5f + normal.x * 0.5f) - cam_pos.x,
        (block_pos.y + 0.5f + normal.y * 0.5f) - cam_pos.y,
```

```
        (block_pos.z + 0.5f + normal.z * 0.5f) - cam_pos.z,
    };
    return (v.x*normal.x + v.y*normal.y + v.z*normal.z) < 0.0f;
}

static float height_eighths_to_world(uint8_t height_eighths)
{
    if (height_eighths < 1)
        height_eighths = 8;
    if (height_eighths > 8)
        height_eighths = 8;
    return (float)height_eighths * (1.0f / 8.0f);
}

static Vec3 merged_face_center(Vec3 block_pos, BlockFace face,
                               int u_size, int v_size,
                               uint8_t height_eighths)
{
    float h = height_eighths_to_world(height_eighths);
    float u = (float)(u_size > 0 ? u_size : 1);
    float v = (float)(v_size > 0 ? v_size : 1);
    float vertical_extent = (v - 1.0f) + h;

    switch (face) {
    case FACE_TOP:
        return (Vec3){ block_pos.x + 0.5f * u,
                       block_pos.y + h,
                       block_pos.z + 0.5f * v };
    case FACE_BOTTOM:
        return (Vec3){ block_pos.x + 0.5f * u,
                       block_pos.y,
                       block_pos.z + 0.5f * v };
    case FACE_LEFT:
        return (Vec3){ block_pos.x,
                       block_pos.y + 0.5f * vertical_extent,
                       block_pos.z + 0.5f * u };
    case FACE_RIGHT:
        return (Vec3){ block_pos.x + 1.0f,
                       block_pos.y + 0.5f * vertical_extent,
                       block_pos.z + 0.5f * u };
    case FACE_FRONT:
        return (Vec3){ block_pos.x + 0.5f * u,
                       block_pos.y + 0.5f * vertical_extent,
                       block_pos.z };
    case FACE_BACK:
        return (Vec3){ block_pos.x + 0.5f * u,
                       block_pos.y + 0.5f * vertical_extent,
                       block_pos.z + 1.0f };
    default:
        return block_pos;
    }
}

static bool merged_face_visible(Vec3 block_pos, BlockFace face,
                                int u_size, int v_size,
                                uint8_t height_eighths, Vec3 cam_pos)
{
    if (face < 0 || face >= NUM_FACES)
        return true;
```

```
795
796     Vec3 center = merged_face_center(block_pos, face, u_size, v_size,
797                                      height_eighths);
798     Vec3 normal = face_normals[face];
799     Vec3 v = {
800         center.x - cam_pos.x,
801         center.y - cam_pos.y,
802         center.z - cam_pos.z,
803     };
804
805     return (v.x * normal.x + v.y * normal.y + v.z * normal.z) < 0.0f;
806 }
807
808 static CameraVertex lerp_camera_vertex(const CameraVertex *a,
809                                        const CameraVertex *b, float t)
810 {
811     CameraVertex out = {
812         .x = a->x + (b->x - a->x) * t,
813         .y = a->y + (b->y - a->y) * t,
814         .z = a->z + (b->z - a->z) * t,
815         .u = a->u + (b->u - a->u) * t,
816         .v = a->v + (b->v - a->v) * t,
817     };
818
819     return out;
820 }
821
822 static Vertex2D lerp_vertex2d(const Vertex2D *a, const Vertex2D *b, float t)
823 {
824     Vertex2D out = {
825         .x = a->x + (b->x - a->x) * t,
826         .y = a->y + (b->y - a->y) * t,
827         .z = a->z + (b->z - a->z) * t,
828         .u_over_w   = a->u_over_w   + (b->u_over_w   - a->u_over_w)   * t,
829         .v_over_w   = a->v_over_w   + (b->v_over_w   - a->v_over_w)   * t,
830         .one_over_w = a->one_over_w + (b->one_over_w - a->one_over_w) * t,
831     };
832
833     return out;
834 }
835
836 static int clip_face_to_near_plane(const CameraVertex in[4], CameraVertex out[6])
837 {
838     int out_count = 0;
839     CameraVertex prev = in[3];
840     bool prev_inside = (prev.z >= NEAR_PLANE);
841
842     for (int i = 0; i < 4; i++) {
843         CameraVertex cur = in[i];
844         bool cur_inside = (cur.z >= NEAR_PLANE);
845
846         if (prev_inside != cur_inside) {
847             float t = (NEAR_PLANE - prev.z) / (cur.z - prev.z);
848             CameraVertex clipped = lerp_camera_vertex(&prev, &cur, t);
849             clipped.z = NEAR_PLANE;
850             out[out_count++] = clipped;
851         }
852
853         if (cur_inside)
```

```
            out[out_count++] = cur;

        prev = cur;
        prev_inside = cur_inside;
    }

    return out_count;
}

typedef enum {
    CLIP_LEFT = 0,
    CLIP_RIGHT,
    CLIP_TOP,
    CLIP_BOTTOM,
} ClipBoundary;

static bool vertex_inside_boundary(const Vertex2D *v, ClipBoundary boundary)
{
    switch (boundary) {
    case CLIP_LEFT:
        return v->x >= VIEW_X_MIN;
    case CLIP_RIGHT:
        return v->x <= VIEW_X_MAX;
    case CLIP_TOP:
        return v->y >= VIEW_Y_MIN;
    case CLIP_BOTTOM:
        return v->y <= VIEW_Y_MAX;
    default:
        return false;
    }
}

static float boundary_value(ClipBoundary boundary)
{
    switch (boundary) {
    case CLIP_LEFT:
        return VIEW_X_MIN;
    case CLIP_RIGHT:
        return VIEW_X_MAX;
    case CLIP_TOP:
        return VIEW_Y_MIN;
    case CLIP_BOTTOM:
        return VIEW_Y_MAX;
    default:
        return 0.0f;
    }
}

static int clip_polygon_against_boundary(const Vertex2D *in, int count,
                                         Vertex2D *out, ClipBoundary boundary)
{
    if (count <= 0)
        return 0;

    int out_count = 0;
    Vertex2D prev = in[count - 1];
    bool prev_inside = vertex_inside_boundary(&prev, boundary);
    float bound = boundary_value(boundary);

```

```
    for (int i = 0; i < count; i++) {
        Vertex2D cur = in[i];
        bool cur_inside = vertex_inside_boundary(&cur, boundary);

        if (prev_inside != cur_inside) {
            float denom;
            float t;
            Vertex2D clipped;

            if (boundary == CLIP_LEFT || boundary == CLIP_RIGHT)
                denom = cur.x - prev.x;
            else
                denom = cur.y - prev.y;

            if (fabsf(denom) < 1e-6f)
                t = 0.0f;
            else if (boundary == CLIP_LEFT || boundary == CLIP_RIGHT)
                t = (bound - prev.x) / denom;
            else
                t = (bound - prev.y) / denom;

            clipped = lerp_vertex2d(&prev, &cur, t);
            if (boundary == CLIP_LEFT || boundary == CLIP_RIGHT)
                clipped.x = bound;
            else
                clipped.y = bound;
            if (out_count >= MAX_VIEW_CLIP_VERTS)
                return 0;
            out[out_count++] = clipped;
        }

        if (cur_inside) {
            if (out_count >= MAX_VIEW_CLIP_VERTS)
                return 0;
            out[out_count++] = cur;
        }

        prev = cur;
        prev_inside = cur_inside;
    }

    return out_count;
}

static int clip_polygon_against_y_bound(const Vertex2D *in, int count,
                                        Vertex2D *out, float bound,
                                        bool keep_greater_equal)
{
    if (count <= 0)
        return 0;

    int out_count = 0;
    Vertex2D prev = in[count - 1];
    bool prev_inside = keep_greater_equal ? (prev.y >= bound) :
                                            (prev.y <= bound);

    for (int i = 0; i < count; i++) {
        Vertex2D cur = in[i];
        bool cur_inside = keep_greater_equal ? (cur.y >= bound) :
```

```
                                                (cur.y <= bound);

        if (prev_inside != cur_inside) {
            float denom = cur.y - prev.y;
            float t = fabsf(denom) < 1e-6f ? 0.0f : (bound - prev.y) / denom;
            Vertex2D clipped = lerp_vertex2d(&prev, &cur, t);

            clipped.y = bound;
            if (out_count >= MAX_VIEW_CLIP_VERTS)
                return 0;
            out[out_count++] = clipped;
        }

        if (cur_inside) {
            if (out_count >= MAX_VIEW_CLIP_VERTS)
                return 0;
            out[out_count++] = cur;
        }

        prev = cur;
        prev_inside = cur_inside;
    }

    return out_count;
}

static int clip_polygon_to_y_range(const Vertex2D *in, int count,
                                   Vertex2D *out, float y_min, float y_max)
{
    Vertex2D tmp[MAX_VIEW_CLIP_VERTS];

    count = clip_polygon_against_y_bound(in, count, tmp, y_min, true);
    if (count == 0)
        return 0;
    count = clip_polygon_against_y_bound(tmp, count, out, y_max, false);
    return count;
}

static int clip_polygon_to_viewport(const Vertex2D in[4], Vertex2D out[MAX_VIEW_CLIP_VERTS])
{
    Vertex2D buf_a[MAX_VIEW_CLIP_VERTS];
    Vertex2D buf_b[MAX_VIEW_CLIP_VERTS];
    int count = 4;

    memcpy(buf_a, in, 4 * sizeof(Vertex2D));

    count = clip_polygon_against_boundary(buf_a, count, buf_b, CLIP_LEFT);
    if (count == 0)
        return 0;
    count = clip_polygon_against_boundary(buf_b, count, buf_a, CLIP_RIGHT);
    if (count == 0)
        return 0;
    count = clip_polygon_against_boundary(buf_a, count, buf_b, CLIP_TOP);
    if (count == 0)
        return 0;
    count = clip_polygon_against_boundary(buf_b, count, buf_a, CLIP_BOTTOM);
    if (count == 0)
        return 0;

```

```
    memcpy(out, buf_a, (size_t)count * sizeof(Vertex2D));
    return count;
}

static uint32_t hash_grid_coord(int x, int y, int z)
{
    return ((uint32_t)x * 73856093u) ^
           ((uint32_t)y * 19349663u) ^
           ((uint32_t)z * 83492791u);
}

static bool ensure_lookup_capacity(RenderContext *ctx, int num_blocks)
{
    int needed = 16;

    while (needed < num_blocks * 4)
        needed <<= 1;

    if (needed <= ctx->lookup_capacity)
        return true;

    LookupEntry *entries = realloc(ctx->lookup_entries,
                                   (size_t)needed * sizeof(*entries));
    if (!entries)
        return false;

    ctx->lookup_entries = entries;
    ctx->lookup_capacity = needed;
    return true;
}

static bool build_block_lookup(RenderContext *ctx, const Block *blocks, int num_blocks,
                               BlockLookup *lookup)
{
    if (!ensure_lookup_capacity(ctx, num_blocks))
        return false;

    memset(ctx->lookup_entries, 0,
           (size_t)ctx->lookup_capacity * sizeof(*ctx->lookup_entries));

    lookup->entries = ctx->lookup_entries;
    lookup->capacity = ctx->lookup_capacity;
    lookup->mask = ctx->lookup_capacity - 1;

    for (int i = 0; i < num_blocks; i++) {
        if (blocks[i].type == BLOCK_AIR)
            continue;

        int x = (int)lroundf(blocks[i].position.x);
        int y = (int)lroundf(blocks[i].position.y);
        int z = (int)lroundf(blocks[i].position.z);
        uint32_t idx = hash_grid_coord(x, y, z) & (uint32_t)lookup->mask;

        while (lookup->entries[idx].occupied) {
            if (lookup->entries[idx].x == x &&
                lookup->entries[idx].y == y &&
                lookup->entries[idx].z == z)
                break;
            idx = (idx + 1u) & (uint32_t)lookup->mask;
```

```
        }

        lookup->entries[idx].x = x;
        lookup->entries[idx].y = y;
        lookup->entries[idx].z = z;
        lookup->entries[idx].type = (uint8_t)blocks[i].type;
        lookup->entries[idx].occupied = 1;
    }

    return true;
}

static BlockID lookup_block_at(const BlockLookup *lookup, Vec3 pos)
{
    int x = (int)lroundf(pos.x);
    int y = (int)lroundf(pos.y);
    int z = (int)lroundf(pos.z);
    uint32_t idx = hash_grid_coord(x, y, z) & (uint32_t)lookup->mask;

    while (lookup->entries[idx].occupied) {
        if (lookup->entries[idx].x == x &&
            lookup->entries[idx].y == y &&
            lookup->entries[idx].z == z)
            return (BlockID)lookup->entries[idx].type;
        idx = (idx + 1u) & (uint32_t)lookup->mask;
    }

    return BLOCK_AIR;
}

static bool is_face_exposed(const Block *block, BlockFace face,
                            const BlockLookup *lookup)
{
    Vec3 neighbor = {
        block->position.x + face_normals[face].x,
        block->position.y + face_normals[face].y,
        block->position.z + face_normals[face].z,
    };

    return block_face_should_render(block->type, lookup_block_at(lookup, neighbor));
}

/* Pack a float edge coefficient into Q24.8 (multiply by 256, round). */
static inline int32_t to_q24_8(float v)
{
    float scaled = v * 256.0f;
    return (int32_t)(scaled >= 0.0f ? scaled + 0.5f : scaled - 0.5f);
}

static inline float snap_q24_8(float v)
{
    return (float)to_q24_8(v) / 256.0f;
}

static inline int floor_div_256_i32(int32_t v)
{
    int q = v / 256;
    int r = v % 256;

```

```
    return (r && v < 0) ? q - 1 : q;
}

static inline int ceil_div_256_i32(int32_t v)
{
    int q = v / 256;
    int r = v % 256;

    return (r && v > 0) ? q + 1 : q;
}

static bool same_screen_xy(const Vertex2D *a, const Vertex2D *b)
{
    return a->x == b->x && a->y == b->y;
}

static int snap_and_compact_polygon(Vertex2D *verts, int count)
{
    Vertex2D compact[MAX_VIEW_CLIP_VERTS];
    int out_count = 0;

    for (int i = 0; i < count; i++) {
        verts[i].x = snap_q24_8(verts[i].x);
        verts[i].y = snap_q24_8(verts[i].y);

        if (out_count > 0 && same_screen_xy(&compact[out_count - 1], &verts[i]))
            continue;

        compact[out_count++] = verts[i];
    }

    if (out_count > 1 && same_screen_xy(&compact[0], &compact[out_count - 1]))
        out_count--;

    memcpy(verts, compact, (size_t)out_count * sizeof(*verts));
    return out_count;
}

static bool split_quads_at_hw_band_edges(void)
{
    static int cached = -1;

    if (cached < 0)
        cached = env_flag("VOXEL_SPLIT_BAND_QUADS", true) ? 1 : 0;
    return cached != 0;
}

static bool split_translucent_quads_at_hw_band_edges(void)
{
    static int cached = -1;

    if (cached < 0)
        cached = env_flag("VOXEL_SPLIT_TRANSLUCENT_QUADS", false) ? 1 : 0;
    return cached != 0;
}

static int textured_world_slice_height_px(void)
{
    static int cached = -1;
```

```
    if (cached < 0) {
        int slice_px = env_int_or_default("VOXEL_TEXTURE_SLICE_PX", 30, 0,
                                          (int)VOXEL_BAND_CACHE_HEIGHT);
        if (slice_px > 0 && slice_px < 4)
            slice_px = 4;
        cached = slice_px;
    }
    return cached;
}

static bool stage_polygon_as_quads(RenderContext *ctx,
                                   const RenderQuad *src,
                                   const Vertex2D *verts,
                                   int count)
{
    Vertex2D compact[MAX_VIEW_CLIP_VERTS];

    if (count > MAX_VIEW_CLIP_VERTS)
        return false;
    memcpy(compact, verts, (size_t)count * sizeof(*compact));
    count = snap_and_compact_polygon(compact, count);
    if (count < 3)
        return false;

    if (count == 3) {
        RenderQuad tri = {
            .vertices = { compact[0], compact[1], compact[2], compact[2] },
            .texture_id = src->texture_id,
            .color_tint = src->color_tint,
            .flags = src->flags,
        };
        return stage_prepared_quad(ctx, tri);
    }

    if (count == 4) {
        RenderQuad quad = {
            .vertices = { compact[0], compact[1], compact[2], compact[3] },
            .texture_id = src->texture_id,
            .color_tint = src->color_tint,
            .flags = src->flags,
        };
        return stage_prepared_quad(ctx, quad);
    }

    bool emitted = false;
    for (int i = 1; i + 1 < count; i++) {
        RenderQuad tri = {
            .vertices = { compact[0], compact[i], compact[i + 1], compact[i + 1] },
            .texture_id = src->texture_id,
            .color_tint = src->color_tint,
            .flags = src->flags,
        };
        if (stage_prepared_quad(ctx, tri))
            emitted = true;
    }
    return emitted;
}
```

```
static bool stage_projected_polygon(RenderContext *ctx,
                                    const RenderQuad *quad,
                                    const Vertex2D *verts,
                                    int count)
{
    if (count < 3)
        return false;

    float y_min = verts[0].y;
    float y_max = verts[0].y;
    for (int i = 1; i < count; i++) {
        if (verts[i].y < y_min) y_min = verts[i].y;
        if (verts[i].y > y_max) y_max = verts[i].y;
    }

    if (!split_quads_at_hw_band_edges())
        return stage_polygon_as_quads(ctx, quad, verts, count);
    if (!(quad->flags & QUAD_FLAG_TEX))
        return stage_polygon_as_quads(ctx, quad, verts, count);

    const bool alpha_blended =
        (quad->flags & QUAD_ALPHA_MASK) != QUAD_ALPHA_OPAQUE;
    if (alpha_blended && !split_translucent_quads_at_hw_band_edges())
        return stage_polygon_as_quads(ctx, quad, verts, count);

    /*
     * The hardware renderer consumes one 60px-tall resident band at a time.
     * Clip in float space before descriptor packing so each per-band descriptor
     * gets its own freshly solved UV/depth planes. This avoids stitching bands
     * together by adding dy to already-quantized Q16.16/Q1.15 starts, which can
     * show up as texture shimmer or a horizontal slice on tall faces. Z-tested
     * textured world geometry can also be split more finely inside each band
     * with VOXEL_TEXTURE_SLICE_PX to reduce fixed-point edge/UV drift.
     */
    const float band_h = (float)VOXEL_BAND_CACHE_HEIGHT;
    const bool ztested_textured = (quad->flags & QUAD_FLAG_ZTEST) != 0;
    const int slice_px = ztested_textured ? textured_world_slice_height_px() : 0;
    const float slice_h = (slice_px > 0) ? (float)slice_px : band_h;
    int first_band = (int)floorf(y_min / band_h);
    int last_band = (int)floorf(y_max / band_h);
    if (first_band < 0) first_band = 0;
    if (last_band < 0) last_band = 0;
    if (first_band >= (int)VOXEL_BAND_COUNT)
        first_band = (int)VOXEL_BAND_COUNT - 1;
    if (last_band >= (int)VOXEL_BAND_COUNT)
        last_band = (int)VOXEL_BAND_COUNT - 1;

    float bucket_y_max = y_max - 1e-4f;
    if (bucket_y_max < y_min)
        bucket_y_max = y_min;
    int first_slice = slice_px > 0 ? (int)floorf(y_min / slice_h) : 0;
    int last_slice = slice_px > 0 ? (int)floorf(bucket_y_max / slice_h) : 0;

    if (first_band == last_band && (slice_px <= 0 || first_slice == last_slice))
        return stage_polygon_as_quads(ctx, quad, verts, count);

    bool emitted = false;
    float emit_y_min = fmaxf(y_min, 0.0f);
    const float emit_y_max = fminf(y_max, (float)VOXEL_RENDER_HEIGHT);
```

```
1326
1327        while (emit_y_min < emit_y_max) {
1328            Vertex2D clipped[MAX_VIEW_CLIP_VERTS];
1329
1330            float next_band_y = (floorf(emit_y_min / band_h) + 1.0f) * band_h;
1331            float emit_y_next = next_band_y;
1332            if (slice_px > 0) {
1333                float next_slice_y = (floorf(emit_y_min / slice_h) + 1.0f) * slice_h;
1334                if (next_slice_y < emit_y_next)
1335                    emit_y_next = next_slice_y;
1336            }
1337            if (emit_y_next > emit_y_max)
1338                emit_y_next = emit_y_max;
1339            if (emit_y_next <= emit_y_min + 1e-4f)
1340                emit_y_next = fminf(emit_y_min + slice_h, emit_y_max);
1341
1342            int clipped_count = clip_polygon_to_y_range(verts, count, clipped,
1343                                                        emit_y_min, emit_y_next);
1344
1345            if (clipped_count >= 3 &&
1346                stage_polygon_as_quads(ctx, quad, clipped, clipped_count))
1347                emitted = true;
1348
1349            emit_y_min = emit_y_next;
1350        }
1351
1352        return emitted;
1353    }
1354
1355    static struct edge_coef make_edge_coef(float x0, float y0, float x1, float y1)
1356    {
1357        struct edge_coef edge;
1358        float A = y0 - y1;
1359        float B = x1 - x0;
1360        float C = -(A * x0 + B * y0);
1361        float dx = x1 - x0;
1362        float dy = y1 - y0;
1363
1364        /*
1365         * The RTL evaluates E(x,y) at integer sample locations. Shift C so those
1366         * integer coordinates correspond to pixel centers (x + 0.5, y + 0.5).
1367         */
1368        C += 0.5f * (A + B);
1369
1370        edge.A = to_q24_8(A);
1371        edge.B = to_q24_8(B);
1372        edge.C = to_q24_8(C);
1373
1374        /*
1375         * Use a top-left fill rule so adjacent quads share edges cleanly.
1376         * For our clockwise screen-space winding in y-down coordinates,
1377         * inclusive edges are:
1378         *   - upward edges
1379         *   - horizontal edges that run left-to-right
1380         *
1381         * Everything else becomes exclusive by subtracting one subpixel LSB.
1382         * This prevents shared-edge cracks and inconsistent side/bottom coverage.
1383         */
1384        if (fabsf(dx) < 1e-6f && fabsf(dy) < 1e-6f)
```

```
        return edge;

    if (!(dy < 0.0f || (fabsf(dy) < 1e-6f && dx > 0.0f)))
        edge.C -= 1;

    return edge;
}

/* Pack a float depth value into Q1.15 unsigned (clamp to [0,2)). */
static inline uint16_t to_q1_15u(float v)
{
    if (v < 0.0f) v = 0.0f;
    if (v > 1.99f) v = 1.99f;
    return (uint16_t)(v * 32768.0f + 0.5f);
}

/* Pack a float depth gradient into Q1.15 signed (clamp to [-1,1)). */
static inline int16_t to_q1_15s(float v)
{
    if (v < -1.0f) v = -1.0f;
    if (v >  0.999f) v = 0.999f;
    {
        float scaled = v * 32768.0f;
        return (int16_t)(scaled >= 0.0f ? scaled + 0.5f : scaled - 0.5f);
    }
}

static inline uint16_t to_q8_8u(float v)
{
    if (v < 0.0f) v = 0.0f;
    if (v > 255.996f) v = 255.996f;
    return (uint16_t)(v * 256.0f + 0.5f);
}

static inline uint16_t to_q0_16u(float v)
{
    if (v < 0.0f) v = 0.0f;
    if (v > 0.9999847f) v = 0.9999847f;
    return (uint16_t)(v * 65536.0f + 0.5f);
}

static inline int32_t to_q16_16(float v)
{
    float scaled = v * 65536.0f;
    return (int32_t)(scaled >= 0.0f ? scaled + 0.5f : scaled - 0.5f);
}

static float quad_signed_area(const Vertex2D v[4])
{
    float area = 0.0f;

    for (int i = 0; i < 4; i++) {
        const Vertex2D *a = &v[i];
        const Vertex2D *b = &v[(i + 1) % 4];
        area += a->x * b->y - b->x * a->y;
    }

    return 0.5f * area;
}
```

```
1444
1445 static void ensure_clockwise_winding(Vertex2D v[4])
1446 {
1447     if (quad_signed_area(v) >= 0.0f)
1448         return;
1449
1450     Vertex2D tmp = v[1];
1451     v[1] = v[3];
1452     v[3] = tmp;
1453 }
1454
1455 /*
1456  * Deterministic triple selection.
1457  *
1458  * For a planar world-space quad, depth (in the 1 - near/z mapping), u/w, v/w
1459  * and 1/w are all exactly linear in screen space, so any 3 of the 4 projected
1460  * vertices should yield the same plane. Clip vertices, Q24.8 snapping and
1461  * floating-point rounding can still produce tiny disagreements between
1462  * triples; picking the first non-degenerate triple in a fixed order removes
1463  * the frame-to-frame jitter where the "winning" triple flipped under
1464  * sub-pixel camera motion.
1465  */
1466 static const int kAttrTriples[4][3] = {
1467     { 0, 1, 2 },
1468     { 0, 1, 3 },
1469     { 0, 2, 3 },
1470     { 1, 2, 3 },
1471 };
1472
1473 typedef struct {
1474     int a;
1475     int b;
1476     int c;
1477     float ax;
1478     float ay;
1479     float bx;
1480     float by;
1481     float det;
1482     bool valid;
1483 } PlaneBasis;
1484
1485 static PlaneBasis choose_plane_basis(const Vertex2D v[4])
1486 {
1487     for (int i = 0; i < 4; i++) {
1488         int a = kAttrTriples[i][0];
1489         int b = kAttrTriples[i][1];
1490         int c = kAttrTriples[i][2];
1491         float ax = v[b].x - v[a].x;
1492         float ay = v[b].y - v[a].y;
1493         float bx = v[c].x - v[a].x;
1494         float by = v[c].y - v[a].y;
1495         float det = ax * by - ay * bx;
1496
1497         if (fabsf(det) > 1e-6f)
1498             return (PlaneBasis){ a, b, c, ax, ay, bx, by, det, true };
1499     }
1500
1501     return (PlaneBasis){ 0 };
1502 }
```

```
1503
1504 static void solve_attr_with_basis(const PlaneBasis *basis,
1505                                   float attr_a, float attr_b, float attr_c,
1506                                   float *ddx, float *ddy)
1507 {
1508     if (!basis->valid) {
1509         *ddx = 0.0f;
1510         *ddy = 0.0f;
1511         return;
1512     }
1513
1514     float av = attr_b - attr_a;
1515     float bv = attr_c - attr_a;
1516
1517     *ddx = ( basis->by * av - basis->ay * bv) / basis->det;
1518     *ddy = (-basis->bx * av + basis->ax * bv) / basis->det;
1519 }
1520
1521 /* Fit a depth plane z(x,y) = z0 + dz_dx*x + dz_dy*y from 3 screen vertices.
1522  * Stores the plane at the requested raster sample point. */
1523 static void fit_depth_plane(const Vertex2D v[4], const PlaneBasis *basis,
1524                             float sample_x, float sample_y,
1525                             uint16_t *z0, int16_t *dz_dx, int16_t *dz_dy)
1526 {
1527     float ddx = 0.0f;
1528     float ddy = 0.0f;
1529     int a = basis->a;
1530
1531     solve_attr_with_basis(basis, v[basis->a].z, v[basis->b].z, v[basis->c].z,
1532                           &ddx, &ddy);
1533
1534     float z_at_origin = v[a].z - ddx * v[a].x - ddy * v[a].y;
1535     float z_at_sample = z_at_origin + ddx * sample_x + ddy * sample_y;
1536
1537     *z0    = to_q1_15u(z_at_sample);
1538     *dz_dx = to_q1_15s(ddx);
1539     *dz_dy = to_q1_15s(ddy);
1540 }
1541
1542 /*
1543  * Fit three screen-space planes for perspective-correct texturing:
1544  *   u/w (x,y), v/w (x,y), 1/w (x,y)
1545  *
1546  * The rasterizer interpolates each plane linearly per pixel and then divides
1547  * to recover true u, v. All three attributes are exactly linear in screen
1548  * space for a planar world-space quad, so a single deterministic 3-vertex fit
1549  * captures the full plane (modulo FP noise).
1550  */
1551 static void fit_uv_plane(const Vertex2D v[4], const PlaneBasis *basis,
1552                          float sample_x, float sample_y,
1553                          struct quad_desc_uv *uv)
1554 {
1555     float duw_dx = 0.0f, duw_dy = 0.0f;
1556     float dvw_dx = 0.0f, dvw_dy = 0.0f;
1557     float diw_dx = 0.0f, diw_dy = 0.0f;
1558     int a = basis->a;
1559
1560     solve_attr_with_basis(basis,
1561                           v[basis->a].u_over_w,
```

```
                          v[basis->b].u_over_w,
                          v[basis->c].u_over_w,
                          &duw_dx, &duw_dy);
    solve_attr_with_basis(basis,
                          v[basis->a].v_over_w,
                          v[basis->b].v_over_w,
                          v[basis->c].v_over_w,
                          &dvw_dx, &dvw_dy);
    solve_attr_with_basis(basis,
                          v[basis->a].one_over_w,
                          v[basis->b].one_over_w,
                          v[basis->c].one_over_w,
                          &diw_dx, &diw_dy);

    float uw_origin = v[a].u_over_w   - duw_dx * v[a].x - duw_dy * v[a].y;
    float vw_origin = v[a].v_over_w   - dvw_dx * v[a].x - dvw_dy * v[a].y;
    float iw_origin = v[a].one_over_w - diw_dx * v[a].x - diw_dy * v[a].y;
    float uw_sample = uw_origin + duw_dx * sample_x + duw_dy * sample_y;
    float vw_sample = vw_origin + dvw_dx * sample_x + dvw_dy * sample_y;
    float iw_sample = iw_origin + diw_dx * sample_x + diw_dy * sample_y;

    uv->u_over_w_0    = to_q16_16(uw_sample);
    uv->u_over_w_dx   = to_q16_16(duw_dx);
    uv->u_over_w_dy   = to_q16_16(duw_dy);
    uv->v_over_w_0    = to_q16_16(vw_sample);
    uv->v_over_w_dx   = to_q16_16(dvw_dx);
    uv->v_over_w_dy   = to_q16_16(dvw_dy);
    uv->one_over_w_0  = to_q16_16(iw_sample);
    uv->one_over_w_dx = to_q16_16(diw_dx);
    uv->one_over_w_dy = to_q16_16(diw_dy);
}

/* Conservative software occlusion:
 * a tile is considered covered only when one opaque, z-tested quad contains
 * all four extreme pixel centers. A later quad is culled only when every tile
 * in its bbox is covered by an occluder whose farthest vertex is still closer
 * than the candidate's nearest vertex. */
static void occlusion_reset(RenderContext *ctx)
{
    memset(ctx->occlusion_covered, 0, sizeof(ctx->occlusion_covered));
    ctx->occlusion_pass = OCCLUSION_PASS_DISABLED;
    ctx->occlusion_tested = 0;
    ctx->occlusion_culled = 0;
    ctx->occlusion_tiles_written = 0;
}

static bool occlusion_quad_eligible(uint8_t flags)
{
    return (flags & QUAD_FLAG_ZTEST) &&
           !(flags & QUAD_FLAG_ALPHA_KEY) &&
           ((flags & QUAD_ALPHA_MASK) == QUAD_ALPHA_OPAQUE);
}

static int occlusion_tile_index(int tx, int ty)
{
    return ty * (int)OCCLUSION_TILES_X + tx;
}

static bool point_inside_convex_quad(const Vertex2D v[4], float px, float py)
```

```
{
    bool saw_positive = false;
    bool saw_negative = false;

    for (int i = 0; i < 4; i++) {
        const Vertex2D *a = &v[i];
        const Vertex2D *b = &v[(i + 1) & 3];
        float dx = b->x - a->x;
        float dy = b->y - a->y;
        float cross;

        if (fabsf(dx) < 1e-5f && fabsf(dy) < 1e-5f)
            continue;

        cross = dx * (py - a->y) - dy * (px - a->x);
        if (fabsf(cross) <= 1e-4f)
            return false;
        if (cross > 0.0f)
            saw_positive = true;
        else
            saw_negative = true;

        if (saw_positive && saw_negative)
            return false;
    }

    return true;
}

static float quad_min_depth(const Vertex2D v[4])
{
    float z = v[0].z;

    for (int i = 1; i < 4; i++) {
        if (v[i].z < z)
            z = v[i].z;
    }
    return z;
}

static float quad_max_depth(const Vertex2D v[4])
{
    float z = v[0].z;

    for (int i = 1; i < 4; i++) {
        if (v[i].z > z)
            z = v[i].z;
    }
    return z;
}

static void occlusion_tile_range(int x_min, int y_min, int x_max, int y_max,
                                 int *tx_min, int *ty_min,
                                 int *tx_max, int *ty_max)
{
    *tx_min = x_min / OCCLUSION_TILE_SIZE;
    *ty_min = y_min / OCCLUSION_TILE_SIZE;
    *tx_max = x_max / OCCLUSION_TILE_SIZE;
    *ty_max = y_max / OCCLUSION_TILE_SIZE;
```

```
    if (*tx_min < 0) *tx_min = 0;
    if (*ty_min < 0) *ty_min = 0;
    if (*tx_max >= (int)OCCLUSION_TILES_X)
        *tx_max = (int)OCCLUSION_TILES_X - 1;
    if (*ty_max >= (int)OCCLUSION_TILES_Y)
        *ty_max = (int)OCCLUSION_TILES_Y - 1;
}

static bool occlusion_quad_hidden(RenderContext *ctx, const Vertex2D v[4],
                                  int x_min, int y_min, int x_max, int y_max)
{
    int tx_min, ty_min, tx_max, ty_max;
    float nearest_z = quad_min_depth(v);

    occlusion_tile_range(x_min, y_min, x_max, y_max,
                         &tx_min, &ty_min, &tx_max, &ty_max);

    for (int ty = ty_min; ty <= ty_max; ty++) {
        for (int tx = tx_min; tx <= tx_max; tx++) {
            int idx = occlusion_tile_index(tx, ty);

            if (!ctx->occlusion_covered[idx])
                return false;
            if (ctx->occlusion_depth[idx] + OCCLUSION_DEPTH_EPSILON >= nearest_z)
                return false;
        }
    }

    return true;
}

static bool occlusion_quad_covers_tile(const Vertex2D v[4], int tx, int ty)
{
    float x0 = (float)(tx * OCCLUSION_TILE_SIZE) + 0.5f;
    float y0 = (float)(ty * OCCLUSION_TILE_SIZE) + 0.5f;
    float x1 = (float)((tx + 1) * OCCLUSION_TILE_SIZE) - 0.5f;
    float y1 = (float)((ty + 1) * OCCLUSION_TILE_SIZE) - 0.5f;

    if (x1 > SCREEN_WIDTH - 0.5f)
        x1 = SCREEN_WIDTH - 0.5f;
    if (y1 > SCREEN_HEIGHT - 0.5f)
        y1 = SCREEN_HEIGHT - 0.5f;

    return point_inside_convex_quad(v, x0, y0) &&
           point_inside_convex_quad(v, x1, y0) &&
           point_inside_convex_quad(v, x0, y1) &&
           point_inside_convex_quad(v, x1, y1);
}

static void occlusion_record_quad(RenderContext *ctx, const Vertex2D v[4],
                                  int x_min, int y_min, int x_max, int y_max)
{
    int tx_min, ty_min, tx_max, ty_max;
    float farthest_z = quad_max_depth(v);

    occlusion_tile_range(x_min, y_min, x_max, y_max,
                         &tx_min, &ty_min, &tx_max, &ty_max);

```

```
    for (int ty = ty_min; ty <= ty_max; ty++) {
        for (int tx = tx_min; tx <= tx_max; tx++) {
            int idx;

            if (!occlusion_quad_covers_tile(v, tx, ty))
                continue;

            idx = occlusion_tile_index(tx, ty);
            if (!ctx->occlusion_covered[idx] ||
                farthest_z < ctx->occlusion_depth[idx]) {
                ctx->occlusion_covered[idx] = 1;
                ctx->occlusion_depth[idx] = farthest_z;
                ctx->occlusion_tiles_written++;
            }
        }
    }
}

static bool projected_quad_fully_inside_viewport(const Vertex2D v[4])
{
    for (int i = 0; i < 4; i++) {
        if (v[i].x < VIEW_X_MIN || v[i].x > VIEW_X_MAX ||
            v[i].y < VIEW_Y_MIN || v[i].y > VIEW_Y_MAX)
            return false;
    }
    return true;
}

static bool stage_projected_quad_no_clip(RenderContext *ctx,
                                         const RenderQuad *quad)
{
    Vertex2D verts[4];

    memcpy(verts, quad->vertices, sizeof(verts));
    return stage_projected_polygon(ctx, quad, verts, 4);
}

static bool emit_camera_quad(RenderContext *ctx, const CameraVertex verts_cam[4],
                             uint8_t texture_id, uint8_t extra_flags)
{
    Vertex2D verts[4];
    uint8_t flags = QUAD_FLAG_ZTEST | QUAD_FLAG_TEX | extra_flags;

    for (int i = 0; i < 4; i++) {
        if (!project_camera_vertex(ctx, &verts_cam[i], &verts[i]))
            return false;
    }

    /* Most world quads are already on-screen; avoid the general clipper. */
    if (projected_quad_fully_inside_viewport(verts)) {
        RenderQuad q = {0};

        memcpy(q.vertices, verts, sizeof(verts));
        q.texture_id = texture_id;
        q.flags = flags;
        return stage_projected_quad_no_clip(ctx, &q);
    }

    RenderQuad q = {0};
```

```
    memcpy(q.vertices, verts, sizeof(verts));
    q.texture_id = texture_id;
    q.flags = flags;
    return renderer_push_quad(ctx, &q);
}

static bool emit_camera_quad_color(RenderContext *ctx,
                                   const CameraVertex verts_cam[4],
                                   uint8_t color,
                                   uint8_t extra_flags)
{
    Vertex2D verts[4];
    uint8_t flags = QUAD_FLAG_ZTEST | extra_flags;

    for (int i = 0; i < 4; i++) {
        if (!project_camera_vertex(ctx, &verts_cam[i], &verts[i]))
            return false;
    }

    if (projected_quad_fully_inside_viewport(verts)) {
        RenderQuad q = {0};

        memcpy(q.vertices, verts, sizeof(verts));
        q.color_tint = color;
        q.flags = flags;
        return stage_projected_quad_no_clip(ctx, &q);
    }

    RenderQuad q = {0};
    memcpy(q.vertices, verts, sizeof(verts));
    q.color_tint = color;
    q.flags = flags;
    return renderer_push_quad(ctx, &q);
}

static void emit_clipped_face(RenderContext *ctx, const CameraVertex *poly,
                              int count, uint8_t texture_id, uint8_t extra_flags)
{
    if (count < 3)
        return;

    if (count == 3) {
        CameraVertex tri[4] = { poly[0], poly[1], poly[2], poly[2] };
        emit_camera_quad(ctx, tri, texture_id, extra_flags);
        return;
    }

    if (count == 4) {
        CameraVertex quad[4] = { poly[0], poly[1], poly[2], poly[3] };
        emit_camera_quad(ctx, quad, texture_id, extra_flags);
        return;
    }

    for (int i = 1; i + 1 < count; i++) {
        CameraVertex tri[4] = { poly[0], poly[i], poly[i + 1], poly[i + 1] };
        emit_camera_quad(ctx, tri, texture_id, extra_flags);
    }
}

```

```c
static void emit_clipped_face_color(RenderContext *ctx,
                                    const CameraVertex *poly,
                                    int count,
                                    uint8_t color,
                                    uint8_t extra_flags)
{
    if (count < 3)
        return;

    if (count == 3) {
        CameraVertex tri[4] = { poly[0], poly[1], poly[2], poly[2] };
        emit_camera_quad_color(ctx, tri, color, extra_flags);
        return;
    }

    if (count == 4) {
        CameraVertex quad[4] = { poly[0], poly[1], poly[2], poly[3] };
        emit_camera_quad_color(ctx, quad, color, extra_flags);
        return;
    }

    for (int i = 1; i + 1 < count; i++) {
        CameraVertex tri[4] = { poly[0], poly[i], poly[i + 1], poly[i + 1] };
        emit_camera_quad_color(ctx, tri, color, extra_flags);
    }
}

static uint8_t choose_face_texture_lod(const RenderContext *ctx,
                                       uint8_t base_tile,
                                       const CameraVertex face_cam[4])
{
    (void)ctx;
    static int texture_lod_enabled = -1;

    if (texture_lod_enabled < 0)
        texture_lod_enabled = env_flag("VOXEL_TEXTURE_LOD", true) ? 1 : 0;
    if (!texture_lod_enabled)
        return base_tile;

    float nearest_z = face_cam[0].z;
    int lod = 0;

    for (int i = 1; i < 4; i++) {
        if (face_cam[i].z < nearest_z)
            nearest_z = face_cam[i].z;
    }

    /*
     * Use a conservative forward-distance rule instead of projected-span
     * math. The old span-based selector was more expensive and could choose
     * low-res tiles for nearby but highly oblique faces because one projected
     * axis collapsed. These thresholds push mip usage farther out so nearby
     * surfaces stay crisp and runtime overhead stays tiny.
     */
    if (nearest_z > 44.0f)
        lod = 2;
    else if (nearest_z > 28.0f)
        lod = 1;

```

```
    return texture_lod_tile_id(base_tile, lod);
}

static uint8_t choose_face_light_flags(const RenderContext *ctx, BlockFace face)
{
    return ctx->face_light_flags[face];
}

static void merged_face_vertices(Vec3 block_pos, BlockFace face,
                                 int u_size, int v_size,
                                 uint8_t height_eighths, Vec3 out[4])
{
    float u = (float)u_size;
    float v = (float)v_size;
    float x = block_pos.x;
    float y = block_pos.y;
    float z = block_pos.z;
    /* "Top" Y of the face. For full-height blocks (height_eighths == 8)
     * this is exactly y + v, which keeps the existing greedy-merge math
     * untouched. For partial-height water (height_eighths < 8) v_size is
     * guaranteed to be 1 (translucent never merges), so the partial top
     * sits at y + height/8. */
    float top_y = y + (v - 1.0f) + (float)height_eighths * (1.0f / 8.0f);
    float top_y_only = y + (float)height_eighths * (1.0f / 8.0f);

    switch (face) {
    case FACE_TOP:
        out[0] = (Vec3){ x,     top_y_only, z + v };
        out[1] = (Vec3){ x + u, top_y_only, z + v };
        out[2] = (Vec3){ x + u, top_y_only, z     };
        out[3] = (Vec3){ x,     top_y_only, z     };
        break;
    case FACE_BOTTOM:
        out[0] = (Vec3){ x,     y, z     };
        out[1] = (Vec3){ x + u, y, z     };
        out[2] = (Vec3){ x + u, y, z + v };
        out[3] = (Vec3){ x,     y, z + v };
        break;
    case FACE_LEFT:
        out[0] = (Vec3){ x, y,     z     };
        out[1] = (Vec3){ x, y,     z + u };
        out[2] = (Vec3){ x, top_y, z + u };
        out[3] = (Vec3){ x, top_y, z     };
        break;
    case FACE_RIGHT:
        out[0] = (Vec3){ x + 1, y,     z + u };
        out[1] = (Vec3){ x + 1, y,     z     };
        out[2] = (Vec3){ x + 1, top_y, z     };
        out[3] = (Vec3){ x + 1, top_y, z + u };
        break;
    case FACE_FRONT:
        out[0] = (Vec3){ x,     y,     z };
        out[1] = (Vec3){ x + u, y,     z };
        out[2] = (Vec3){ x + u, top_y, z };
        out[3] = (Vec3){ x,     top_y, z };
        break;
    case FACE_BACK:
        out[0] = (Vec3){ x + u, y,     z + 1 };
        out[1] = (Vec3){ x,     y,     z + 1 };
```

```
        out[2] = (Vec3){ x,     top_y, z + 1 };
        out[3] = (Vec3){ x + u, top_y, z + 1 };
        break;
    default:
        for (int i = 0; i < 4; i++)
            out[i] = block_pos;
        break;
    }
}

/*
 * Runtime toggle for merged quad emission with QUAD_TEX_REPEAT_UV. Default ON:
 * far chunks were greedily meshed specifically to keep the descriptor stream
 * small enough for steady input/frame pacing. Set VOXEL_MERGE_FAR_QUADS=0 to
 * fall back to unit-quad expansion for visual A/B testing.
 */
static bool merged_emit_repeat_uv_enabled(void)
{
    static int cached = -1;

    if (cached < 0)
        cached = env_flag_fallback("VOXEL_MERGE_FAR_QUADS",
                                   "BLOCK_GAME_MERGE_FAR_QUADS",
                                   true) ? 1 : 0;
    return cached != 0;
}

static Vec3 merged_face_sub_origin(Vec3 block_pos, BlockFace face,
                                   int du, int dv)
{
    switch (face) {
    case FACE_TOP:
    case FACE_BOTTOM:
        block_pos.x += (float)du;
        block_pos.z += (float)dv;
        break;
    case FACE_LEFT:
    case FACE_RIGHT:
        block_pos.z += (float)du;
        block_pos.y += (float)dv;
        break;
    case FACE_FRONT:
    case FACE_BACK:
        block_pos.x += (float)du;
        block_pos.y += (float)dv;
        break;
    default:
        break;
    }
    return block_pos;
}

static void emit_merged_block_face_lit(RenderContext *ctx, BlockID type,
                                       Vec3 block_pos, BlockFace face,
                                       int u_size, int v_size,
                                       uint8_t height_eighths,
                                       uint8_t light_flags);

static void expand_merged_face_to_unit_quads(RenderContext *ctx, BlockID type,
```

```
                                          Vec3 block_pos, BlockFace face,
                                          int u_size, int v_size,
                                          uint8_t light_flags)
{
    for (int dv = 0; dv < v_size; dv++) {
        for (int du = 0; du < u_size; du++) {
            Vec3 unit = merged_face_sub_origin(block_pos, face, du, dv);
            emit_merged_block_face_lit(ctx, type, unit, face,
                                       1, 1, 8, light_flags);
            if (ctx->n_quads >= MAX_QUADS_IN_FLIGHT)
                return;
        }
    }
}

static void emit_merged_block_face_lit(RenderContext *ctx, BlockID type,
                                       Vec3 block_pos, BlockFace face,
                                       int u_size, int v_size,
                                       uint8_t height_eighths,
                                       uint8_t light_flags)
{
    static const float tile_span = 16.0f;
    Vec3 face_world[4];
    CameraVertex face_cam[4];
    uint8_t face_flags;
    uint8_t texture_id;
    uint8_t base_tile;

    if (type == BLOCK_AIR)
        return;
    if (face < 0 || face >= NUM_FACES)
        return;

    /*
     * A/B fallback: decomposing a merged run avoids QUAD_TEX_REPEAT_UV entirely,
     * but it is intentionally not the default because it can multiply the far
     * terrain descriptor count and make input pacing noticeably worse.
     */
    if ((u_size > 1 || v_size > 1) && !merged_emit_repeat_uv_enabled()) {
        /*
         * Unit emits below re-run is_face_visible() with their own centers,
         * so we don't pre-cull the anchor here: a merged run can straddle a
         * face whose center is behind the camera while individual cells are
         * not, and vice versa. The per-unit check is the same one used in
         * the non-merged path.
         */
        expand_merged_face_to_unit_quads(ctx, type, block_pos, face,
                                         u_size, v_size, light_flags);
        return;
    }

    if (!merged_face_visible(block_pos, face, u_size, v_size,
                             height_eighths, ctx->current_camera.position))
        return;

    merged_face_vertices(block_pos, face, u_size, v_size, height_eighths, face_world);
    for (int i = 0; i < 4; i++) {
        world_to_camera(ctx, face_world[i], &face_cam[i]);
        face_cam[i].u = (i == 1 || i == 2) ? tile_span * (float)u_size : 0.0f;
```

```
        /*
         * Atlas row 0 sits at the top of each tile image, and the side faces'
         * vertex 0 is world-bottom. Invert V so world-top maps to texture-top.
         */
        face_cam[i].v = (i == 2 || i == 3) ? 0.0f : tile_span * (float)v_size;
    }
    if (camera_quad_outside_view(ctx, face_cam))
        return;

    base_tile = block_face_texture_id(type, face);
    texture_id = choose_face_texture_lod(ctx, base_tile, face_cam);
    if (u_size > 1 || v_size > 1)
        texture_id |= QUAD_TEX_REPEAT_UV;
    face_flags = light_flags | QUAD_FLAG_FOG;
    /*
     * Translucent blocks (glass) ride the hardware alpha-blend path. 50% src
     * / 50% dst gives an obviously see-through look while still leaving the
     * glass texture itself clearly readable. We keep ZTEST on so the glass
     * occludes things behind it at the correct depth; because chunks are
     * drawn nearest-first the destination pixel will usually already hold an
     * opaque fragment that the blend can mix against.
     */
    if (block_is_translucent(type))
        face_flags |= QUAD_ALPHA_50;
    /*
     * Cutout textures (leaves, flowers, doors, torches) carry palette index 0
     * for their holes. Without ALPHA_KEY those texels render as the palette-0
     * background color and look like solid sky-tinted blobs instead of holes.
     */
    if (block_is_alpha_keyed(type))
        face_flags |= QUAD_FLAG_ALPHA_KEY;

    CameraVertex clipped[6];
    int clipped_count = clip_face_to_near_plane(face_cam, clipped);
    emit_clipped_face(ctx, clipped, clipped_count, texture_id, face_flags);
}

static void cross_face_vertices(Vec3 block_pos, uint8_t face,
                                Vec3 out[4])
{
    const float inset = 0.08f;
    float x = block_pos.x;
    float y = block_pos.y;
    float z = block_pos.z;

    if (face == CHUNK_FACE_CROSS_B) {
        out[0] = (Vec3){ x + 1.0f - inset, y,        z + inset };
        out[1] = (Vec3){ x + inset,        y,        z + 1.0f - inset };
        out[2] = (Vec3){ x + inset,        y + 1.0f, z + 1.0f - inset };
        out[3] = (Vec3){ x + 1.0f - inset, y + 1.0f, z + inset };
        return;
    }

    out[0] = (Vec3){ x + inset,        y,        z + inset };
    out[1] = (Vec3){ x + 1.0f - inset, y,        z + 1.0f - inset };
    out[2] = (Vec3){ x + 1.0f - inset, y + 1.0f, z + 1.0f - inset };
    out[3] = (Vec3){ x + inset,        y + 1.0f, z + inset };
}

```

```
static void emit_cross_block_face_lit(RenderContext *ctx, BlockID type,
                                      Vec3 block_pos, uint8_t face,
                                      uint8_t light_flags)
{
    static const float tile_span = 16.0f;
    Vec3 face_world[4];
    CameraVertex face_cam[4];
    uint8_t texture_id;
    uint8_t face_flags = light_flags | QUAD_FLAG_FOG | QUAD_FLAG_ALPHA_KEY;

    if (type == BLOCK_AIR)
        return;

    cross_face_vertices(block_pos, face, face_world);
    for (int i = 0; i < 4; i++) {
        world_to_camera(ctx, face_world[i], &face_cam[i]);
        face_cam[i].u = (i == 1 || i == 2) ? tile_span : 0.0f;
        face_cam[i].v = (i == 2 || i == 3) ? 0.0f : tile_span;
    }
    if (camera_quad_outside_view(ctx, face_cam))
        return;

    texture_id = choose_face_texture_lod(ctx,
                                         block_face_texture_id(type, FACE_FRONT),
                                         face_cam);

    CameraVertex clipped[6];
    int clipped_count = clip_face_to_near_plane(face_cam, clipped);
    emit_clipped_face(ctx, clipped, clipped_count, texture_id, face_flags);
}

static void emit_model_quad_lit(RenderContext *ctx,
                                Vec3 face_world[4], uint8_t texture_tile,
                                uint8_t light_flags);
static void emit_model_quad_lit_rotated(RenderContext *ctx,
                                        Vec3 face_world[4],
                                        uint8_t texture_tile,
                                        uint8_t light_flags,
                                        int uv_rotation);

static void flat_face_vertices(Vec3 block_pos,
                               BlockID type,
                               Vec3 out[4])
{
    float inset = 0.05f;
    float y = block_pos.y + 0.025f;

    if (type == BLOCK_BUTTON) {
        inset = 0.24f;
        y = block_pos.y + 0.085f;
    } else if (type == BLOCK_BUTTON_PRESSED) {
        inset = 0.29f;
        y = block_pos.y + 0.025f;
    } else if (block_is_pressure_plate(type) &&
               !block_pressure_plate_powered(type)) {
        inset = 0.08f;
        y = block_pos.y + 0.055f;
    } else if (block_pressure_plate_powered(type)) {
        inset = 0.08f;
```

```
        y = block_pos.y + 0.015f;
    }

    out[0] = (Vec3){ block_pos.x + inset,        y,
                     block_pos.z + inset };
    out[1] = (Vec3){ block_pos.x + 1.0f - inset, y,
                     block_pos.z + inset };
    out[2] = (Vec3){ block_pos.x + 1.0f - inset, y,
                     block_pos.z + 1.0f - inset };
    out[3] = (Vec3){ block_pos.x + inset,        y,
                     block_pos.z + 1.0f - inset };
}

enum {
    REDSTONE_WIRE_N = 1u << 0,
    REDSTONE_WIRE_E = 1u << 1,
    REDSTONE_WIRE_S = 1u << 2,
    REDSTONE_WIRE_W = 1u << 3,
};

static bool render_block_is_redstone_wire(BlockID id)
{
    return id == BLOCK_REDSTONE_WIRE_UNCONNECTED ||
           id == BLOCK_REDSTONE_WIRE_OFF ||
           id == BLOCK_REDSTONE_WIRE_ON;
}

static bool render_block_is_redstone_component(BlockID id)
{
    return render_block_is_redstone_wire(id) ||
           id == BLOCK_REDSTONE_TORCH_OFF ||
           id == BLOCK_REDSTONE_TORCH_ON ||
           block_is_repeater(id) ||
           block_is_comparator(id) ||
           block_is_lever(id) ||
           block_is_pressure_plate(id) ||
           id == BLOCK_LAMP_OFF ||
           id == BLOCK_LAMP ||
           id == BLOCK_REDSTONE_BLOCK ||
           id == BLOCK_BUTTON ||
           id == BLOCK_BUTTON_PRESSED;
}

static bool render_redstone_wire_connects_to(BlockID id)
{
    if (render_block_is_redstone_component(id))
        return true;
    return id != BLOCK_AIR &&
           block_render_model(id) == BLOCK_RENDER_CUBE &&
           !block_is_passable(id);
}

static uint8_t render_redstone_wire_mask(const VoxelWorld *world,
                                         int wx,
                                         int wy,
                                         int wz)
{
    uint8_t mask = 0;

```

```
    if (render_redstone_wire_connects_to(world_get_block(world, wx, wy, wz - 1)))
        mask |= REDSTONE_WIRE_N;
    if (render_redstone_wire_connects_to(world_get_block(world, wx + 1, wy, wz)))
        mask |= REDSTONE_WIRE_E;
    if (render_redstone_wire_connects_to(world_get_block(world, wx, wy, wz + 1)))
        mask |= REDSTONE_WIRE_S;
    if (render_redstone_wire_connects_to(world_get_block(world, wx - 1, wy, wz)))
        mask |= REDSTONE_WIRE_W;
    return mask;
}

static int redstone_mask_bit_count(uint8_t mask)
{
    int count = 0;

    for (int bit = 0; bit < 4; bit++) {
        if (mask & (1u << bit))
            count++;
    }
    return count;
}

static uint8_t redstone_wire_texture_for_mask(BlockID type,
                                              uint8_t mask,
                                              int *uv_rotation_out)
{
    bool powered = type == BLOCK_REDSTONE_WIRE_ON;
    int count = redstone_mask_bit_count(mask);

    if (uv_rotation_out)
        *uv_rotation_out = 0;

    if (count == 0)
        return powered ? TEX_TILE_REDSTONE_WIRE_DOT_ON :
                         TEX_TILE_REDSTONE_WIRE_UNCONNECTED;

    if (count == 1 ||
        mask == (REDSTONE_WIRE_E | REDSTONE_WIRE_W) ||
        mask == (REDSTONE_WIRE_N | REDSTONE_WIRE_S)) {
        if ((mask & (REDSTONE_WIRE_N | REDSTONE_WIRE_S)) &&
            !(mask & (REDSTONE_WIRE_E | REDSTONE_WIRE_W)) &&
            uv_rotation_out)
            *uv_rotation_out = 1;
        return powered ? TEX_TILE_REDSTONE_WIRE_ON :
                         TEX_TILE_REDSTONE_WIRE_OFF;
    }

    if (count == 2) {
        if (uv_rotation_out) {
            /*
             * Flat top-face UVs map texture north to world south at rotation
             * zero, so the canonical texture N+E corner appears as S+E until
             * rotated.
             */
            if (mask == (REDSTONE_WIRE_N | REDSTONE_WIRE_E))
                *uv_rotation_out = 1;
            else if (mask == (REDSTONE_WIRE_W | REDSTONE_WIRE_N))
                *uv_rotation_out = 2;
            else if (mask == (REDSTONE_WIRE_S | REDSTONE_WIRE_W))
```

```
                *uv_rotation_out = 3;
        }
        return powered ? TEX_TILE_REDSTONE_WIRE_CORNER_ON :
                         TEX_TILE_REDSTONE_WIRE_CORNER_OFF;
    }

    if (count == 3) {
        if (uv_rotation_out) {
            uint8_t missing = (uint8_t)(~mask & 0x0f);

            if (missing & REDSTONE_WIRE_W)
                *uv_rotation_out = 1;
            else if (missing & REDSTONE_WIRE_S)
                *uv_rotation_out = 2;
            else if (missing & REDSTONE_WIRE_E)
                *uv_rotation_out = 3;
        }
        return powered ? TEX_TILE_REDSTONE_WIRE_T_ON :
                         TEX_TILE_REDSTONE_WIRE_T_OFF;
    }

    return powered ? TEX_TILE_REDSTONE_WIRE_CROSS_ON :
                     TEX_TILE_REDSTONE_WIRE_CROSS_OFF;
}

static int redstone_directional_uv_rotation(BlockID type)
{
    /*
     * Directional redstone textures in the atlas point toward their bottom edge.
     * Repeater source textures are rotated into that convention by the atlas
     * generator so repeaters and comparators can share this map.
     */
    switch (block_redstone_facing(type)) {
    case BLOCK_DOOR_FACING_EAST:
        return 3;
    case BLOCK_DOOR_FACING_SOUTH:
        return 2;
    case BLOCK_DOOR_FACING_WEST:
        return 1;
    case BLOCK_DOOR_FACING_NORTH:
    default:
        return 0;
    }
}

static Vec3 redstone_device_local_point(Vec3 block_pos,
                                        BlockDoorFacing facing,
                                        float side,
                                        float forward,
                                        float y)
{
    int fdx;
    int fdz;
    float rdx;
    float rdz;

    switch (facing) {
    case BLOCK_DOOR_FACING_EAST:
        fdx = 1;
```

```
        fdz = 0;
        break;
    case BLOCK_DOOR_FACING_SOUTH:
        fdx = 0;
        fdz = 1;
        break;
    case BLOCK_DOOR_FACING_WEST:
        fdx = -1;
        fdz = 0;
        break;
    case BLOCK_DOOR_FACING_NORTH:
    default:
        fdx = 0;
        fdz = -1;
        break;
    }

    rdx = (float)-fdz;
    rdz = (float)fdx;
    return (Vec3){
        block_pos.x + 0.5f + rdx * side + (float)fdx * forward,
        block_pos.y + y,
        block_pos.z + 0.5f + rdz * side + (float)fdz * forward,
    };
}

static void redstone_device_plate_top_vertices(Vec3 block_pos, Vec3 out[4])
{
    const float inset = REDSTONE_DEVICE_PLATE_INSET;
    const float y = 0.080f;

    out[0] = (Vec3){ block_pos.x + inset,        block_pos.y + y,
                     block_pos.z + inset };
    out[1] = (Vec3){ block_pos.x + 1.0f - inset, block_pos.y + y,
                     block_pos.z + inset };
    out[2] = (Vec3){ block_pos.x + 1.0f - inset, block_pos.y + y,
                     block_pos.z + 1.0f - inset };
    out[3] = (Vec3){ block_pos.x + inset,        block_pos.y + y,
                     block_pos.z + 1.0f - inset };
}

static void emit_redstone_device_plate_side(RenderContext *ctx,
                                            Vec3 block_pos,
                                            BlockDoorFacing facing,
                                            int side_index,
                                            uint8_t light_flags)
{
    const float half_w = REDSTONE_DEVICE_PLATE_HALF;
    const float half_l = REDSTONE_DEVICE_PLATE_HALF;
    const float bottom = 0.018f;
    const float top = 0.080f;
    Vec3 face_world[4];

    switch (side_index) {
    case 0:
        face_world[0] = redstone_device_local_point(block_pos, facing,
                                                    -half_w, -half_l, bottom);
        face_world[1] = redstone_device_local_point(block_pos, facing,
                                                     half_w, -half_l, bottom);
```

```
        face_world[2] = redstone_device_local_point(block_pos, facing,
                                                    half_w, -half_l, top);
        face_world[3] = redstone_device_local_point(block_pos, facing,
                                                   -half_w, -half_l, top);
        break;
    case 1:
        face_world[0] = redstone_device_local_point(block_pos, facing,
                                                    half_w, -half_l, bottom);
        face_world[1] = redstone_device_local_point(block_pos, facing,
                                                    half_w,  half_l, bottom);
        face_world[2] = redstone_device_local_point(block_pos, facing,
                                                    half_w,  half_l, top);
        face_world[3] = redstone_device_local_point(block_pos, facing,
                                                    half_w, -half_l, top);
        break;
    case 2:
        face_world[0] = redstone_device_local_point(block_pos, facing,
                                                    half_w,  half_l, bottom);
        face_world[1] = redstone_device_local_point(block_pos, facing,
                                                   -half_w,  half_l, bottom);
        face_world[2] = redstone_device_local_point(block_pos, facing,
                                                   -half_w,  half_l, top);
        face_world[3] = redstone_device_local_point(block_pos, facing,
                                                    half_w,  half_l, top);
        break;
    default:
        face_world[0] = redstone_device_local_point(block_pos, facing,
                                                   -half_w,  half_l, bottom);
        face_world[1] = redstone_device_local_point(block_pos, facing,
                                                   -half_w, -half_l, bottom);
        face_world[2] = redstone_device_local_point(block_pos, facing,
                                                   -half_w, -half_l, top);
        face_world[3] = redstone_device_local_point(block_pos, facing,
                                                   -half_w,  half_l, top);
        break;
    }

    emit_model_quad_lit(ctx, face_world, TEX_TILE_STONE, light_flags);
}

static void emit_redstone_device_post(RenderContext *ctx,
                                      Vec3 block_pos,
                                      BlockDoorFacing facing,
                                      float side,
                                      float forward,
                                      uint8_t texture_tile,
                                      uint8_t light_flags)
{
    const float half = REDSTONE_DEVICE_POST_HALF;
    const float bottom = 0.080f;
    const float top = 0.430f;
    Vec3 face_world[4];

    face_world[0] = redstone_device_local_point(block_pos, facing,
                                                side - half, forward, bottom);
    face_world[1] = redstone_device_local_point(block_pos, facing,
                                                side + half, forward, bottom);
    face_world[2] = redstone_device_local_point(block_pos, facing,
                                                side + half, forward, top);
```

```
    face_world[3] = redstone_device_local_point(block_pos, facing,
                                                side - half, forward, top);
    emit_model_quad_lit(ctx, face_world, texture_tile, light_flags);

    face_world[0] = redstone_device_local_point(block_pos, facing,
                                                side, forward - half, bottom);
    face_world[1] = redstone_device_local_point(block_pos, facing,
                                                side, forward + half, bottom);
    face_world[2] = redstone_device_local_point(block_pos, facing,
                                                side, forward + half, top);
    face_world[3] = redstone_device_local_point(block_pos, facing,
                                                side, forward - half, top);
    emit_model_quad_lit(ctx, face_world, texture_tile, light_flags);
}

static float repeater_moving_post_forward(uint8_t delay_ticks)
{
    static const float forward_texels_by_delay[4] = {
        4.0f, 2.0f, 0.0f, -2.0f,
    };

    if (delay_ticks < 1)
        delay_ticks = 1;
    if (delay_ticks > 4)
        delay_ticks = 4;

    /*
     * Move the adjustable torch two texture texels toward the input side per
     * delay step. Delay 1 should sit close to the fixed torch, then walk back
     * along the inset red strip as the delay increases.
     */
    return forward_texels_by_delay[delay_ticks - 1u] *
           REDSTONE_DEVICE_PLATE_TEXEL;
}

static void emit_redstone_device_block_lit(RenderContext *ctx,
                                           BlockID type,
                                           Vec3 block_pos,
                                           uint8_t visual_state,
                                           uint8_t light_flags)
{
    Vec3 face_world[4];
    BlockDoorFacing facing = block_redstone_facing(type);
    uint8_t top_tile = block_face_texture_id(type, FACE_TOP);
    uint8_t post_tile = block_redstone_directional_powered(type) ?
                        TEX_TILE_REDSTONE_TORCH_ON :
                        TEX_TILE_REDSTONE_TORCH_OFF;
    int uv_rotation = redstone_directional_uv_rotation(type);

    redstone_device_plate_top_vertices(block_pos, face_world);
    emit_model_quad_lit_rotated(ctx, face_world, top_tile, light_flags,
                                uv_rotation);
    for (int side = 0; side < 4; side++)
        emit_redstone_device_plate_side(ctx, block_pos, facing, side,
                                        light_flags);

    if (block_is_comparator(type)) {
        emit_redstone_device_post(ctx, block_pos, facing,
                                  -0.18f, -0.16f, post_tile, light_flags);
```

```
        emit_redstone_device_post(ctx, block_pos, facing,
                                  0.18f, -0.16f, post_tile, light_flags);
        emit_redstone_device_post(ctx, block_pos, facing,
                                  0.0f, 0.20f, post_tile, light_flags);
    } else {
        float moving_post_forward;

        moving_post_forward = repeater_moving_post_forward(visual_state);
        emit_redstone_device_post(ctx, block_pos, facing,
                                  0.0f, 5.0f * REDSTONE_DEVICE_PLATE_TEXEL,
                                  post_tile, light_flags);
        emit_redstone_device_post(ctx, block_pos, facing,
                                  0.0f, moving_post_forward,
                                  post_tile, light_flags);
    }
}

static void emit_flat_block_face_lit(RenderContext *ctx, BlockID type,
                                     const VoxelWorld *world,
                                     Vec3 block_pos,
                                     int wx,
                                     int wy,
                                     int wz,
                                     uint8_t visual_state,
                                     uint8_t light_flags)
{
    Vec3 face_world[4];
    uint8_t texture_tile;
    int uv_rotation = 0;

    if (type == BLOCK_AIR)
        return;

    if (block_is_redstone_directional(type)) {
        if (block_is_repeater(type) && world) {
            uint8_t live_delay = world_repeater_delay_ticks(world, wx, wy, wz);

            if (live_delay)
                visual_state = live_delay;
        }
        emit_redstone_device_block_lit(ctx, type, block_pos, visual_state,
                                       light_flags);
        return;
    }

    flat_face_vertices(block_pos, type, face_world);
    texture_tile = block_face_texture_id(type, FACE_TOP);
    if (render_block_is_redstone_wire(type)) {
        uint8_t mask = render_redstone_wire_mask(world, wx, wy, wz);

        texture_tile = redstone_wire_texture_for_mask(type, mask,
                                                      &uv_rotation);
    }

    emit_model_quad_lit_rotated(ctx, face_world, texture_tile, light_flags,
                                uv_rotation);
}

static void torch_axis_for_support(Vec3 block_pos,
```

```
                                    uint8_t support_face,
                                    Vec3 *base_out,
                                    Vec3 *tip_out)
{
    const float wall_base = 0.08f;
    const float wall_tip = 0.42f;
    Vec3 base = { block_pos.x + 0.5f,
                  block_pos.y,
                  block_pos.z + 0.5f };
    Vec3 tip = { block_pos.x + 0.5f,
                 block_pos.y + 0.86f,
                 block_pos.z + 0.5f };

    switch (support_face) {
    case FACE_LEFT:
        base.x = block_pos.x + wall_base;
        base.y = block_pos.y + 0.22f;
        tip.x = block_pos.x + wall_tip;
        break;
    case FACE_RIGHT:
        base.x = block_pos.x + 1.0f - wall_base;
        base.y = block_pos.y + 0.22f;
        tip.x = block_pos.x + 1.0f - wall_tip;
        break;
    case FACE_FRONT:
        base.z = block_pos.z + wall_base;
        base.y = block_pos.y + 0.22f;
        tip.z = block_pos.z + wall_tip;
        break;
    case FACE_BACK:
        base.z = block_pos.z + 1.0f - wall_base;
        base.y = block_pos.y + 0.22f;
        tip.z = block_pos.z + 1.0f - wall_tip;
        break;
    default:
        break;
    }

    if (base_out)
        *base_out = base;
    if (tip_out)
        *tip_out = tip;
}

static void torch_face_vertices(Vec3 block_pos, uint8_t face,
                                uint8_t support_face, Vec3 out[4])
{
    const bool side_support =
        support_face >= FACE_LEFT && support_face <= FACE_BACK;
    const float half = side_support ? 0.18f : 0.24f;
    Vec3 base;
    Vec3 tip;
    float ax;
    float az;

    torch_axis_for_support(block_pos, support_face, &base, &tip);

    if (face == CHUNK_FACE_CROSS_B) {
        ax = half;
```

```
        az = -half;
        out[0] = (Vec3){ base.x + ax, base.y, base.z + az };
        out[1] = (Vec3){ base.x - ax, base.y, base.z - az };
        out[2] = (Vec3){ tip.x - ax,  tip.y,  tip.z - az };
        out[3] = (Vec3){ tip.x + ax,  tip.y,  tip.z + az };
        return;
    }

    ax = half;
    az = half;
    out[0] = (Vec3){ base.x - ax, base.y, base.z - az };
    out[1] = (Vec3){ base.x + ax, base.y, base.z + az };
    out[2] = (Vec3){ tip.x + ax,  tip.y,  tip.z + az };
    out[3] = (Vec3){ tip.x - ax,  tip.y,  tip.z - az };
}

static void door_slab_planes(BlockID type,
                             bool *span_x_out,
                             float *outer_plane_out,
                             float *inner_plane_out)
{
    const float thickness = 0.10f;
    const float near_edge = 0.05f;
    const float far_edge = 0.95f;
    BlockDoorFacing facing = block_door_facing(type);
    bool open = block_is_door_open(type);
    bool span_x = true;
    float plane = near_edge;

    if (!open) {
        switch (facing) {
        case BLOCK_DOOR_FACING_EAST:
            span_x = false;
            plane = far_edge;
            break;
        case BLOCK_DOOR_FACING_SOUTH:
            span_x = true;
            plane = far_edge;
            break;
        case BLOCK_DOOR_FACING_WEST:
            span_x = false;
            plane = near_edge;
            break;
        case BLOCK_DOOR_FACING_NORTH:
        default:
            span_x = true;
            plane = near_edge;
            break;
        }
    } else {
        switch (facing) {
        case BLOCK_DOOR_FACING_EAST:
            span_x = true;
            plane = near_edge;
            break;
        case BLOCK_DOOR_FACING_SOUTH:
            span_x = false;
            plane = far_edge;
            break;
```

```
        case BLOCK_DOOR_FACING_WEST:
            span_x = true;
            plane = far_edge;
            break;
        case BLOCK_DOOR_FACING_NORTH:
        default:
            span_x = false;
            plane = near_edge;
            break;
        }
    }

    if (span_x_out)
        *span_x_out = span_x;
    if (outer_plane_out)
        *outer_plane_out = plane;
    if (inner_plane_out)
        *inner_plane_out = plane + (plane < 0.5f ? thickness : -thickness);
}

static void door_face_vertices(Vec3 block_pos, BlockID type, uint8_t face,
                               Vec3 out[4])
{
    bool span_x;
    float outer_plane;
    float inner_plane;
    float plane;

    door_slab_planes(type, &span_x, &outer_plane, &inner_plane);
    plane = face == CHUNK_FACE_CROSS_B ? inner_plane : outer_plane;

    if (span_x) {
        float z = block_pos.z + plane;

        out[0] = (Vec3){ block_pos.x,        block_pos.y,        z };
        out[1] = (Vec3){ block_pos.x + 1.0f, block_pos.y,        z };
        out[2] = (Vec3){ block_pos.x + 1.0f, block_pos.y + 1.0f, z };
        out[3] = (Vec3){ block_pos.x,        block_pos.y + 1.0f, z };
        return;
    }

    {
        float x = block_pos.x + plane;

        out[0] = (Vec3){ x, block_pos.y,        block_pos.z + 1.0f };
        out[1] = (Vec3){ x, block_pos.y,        block_pos.z };
        out[2] = (Vec3){ x, block_pos.y + 1.0f, block_pos.z };
        out[3] = (Vec3){ x, block_pos.y + 1.0f, block_pos.z + 1.0f };
    }
}

static void door_edge_vertices(Vec3 block_pos, BlockID type, int edge,
                               Vec3 out[4])
{
    bool span_x;
    float outer_plane;
    float inner_plane;
    float min_plane;
    float max_plane;
```

```
    float x0 = block_pos.x;
    float x1 = block_pos.x + 1.0f;
    float y0 = block_pos.y;
    float y1 = block_pos.y + 1.0f;
    float z0 = block_pos.z;
    float z1 = block_pos.z + 1.0f;

    door_slab_planes(type, &span_x, &outer_plane, &inner_plane);
    min_plane = outer_plane < inner_plane ? outer_plane : inner_plane;
    max_plane = outer_plane < inner_plane ? inner_plane : outer_plane;

    if (span_x) {
        float za = block_pos.z + min_plane;
        float zb = block_pos.z + max_plane;

        switch (edge) {
        case 0:
            out[0] = (Vec3){ x0, y0, za };
            out[1] = (Vec3){ x0, y0, zb };
            out[2] = (Vec3){ x0, y1, zb };
            out[3] = (Vec3){ x0, y1, za };
            return;
        case 1:
            out[0] = (Vec3){ x1, y0, zb };
            out[1] = (Vec3){ x1, y0, za };
            out[2] = (Vec3){ x1, y1, za };
            out[3] = (Vec3){ x1, y1, zb };
            return;
        case 2:
            out[0] = (Vec3){ x0, y0, za };
            out[1] = (Vec3){ x1, y0, za };
            out[2] = (Vec3){ x1, y0, zb };
            out[3] = (Vec3){ x0, y0, zb };
            return;
        default:
            out[0] = (Vec3){ x0, y1, zb };
            out[1] = (Vec3){ x1, y1, zb };
            out[2] = (Vec3){ x1, y1, za };
            out[3] = (Vec3){ x0, y1, za };
            return;
        }
    }

    {
        float xa = block_pos.x + min_plane;
        float xb = block_pos.x + max_plane;

        switch (edge) {
        case 0:
            out[0] = (Vec3){ xa, y0, z0 };
            out[1] = (Vec3){ xb, y0, z0 };
            out[2] = (Vec3){ xb, y1, z0 };
            out[3] = (Vec3){ xa, y1, z0 };
            return;
        case 1:
            out[0] = (Vec3){ xb, y0, z1 };
            out[1] = (Vec3){ xa, y0, z1 };
            out[2] = (Vec3){ xa, y1, z1 };
            out[3] = (Vec3){ xb, y1, z1 };
```

```
            return;
        case 2:
            out[0] = (Vec3){ xa, y0, z0 };
            out[1] = (Vec3){ xb, y0, z0 };
            out[2] = (Vec3){ xb, y0, z1 };
            out[3] = (Vec3){ xa, y0, z1 };
            return;
        default:
            out[0] = (Vec3){ xa, y1, z1 };
            out[1] = (Vec3){ xb, y1, z1 };
            out[2] = (Vec3){ xb, y1, z0 };
            out[3] = (Vec3){ xa, y1, z0 };
            return;
        }
    }
}

static void emit_model_quad_lit(RenderContext *ctx,
                                Vec3 face_world[4], uint8_t texture_tile,
                                uint8_t light_flags)
{
    emit_model_quad_lit_rotated(ctx, face_world, texture_tile, light_flags, 0);
}

static void emit_model_quad_lit_rotated(RenderContext *ctx,
                                        Vec3 face_world[4],
                                        uint8_t texture_tile,
                                        uint8_t light_flags,
                                        int uv_rotation)
{
    static const float base_uv[4][2] = {
        { 0.0f,  16.0f },
        { 16.0f, 16.0f },
        { 16.0f, 0.0f  },
        { 0.0f,  0.0f  },
    };
    CameraVertex face_cam[4];
    uint8_t texture_id;
    uint8_t face_flags = light_flags | QUAD_FLAG_FOG | QUAD_FLAG_ALPHA_KEY;

    uv_rotation &= 3;
    for (int i = 0; i < 4; i++) {
        int uv_index = (i + uv_rotation) & 3;

        world_to_camera(ctx, face_world[i], &face_cam[i]);
        face_cam[i].u = base_uv[uv_index][0];
        face_cam[i].v = base_uv[uv_index][1];
    }
    if (camera_quad_outside_view(ctx, face_cam))
        return;

    texture_id = choose_face_texture_lod(ctx, texture_tile, face_cam);

    CameraVertex clipped[6];
    int clipped_count = clip_face_to_near_plane(face_cam, clipped);
    emit_clipped_face(ctx, clipped, clipped_count, texture_id, face_flags);
}

static void emit_torch_block_face_lit(RenderContext *ctx, BlockID type,
```

```
                                      Vec3 block_pos, uint8_t face,
                                      uint8_t support_face,
                                      uint8_t light_flags)
{
    Vec3 face_world[4];

    if (type == BLOCK_AIR)
        return;

    torch_face_vertices(block_pos, face, support_face, face_world);
    emit_model_quad_lit(ctx, face_world,
                        block_face_texture_id(type, FACE_FRONT),
                        light_flags);
}

static void emit_door_block_face_lit(RenderContext *ctx, BlockID type,
                                     Vec3 block_pos, uint8_t face,
                                     uint8_t light_flags)
{
    Vec3 face_world[4];

    if (!block_is_door(type))
        return;

    door_face_vertices(block_pos, type, face, face_world);
    emit_model_quad_lit(ctx, face_world,
                        block_face_texture_id(type, FACE_FRONT),
                        light_flags);

    if (face == CHUNK_FACE_CROSS_A) {
        for (int edge = 0; edge < 4; edge++) {
            door_edge_vertices(block_pos, type, edge, face_world);
            emit_model_quad_lit(ctx, face_world, TEX_TILE_WOOD_PLANK,
                                light_flags);
        }
    }
}

static void emit_merged_block_face(RenderContext *ctx, BlockID type,
                                   Vec3 block_pos, BlockFace face,
                                   int u_size, int v_size)
{
    uint8_t light_flags = block_is_self_lit(type)
                              ? QUAD_LIGHT_LEVEL(0)
                              : choose_face_light_flags(ctx, face);

    emit_merged_block_face_lit(ctx, type, block_pos, face, u_size, v_size,
                               8, light_flags);
}

static void emit_block_face(RenderContext *ctx, BlockID type,
                            Vec3 block_pos, BlockFace face)
{
    emit_merged_block_face(ctx, type, block_pos, face, 1, 1);
}

typedef struct {
    uint8_t unlock_stage;
    uint8_t x;
```

```
    uint8_t y;
    uint8_t w;
    uint8_t h;
} BreakCrackPatch;

/* 16x16 grid-space crack pieces. Hard rectangular runs keep the break overlay
 * pixelated and shard-like instead of looking hand-drawn. */
static const BreakCrackPatch BREAK_CRACK_PATCHES[] = {
    { 0,  7,  7, 2, 2 },
    { 1,  7,  5, 1, 2 },
    { 1,  6,  7, 1, 1 },
    { 1,  9,  8, 1, 1 },
    { 2,  5,  6, 2, 1 },
    { 2,  4,  5, 1, 1 },
    { 2,  9,  6, 2, 1 },
    { 2, 11,  5, 1, 1 },
    { 3,  6,  9, 1, 2 },
    { 3,  5, 11, 1, 1 },
    { 3,  9,  9, 2, 1 },
    { 3, 11, 10, 1, 1 },
    { 4,  3,  4, 1, 2 },
    { 4,  2,  3, 1, 1 },
    { 4, 12,  4, 1, 2 },
    { 4, 13,  3, 1, 1 },
    { 5,  4, 12, 1, 2 },
    { 5,  3, 14, 1, 1 },
    { 5, 12, 10, 1, 2 },
    { 5, 13, 12, 1, 1 },
    { 6,  7,  3, 1, 2 },
    { 6,  7,  1, 1, 2 },
    { 6, 10,  2, 2, 1 },
    { 6, 12,  1, 1, 1 },
    { 7,  1,  2, 1, 1 },
    { 7,  0,  1, 1, 1 },
    { 7, 14,  2, 1, 1 },
    { 7, 15,  1, 1, 1 },
    { 7, 14, 13, 1, 2 },
    { 8,  2, 14, 1, 2 },
    { 8,  0, 15, 2, 1 },
    { 8, 14, 14, 2, 1 },
    { 8, 15, 15, 1, 1 },
    { 9,  0,  0, 1, 1 },
    { 9, 15,  0, 1, 1 },
    { 9,  0,  8, 2, 1 },
    { 9, 14,  8, 2, 1 },
    { 9,  6,  6, 1, 1 },
    { 9,  9,  7, 1, 1 },
    { 9,  8,  9, 1, 1 },
};

static Vec3 vec3_add(Vec3 a, Vec3 b)
{
    return (Vec3){ a.x + b.x, a.y + b.y, a.z + b.z };
}

static Vec3 vec3_sub(Vec3 a, Vec3 b)
{
    return (Vec3){ a.x - b.x, a.y - b.y, a.z - b.z };
}
```

```
3037
3038 static Vec3 vec3_scale(Vec3 v, float scale)
3039 {
3040     return (Vec3){ v.x * scale, v.y * scale, v.z * scale };
3041 }
3042
3043 static Vec3 break_face_point_from_uv(const Vec3 corners[4], float u, float v)
3044 {
3045     Vec3 du = vec3_sub(corners[1], corners[0]);
3046     Vec3 dv = vec3_sub(corners[3], corners[0]);
3047     Vec3 p = corners[0];
3048
3049     p = vec3_add(p, vec3_scale(du, u / 16.0f));
3050     p = vec3_add(p, vec3_scale(dv, (16.0f - v) / 16.0f));
3051     return p;
3052 }
3053
3054 static void emit_break_crack_patch(RenderContext *ctx,
3055                                    const Vec3 corners[4],
3056                                    const BreakCrackPatch *patch,
3057                                    uint8_t color,
3058                                    uint8_t alpha)
3059 {
3060     float x0 = (float)patch->x;
3061     float y0 = (float)patch->y;
3062     float x1 = x0 + (float)patch->w;
3063     float y1 = y0 + (float)patch->h;
3064     Vec3 patch_world[4];
3065     CameraVertex patch_cam[4];
3066     CameraVertex clipped[6];
3067
3068     patch_world[0] = break_face_point_from_uv(corners, x0, y1);
3069     patch_world[1] = break_face_point_from_uv(corners, x1, y1);
3070     patch_world[2] = break_face_point_from_uv(corners, x1, y0);
3071     patch_world[3] = break_face_point_from_uv(corners, x0, y0);
3072
3073     for (int i = 0; i < 4; i++)
3074         world_to_camera(ctx, patch_world[i], &patch_cam[i]);
3075     if (camera_quad_outside_view(ctx, patch_cam))
3076         return;
3077
3078     int clipped_count = clip_face_to_near_plane(patch_cam, clipped);
3079     emit_clipped_face_color(ctx, clipped, clipped_count, color, alpha);
3080 }
3081
3082 int renderer_draw_block_break_overlay(RenderContext *ctx,
3083                                       int block_x, int block_y, int block_z,
3084                                       float progress)
3085 {
3086     const float face_epsilon = 0.004f;
3087     int before;
3088     int stage;
3089     uint8_t color;
3090     uint8_t alpha;
3091     Vec3 block_pos;
3092
3093     if (!ctx)
3094         return 0;
3095
```

```
    progress = clamp01(progress);
    if (progress <= 0.0f)
        return 0;

    before = ctx->n_quads;
    stage = (int)floorf(progress * 10.0f);
    if (stage < 0)
        stage = 0;
    if (stage > 9)
        stage = 9;
    color = 14;
    alpha = QUAD_ALPHA_75;
    block_pos = (Vec3){ (float)block_x, (float)block_y, (float)block_z };

    for (int face = 0; face < NUM_FACES; face++) {
        Vec3 corners[4];
        Vec3 normal = face_normals[face];

        if (!is_face_visible(block_pos, normal, ctx->current_camera.position))
            continue;

        merged_face_vertices(block_pos, (BlockFace)face, 1, 1, 8, corners);
        for (int i = 0; i < 4; i++) {
            corners[i].x += normal.x * face_epsilon;
            corners[i].y += normal.y * face_epsilon;
            corners[i].z += normal.z * face_epsilon;
        }

        for (size_t i = 0;
             i < sizeof(BREAK_CRACK_PATCHES) / sizeof(BREAK_CRACK_PATCHES[0]);
             i++) {
            if (BREAK_CRACK_PATCHES[i].unlock_stage > stage)
                continue;
            emit_break_crack_patch(ctx, corners, &BREAK_CRACK_PATCHES[i],
                                   color, alpha);
            if (ctx->n_quads >= MAX_QUADS_IN_FLIGHT)
                return ctx->n_quads - before;
        }
    }

    return ctx->n_quads - before;
}

static float distance_to_interval(float value, float min, float max)
{
    if (value < min)
        return min - value;
    if (value > max)
        return value - max;
    return 0.0f;
}

typedef struct {
    CameraVertex center;
    float ex;
    float ey;
    float ez;
} ChunkCameraBounds;

```

```
static void chunk_camera_bounds(RenderContext *ctx, const Chunk *chunk,
                                ChunkCameraBounds *bounds)
{
    float half_x = WORLD_CHUNK_SIZE * 0.5f;
    float half_y = WORLD_CHUNK_HEIGHT * 0.5f;
    float half_z = WORLD_CHUNK_SIZE * 0.5f;
    float cy = ctx->cos_yaw;
    float sy = ctx->sin_yaw;
    float cp = ctx->cos_pitch;
    float sp = ctx->sin_pitch;
    Vec3 chunk_center = {
        (float)(chunk->chunk_x * WORLD_CHUNK_SIZE) + half_x,
        half_y,
        (float)(chunk->chunk_z * WORLD_CHUNK_SIZE) + half_z,
    };

    world_to_camera(ctx, chunk_center, &bounds->center);

    /* Project world-space half-extents into camera space using abs(R) * e. */
    bounds->ex = fabsf(cy) * half_x + fabsf(sy) * half_z;
    bounds->ey = fabsf(sy * sp) * half_x +
                 fabsf(cp) * half_y +
                 fabsf(cy * sp) * half_z;
    bounds->ez = fabsf(sy * cp) * half_x +
                 fabsf(sp) * half_y +
                 fabsf(cy * cp) * half_z;
}

static float chunk_distance_sq_to_camera(RenderContext *ctx, const Chunk *chunk);

static bool ensure_translucent_scratch(RenderContext *ctx, int needed)
{
    int cap;
    TranslucentFaceRef *buf;

    if (needed <= ctx->translucent_scratch_capacity)
        return true;

    cap = ctx->translucent_scratch_capacity ? ctx->translucent_scratch_capacity : 64;
    while (cap < needed)
        cap *= 2;

    buf = realloc(ctx->translucent_scratch, (size_t)cap * sizeof(*buf));
    if (!buf)
        return false;

    ctx->translucent_scratch = buf;
    ctx->translucent_scratch_capacity = cap;
    return true;
}

static uint32_t translucent_face_stable_order(const Chunk *chunk,
                                              const ChunkFace *face)
{
    uint32_t h = 2166136261u;

#define MIX_U32(v) do { h ^= (uint32_t)(v); h *= 16777619u; } while (0)
    MIX_U32(chunk->chunk_x);
    MIX_U32(chunk->chunk_z);
```

```
    MIX_U32(face->x);
    MIX_U32(face->y);
    MIX_U32(face->z);
    MIX_U32(face->face);
#undef MIX_U32

    return h;
}

static int compare_translucent_back_to_front(const void *a, const void *b)
{
    const TranslucentFaceRef *ra = a;
    const TranslucentFaceRef *rb = b;
    float dz = ra->view_z - rb->view_z;

    /* Larger view_z (farther) comes first. */
    if (dz > 1e-4f) return -1;
    if (dz < -1e-4f) return 1;
    if (ra->stable_order < rb->stable_order) return -1;
    if (ra->stable_order > rb->stable_order) return 1;
    return 0;
}

static bool chunk_within_render_distance(RenderContext *ctx,
                                         const VoxelWorld *world,
                                         const Chunk *chunk)
{
    float max_distance;
    float max_distance_sq;

    if (world->render_distance_chunks <= 0)
        return true;

    max_distance = (float)(world->render_distance_chunks * WORLD_CHUNK_SIZE);
    max_distance_sq = max_distance * max_distance;
    return chunk_distance_sq_to_camera(ctx, chunk) <= max_distance_sq;
}

static void configure_world_fog(RenderContext *ctx, const VoxelWorld *world)
{
    struct voxel_fog_state fog = {0};

    if (!ctx || !world || world->render_distance_chunks <= 0) {
        if (ctx)
            renderer_set_fog_state(ctx, &fog);
        return;
    }

    float end_distance = (float)(world->render_distance_chunks * WORLD_CHUNK_SIZE);
    float fade_span = (float)WORLD_CHUNK_SIZE;
    float start_distance;
    float inv_proj_sq;

    start_distance = end_distance - fade_span;
    if (start_distance < NEAR_PLANE)
        start_distance = NEAR_PLANE;

    inv_proj_sq = 1.0f / (ctx->current_camera.depth * ctx->current_camera.depth);

```

```
    fog.start_dist = to_q8_8u(start_distance);
    fog.end_dist = to_q8_8u(end_distance);
    fog.color_index = PAL_SKY_HORIZON;
    fog.enabled = fog.end_dist > fog.start_dist;
    fog.inv_proj_sq = to_q0_16u(inv_proj_sq);
    renderer_set_fog_state(ctx, &fog);
}

static bool chunk_intersects_frustum(RenderContext *ctx, const Chunk *chunk)
{
    float x_slope = (SCREEN_WIDTH * 0.5f) / ctx->current_camera.depth;
    float y_slope = (SCREEN_HEIGHT * 0.5f) / ctx->current_camera.depth;
    ChunkCameraBounds bounds;

    chunk_camera_bounds(ctx, chunk, &bounds);

    if (bounds.center.z + bounds.ez < NEAR_PLANE)
        return false;
    if (bounds.center.x + x_slope * bounds.center.z +
        (bounds.ex + x_slope * bounds.ez) < 0.0f)
        return false;
    if (-bounds.center.x + x_slope * bounds.center.z +
        (bounds.ex + x_slope * bounds.ez) < 0.0f)
        return false;
    if (bounds.center.y + y_slope * bounds.center.z +
        (bounds.ey + y_slope * bounds.ez) < 0.0f)
        return false;
    if (-bounds.center.y + y_slope * bounds.center.z +
        (bounds.ey + y_slope * bounds.ez) < 0.0f)
        return false;

    return true;
}

typedef struct {
    const Chunk *chunk;
    /* Cached at the start of draw_world so the entire frame iterates a
     * stable mesh pointer while workers publish replacements. */
    const ChunkMesh *mesh;
    float distance_sq;
} ChunkDrawCandidate;

static float chunk_distance_sq_to_camera(RenderContext *ctx, const Chunk *chunk)
{
    float min_x = (float)(chunk->chunk_x * WORLD_CHUNK_SIZE);
    float max_x = min_x + (float)WORLD_CHUNK_SIZE;
    float min_y = 0.0f;
    float max_y = (float)WORLD_CHUNK_HEIGHT;
    float min_z = (float)(chunk->chunk_z * WORLD_CHUNK_SIZE);
    float max_z = min_z + (float)WORLD_CHUNK_SIZE;
    float dx = distance_to_interval(ctx->current_camera.position.x, min_x, max_x);
    float dy = distance_to_interval(ctx->current_camera.position.y, min_y, max_y);
    float dz = distance_to_interval(ctx->current_camera.position.z, min_z, max_z);

    return dx * dx + dy * dy + dz * dz;
}

/* --- Public API --- */

```

```
RenderContext *renderer_init(void)
{
    RenderContext *ctx = calloc(1, sizeof(*ctx));
    if (!ctx) return NULL;

    ctx->transport = gpu_transport_open();
    if (!ctx->transport) {
        free(ctx);
        return NULL;
    }

    ctx->submit_capacity = MAX_QUADS_IN_FLIGHT *
                           (sizeof(struct quad_desc) + sizeof(struct quad_desc_uv));
    ctx->submit_buffer = malloc(ctx->submit_capacity);
    ctx->lookup_entries = NULL;
    ctx->lookup_capacity = 0;
    ctx->translucent_scratch = NULL;
    ctx->translucent_scratch_capacity = 0;
    ctx->occlusion_enabled = env_flag("VOXEL_OCCLUSION_CULL", false) ? 1 : 0;
    ctx->occlusion_diag = env_flag("VOXEL_DIAG_OCCLUSION", false) ? 1 : 0;
    if (!ctx->submit_buffer) {
        free(ctx->submit_buffer);
        gpu_transport_close(ctx->transport);
        free(ctx);
        return NULL;
    }

    ctx->light_dir = normalize_vec3((Vec3){ 0.35f, 0.92f, 0.20f });
    ctx->world_daylight = 1.0f;
    update_face_light_flags(ctx);
    upload_default_palette(ctx->transport);
    upload_light_palette_banks(ctx, ctx->world_daylight);
    renderer_set_fog_state(ctx, &(struct voxel_fog_state){0});
    return ctx;
}

void renderer_shutdown(RenderContext *ctx)
{
    if (!ctx) return;
    free(ctx->translucent_scratch);
    free(ctx->lookup_entries);
    free(ctx->submit_buffer);
    gpu_transport_close(ctx->transport);
    free(ctx);
}

void renderer_begin_frame(RenderContext *ctx)
{
    ctx->n_quads = 0;
    ctx->submit_bytes = 0;
    occlusion_reset(ctx);
    /* Front-load the per-quad band binning into emit time: stage_prepared_quad
     * will hand each finished descriptor to gpu_transport_bin_descriptor as
     * soon as it is built (descriptor still hot in L1), and the corresponding
     * second-pass walk inside submit_hw_binned is skipped. */
    gpu_transport_begin_descriptors(ctx->transport);
}

void renderer_end_frame(RenderContext *ctx)
```

```
{
    if (ctx->occlusion_diag && ((ctx->occlusion_frame_count++ % 60u) == 0u)) {
        fprintf(stderr,
                "renderer: occlusion tested=%u culled=%u tiles=%u enabled=%u\n",
                ctx->occlusion_tested,
                ctx->occlusion_culled,
                ctx->occlusion_tiles_written,
                (unsigned)ctx->occlusion_enabled);
    }

    if (gpu_transport_clear(ctx->transport) < 0)
        return;
    if (!renderer_flush_gpu_state(ctx))
        return;

    if (ctx->submit_bytes == 0) {
        gpu_transport_flip(ctx->transport);
        return;
    }

    if (gpu_transport_submit_descriptors(ctx->transport,
                                         ctx->submit_buffer,
                                         ctx->submit_bytes) < 0)
        return;

    gpu_transport_flip(ctx->transport);
}

void renderer_set_camera(RenderContext *ctx, const Camera *camera)
{
    ctx->current_camera = *camera;
    ctx->cos_yaw   = cosf(camera->yaw);
    ctx->sin_yaw   = sinf(camera->yaw);
    ctx->cos_pitch = cosf(camera->pitch);
    ctx->sin_pitch = sinf(camera->pitch);
}

static bool stage_prepared_quad(RenderContext *ctx, RenderQuad quad)
{
    const bool input_textured = (quad.flags & QUAD_FLAG_TEX) != 0;
    const size_t textured_descriptor_bytes =
        gpu_transport_textured_descriptor_size(ctx->transport);
    const size_t max_descriptor_bytes =
        input_textured ? textured_descriptor_bytes : sizeof(struct quad_desc);

    if (ctx->n_quads >= MAX_QUADS_IN_FLIGHT)
        return false;
    if (ctx->submit_bytes + max_descriptor_bytes > ctx->submit_capacity)
        return false;

    /*
     * The rasterizer only sees Q24.8 edge math. Snap screen-space vertices to
     * that same grid before bbox, edge, and depth setup so all three stages
     * agree on which pixels a thin quad should cover.
     */
    for (int i = 0; i < 4; i++) {
        quad.vertices[i].x = snap_q24_8(quad.vertices[i].x);
        quad.vertices[i].y = snap_q24_8(quad.vertices[i].y);
    }
```

```
    ensure_clockwise_winding(quad.vertices);

    const Vertex2D *v = quad.vertices;
    PlaneBasis basis = choose_plane_basis(v);

    /* Inclusive integer bbox over pixel-center samples. */
    int32_t qxmin = to_q24_8(v[0].x), qxmax = qxmin;
    int32_t qymin = to_q24_8(v[0].y), qymax = qymin;
    for (int i = 1; i < 4; i++) {
        int32_t qx = to_q24_8(v[i].x);
        int32_t qy = to_q24_8(v[i].y);

        if (qx < qxmin) qxmin = qx;
        if (qx > qxmax) qxmax = qx;
        if (qy < qymin) qymin = qy;
        if (qy > qymax) qymax = qy;
    }
    int x_min = ceil_div_256_i32(qxmin - 128);  if (x_min <   0) x_min =   0;
    int y_min = ceil_div_256_i32(qymin - 128);  if (y_min <   0) y_min =   0;
    int x_max = floor_div_256_i32(qxmax - 128);
    int y_max = floor_div_256_i32(qymax - 128);
    if (x_max > (int)VOXEL_RENDER_WIDTH - 1)
        x_max = (int)VOXEL_RENDER_WIDTH - 1;
    if (y_max > (int)VOXEL_RENDER_HEIGHT - 1)
        y_max = (int)VOXEL_RENDER_HEIGHT - 1;

    if (x_min > x_max || y_min > y_max)
        return false;   /* degenerate / off-screen */

    if (ctx->occlusion_enabled &&
        ctx->occlusion_pass == OCCLUSION_PASS_OPAQUE_WORLD &&
        occlusion_quad_eligible(quad.flags)) {
        ctx->occlusion_tested++;
        if (occlusion_quad_hidden(ctx, v, x_min, y_min, x_max, y_max)) {
            ctx->occlusion_culled++;
            return false;
        }
    }

    struct quad_desc *d = (struct quad_desc *)(ctx->submit_buffer + ctx->submit_bytes);
    memset(d, 0, sizeof(*d));

    d->x_min = (int16_t)x_min;
    d->y_min = (int16_t)y_min;
    d->x_max = (int16_t)x_max;
    d->y_max = (int16_t)y_max;

    for (int i = 0; i < 4; i++) {
        float x0 = v[i].x,         y0 = v[i].y;
        float x1 = v[(i+1)%4].x,   y1 = v[(i+1)%4].y;
        d->edges[i] = make_edge_coef(x0, y0, x1, y1);
    }

    /* Compute into locals first to avoid taking addresses of packed members. */
    uint16_t z0;
    int16_t dz_dx;
    int16_t dz_dy;
    fit_depth_plane(v, &basis, (float)x_min + 0.5f, (float)y_min + 0.5f,
```

```
                    &z0, &dz_dx, &dz_dy);
    d->z0 = z0;
    d->dz_dx = dz_dx;
    d->dz_dy = dz_dy;

    d->tex_or_color = (quad.flags & QUAD_FLAG_TEX) ? quad.texture_id : quad.color_tint;
    d->flags = quad.flags;

#ifdef DEBUG_FLAT_COLOR
#ifndef DEBUG_FLAT_COLOR_INDEX
#define DEBUG_FLAT_COLOR_INDEX 24
#endif
    d->flags = (uint16_t)(quad.flags & ~(QUAD_FLAG_TEX));
    d->tex_or_color = (uint8_t)DEBUG_FLAT_COLOR_INDEX;
#endif

    size_t emitted_size;
    if (d->flags & QUAD_FLAG_TEX) {
        struct quad_desc_uv *uv = (struct quad_desc_uv *)
            (ctx->submit_buffer + ctx->submit_bytes + sizeof(*d));
        memset(uv, 0, sizeof(*uv));
        fit_uv_plane(v, &basis, (float)x_min + 0.5f, (float)y_min + 0.5f, uv);
        emitted_size = textured_descriptor_bytes;
    } else {
        emitted_size = sizeof(*d);
    }
    ctx->submit_bytes += emitted_size;
    ctx->n_quads++;

    /* Bin-during-emit: route this just-built descriptor into the per-band
     * bins now while it is still in L1, instead of having submit_hw_binned
     * walk the contiguous stream a second time. No-op when the transport is
     * not in HW mode. */
    gpu_transport_bin_descriptor(ctx->transport, d, emitted_size);

    if (ctx->occlusion_enabled &&
        ctx->occlusion_pass == OCCLUSION_PASS_OPAQUE_WORLD &&
        occlusion_quad_eligible(d->flags)) {
        occlusion_record_quad(ctx, v, x_min, y_min, x_max, y_max);
    }

    return true;
}

bool renderer_push_quad(RenderContext *ctx, const RenderQuad *quad)
{
    Vertex2D clipped[MAX_VIEW_CLIP_VERTS];
    int count = clip_polygon_to_viewport(quad->vertices, clipped);

    return stage_projected_polygon(ctx, quad, clipped, count);
}

static void renderer_draw_block_faces(RenderContext *ctx, const Block *block,
                                      const BlockLookup *lookup)
{
    if (block->type == BLOCK_AIR) return;

    for (int f = 0; f < NUM_FACES; f++) {
        if (!is_face_exposed(block, (BlockFace)f, lookup))
```

```
            continue;
        emit_block_face(ctx, block->type, block->position, (BlockFace)f);
    }
}

int renderer_draw_chunk(RenderContext *ctx, const Block *blocks, int num_blocks)
{
    BlockLookup lookup;
    int before = ctx->n_quads;

    if (!build_block_lookup(ctx, blocks, num_blocks, &lookup))
        return 0;

    for (int i = 0; i < num_blocks; i++)
        renderer_draw_block_faces(ctx, &blocks[i], &lookup);

    return ctx->n_quads - before;
}

static bool falling_block_hides_mesh_cell(const VoxelWorld *world,
                                          int wx, int wy, int wz)
{
    int seen = 0;

    if (!world || world->falling_block_count <= 0)
        return false;

    for (int i = 0; i < WORLD_MAX_FALLING_BLOCKS &&
                    seen < world->falling_block_count; i++) {
        const FallingBlock *falling = &world->falling_blocks[i];

        if (!falling->active)
            continue;
        seen++;
        if (falling->wx == wx &&
            falling->origin_y == wy &&
            falling->wz == wz)
            return true;
    }

    return false;
}

static int renderer_draw_falling_blocks(RenderContext *ctx,
                                        const VoxelWorld *world)
{
    int before = ctx->n_quads;
    int seen = 0;

    if (!world || world->falling_block_count <= 0)
        return 0;

    for (int i = 0; i < WORLD_MAX_FALLING_BLOCKS &&
                    seen < world->falling_block_count; i++) {
        const FallingBlock *falling = &world->falling_blocks[i];
        if (!falling->active)
            continue;
        seen++;

```

```
        Vec3 block_pos = {
            (float)falling->wx,
            falling->y,
            (float)falling->wz,
        };
        for (int f = 0; f < NUM_FACES; f++) {
            emit_block_face(ctx, falling->type, block_pos, (BlockFace)f);
            if (ctx->n_quads >= MAX_QUADS_IN_FLIGHT)
                return ctx->n_quads - before;
        }
    }

    return ctx->n_quads - before;
}

static bool cube_face_back_facing(BlockFace face,
                                  float wx,
                                  float wy,
                                  float wz,
                                  Vec3 cam_pos)
{
    switch (face) {
    case FACE_TOP:    return cam_pos.y < wy + 1.0f;
    case FACE_BOTTOM: return cam_pos.y > wy;
    case FACE_RIGHT:  return cam_pos.x < wx + 1.0f;
    case FACE_LEFT:   return cam_pos.x > wx;
    case FACE_BACK:   return cam_pos.z < wz + 1.0f;
    case FACE_FRONT:  return cam_pos.z > wz;
    default:          return false;
    }
}

static bool fast_cube_backface_cull_enabled(void)
{
    static int cached = -1;

    if (cached < 0)
        cached = env_flag("VOXEL_FAST_BACKFACE_CULL", true) ? 1 : 0;
    return cached != 0;
}

int renderer_draw_world(RenderContext *ctx, const VoxelWorld *world,
                        float time_seconds)
{
    int before = ctx->n_quads;
    int candidate_count = 0;

    if (!world || world->chunk_count <= 0)
        return 0;

    update_world_light_state(ctx, time_seconds);
    configure_world_fog(ctx, world);

    ChunkDrawCandidate candidates[world->chunk_count];

    for (int i = 0; i < world->chunk_count; i++) {
        const Chunk *chunk = &world->chunks[i];
        const ChunkMesh *mesh;

```

```
        if (!(chunk->flags & CHUNK_FLAG_LOADED))
            continue;
        mesh = atomic_load_explicit(&chunk->live_mesh, memory_order_acquire);
        if (!mesh || mesh->face_count <= 0)
            continue;
        if (!chunk_within_render_distance(ctx, world, chunk))
            continue;
        if (!chunk_intersects_frustum(ctx, chunk))
            continue;

        candidates[candidate_count++] = (ChunkDrawCandidate){
            .chunk = chunk,
            .mesh = mesh,
            .distance_sq = chunk_distance_sq_to_camera(ctx, chunk),
        };
    }

    for (int i = 1; i < candidate_count; i++) {
        ChunkDrawCandidate key = candidates[i];
        int j = i - 1;

        while (j >= 0 && candidates[j].distance_sq > key.distance_sq) {
            candidates[j + 1] = candidates[j];
            j--;
        }
        candidates[j + 1] = key;
    }

    /*
     * Opaque pass: emit all non-translucent faces first, chunks nearest
     * first. Glass/translucent faces also write Z through QUAD_FLAG_ZTEST,
     * so emitting them first would cause opaque geometry behind the glass
     * to be Z-rejected and we'd blend glass against sky instead of against
     * the stone behind it.
     *
     * Optional early back-face cull: face normals are axis-aligned (exactly one
     * component 1, rest zero). The full dot-product in is_face_visible
     * reduces to a single comparison per face direction:
     *   sign > 0 axis: skip if (block_pos[axis] + 1.0) - cam[axis] >= 0
     *   sign < 0 axis: skip if   block_pos[axis]        - cam[axis] <= 0
     * Doing this before choose_chunk_face_light_flags saves the light
     * lookup + function call for all back-facing faces (~50% of faces). */
    const Vec3 cam_pos = ctx->current_camera.position;
    ctx->occlusion_pass = ctx->occlusion_enabled ?
                          OCCLUSION_PASS_OPAQUE_WORLD :
                          OCCLUSION_PASS_DISABLED;
    for (int i = 0; i < candidate_count; i++) {
        const Chunk *chunk = candidates[i].chunk;
        const ChunkMesh *mesh = candidates[i].mesh;
        const int opaque_cube_count = mesh->opaque_cube_face_count;
        const int opaque_face_count = mesh->opaque_face_count;

        for (int face_index = 0; face_index < opaque_cube_count; face_index++) {
            const ChunkFace *face = &mesh->faces[face_index];
            BlockID id = (BlockID)face->type;
            int wxi = chunk->chunk_x * WORLD_CHUNK_SIZE + (int)face->x;
            int wyi = (int)face->y;
            int wzi = chunk->chunk_z * WORLD_CHUNK_SIZE + (int)face->z;
            if (falling_block_hides_mesh_cell(world, wxi, wyi, wzi))
```

```
            continue;

        float wx = (float)wxi;
        float wy = (float)wyi;
        float wz = (float)wzi;

        if (fast_cube_backface_cull_enabled() &&
            cube_face_back_facing((BlockFace)face->face,
                                  wx, wy, wz, cam_pos))
            continue;

        Vec3 block_pos = { wx, wy, wz };
        uint8_t light_flags = choose_chunk_face_light_flags(ctx, id,
                                                            (BlockFace)face->face,
                                                            face->sky_light,
                                                            face->block_light);

        emit_merged_block_face_lit(ctx, id, block_pos,
                                   (BlockFace)face->face,
                                   face->u_size ? face->u_size : 1,
                                   face->v_size ? face->v_size : 1,
                                   face->height ? face->height : 8,
                                   light_flags);
        if (ctx->n_quads >= MAX_QUADS_IN_FLIGHT) {
            ctx->occlusion_pass = OCCLUSION_PASS_DISABLED;
            return ctx->n_quads - before;
        }
    }

    for (int face_index = opaque_cube_count;
         face_index < opaque_face_count;
         face_index++) {
        const ChunkFace *face = &mesh->faces[face_index];
        BlockID id = (BlockID)face->type;
        int wxi = chunk->chunk_x * WORLD_CHUNK_SIZE + (int)face->x;
        int wyi = (int)face->y;
        int wzi = chunk->chunk_z * WORLD_CHUNK_SIZE + (int)face->z;
        if (falling_block_hides_mesh_cell(world, wxi, wyi, wzi))
            continue;

        Vec3 block_pos = {
            (float)wxi,
            (float)wyi,
            (float)wzi,
        };
        BlockRenderModel model = block_render_model(id);
        uint8_t light_flags = choose_chunk_face_light_flags(ctx, id,
                                                            FACE_TOP,
                                                            face->sky_light,
                                                            face->block_light);

        if (model == BLOCK_RENDER_CROSS)
            emit_cross_block_face_lit(ctx, id, block_pos,
                                      face->face, light_flags);
        else if (model == BLOCK_RENDER_TORCH)
            emit_torch_block_face_lit(ctx, id, block_pos,
                                      face->face, face->height,
                                      light_flags);
        else if (model == BLOCK_RENDER_DOOR)
```

```
3804                emit_door_block_face_lit(ctx, id, block_pos,
3805                                         face->face, light_flags);
3806            else
3807                emit_flat_block_face_lit(ctx, id, world, block_pos,
3808                                         wxi, wyi, wzi, face->height,
3809                                         light_flags);
3810            if (ctx->n_quads >= MAX_QUADS_IN_FLIGHT) {
3811                ctx->occlusion_pass = OCCLUSION_PASS_DISABLED;
3812                return ctx->n_quads - before;
3813            }
3814        }
3815    }
3816    ctx->occlusion_pass = OCCLUSION_PASS_DISABLED;
3817
3818    renderer_draw_falling_blocks(ctx, world);
3819    if (ctx->n_quads >= MAX_QUADS_IN_FLIGHT)
3820        return ctx->n_quads - before;
3821
3822    /*
3823     * Translucent pass: collect every glass face, sort back-to-front by
3824     * view-space depth of its block center, then emit. Two alpha-blended
3825     * quads at the same pixel must be drawn farthest-first so each blend
3826     * mixes against the correctly-composited destination. Block-center as
3827     * the sort key is fine because we back-face cull: the three (or fewer)
3828     * faces of a single glass block visible at once never overlap in
3829     * screen space, so their within-block order doesn't matter.
3830     */
3831    int w = 0;
3832    for (int i = 0; i < candidate_count; i++) {
3833        const Chunk *chunk = candidates[i].chunk;
3834        const ChunkMesh *mesh = candidates[i].mesh;
3835        const float chunk_ox = (float)(chunk->chunk_x * WORLD_CHUNK_SIZE);
3836        const float chunk_oz = (float)(chunk->chunk_z * WORLD_CHUNK_SIZE);
3837
3838        if (mesh->opaque_face_count >= mesh->face_count)
3839            continue;
3840
3841        for (int face_index = mesh->opaque_face_count;
3842             face_index < mesh->face_count;
3843             face_index++) {
3844            const ChunkFace *face = &mesh->faces[face_index];
3845
3846            float wx = chunk_ox + (float)face->x;
3847            float wy = (float)face->y;
3848            float wz = chunk_oz + (float)face->z;
3849            Vec3 block_pos = { wx, wy, wz };
3850            int u_size = face->u_size ? face->u_size : 1;
3851            int v_size = face->v_size ? face->v_size : 1;
3852            uint8_t height = face->height ? face->height : 8;
3853            BlockFace face_dir = (BlockFace)face->face;
3854
3855            if (!merged_face_visible(block_pos, face_dir, u_size, v_size,
3856                                     height, cam_pos))
3857                continue;
3858
3859            if (!ensure_translucent_scratch(ctx, w + 1))
3860                continue;
3861
3862            Vec3 face_center = merged_face_center(block_pos, face_dir, u_size,
```

```
                                                v_size, height);
            CameraVertex cv;
            world_to_camera(ctx, face_center, &cv);

            ctx->translucent_scratch[w++] = (TranslucentFaceRef){
                .chunk = chunk,
                .face = face,
                .view_z = cv.z,
                .stable_order = translucent_face_stable_order(chunk, face),
            };
        }
    }

    if (w > 0) {
        if (w > 1) {
            qsort(ctx->translucent_scratch, (size_t)w,
                  sizeof(*ctx->translucent_scratch),
                  compare_translucent_back_to_front);
        }

        for (int i = 0; i < w; i++) {
            const TranslucentFaceRef *ref = &ctx->translucent_scratch[i];
            Vec3 block_pos = {
                (float)(ref->chunk->chunk_x * WORLD_CHUNK_SIZE + ref->face->x),
                (float)ref->face->y,
                (float)(ref->chunk->chunk_z * WORLD_CHUNK_SIZE + ref->face->z),
            };
            uint8_t light_flags = choose_chunk_face_light_flags(
                ctx,
                (BlockID)ref->face->type,
                (BlockFace)ref->face->face,
                ref->face->sky_light,
                ref->face->block_light);

            emit_merged_block_face_lit(ctx, (BlockID)ref->face->type,
                                       block_pos, (BlockFace)ref->face->face,
                                       ref->face->u_size ? ref->face->u_size : 1,
                                       ref->face->v_size ? ref->face->v_size : 1,
                                       ref->face->height ? ref->face->height : 8,
                                       light_flags);
            if (ctx->n_quads >= MAX_QUADS_IN_FLIGHT)
                return ctx->n_quads - before;
        }
    }

    return ctx->n_quads - before;
}

static bool push_screen_textured_quad(RenderContext *ctx,
                                      float x0, float y0,
                                      float x1, float y1,
                                      float u0, float v0,
                                      float u1, float v1,
                                      uint8_t texture_id,
                                      uint8_t extra_flags)
{
    RenderQuad quad = {0};

    quad.texture_id = texture_id;
```

```
    quad.flags = QUAD_FLAG_TEX | extra_flags;
    quad.vertices[0] = (Vertex2D){ x0, y0, 0.0f, u0, v0, 1.0f };
    quad.vertices[1] = (Vertex2D){ x1, y0, 0.0f, u1, v0, 1.0f };
    quad.vertices[2] = (Vertex2D){ x1, y1, 0.0f, u1, v1, 1.0f };
    quad.vertices[3] = (Vertex2D){ x0, y1, 0.0f, u0, v1, 1.0f };
    if (projected_quad_fully_inside_viewport(quad.vertices))
        return stage_projected_quad_no_clip(ctx, &quad);
    return renderer_push_quad(ctx, &quad);
}

static bool push_screen_flat_quad(RenderContext *ctx,
                                  float x0, float y0,
                                  float x1, float y1,
                                  uint8_t palette_index)
{
    return renderer_fill_rect(ctx, x0, y0, x1, y1, palette_index, 0);
}

bool renderer_draw_screen_tile(RenderContext *ctx,
                               float x0, float y0, float x1, float y1,
                               uint8_t texture_id, uint8_t extra_flags)
{
    return push_screen_textured_quad(ctx,
                                     x0, y0, x1, y1,
                                     0.0f, 0.0f, 16.0f, 16.0f,
                                     texture_id,
                                     extra_flags);
}

bool renderer_draw_custom_screen_quad(RenderContext *ctx,
                                      float x0, float y0,
                                      float x1, float y1,
                                      float x2, float y2,
                                      float x3, float y3,
                                      uint8_t texture_id, uint8_t extra_flags)
{
    RenderQuad quad = {0};

    quad.texture_id = texture_id;
    quad.flags = QUAD_FLAG_TEX | extra_flags;
    quad.vertices[0] = (Vertex2D){ x0, y0, 0.0f, 0.0f,  0.0f,  1.0f };
    quad.vertices[1] = (Vertex2D){ x1, y1, 0.0f, 16.0f, 0.0f,  1.0f };
    quad.vertices[2] = (Vertex2D){ x2, y2, 0.0f, 16.0f, 16.0f, 1.0f };
    quad.vertices[3] = (Vertex2D){ x3, y3, 0.0f, 0.0f,  16.0f, 1.0f };
    if (projected_quad_fully_inside_viewport(quad.vertices))
        return stage_projected_quad_no_clip(ctx, &quad);
    return renderer_push_quad(ctx, &quad);
}

bool renderer_draw_world_billboard_tile(RenderContext *ctx,
                                        Vec3 center,
                                        float size_world,
                                        uint8_t texture_id,
                                        uint8_t extra_flags)
{
    CameraVertex camera;
    Vertex2D projected;
    RenderQuad quad = {0};
    float inv_w;
```

```
    float half_px;

    if (!ctx || size_world <= 0.0f)
        return false;

    world_to_camera(ctx, center, &camera);
    if (!project_camera_vertex(ctx, &camera, &projected))
        return false;

    inv_w = 1.0f / camera.z;
    half_px = ctx->current_camera.depth * size_world * 0.5f * inv_w;
    if (half_px < 1.0f)
        half_px = 1.0f;

    quad.texture_id = texture_id;
    quad.flags = QUAD_FLAG_TEX | extra_flags;
    quad.vertices[0] = (Vertex2D){
        projected.x - half_px, projected.y - half_px, projected.z,
        0.0f * inv_w, 0.0f * inv_w, inv_w,
    };
    quad.vertices[1] = (Vertex2D){
        projected.x + half_px, projected.y - half_px, projected.z,
        16.0f * inv_w, 0.0f * inv_w, inv_w,
    };
    quad.vertices[2] = (Vertex2D){
        projected.x + half_px, projected.y + half_px, projected.z,
        16.0f * inv_w, 16.0f * inv_w, inv_w,
    };
    quad.vertices[3] = (Vertex2D){
        projected.x - half_px, projected.y + half_px, projected.z,
        0.0f * inv_w, 16.0f * inv_w, inv_w,
    };

    if (projected_quad_fully_inside_viewport(quad.vertices))
        return stage_projected_quad_no_clip(ctx, &quad);
    return renderer_push_quad(ctx, &quad);
}

bool renderer_fill_rect(RenderContext *ctx,
                        float x0, float y0, float x1, float y1,
                        uint8_t palette_index, uint8_t extra_flags)
{
    RenderQuad quad = {0};

    quad.color_tint = palette_index;
    quad.flags = extra_flags;
    quad.vertices[0] = (Vertex2D){ x0, y0, 0.0f, 0.0f, 0.0f, 1.0f };
    quad.vertices[1] = (Vertex2D){ x1, y0, 0.0f, 0.0f, 0.0f, 1.0f };
    quad.vertices[2] = (Vertex2D){ x1, y1, 0.0f, 0.0f, 0.0f, 1.0f };
    quad.vertices[3] = (Vertex2D){ x0, y1, 0.0f, 0.0f, 0.0f, 1.0f };
    if (projected_quad_fully_inside_viewport(quad.vertices))
        return stage_projected_quad_no_clip(ctx, &quad);
    return renderer_push_quad(ctx, &quad);
}

static bool project_sky_direction(RenderContext *ctx, Vec3 dir, float *screen_x, float *screen_y)
{
    Vec3 world = {
        ctx->current_camera.position.x + dir.x * SKY_DOME_DISTANCE,
```

```
        ctx->current_camera.position.y + dir.y * SKY_DOME_DISTANCE,
        ctx->current_camera.position.z + dir.z * SKY_DOME_DISTANCE,
    };
    CameraVertex camera = {0};
    Vertex2D projected;

    world_to_camera(ctx, world, &camera);
    if (!project_camera_vertex(ctx, &camera, &projected))
        return false;

    *screen_x = projected.x;
    *screen_y = projected.y;
    return true;
}

static uint32_t sky_hash_u32(uint32_t value)
{
    value ^= value >> 16;
    value *= 0x7feb352du;
    value ^= value >> 15;
    value *= 0x846ca68bu;
    value ^= value >> 16;
    return value;
}

static float sky_hash01(uint32_t value)
{
    return (float)(sky_hash_u32(value) & 0x00ffffffu) *
           (1.0f / 16777215.0f);
}

static bool draw_sky_sprite_rotated(RenderContext *ctx,
                                    Vec3 dir,
                                    float width_px,
                                    float height_px,
                                    float rotation,
                                    uint8_t texture_id,
                                    uint8_t extra_flags)
{
    float center_x;
    float center_y;
    float half_w = width_px * 0.5f;
    float half_h = height_px * 0.5f;
    float cr = cosf(rotation);
    float sr = sinf(rotation);
    float rx = cr * half_w;
    float ry = sr * half_w;
    float dx = -sr * half_h;
    float dy = cr * half_h;

    if (!project_sky_direction(ctx, dir, &center_x, &center_y))
        return false;

    return renderer_draw_custom_screen_quad(ctx,
                                            center_x - rx - dx,
                                            center_y - ry - dy,
                                            center_x + rx - dx,
                                            center_y + ry - dy,
                                            center_x + rx + dx,
```

```
                                          center_y + ry + dy,
                                          center_x - rx + dx,
                                          center_y - ry + dy,
                                          texture_id,
                                          extra_flags);
}

static bool draw_sky_sprite(RenderContext *ctx, Vec3 dir, float size_px, uint8_t texture_id)
{
    float center_x;
    float center_y;
    float half = size_px * 0.5f;

    if (!project_sky_direction(ctx, dir, &center_x, &center_y))
        return false;

    return push_screen_textured_quad(ctx,
                                     center_x - half, center_y - half,
                                     center_x + half, center_y + half,
                                     0.0f, 0.0f, 16.0f, 16.0f,
                                     texture_id,
                                     QUAD_FLAG_ALPHA_KEY);
}

static void draw_randomized_stars(RenderContext *ctx)
{
    for (uint32_t i = 0; i < SKY_STAR_COUNT; i++) {
        float azimuth = sky_hash01(i * 13u + 0x51a7u) * (2.0f * PI_F);
        float elevation_t = sky_hash01(i * 17u + 0x932bu);
        float size_t = sky_hash01(i * 19u + 0xb31du);
        float aspect_t = sky_hash01(i * 23u + 0xc0deu);
        float alpha_t = sky_hash01(i * 29u + 0x57a1u);
        float elevation = asinf(0.10f + 0.86f * elevation_t);
        float size_px = 10.0f + 8.0f * size_t;
        float width_px = size_px * (0.86f + 0.28f * aspect_t);
        float height_px = size_px * (1.14f - 0.28f * aspect_t);
        float rotation = sky_hash01(i * 31u + 0x6f3du) * (2.0f * PI_F);
        uint8_t alpha = QUAD_ALPHA_OPAQUE;

        if (alpha_t < 0.18f)
            alpha = QUAD_ALPHA_50;
        else if (alpha_t < 0.62f)
            alpha = QUAD_ALPHA_75;

        draw_sky_sprite_rotated(ctx,
                                direction_from_azimuth_elevation(azimuth,
                                                                 elevation),
                                width_px,
                                height_px,
                                rotation,
                                TEX_TILE_STARS,
                                QUAD_FLAG_ALPHA_KEY | alpha);
    }
}

static RGB24 sample_sky_gradient(const SkyPalette *palette, float t)
{
    if (t <= 0.22f)
        return lerp_rgb24(palette->zenith, palette->high, smoothstepf(0.0f, 0.22f, t));
```

```
    if (t <= 0.62f)
        return lerp_rgb24(palette->high, palette->mid, smoothstepf(0.22f, 0.62f, t));
    return lerp_rgb24(palette->mid, palette->horizon, smoothstepf(0.62f, 1.0f, t));
}

static void draw_sky_gradient(RenderContext *ctx, const SkyPalette *palette)
{
    for (int band = 0; band < SKY_GRADIENT_BANDS; band++) {
        float y0 = SCREEN_HEIGHT * (float)band / (float)SKY_GRADIENT_BANDS;
        float y1 = SCREEN_HEIGHT * (float)(band + 1) / (float)SKY_GRADIENT_BANDS;
        float t = ((float)band + 0.5f) / (float)SKY_GRADIENT_BANDS;
        uint8_t palette_index = (uint8_t)(PAL_SKY_GRADIENT_BASE + band);
        RGB24 color = sample_sky_gradient(palette, t);

        renderer_set_sky_palette_rgb(ctx, (uint8_t)band, color);
        push_screen_flat_quad(ctx, 0.0f, y0, SCREEN_WIDTH, y1, palette_index);
    }
}

static Vec3 sun_direction_for_time(float time_seconds)
{
    float cycle = fmodf(time_seconds / SKY_DAY_LENGTH_SECONDS + 0.18f, 1.0f);

    if (cycle < 0.0f)
        cycle += 1.0f;

    float orbit = cycle * (2.0f * PI_F);
    return normalize_vec3((Vec3){
        cosf(orbit),
        0.92f * sinf(orbit),
        0.35f,
    });
}

static SkyPalette make_sky_palette(Vec3 sun_dir)
{
    float daylight = smoothstepf(-0.18f, 0.08f, sun_dir.y);
    float twilight = clamp01(1.0f - fabsf(sun_dir.y) * 4.5f);
    float night = 1.0f - daylight;
    SkyPalette palette = {
        .zenith = lerp_rgb24(rgb24(0x08, 0x0a, 0x16), rgb24(0x58, 0x96, 0xdb), daylight),
        .high = lerp_rgb24(rgb24(0x15, 0x18, 0x31), rgb24(0x7f, 0xbd, 0xff), daylight),
        .mid = lerp_rgb24(rgb24(0x22, 0x24, 0x46), rgb24(0xad, 0xd8, 0xff), daylight),
        .horizon = lerp_rgb24(rgb24(0x2c, 0x1f, 0x2d), rgb24(0xdf, 0xee, 0xff), daylight),
        .cloud = lerp_rgb24(rgb24(0x54, 0x5c, 0x79), rgb24(0xf5, 0xfa, 0xff), daylight),
        .cloud_shadow = lerp_rgb24(rgb24(0x34, 0x3b, 0x53), rgb24(0xcb, 0xd8, 0xec), daylight),
        .sun_core = rgb24(0xff, 0xee, 0xaa),
        .sun_glow = rgb24(0xff, 0xbd, 0x58),
        .moon = lerp_rgb24(rgb24(0xb9, 0xc3, 0xe0), rgb24(0xe7, 0xeb, 0xf8), night),
        .moon_shadow = lerp_rgb24(rgb24(0x5d, 0x67, 0x85), rgb24(0x9c, 0xa4, 0xc0), night),
    };
    RGB24 sunset = rgb24(0xff, 0xab, 0x61);

    twilight *= 0.35f + 0.65f * night;
    palette.zenith = mix_rgb24(palette.zenith, rgb24(0x52, 0x3d, 0x5d), twilight * 0.22f);
    palette.high = mix_rgb24(palette.high, rgb24(0xa0, 0x5f, 0x62), twilight * 0.28f);
    palette.mid = mix_rgb24(palette.mid, rgb24(0xe7, 0x88, 0x56), twilight * 0.45f);
    palette.horizon = mix_rgb24(palette.horizon, sunset, twilight * 0.78f);
    palette.cloud = mix_rgb24(palette.cloud, rgb24(0xff, 0xc6, 0x8c), twilight * 0.18f);
```

```
4217        palette.cloud_shadow = mix_rgb24(palette.cloud_shadow, rgb24(0xc7, 0x8d, 0x58), twilight * 0.22
            f);
4218        palette.sun_core = mix_rgb24(palette.sun_core, rgb24(0xff, 0xdb, 0x91), twilight * 0.35f);
4219        palette.sun_glow = mix_rgb24(palette.sun_glow, rgb24(0xff, 0x97, 0x4a), twilight * 0.50f);
4220        palette.star = lerp_rgb24(palette.zenith, rgb24(0xff, 0xff, 0xff),
4221                                  smoothstepf(0.15f, 0.95f, night));
4222
4223        return palette;
4224    }
4225
4226    static void upload_sky_palette(RenderContext *ctx, const SkyPalette *palette)
4227    {
4228        renderer_set_palette_rgb(ctx, 0, palette->zenith);
4229        renderer_set_palette_rgb(ctx, PAL_SKY_HIGH, palette->high);
4230        renderer_set_palette_rgb(ctx, PAL_SKY_MID, palette->mid);
4231        renderer_set_palette_rgb(ctx, PAL_SKY_HORIZON, palette->horizon);
4232        renderer_set_palette_rgb(ctx, PAL_CLOUD, palette->cloud);
4233        renderer_set_palette_rgb(ctx, PAL_CLOUD_SHADOW, palette->cloud_shadow);
4234        renderer_set_palette_rgb(ctx, PAL_SUN_CORE, palette->sun_core);
4235        renderer_set_palette_rgb(ctx, PAL_SUN_GLOW, palette->sun_glow);
4236        renderer_set_palette_rgb(ctx, PAL_MOON, palette->moon);
4237        renderer_set_palette_rgb(ctx, PAL_MOON_SHADOW, palette->moon_shadow);
4238        renderer_set_palette_rgb(ctx, PAL_STAR, palette->star);
4239    }
4240
4241    int renderer_draw_sky(RenderContext *ctx, float time_seconds)
4242    {
4243        typedef struct {
4244            float azimuth;
4245            float elevation;
4246            float size_px;
4247            float drift;
4248            float wobble;
4249        } SkySpriteDef;
4250
4251        static const SkySpriteDef cloud_defs[] = {
4252            { 0.15f, 0.48f, 62.0f,  0.014f, 0.23f },
4253            { 1.65f, 0.41f, 54.0f, -0.010f, 0.19f },
4254            { 3.20f, 0.52f, 68.0f,  0.012f, 0.17f },
4255            { 4.85f, 0.37f, 58.0f, -0.008f, 0.21f },
4256        };
4257        int before = ctx->n_quads;
4258        Vec3 sun_dir;
4259        Vec3 moon_dir;
4260        SkyPalette palette;
4261        float daylight;
4262        float night;
4263
4264        if (!ctx)
4265            return 0;
4266
4267        /* Continuous time drives sprite drift below; quantized time drives every
4268         * palette computation so hw_sky_epoch stays stable for ~15 frames at
4269         * 30 FPS and sky_band_reuse can hit on primer-only bands. Both this path
4270         * and renderer_draw_world use palette_time_for() so they cannot disagree
4271         * across a 32-step daylight_key boundary and re-flush the palette. */
4272        float palette_time = palette_time_for(time_seconds);
4273        /* Palette and lighting use quantized time so hw_sky_epoch stays stable. */
4274        Vec3 palette_sun_dir = sun_direction_for_time(palette_time);
```

```
    update_world_light_from_sun(ctx, palette_sun_dir);
    daylight = ctx->world_daylight;
    night = 1.0f - daylight;
    palette = make_sky_palette(palette_sun_dir);
    upload_sky_palette(ctx, &palette);
    draw_sky_gradient(ctx, &palette);

    /* Sprite positions use continuous time so movement is smooth. */
    sun_dir = sun_direction_for_time(time_seconds);
    moon_dir = (Vec3){ -sun_dir.x, -sun_dir.y, -sun_dir.z };

    if (night > 0.2f)
        draw_randomized_stars(ctx);

    if (sun_dir.y > 0.0f)
        draw_sky_sprite(ctx, sun_dir, 34.0f, TEX_TILE_SUN);
    if (moon_dir.y > 0.0f)
        draw_sky_sprite(ctx, moon_dir, 30.0f, TEX_TILE_MOON);

    for (size_t i = 0; i < sizeof(cloud_defs) / sizeof(cloud_defs[0]); i++) {
        float azimuth = cloud_defs[i].azimuth + cloud_defs[i].drift * time_seconds;
        float elevation = cloud_defs[i].elevation +
                          0.04f * sinf(time_seconds * cloud_defs[i].wobble + (float)i * 1.7f);

        draw_sky_sprite(ctx,
                        direction_from_azimuth_elevation(azimuth, elevation),
                        cloud_defs[i].size_px,
                        TEX_TILE_CLOUD);
    }

    return ctx->n_quads - before;
}

/* Project a world-space vertical line segment to screen and draw it as a
 * 1-pixel-wide line using a screen-space quad. Returns true if visible. */
static bool draw_projected_line(RenderContext *ctx,
                                float wx0, float wy0, float wz0,
                                float wx1, float wy1, float wz1,
                                uint8_t palette_index)
{
    CameraVertex cam0, cam1;
    Vertex2D scr0, scr1;

    world_to_camera(ctx, (Vec3){wx0, wy0, wz0}, &cam0);
    world_to_camera(ctx, (Vec3){wx1, wy1, wz1}, &cam1);
    if (!project_camera_vertex(ctx, &cam0, &scr0) ||
        !project_camera_vertex(ctx, &cam1, &scr1))
        return false;

    float dx = scr1.x - scr0.x;
    float dy = scr1.y - scr0.y;
    float len = sqrtf(dx * dx + dy * dy);
    if (len < 0.1f) return false;

    /* Normal vector to the line in screen space */
    float nx = -dy / len;
    float ny = dx / len;

    /* 0.5px half-width for a 1-pixel wide line */
```

```
    float hw = 0.5f;
    nx *= hw;
    ny *= hw;

    /* Build a quad connecting the two points, 1px thick.
     * Force z=0.0f (nearest possible depth) so the debug lines
     * draw cleanly over all scene blocks. */
    RenderQuad quad = {0};
    quad.color_tint = palette_index;
    quad.flags = 0;

    float inv_w0 = scr0.one_over_w;
    float inv_w1 = scr1.one_over_w;

    quad.vertices[0] = (Vertex2D){ scr0.x - nx, scr0.y - ny, 0.0f, 0.0f, inv_w0, 0.0f };
    quad.vertices[1] = (Vertex2D){ scr0.x + nx, scr0.y + ny, 0.0f, 0.0f, inv_w0, 0.0f };
    quad.vertices[2] = (Vertex2D){ scr1.x + nx, scr1.y + ny, 0.0f, 0.0f, inv_w1, 0.0f };
    quad.vertices[3] = (Vertex2D){ scr1.x - nx, scr1.y - ny, 0.0f, 0.0f, inv_w1, 0.0f };

    if (projected_quad_fully_inside_viewport(quad.vertices))
        return stage_projected_quad_no_clip(ctx, &quad);
    return renderer_push_quad(ctx, &quad);
}

int renderer_draw_chunk_borders(RenderContext *ctx,
                                float player_x, float player_z,
                                int render_distance)
{
    if (!ctx || render_distance <= 0)
        return 0;

    int drawn = 0;
    int player_cx = (int)floorf(player_x / (float)WORLD_CHUNK_SIZE);
    int player_cz = (int)floorf(player_z / (float)WORLD_CHUNK_SIZE);

    /* Draw vertical lines at chunk boundaries within render distance.
     * Lines run from y=0 to y=WORLD_CHUNK_HEIGHT at each chunk edge. */
    float y_bot = 0.0f;
    float y_top = (float)WORLD_CHUNK_HEIGHT;

    for (int dx = -render_distance; dx <= render_distance + 1; dx++) {
        float wx = (float)((player_cx + dx) * WORLD_CHUNK_SIZE);
        /* Highlight the player's chunk borders in yellow (palette 8),
         * other chunk borders in white (palette 5). */
        bool is_player_edge = (dx == 0 || dx == 1);
        uint8_t color = is_player_edge ? 8 : 5;

        for (int dz = -render_distance; dz <= render_distance + 1; dz++) {
            float wz = (float)((player_cz + dz) * WORLD_CHUNK_SIZE);
            bool is_player_z = (dz == 0 || dz == 1);
            uint8_t line_color = (is_player_edge && is_player_z) ? 8 : color;

            /* Vertical line at this grid intersection. */
            if (draw_projected_line(ctx, wx, y_bot, wz, wx, y_top, wz,
                                    line_color))
                drawn++;
        }
    }

```

```
    return drawn;
}

bool renderer_draw_crosshair(RenderContext *ctx)
{
    const float cx = SCREEN_WIDTH * 0.5f;
    const float cy = SCREEN_HEIGHT * 0.5f;
    const float half = 4.5f * HUD_SCALE;
    const float thickness = 0.75f * HUD_SCALE;
    const uint8_t white = 5;
    bool ok = true;

    ok &= renderer_fill_rect(ctx,
                             cx - half, cy - thickness * 0.5f,
                             cx + half, cy + thickness * 0.5f,
                             white, QUAD_ALPHA_75);
    ok &= renderer_fill_rect(ctx,
                             cx - thickness * 0.5f, cy - half,
                             cx + thickness * 0.5f, cy + half,
                             white, QUAD_ALPHA_75);
    return ok;
}
```

Complete `voxel_gpu.sv` RTL listing

Listing 3: Complete `voxel_gpu.sv` RTL source.

```
// voxel_gpu.sv - SDRAM-backed display path with RGB565 external frames.
//
// The rasterizer renders at true 640x480 through an explicit 640x60 BRAM band
// cache. Userspace bins descriptors into eight vertical passes and brackets each
// pass with BEGIN_BAND / END_BAND CSRs. Palette entries are still the
// source-color ABI, but resolved pixels are stored as RGB565 so translucent
// quads can alpha blend against the existing destination pixel.
// FLIP no longer copies a full BRAM framebuffer. Instead:
//   * BEGIN_BAND clears the resident Z band; untouched color is lazy and
//     regenerated as sky/clear where Z still equals 0xFFFF,
//   * the rasterizer writes only pixels that fall inside that resident band,
//   * END_BAND flushes the color band to the inactive SDRAM color frame, and
//   * scanout reads the active SDRAM frame through small line buffers.
//
// This keeps BRAM usage close to the indexed design while making the external
// framebuffer itself true color.
//
// Reading map for teammates:
//   1. Register/FIFO front door: HPS writes CSRs and descriptor words.
//   2. Main engine FSM: fetches one descriptor, prepares one resident band,
//      rasterizes pixels, and kicks background band flushes.
//   3. Pixel pipeline: edge test -> reciprocal/texture -> palette -> fog ->
//      color/Z commit. Valid bits move alongside data through every stage.
//   4. SDRAM background flush and VGA scanout run beside the main FSM. Most
//      "weird" gates below protect ownership between those concurrent users.
//   5. Leaf helper modules/`.svh` files contain pure math/color helpers only;
//      stateful timing stays in this module so the control flow remains visible.

module voxel_gpu (
    input  logic        clk,
```

```
    input  logic        reset,

    // Avalon-MM slave (address units = WORDS; writedata is 32 bits).
    input  logic [12:0] address,
    input  logic        chipselect,
    input  logic        write,
    input  logic [31:0] writedata,
    input  logic  [3:0] byteenable,
    output logic [31:0] readdata,

    // VGA conduit
    output logic  [7:0] VGA_R,
    output logic  [7:0] VGA_G,
    output logic  [7:0] VGA_B,
    output logic        VGA_CLK,
    output logic        VGA_HS,
    output logic        VGA_VS,
    output logic        VGA_BLANK_n,
    output logic        VGA_SYNC_n,

    // Board SDR SDRAM conduit
    output logic [12:0] DRAM_ADDR,
    output logic  [1:0] DRAM_BA,
    output logic        DRAM_CAS_N,
    output logic        DRAM_CKE,
    output logic        DRAM_CLK,
    output logic        DRAM_CS_N,
    inout  wire [15:0]  DRAM_DQ,
    output logic        DRAM_LDQM,
    output logic        DRAM_RAS_N,
    output logic        DRAM_UDQM,
    output logic        DRAM_WE_N
);

    // ------------------------------------------------------------------------
    // Register Map, Frame Geometry, and Packed Descriptor Constants
    // ------------------------------------------------------------------------

    localparam logic [12:0] ADDR_CONTROL  = 13'h000;
    localparam logic [12:0] ADDR_STATUS   = 13'h001;
    localparam logic [12:0] ADDR_FRAMECNT = 13'h002;
    localparam logic [12:0] ADDR_PAL_ADDR = 13'h003;
    localparam logic [12:0] ADDR_PAL_DATA = 13'h004;
    localparam logic [12:0] ADDR_FOG_RANGE = 13'h005;
    localparam logic [12:0] ADDR_FOG_CTRL = 13'h006;
    localparam logic [12:0] ADDR_EXTMEM_CTRL = 13'h007;
    localparam logic [12:0] ADDR_EXTMEM_FRONT = 13'h008;
    localparam logic [12:0] ADDR_EXTMEM_BACK = 13'h009;
    localparam logic [12:0] ADDR_EXTMEM_STRIDE = 13'h00A;
    localparam logic [12:0] ADDR_EXTMEM_TILE = 13'h00B;
    localparam logic [12:0] ADDR_EXTMEM_STAT = 13'h00C;
    localparam logic [12:0] ADDR_BAND_INDEX = 13'h00D;
    localparam logic [12:0] ADDR_BAND_CTRL = 13'h00E;
    localparam logic [12:0] ADDR_BAND_WINDOW = 13'h00F;
    localparam logic [12:0] ADDR_PERF_DRAW_ACT  = 13'h010;
    localparam logic [12:0] ADDR_PERF_DRAW_IDLE = 13'h011;
    localparam logic [12:0] ADDR_PERF_FLUSH_ACT = 13'h012;
    localparam logic [12:0] ADDR_PERF_FLUSH_STL = 13'h013;
    localparam logic [12:0] ADDR_PERF_INIT      = 13'h014;
```

```
    localparam logic [12:0] ADDR_PERF_LOAD      = 13'h015;
    localparam logic [12:0] ADDR_PERF_FLUSH_LOAD  = 13'h016;
    localparam logic [12:0] ADDR_PERF_FLUSH_FIFO  = 13'h017;
    localparam logic [12:0] ADDR_PERF_FLUSH_DATA  = 13'h018;
    localparam logic [12:0] ADDR_PERF_FLUSH_DRAIN = 13'h019;
    localparam logic [12:0] ADDR_SKY_PAL_ADDR = 13'h01A;
    localparam logic [12:0] ADDR_SKY_PAL_DATA = 13'h01B;
    localparam logic [12:0] ADDR_FIFO_LO  = 13'h400;  // 0x1000
    localparam logic [12:0] ADDR_FIFO_HI  = 13'h800;  // 0x2000 (exclusive)

    localparam int FB_WIDTH       = 640;
    localparam int BAND_HEIGHT    = 60;
    localparam int BAND_PIXELS    = FB_WIDTH * BAND_HEIGHT;
    localparam int FIFO_DEPTH     = 1024;
    localparam int BASE_QUAD_WORDS = 16;
    localparam int UV_QUAD_WORDS   = 9;
    localparam int MAX_DESC_WORDS  = BASE_QUAD_WORDS + UV_QUAD_WORDS;
    localparam logic [5:0] BASE_QUAD_WORDS_6 = 6'd16;
    localparam logic [5:0] MAX_DESC_WORDS_6  = 6'd25;
    localparam int TEXTURE_BYTES   = 128 * 16 * 16;
    localparam int LINE_WORDS      = FB_WIDTH;
    /* Stage background-flush words into the 512-word SDRAM WR FIFO while
     * scanout owns the bus; keep headroom for in-flight pushes. */
    localparam logic [8:0] COPY_WR_FIFO_HIGH_WATER = 9'd224;
    localparam logic [7:0] COPY_DRAIN_CYCLES = 8'd128;
    /* Drain every valid pipeline stage between fetch and color commit. */
    localparam logic [3:0] DRAW_FLUSH_CYCLES = 4'd14;
    localparam logic [24:0] READ_BURST_WORDS_25 = 25'd64;
    localparam logic [9:0]  LINE_WORDS_10 = 10'd640;
    localparam logic [8:0]  COPY_BURST_WORDS_9 = 9'd128;
    localparam logic [8:0]  READ_BURST_WORDS_9 = 9'd64;
    localparam logic [10:0] ACTIVE_WRITE_END_HCOUNT = 11'd1120;
    localparam int EXTMEM_SKY_GRADIENT_CLEAR_BIT = 5;
    localparam logic [7:0] PAL_SKY_GRADIENT_BASE = 8'd40;
    localparam logic [7:0] PAL_SKY_GRADIENT_LAST = 8'd63;
    localparam int SKY_GRADIENT_COLORS = 24;
    localparam logic [4:0] SKY_GRADIENT_LAST_INDEX = 5'd23;
    localparam logic [5:0] SKY_GRADIENT_COLORS_6 = 6'd24;
    localparam logic [15:0] Z_CLEAR_SENTINEL = 16'hFFFF;
    localparam logic [15:0] Z_VALID_FAR      = 16'hFFFE;
    localparam logic [31:0] DEFAULT_EXTMEM_CTRL = 32'h0000_002B;
    localparam logic [31:0] DEFAULT_EXTMEM_FRONT_BASE = 32'd0;
    localparam logic [31:0] DEFAULT_EXTMEM_BACK_BASE = 32'd1048576; // 1MB
    localparam logic [31:0] DEFAULT_EXTMEM_STRIDE = 32'd1280;
    localparam logic [17:0] SDRAM_POWERUP_HOLD_LAST = 18'd199999;
    localparam logic [15:0] SDRAM_INIT_WAIT_LAST = 16'd31999;
    localparam int FLAG_TEX_BIT        = 0;
    localparam int FLAG_ZTEST_BIT      = 1;
    localparam int FLAG_ALPHA_KEY_BIT  = 2;
    localparam int FLAG_FOG_BIT        = 3;
    localparam int FLAG_LIGHT_LSB      = 4;
    localparam int FLAG_LIGHT_MSB      = 5;
    localparam int FLAG_ALPHA_LSB      = 6;
    localparam int FLAG_ALPHA_MSB      = 7;

    // ------------------------------------------------------------------------
    // Main Engine State Map
    // ------------------------------------------------------------------------
    //
```

```
149     // Descriptor/raster path:
150     //   IDLE -> FETCH -> SETUP -> DRAW -> DRAW_FLUSH -> IDLE
151     //
152     // Cache path:
153     //   BEGIN_BAND enters CACHE_INIT to clear the resident Z band. END_BAND
154     //   starts the independent background flush controller and does not consume
155     //   a main-FSM state.
156     typedef enum logic [2:0] {
157         ST_IDLE             = 3'd0,
158         ST_CLEAR            = 3'd1,
159         ST_FETCH            = 3'd2,
160         ST_SETUP            = 3'd3,
161         ST_DRAW             = 3'd4,
162         ST_DRAW_FLUSH       = 3'd5,
163         ST_CACHE_INIT       = 3'd6
164     } engine_state_t;
165
166     engine_state_t state;
167
168     // ------------------------------------------------------------------------
169     // CSRs, Frame Selection, and Debug Counters
170     // ------------------------------------------------------------------------
171
172     logic        ctrl_en;
173     logic        ctrl_ien;
174     logic        ctrl_flp_pending;
175     logic        clear_pending;
176     logic [31:0] frame_count;
177     logic  [7:0] pal_addr;
178     logic  [4:0] sky_pal_addr;
179     logic        fog_enable;
180     logic  [7:0] fog_color;
181     logic [15:0] fog_start_dist;
182     logic [15:0] fog_end_dist;
183     logic [15:0] fog_inv_proj_sq;
184     logic [31:0] extmem_ctrl;
185     logic [31:0] extmem_front_base;
186     logic [31:0] extmem_back_base;
187     logic [31:0] extmem_stride_bytes;
188     logic [31:0] extmem_tile_cfg;
189     logic [31:0] extmem_dma_status;
190     logic        vsy_latch;
191     logic        display_sel;             // 0 => extmem_front_base is visible
192     logic        display_valid;
193     logic        copy_target_sel;         // inactive SDRAM color frame rendered this frame
194     logic        copy_complete_pending;   // completed render frame waits for vsync display swap
195
196     // Per-frame perf counters (free-running, reset on FLIP write).
197     // Software reads via ADDR_PERF_* before issuing the FLIP that ends
198     // the frame, so the values reflect the just-completed frame.
199     // 50 MHz × 50000 cycles = 1 ms.
200     logic [31:0] perf_draw_active;   // ST_DRAW/ST_DRAW_FLUSH committing pixel
201     logic [31:0] perf_draw_idle;     // ST_DRAW/ST_DRAW_FLUSH no commit (starved)
202     logic [31:0] perf_flush_active;  // bg flush running AND word pushed this cyc
203     logic [31:0] perf_flush_stall;   // bg flush running AND no push (SDRAM stall)
204     logic [31:0] perf_init;          // ST_CACHE_INIT
205     logic [31:0] perf_load;          // reserved ABI counter
206     logic [31:0] perf_flush_wait_load;  // bg flush waiting to launch WR stream
207     logic [31:0] perf_flush_wait_fifo;  // bg flush blocked by write FIFO/headroom
```

```
208     logic [31:0] perf_flush_wait_data;  // bg flush waiting on cache/sky word
209     logic [31:0] perf_flush_wait_drain; // bg flush final SDRAM/FIFO drain
210
211     // Palette path: draw_pipe_color -> pal_rd address flop -> palette[] ->
212     // plr data flop -> RGB565/fog. The explicit address/data split keeps
213     // the timing fixed whether Quartus maps the array to logic or memory.
214     (* ramstyle = "logic" *) logic [23:0] palette [0:255];
215     (* ramstyle = "logic" *) logic [23:0] sky_palette [0:SKY_GRADIENT_COLORS-1];
216     (* ramstyle = "M10K" *) logic [31:0] fifo_mem [0:FIFO_DEPTH-1];
217     (* ramstyle = "logic" *) logic [31:0] recip_lut [0:1024];
218
219     logic [9:0]  fifo_wr_ptr;
220     logic [9:0]  fifo_rd_ptr;
221     logic [10:0] fifo_count;
222     wire         fifo_full  = (fifo_count == 11'd1024);
223     wire         fifo_empty = (fifo_count == 0);
224     wire [31:0]  fifo_head  = fifo_mem[fifo_rd_ptr];
225
226     // ------------------------------------------------------------------
227     // Descriptor Fetch and Raster Setup Registers
228     // ------------------------------------------------------------------
229
230     logic [31:0] desc_words [0:MAX_DESC_WORDS-1];
231     logic  [5:0] fetch_count;
232     logic [31:0] prefetch_words [0:MAX_DESC_WORDS-1];
233     logic  [5:0] prefetch_count;
234     logic  [5:0] prefetch_target_words;
235     logic        prefetch_active;
236     logic        prefetch_valid;
237     logic  [3:0] draw_flush_count;
238
239     logic [15:0] clear_addr;
240     logic [15:0] draw_row_base;
241     logic  [9:0] draw_x_min, draw_x_max, draw_x_cur;
242     logic  [8:0] draw_y_min;
243     logic  [8:0] draw_y_max, draw_y_cur;
244     logic        draw_row_inside;  // set when draw_inside seen on current row
245     logic  [7:0] draw_tex_or_color;
246     logic  [7:0] draw_flags;
247     logic [15:0] draw_z0;
248     logic signed [15:0] draw_dz_dx;
249     logic signed [15:0] draw_dz_dy;
250     logic signed [31:0] draw_uw_0;
251     logic signed [31:0] draw_uw_dx;
252     logic signed [31:0] draw_uw_dy;
253     logic signed [31:0] draw_vw_0;
254     logic signed [31:0] draw_vw_dx;
255     logic signed [31:0] draw_vw_dy;
256     logic signed [31:0] draw_iw_0;
257     logic signed [31:0] draw_iw_dx;
258     logic signed [31:0] draw_iw_dy;
259     logic signed [31:0] edge_A [0:3];
260     logic signed [31:0] edge_B [0:3];
261     logic signed [31:0] edge_C [0:3];
262     logic signed [63:0] edge_row_val [0:3];
263     logic signed [63:0] edge_cur_val [0:3];
264     logic signed [47:0] z_row_val;
265     logic signed [47:0] z_cur_val;
266     logic signed [63:0] uw_row_val, uw_cur_val;
```

```
    logic signed [63:0] vw_row_val, vw_cur_val;
    logic signed [63:0] iw_row_val, iw_cur_val;
    // ---------------------------------------------------------------------
    // Pixel Pipeline Registers
    // ---------------------------------------------------------------------
    //
    // Unsuffixed lane = even pixel of the 2-pixel pair.
    // `_o` lane       = odd pixel of the same pair.
    //
    // Stage notes name the field that changes; unmentioned metadata is carried
    // forward to keep the two lanes aligned.

    // pipe0: raster pair accepted; cache read addresses, z, and UV/w are sampled.
    logic        pipe0_valid;
    logic        pipe0_inside;
    logic        pipe0_ztest;
    logic        pipe0_textured;
    logic        pipe0_alpha_key;
    logic  [1:0] pipe0_alpha;
    logic        pipe0_fog;
    logic  [1:0] pipe0_light_bank;
    logic  [7:0] pipe0_tex_or_color;
    logic [15:0] pipe0_addr;
    logic [15:0] pipe0_z;
    logic  [9:0] pipe0_x;
    logic  [8:0] pipe0_y;
    logic signed [31:0] pipe0_uw_q;
    logic signed [31:0] pipe0_vw_q;
    logic [31:0] pipe0_iw_q;
    // recip0: 1/w is normalized and the LUT index/fraction is derived.
    logic        recip0_valid;
    logic        recip0_inside;
    logic        recip0_ztest;
    logic        recip0_textured;
    logic        recip0_alpha_key;
    logic  [1:0] recip0_alpha;
    logic        recip0_fog;
    logic  [1:0] recip0_light_bank;
    logic  [7:0] recip0_tex_or_color;
    logic [15:0] recip0_addr;
    logic [15:0] recip0_z;
    logic  [9:0] recip0_x;
    logic  [8:0] recip0_y;
    logic signed [31:0] recip0_uw_q;
    logic signed [31:0] recip0_vw_q;
    logic        recip0_iw_zero;
    logic  [5:0] recip0_iw_msb;
    logic [31:0] recip0_iw_norm_q;
    // recip1: destination color/Z return from the band cache; reciprocal LUT read.
    logic        recip1_valid;
    logic        recip1_inside;
    logic        recip1_ztest;
    logic        recip1_textured;
    logic        recip1_alpha_key;
    logic  [1:0] recip1_alpha;
    logic        recip1_fog;
    logic  [1:0] recip1_light_bank;
    logic  [7:0] recip1_tex_or_color;
    logic [15:0] recip1_addr;
```

```
    logic [15:0] recip1_z;
    logic [15:0] recip1_z_ref;
    logic [15:0] recip1_dst_rgb565;
    logic  [9:0] recip1_x;
    logic  [8:0] recip1_y;
    logic signed [31:0] recip1_uw_q;
    logic signed [31:0] recip1_vw_q;
    logic        recip1_iw_zero;
    logic  [5:0] recip1_iw_msb;
    logic  [5:0] recip1_iw_lut_frac;
    logic [31:0] recip1_w_norm_lo;
    logic [31:0] recip1_w_norm_hi;
    // recip2: reciprocal interpolation finishes; normalized w is registered.
    logic        recip2_valid;
    logic        recip2_inside;
    logic        recip2_ztest;
    logic        recip2_textured;
    logic        recip2_alpha_key;
    logic  [1:0] recip2_alpha;
    logic        recip2_fog;
    logic  [1:0] recip2_light_bank;
    logic  [7:0] recip2_tex_or_color;
    logic [15:0] recip2_addr;
    logic [15:0] recip2_z;
    logic [15:0] recip2_z_ref;
    logic [15:0] recip2_dst_rgb565;
    logic  [9:0] recip2_x;
    logic  [8:0] recip2_y;
    logic signed [31:0] recip2_uw_q;
    logic signed [31:0] recip2_vw_q;
    logic        recip2_iw_zero;
    logic  [5:0] recip2_iw_msb;
    logic [31:0] recip2_w_norm_q;
    // pipe1: w is denormalized and aligned with UV and destination metadata.
    logic        pipe1_valid;
    logic        pipe1_inside;
    logic        pipe1_ztest;
    logic        pipe1_textured;
    logic        pipe1_alpha_key;
    logic  [1:0] pipe1_alpha;
    logic        pipe1_fog;
    logic  [1:0] pipe1_light_bank;
    logic  [7:0] pipe1_tex_or_color;
    logic [15:0] pipe1_addr;
    logic [15:0] pipe1_z;
    logic [15:0] pipe1_z_ref;
    logic [15:0] pipe1_dst_rgb565;
    logic  [9:0] pipe1_x;
    logic  [8:0] pipe1_y;
    logic signed [31:0] pipe1_uw_q;
    logic signed [31:0] pipe1_vw_q;
    logic [31:0] pipe1_w_q;
    // tex0: perspective UV products are registered.
    logic        tex0_valid;
    logic        tex0_inside;
    logic        tex0_ztest;
    logic        tex0_textured;
    logic        tex0_alpha_key;
    logic  [1:0] tex0_alpha;
```

```
    logic        tex0_fog;
    logic  [1:0] tex0_light_bank;
    logic  [7:0] tex0_tex_or_color;
    logic [15:0] tex0_addr;
    logic [15:0] tex0_z;
    logic [15:0] tex0_z_ref;
    logic [15:0] tex0_dst_rgb565;
    logic  [9:0] tex0_x;
    logic  [8:0] tex0_y;
    logic [31:0] tex0_w_q;
    logic signed [63:0] tex0_u_prod;
    logic signed [63:0] tex0_v_prod;
    // pipe2: texture coordinates become atlas addresses.
    logic        pipe2_valid;
    logic        pipe2_inside;
    logic        pipe2_ztest;
    logic        pipe2_textured;
    logic        pipe2_alpha_key;
    logic  [1:0] pipe2_alpha;
    logic        pipe2_fog;
    logic  [1:0] pipe2_light_bank;
    logic  [7:0] pipe2_tex_or_color;
    logic [15:0] pipe2_addr;
    logic [15:0] pipe2_z;
    logic [15:0] pipe2_z_ref;
    logic [15:0] pipe2_dst_rgb565;
    logic  [9:0] pipe2_x;
    logic  [8:0] pipe2_y;
    logic [31:0] pipe2_w_q;
    logic [14:0] pipe2_tex_addr;
    // draw_pipe: ROM texel data is aligned with color/Z metadata.
    logic        draw_pipe_valid;
    logic        draw_pipe_inside;
    logic        draw_pipe_ztest;
    logic        draw_pipe_textured;
    logic        draw_pipe_alpha_key;
    logic  [1:0] draw_pipe_alpha;
    logic        draw_pipe_fog;
    logic  [1:0] draw_pipe_light_bank;
    logic  [7:0] draw_pipe_tex_or_color;
    logic [15:0] draw_pipe_addr;
    logic [15:0] draw_pipe_z;
    logic [15:0] draw_pipe_z_ref;
    logic [15:0] draw_pipe_dst_rgb565;
    logic  [9:0] draw_pipe_x;
    logic  [8:0] draw_pipe_y;
    logic [31:0] draw_pipe_w_q;
    logic        draw_is_band_primer;

    // pal_rd: palette/fog addresses are flopped before the array read.
    logic        pal_rd_valid;
    logic        pal_rd_pass;
    logic        pal_rd_ztest;
    logic  [1:0] pal_rd_alpha;
    logic        pal_rd_fog;
    logic [15:0] pal_rd_addr;
    logic [15:0] pal_rd_z;
    logic  [7:0] pal_rd_src_addr;
    logic  [7:0] pal_rd_fog_addr;
```

```
    logic        pal_rd_src_sky;
    logic  [4:0] pal_rd_sky_index;
    logic [15:0] pal_rd_dst_rgb565;
    logic [31:0] pal_rd_w_q;
    logic [33:0] pal_rd_ray_scale_q16;

    // plr: palette/fog RGB values are captured and kept with fog metadata.
    logic        plr_valid;
    logic        plr_pass;
    logic        plr_ztest;
    logic  [1:0] plr_alpha;
    logic        plr_fog;
    logic [15:0] plr_addr;
    logic [15:0] plr_z;
    logic [23:0] plr_src_rgb;
    logic [15:0] plr_dst_rgb565;
    logic [23:0] plr_fog_rgb;
    logic [31:0] plr_w_q;
    logic [33:0] plr_ray_scale_q16;

    // fog0: RGB888 converts to RGB565; radial distance multiply starts.
    logic        fog0_valid;
    logic        fog0_pass;
    logic        fog0_ztest;
    logic  [1:0] fog0_alpha;
    logic        fog0_fog;
    logic [15:0] fog0_addr;
    logic [15:0] fog0_z;
    logic [15:0] fog0_src_rgb565;
    logic [15:0] fog0_dst_rgb565;
    logic [15:0] fog0_fog_rgb565;
    logic [31:0] fog0_w_q;
    logic [33:0] fog0_ray_scale_q16;
    // fog1: radial distance is in Q8.8 for fog blend.
    logic        fog1_valid;
    logic        fog1_pass;
    logic        fog1_ztest;
    logic  [1:0] fog1_alpha;
    logic        fog1_fog;
    logic [15:0] fog1_addr;
    logic [15:0] fog1_z;
    logic [15:0] fog1_src_rgb565;
    logic [15:0] fog1_dst_rgb565;
    logic [15:0] fog1_fog_rgb565;
    logic [15:0] fog1_radial_q8_8;
    // commit: final RGB565 and Z write-back controls.
    logic        commit_valid;
    logic        commit_pass;
    logic        commit_ztest;
    logic [15:0] commit_addr;
    logic [15:0] commit_z;
    logic [15:0] commit_color;

    /* Odd lane for the 2 px/cycle raster pipe. Existing unsuffixed
     * registers are lane0/even; `_o` registers carry lane1/odd. */
    logic        pipe0_valid_o;
    logic        pipe0_inside_o;
    logic        pipe0_ztest_o;
    logic        pipe0_textured_o;
```

```
    logic        pipe0_alpha_key_o;
    logic  [1:0] pipe0_alpha_o;
    logic        pipe0_fog_o;
    logic  [1:0] pipe0_light_bank_o;
    logic  [7:0] pipe0_tex_or_color_o;
    logic [15:0] pipe0_addr_o;
    logic [15:0] pipe0_z_o;
    logic  [9:0] pipe0_x_o;
    logic  [8:0] pipe0_y_o;
    logic signed [31:0] pipe0_uw_q_o;
    logic signed [31:0] pipe0_vw_q_o;
    logic [31:0] pipe0_iw_q_o;
    logic        recip0_valid_o;
    logic        recip0_inside_o;
    logic        recip0_ztest_o;
    logic        recip0_textured_o;
    logic        recip0_alpha_key_o;
    logic  [1:0] recip0_alpha_o;
    logic        recip0_fog_o;
    logic  [1:0] recip0_light_bank_o;
    logic  [7:0] recip0_tex_or_color_o;
    logic [15:0] recip0_addr_o;
    logic [15:0] recip0_z_o;
    logic  [9:0] recip0_x_o;
    logic  [8:0] recip0_y_o;
    logic signed [31:0] recip0_uw_q_o;
    logic signed [31:0] recip0_vw_q_o;
    logic        recip0_iw_zero_o;
    logic  [5:0] recip0_iw_msb_o;
    logic [31:0] recip0_iw_norm_q_o;
    logic        recip1_valid_o;
    logic        recip1_inside_o;
    logic        recip1_ztest_o;
    logic        recip1_textured_o;
    logic        recip1_alpha_key_o;
    logic  [1:0] recip1_alpha_o;
    logic        recip1_fog_o;
    logic  [1:0] recip1_light_bank_o;
    logic  [7:0] recip1_tex_or_color_o;
    logic [15:0] recip1_addr_o;
    logic [15:0] recip1_z_o;
    logic [15:0] recip1_z_ref_o;
    logic [15:0] recip1_dst_rgb565_o;
    logic  [9:0] recip1_x_o;
    logic  [8:0] recip1_y_o;
    logic signed [31:0] recip1_uw_q_o;
    logic signed [31:0] recip1_vw_q_o;
    logic        recip1_iw_zero_o;
    logic  [5:0] recip1_iw_msb_o;
    logic  [5:0] recip1_iw_lut_frac_o;
    logic [31:0] recip1_w_norm_lo_o;
    logic [31:0] recip1_w_norm_hi_o;
    logic        recip2_valid_o;
    logic        recip2_inside_o;
    logic        recip2_ztest_o;
    logic        recip2_textured_o;
    logic        recip2_alpha_key_o;
    logic  [1:0] recip2_alpha_o;
    logic        recip2_fog_o;
```

```
    logic  [1:0] recip2_light_bank_o;
    logic  [7:0] recip2_tex_or_color_o;
    logic [15:0] recip2_addr_o;
    logic [15:0] recip2_z_o;
    logic [15:0] recip2_z_ref_o;
    logic [15:0] recip2_dst_rgb565_o;
    logic  [9:0] recip2_x_o;
    logic  [8:0] recip2_y_o;
    logic signed [31:0] recip2_uw_q_o;
    logic signed [31:0] recip2_vw_q_o;
    logic        recip2_iw_zero_o;
    logic  [5:0] recip2_iw_msb_o;
    logic [31:0] recip2_w_norm_q_o;
    logic        pipe1_valid_o;
    logic        pipe1_inside_o;
    logic        pipe1_ztest_o;
    logic        pipe1_textured_o;
    logic        pipe1_alpha_key_o;
    logic  [1:0] pipe1_alpha_o;
    logic        pipe1_fog_o;
    logic  [1:0] pipe1_light_bank_o;
    logic  [7:0] pipe1_tex_or_color_o;
    logic [15:0] pipe1_addr_o;
    logic [15:0] pipe1_z_o;
    logic [15:0] pipe1_z_ref_o;
    logic [15:0] pipe1_dst_rgb565_o;
    logic  [9:0] pipe1_x_o;
    logic  [8:0] pipe1_y_o;
    logic signed [31:0] pipe1_uw_q_o;
    logic signed [31:0] pipe1_vw_q_o;
    logic [31:0] pipe1_w_q_o;
    logic        tex0_valid_o;
    logic        tex0_inside_o;
    logic        tex0_ztest_o;
    logic        tex0_textured_o;
    logic        tex0_alpha_key_o;
    logic  [1:0] tex0_alpha_o;
    logic        tex0_fog_o;
    logic  [1:0] tex0_light_bank_o;
    logic  [7:0] tex0_tex_or_color_o;
    logic [15:0] tex0_addr_o;
    logic [15:0] tex0_z_o;
    logic [15:0] tex0_z_ref_o;
    logic [15:0] tex0_dst_rgb565_o;
    logic  [9:0] tex0_x_o;
    logic  [8:0] tex0_y_o;
    logic [31:0] tex0_w_q_o;
    logic signed [63:0] tex0_u_prod_o;
    logic signed [63:0] tex0_v_prod_o;
    logic        pipe2_valid_o;
    logic        pipe2_inside_o;
    logic        pipe2_ztest_o;
    logic        pipe2_textured_o;
    logic        pipe2_alpha_key_o;
    logic  [1:0] pipe2_alpha_o;
    logic        pipe2_fog_o;
    logic  [1:0] pipe2_light_bank_o;
    logic  [7:0] pipe2_tex_or_color_o;
    logic [15:0] pipe2_addr_o;
```

```
    logic [15:0] pipe2_z_o;
    logic [15:0] pipe2_z_ref_o;
    logic [15:0] pipe2_dst_rgb565_o;
    logic  [9:0] pipe2_x_o;
    logic  [8:0] pipe2_y_o;
    logic [31:0] pipe2_w_q_o;
    logic [14:0] pipe2_tex_addr_o;
    logic        draw_pipe_valid_o;
    logic        draw_pipe_inside_o;
    logic        draw_pipe_ztest_o;
    logic        draw_pipe_textured_o;
    logic        draw_pipe_alpha_key_o;
    logic  [1:0] draw_pipe_alpha_o;
    logic        draw_pipe_fog_o;
    logic  [1:0] draw_pipe_light_bank_o;
    logic  [7:0] draw_pipe_tex_or_color_o;
    logic [15:0] draw_pipe_addr_o;
    logic [15:0] draw_pipe_z_o;
    logic [15:0] draw_pipe_z_ref_o;
    logic [15:0] draw_pipe_dst_rgb565_o;
    logic  [9:0] draw_pipe_x_o;
    logic  [8:0] draw_pipe_y_o;
    logic [31:0] draw_pipe_w_q_o;
    logic        pal_rd_valid_o;
    logic        pal_rd_pass_o;
    logic        pal_rd_ztest_o;
    logic  [1:0] pal_rd_alpha_o;
    logic        pal_rd_fog_o;
    logic [15:0] pal_rd_addr_o;
    logic [15:0] pal_rd_z_o;
    logic  [7:0] pal_rd_src_addr_o;
    logic  [7:0] pal_rd_fog_addr_o;
    logic        pal_rd_src_sky_o;
    logic  [4:0] pal_rd_sky_index_o;
    logic [15:0] pal_rd_dst_rgb565_o;
    logic [31:0] pal_rd_w_q_o;
    logic [33:0] pal_rd_ray_scale_q16_o;
    logic        plr_valid_o;
    logic        plr_pass_o;
    logic        plr_ztest_o;
    logic  [1:0] plr_alpha_o;
    logic        plr_fog_o;
    logic [15:0] plr_addr_o;
    logic [15:0] plr_z_o;
    logic [23:0] plr_src_rgb_o;
    logic [15:0] plr_dst_rgb565_o;
    logic [23:0] plr_fog_rgb_o;
    logic [31:0] plr_w_q_o;
    logic [33:0] plr_ray_scale_q16_o;
    logic        fog0_valid_o;
    logic        fog0_pass_o;
    logic        fog0_ztest_o;
    logic  [1:0] fog0_alpha_o;
    logic        fog0_fog_o;
    logic [15:0] fog0_addr_o;
    logic [15:0] fog0_z_o;
    logic [15:0] fog0_src_rgb565_o;
    logic [15:0] fog0_dst_rgb565_o;
    logic [15:0] fog0_fog_rgb565_o;
```

```
    logic [31:0] fog0_w_q_o;
    logic [33:0] fog0_ray_scale_q16_o;
    logic        fog1_valid_o;
    logic        fog1_pass_o;
    logic        fog1_ztest_o;
    logic  [1:0] fog1_alpha_o;
    logic        fog1_fog_o;
    logic [15:0] fog1_addr_o;
    logic [15:0] fog1_z_o;
    logic [15:0] fog1_src_rgb565_o;
    logic [15:0] fog1_dst_rgb565_o;
    logic [15:0] fog1_fog_rgb565_o;
    logic [15:0] fog1_radial_q8_8_o;
    logic        commit_valid_o;
    logic        commit_pass_o;
    logic        commit_ztest_o;
    logic [15:0] commit_addr_o;
    logic [15:0] commit_z_o;
    logic [15:0] commit_color_o;
    // tex_rd_data is driven combinationally by voxel_texture_rom's
    // registered output. The ROM takes pipe2_tex_addr on cycle T and
    // presents mem[pipe2_tex_addr[T]] on cycle T+1, which is the same
    // 1-cycle latency the draw_pipe stage expects (see the instance
    // below and the voxel_texture_rom module header for rationale).
    wire   [7:0] tex_rd_data;
    wire   [7:0] tex_rd_data_o;

    logic [10:0] hcount;
    logic  [9:0] vcount;

    logic        scan_visible_now;
    logic [15:0] scan_rgb565_r;
    logic        scan_visible_r;
    logic [15:0] draw_addr;
    logic [15:0] draw_addr_o;
    logic [15:0] fb_back_rd_addr;
    logic [15:0] fb_back_rd_addr_o;
    // fb_back_rd_data is now a wire driven by the ping-pong cache mux
    logic [15:0] fb_wr_addr;
    logic [15:0] fb_wr_data;
    logic        fb_back_wr_en;
    logic [15:0] fb_wr_addr_e;
    logic [15:0] fb_wr_data_e;
    logic        fb_back_wr_en_e;
    logic [15:0] fb_wr_addr_o;
    logic [15:0] fb_wr_data_o;
    logic        fb_back_wr_en_o;
    logic [15:0] z_rd_addr;
    logic [15:0] z_rd_addr_o;
    // z_rd_data is now a wire driven by the ping-pong cache mux
    logic [15:0] z_wr_addr;
    logic [15:0] z_wr_data;
    logic        z_wr_en;
    logic [15:0] z_wr_addr_e;
    logic [15:0] z_wr_data_e;
    logic        z_wr_en_e;
    logic [15:0] z_wr_addr_o;
    logic [15:0] z_wr_data_o;
    logic        z_wr_en_o;
```

```
739
740     (* ramstyle = "MLAB, no_rw_check" *) logic [15:0] scan_linebuf0 [0:LINE_WORDS-1];
741     (* ramstyle = "MLAB, no_rw_check" *) logic [15:0] scan_linebuf1 [0:LINE_WORDS-1];
742     (* ramstyle = "MLAB, no_rw_check" *) logic [15:0] scan_linebuf2 [0:LINE_WORDS-1];
743     logic        scan_line0_ready;
744     logic        scan_line1_ready;
745     logic        scan_line2_ready;
746     logic  [8:0] scan_line0_row;
747     logic  [8:0] scan_line1_row;
748     logic  [8:0] scan_line2_row;
749     logic  [1:0] scan_active_bank;
750     logic        scan_active_valid;
751     logic  [8:0] scan_active_row;
752     logic        scan_fill_active;
753     logic        scan_fill_armed;
754     logic        scan_fill_load_pending;
755     logic  [1:0] scan_fill_bank;
756     logic  [8:0] scan_fill_row;
757     logic [24:0] scan_fill_base_words;
758     logic  [9:0] scan_fill_store_idx;
759     logic        scan_rd_capture;
760     logic [15:0] scan_rgb565_now;
761     logic [15:0] scan_late_count;
762     logic [24:0] sdram_rd_addr_cfg;
763     logic [24:0] sdram_rd_max_addr_cfg;
764     logic        sdram_rd_load_pulse;
765     /*
766      * Some RD_LOADs need a multi-cycle FIFO clear, but stretching every
767      * 64-word scanline chunk is expensive. Use the long clear for scanline
768      * starts and cache-load starts, where stale tail data can contaminate the
769      * first visible chunk; continuation chunks use the normal one-cycle pulse.
770      */
771     logic        sdram_rd_load_stretch_req;
772     logic  [3:0] sdram_rd_load_hold;
773     wire         sdram_rd_load_out = sdram_rd_load_pulse ||
774                                      (sdram_rd_load_hold != 4'd0);
775
776     logic  [2:0] cache_band_index;
777     logic  [2:0] cache_target_band;
778     logic  [2:0] band_index_cfg;
779     logic  [5:0] band_flush_y_min_cfg;
780     logic  [5:0] band_flush_y_max_cfg;
781     logic  [5:0] cache_flush_y_min;
782     logic [15:0] cache_window_start;
783     logic [15:0] cache_window_pixels;
784     logic [15:0] cache_window_end;
785     logic        band_begin_pending;
786     logic        band_flush_pending;
787     logic        cache_valid;
788     logic        cache_dirty;
789     logic        cache_draw_dirty;
790     logic [15:0] cache_maint_addr;
791
792     /* Ping-pong band cache */
793     logic        draw_cache_sel;           // 0 = A active, 1 = B active
794     logic        flush_active;             // background flush running
795     logic [15:0] flush_maint_addr;         // read address into inactive cache
796     logic [15:0] flush_window_start;       // first local pixel address to flush
797     logic [15:0] flush_pixels_total;       // pixels in the flushing band
```

```
798     logic [15:0] flush_words_issued;       // cache reads / generated words issued
799     logic [15:0] flush_words_done;         // words pushed to SDRAM wr FIFO
800     logic        flush_fetch_inflight;     // one-cycle read latency pending
801     logic        flush_word_pending_valid; // captured pixel waiting for SDRAM push
802     logic [15:0] flush_word_pending;       // the captured pixel value
803     logic        flush_load_pending;       // kick the SDRAM write burst
804     logic  [7:0] flush_drain_count;        // wait after WR FIFO empty for final SDRAM burst
805     logic  [2:0] flush_band_index;         // which band is being flushed
806     logic [24:0] flush_sdram_wr_addr;      // SDRAM write base for flush
807     logic [24:0] flush_sdram_wr_max_addr;  // SDRAM write end for flush
808     logic        flush_cache_sel;          // which cache to flush (0=A, 1=B)
809     logic        flush_generated_sky;      // source is sky palette, not local cache
810     logic  [9:0] flush_sky_x;
811     logic  [4:0] flush_sky_row_count;
812     logic  [4:0] flush_sky_palette;
813     logic [15:0] flush_fetch_clear_rgb565; // clear color aligned with cache read
814
815     /*
816      * Per-cache, per-bank read/write signals. `_e` and `_o` are the even/odd
817      * linear-address banks, and `*_rd_sel_q` selects the bank whose read data
818      * returns one cycle after the address.
819      */
820     logic [15:0] fb_A_e_rd_addr, fb_A_o_rd_addr;
821     logic [15:0] fb_A_e_rd_data, fb_A_o_rd_data;
822     logic [15:0] fb_A_e_wr_addr, fb_A_o_wr_addr;
823     logic [15:0] fb_A_e_wr_data, fb_A_o_wr_data;
824     logic        fb_A_e_wr_en,   fb_A_o_wr_en;
825     logic [15:0] fb_B_e_rd_addr, fb_B_o_rd_addr;
826     logic [15:0] fb_B_e_rd_data, fb_B_o_rd_data;
827     logic [15:0] fb_B_e_wr_addr, fb_B_o_wr_addr;
828     logic [15:0] fb_B_e_wr_data, fb_B_o_wr_data;
829     logic        fb_B_e_wr_en,   fb_B_o_wr_en;
830     logic [15:0] z_A_e_rd_addr,  z_A_o_rd_addr;
831     logic [15:0] z_A_e_rd_data,  z_A_o_rd_data;
832     logic [15:0] z_A_e_wr_addr,  z_A_o_wr_addr;
833     logic [15:0] z_A_e_wr_data,  z_A_o_wr_data;
834     logic        z_A_e_wr_en,    z_A_o_wr_en;
835     logic [15:0] z_B_e_rd_addr,  z_B_o_rd_addr;
836     logic [15:0] z_B_e_rd_data,  z_B_o_rd_data;
837     logic [15:0] z_B_e_wr_addr,  z_B_o_wr_addr;
838     logic [15:0] z_B_e_wr_data,  z_B_o_wr_data;
839     logic        z_B_e_wr_en,    z_B_o_wr_en;
840     logic        fb_A_rd_sel_q, fb_B_rd_sel_q;
841     logic        z_A_rd_sel_q,  z_B_rd_sel_q;
842     logic [15:0] fb_A_rd_data, fb_B_rd_data;
843     logic [15:0] z_A_rd_data,  z_B_rd_data;
844
845     logic [17:0] sdram_powerup_counter;
846     logic [15:0] sdram_init_wait_counter;
847     logic        sdram_ctrl_reset_n;
848     logic        sdram_ready;
849     logic        sdram_ctrl_clk;
850     logic        sdram_wr_full;
851     logic [15:0] sdram_wr_use;
852     logic [15:0] sdram_rd_data;
853     logic        sdram_rd_empty;
854     logic [15:0] sdram_rd_use;
855     logic  [1:0] dram_cs_n_bus;
856
```

```
857     logic vga_vs_d;
858
859     wire wr = chipselect & write;
860     wire ctrl_clear_write = wr && (address == ADDR_CONTROL) && writedata[3];
861     wire fifo_push_req = wr && (address >= ADDR_FIFO_LO) && (address < ADDR_FIFO_HI) && !fifo_full;
862     wire desc_has_uv = desc_flags[FLAG_TEX_BIT];
863     wire [5:0] fetch_target_words =
864         ((fetch_count >= BASE_QUAD_WORDS_6) && desc_has_uv) ?
865         MAX_DESC_WORDS_6 : BASE_QUAD_WORDS_6;
866     wire fetch_pop_req =
867         (state == ST_FETCH) && (fetch_count < fetch_target_words) && !fifo_empty;
868     /*
869      * Keep the prefetched descriptor separate from desc_words. The draw path
870      * mostly uses latched draw_* / edge_* registers, but desc_* remains live
871      * combinational glue for fetch/setup and cache helpers; separating these
872      * arrays avoids any accidental next-descriptor bleed while a quad drains.
873      */
874     wire [7:0] prefetch_flags = prefetch_words[15][31:24];
875     wire prefetch_has_uv = prefetch_flags[FLAG_TEX_BIT];
876     wire [5:0] prefetch_target_words_current =
877         ((prefetch_count >= BASE_QUAD_WORDS_6) && prefetch_has_uv) ?
878         MAX_DESC_WORDS_6 : BASE_QUAD_WORDS_6;
879     wire [5:0] prefetch_capture_target_words =
880         ((prefetch_count == (BASE_QUAD_WORDS_6 - 6'd1)) &&
881          fifo_head[24 + FLAG_TEX_BIT]) ?
882         MAX_DESC_WORDS_6 : prefetch_target_words_current;
883     wire prefetch_can_start =
884         ctrl_en && !prefetch_active && !prefetch_valid && !band_flush_pending &&
885         ((state == ST_DRAW) || (state == ST_DRAW_FLUSH)) &&
886         (fifo_count >= {5'd0, BASE_QUAD_WORDS_6});
887     wire prefetch_pop_req =
888         prefetch_active && (prefetch_count < prefetch_target_words_current) &&
889         !fifo_empty;
890     wire prefetch_finishes_on_pop =
891         prefetch_pop_req &&
892         ((prefetch_count + 6'd1) >= prefetch_capture_target_words);
893     wire fifo_pop_req = fetch_pop_req || prefetch_pop_req;
894     /* Background flush ordering is handled by the FSM; keep it out of BSY so
895      * userspace can pipeline the next band. */
896     wire engine_busy = (state != ST_IDLE) || clear_pending ||
897                        band_begin_pending || prefetch_active ||
898                        prefetch_valid;
899     wire vsync_pulse = vga_vs_d & ~VGA_VS;
900     wire [24:0] extmem_front_base_words = extmem_front_base[25:1];
901     wire [24:0] extmem_back_base_words  = extmem_back_base[25:1];
902     wire [24:0] display_base_words      = display_sel ? extmem_back_base_words :
        extmem_front_base_words;
903     wire [24:0] copy_target_base_words  = copy_target_sel ? extmem_back_base_words :
        extmem_front_base_words;
904     /*
905      * vcount counts 0..524 (VTOTAL-1) but scan_current_row is only 9 bits, so
906      * a naive vcount[8:0] wraps at vcount=512 -- during the back porch the
907      * scanout consumer would see scan_current_row=0..12 and race the advance
908      * logic ahead, popping ~13 prefetched rows before the visible frame even
909      * starts. That shows up as black at the top (source rows 0..12 gone) and
910      * black at the bottom (prefetcher caps at row 479, so active_row stalls
911      * there for the last 13 visible vcounts). Pin scan_current_row to 0 outside
912      * the active region so neither the comparator nor the +1 target lookahead
913      * fires while vcount is past 479.
```

```
914      */
915     wire        vcount_visible          = (vcount < 10'd480);
916     wire [8:0]  scan_current_row        = vcount_visible ? vcount[8:0] : 9'd0;
917     wire        scan_hblank_window      = vcount_visible && (hcount >= 11'd1280);
918     wire        scan_hblank_start       = vcount_visible && (hcount == 11'd1280);
919     wire [8:0]  scan_target_row         = scan_hblank_window ?
920                                           (scan_current_row + 9'd1) : scan_current_row;
921     wire        scan_target_valid       = scan_target_row < 9'd480;
922     wire [8:0]  scan_immediate_next_row = scan_current_row + 9'd1;
923     wire        scan_immediate_next_valid =
924         vcount_visible && (scan_current_row < 9'd479);
925     wire [8:0]  scan_far_next_row       = scan_current_row + 9'd2;
926     wire        scan_far_next_valid     =
927         vcount_visible && (scan_current_row < 9'd478);
928     wire [9:0]  scan_current_x          = hcount[10:1];
929     wire        scan_current_x_valid    = (scan_current_x < 10'd640);
930     wire [8:0]  scan_active_next_row    = scan_active_row + 9'd1;
931     wire        scan_active_bank_ready  = (scan_active_bank == 2'd0) ? scan_line0_ready :
932                                           (scan_active_bank == 2'd1) ? scan_line1_ready :
933                                                                        scan_line2_ready;
934     wire [8:0]  scan_active_bank_row    = (scan_active_bank == 2'd0) ? scan_line0_row :
935                                           (scan_active_bank == 2'd1) ? scan_line1_row :
936                                                                        scan_line2_row;
937     /*
938      * Drive scanout prefetch from the VGA row, not from the last active
939      * linebuffer row. If scanout ever falls behind, active-row-relative
940      * prefetch keeps chasing the stale row and the monitor repeats bands.
941      */
942     wire        scan_current_line_ready =
943         (scan_line0_ready && (scan_line0_row == scan_current_row)) ||
944         (scan_line1_ready && (scan_line1_row == scan_current_row)) ||
945         (scan_line2_ready && (scan_line2_row == scan_current_row));
946     wire        scan_target_line_ready =
947         !scan_target_valid ||
948         (scan_line0_ready && (scan_line0_row == scan_target_row)) ||
949         (scan_line1_ready && (scan_line1_row == scan_target_row)) ||
950         (scan_line2_ready && (scan_line2_row == scan_target_row));
951     wire        scan_immediate_next_line_ready =
952         !scan_immediate_next_valid ||
953         (scan_line0_ready && (scan_line0_row == scan_immediate_next_row)) ||
954         (scan_line1_ready && (scan_line1_row == scan_immediate_next_row)) ||
955         (scan_line2_ready && (scan_line2_row == scan_immediate_next_row));
956     wire        scan_far_next_line_ready =
957         !scan_far_next_valid ||
958         (scan_line0_ready && (scan_line0_row == scan_far_next_row)) ||
959         (scan_line1_ready && (scan_line1_row == scan_far_next_row)) ||
960         (scan_line2_ready && (scan_line2_row == scan_far_next_row));
961     wire        scan_target_or_active_ready =
962         !scan_active_valid || scan_target_line_ready;
963     wire        scan_prefetch_recover_current =
964         vcount_visible && !scan_current_line_ready;
965     wire        scan_prefetch_need_target =
966         vcount_visible && scan_target_valid &&
967         scan_current_line_ready && !scan_target_line_ready;
968     wire        scan_prefetch_need_next =
969         vcount_visible && scan_current_line_ready &&
970         scan_immediate_next_valid && !scan_immediate_next_line_ready;
971     wire        scan_prefetch_need_far =
972         vcount_visible && scan_current_line_ready &&
```

```
973         scan_immediate_next_line_ready &&
974         scan_far_next_valid && !scan_far_next_line_ready;
975     wire [8:0]  scan_prefetch_row =
976         scan_prefetch_recover_current ? scan_current_row :
977         scan_prefetch_need_target     ? scan_target_row  :
978         scan_prefetch_need_next       ? scan_immediate_next_row :
979                                         scan_far_next_row;
980     wire [24:0] scan_prefetch_base_words =
981         display_base_words +
982         {7'd0, scan_prefetch_row, 9'd0} +
983         {9'd0, scan_prefetch_row, 7'd0};
984     wire        scan_prefetch_valid     =
985         vcount_visible && scan_active_valid &&
986         (scan_prefetch_recover_current ||
987          scan_prefetch_need_target ||
988          scan_prefetch_need_next ||
989          scan_prefetch_need_far);
990     wire        scan_prefetch_ready     =
991         (scan_line0_ready && (scan_line0_row == scan_prefetch_row)) ||
992         (scan_line1_ready && (scan_line1_row == scan_prefetch_row)) ||
993         (scan_line2_ready && (scan_line2_row == scan_prefetch_row));
994     /* Protect any bank holding a row scanout will need before it can be
995      * legitimately reloaded: current (showing now), target (next at hblank),
996      * immediate_next (current+1), and far_next (current+2). Without far_next
997      * protection a 2-line prefetch could land in bank X and then be evicted
998      * by the next prefetch tick -- wasting the SDRAM burst and increasing
999      * the chance scanout falls behind. */
1000    wire        scan_bank0_protected    =
1001        scan_line0_ready &&
1002        ((scan_line0_row == scan_current_row) ||
1003         (scan_target_valid &&
1004          (scan_line0_row == scan_target_row)) ||
1005         (scan_immediate_next_valid &&
1006          (scan_line0_row == scan_immediate_next_row)) ||
1007         (scan_far_next_valid &&
1008          (scan_line0_row == scan_far_next_row)));
1009    wire        scan_bank1_protected    =
1010        scan_line1_ready &&
1011        ((scan_line1_row == scan_current_row) ||
1012         (scan_target_valid &&
1013          (scan_line1_row == scan_target_row)) ||
1014         (scan_immediate_next_valid &&
1015          (scan_line1_row == scan_immediate_next_row)) ||
1016         (scan_far_next_valid &&
1017          (scan_line1_row == scan_far_next_row)));
1018    wire        scan_bank2_protected    =
1019        scan_line2_ready &&
1020        ((scan_line2_row == scan_current_row) ||
1021         (scan_target_valid &&
1022          (scan_line2_row == scan_target_row)) ||
1023         (scan_immediate_next_valid &&
1024          (scan_line2_row == scan_immediate_next_row)) ||
1025         (scan_far_next_valid &&
1026          (scan_line2_row == scan_far_next_row)));
1027    wire        scan_bank0_free         = (scan_active_bank != 2'd0) &&
1028                                          !scan_bank0_protected;
1029    wire        scan_bank1_free         = (scan_active_bank != 2'd1) &&
1030                                          !scan_bank1_protected;
1031    wire        scan_bank2_free         = (scan_active_bank != 2'd2) &&
```

```
                                      !scan_bank2_protected;
    wire [1:0]  scan_prefetch_bank      = scan_bank0_free ? 2'd0 :
                                          scan_bank1_free ? 2'd1 :
                                          scan_bank2_free ? 2'd2 : 2'd3;
    wire        scan_read_idle          = !scan_fill_active && !scan_fill_armed &&
                                          !scan_fill_load_pending &&
                                          sdram_rd_empty;
    wire        cache_init_state        = (state == ST_CACHE_INIT);
    /* True when the main FSM still owns the active cache port. */
    wire        cache_used_by_main      = cache_sky_patch_state ||
                                          (state == ST_DRAW) ||
                                          (state == ST_DRAW_FLUSH) ||
                                          cache_init_state;
    /* Hold BEGIN until any queued END_BAND has captured its flush indices. */
    wire        band_begin_cache_available =
        !band_flush_pending &&
        (!flush_active || flush_generated_sky || ((~draw_cache_sel) != flush_cache_sel));
    wire        scan_vsync_read_req     = vsync_pulse && sdram_ready &&
                                          (copy_complete_pending || display_valid);
    /*
     * Defense-in-depth: never spawn a scan-fill burst while the SDRAM RD
     * FIFO still has residuals. This keeps one scanline burst from poisoning
     * the next linebuf fill with stale tail words.
     */
    wire        scan_prefetch_critical  =
        scan_prefetch_recover_current ||
        scan_prefetch_need_target ||
        scan_prefetch_need_next;
    wire        scan_prefetch_best_effort =
        scan_prefetch_need_far && !scan_prefetch_critical;
    wire        scan_write_pressure =
        flush_active ||
        band_flush_pending ||
        flush_word_pending_valid ||
        (sdram_wr_use[8:0] != 9'd0);
    wire        scan_prefetch_req       = !scan_fill_active && display_valid &&
                                          sdram_ready && scan_active_valid &&
                                          scan_prefetch_valid && !scan_prefetch_ready &&
                                          (scan_prefetch_bank != 2'd3) &&
                                          (!scan_prefetch_best_effort ||
                                           !scan_write_pressure) &&
                                          sdram_rd_empty;
    /* Let writes use active-video time once scanout has enough line headroom. */
    wire        scan_prefetch_margin_ready =
        scan_target_or_active_ready &&
        scan_immediate_next_line_ready &&
        (!scan_prefetch_critical || scan_prefetch_ready);
    wire        scan_active_write_window = vcount_visible &&
                                           (hcount < ACTIVE_WRITE_END_HCOUNT);
    wire        scanout_write_slack     = !display_valid ||
                                          (scan_read_idle &&
                                           !scan_prefetch_req &&
                                           (!VGA_BLANK_n ||
                                            (scan_prefetch_margin_ready &&
                                             scan_active_write_window)));
    wire        scanout_read_load_req   = scan_vsync_read_req ||
                                          scan_fill_load_pending ||
                                          scan_prefetch_req;
    wire        bg_flush_wr_load_req    = flush_active && flush_load_pending &&
```

```
                                        scanout_write_slack &&
                                        !scanout_read_load_req;
    wire        bg_flush_stream_active  = flush_active && !flush_load_pending;
    wire        scan_visible_data_ready = display_valid && sdram_ready &&
                                        scan_active_bank_ready &&
                                        (scan_active_bank_row == scan_active_row);
    wire [9:0]  scan_fill_words_complete =
        scan_fill_store_idx + (scan_rd_capture ? 10'd1 : 10'd0);
    wire        scan_fill_line_done =
        scan_rd_capture && (scan_fill_words_complete == LINE_WORDS_10);
    wire        scan_fill_chunk_done =
        scan_rd_capture && (scan_fill_words_complete[5:0] == 6'd0);
    /* Background flush may stage into the internal WR FIFO while scanout reads. */
    wire        bg_flush_wr_push   = bg_flush_stream_active && flush_word_pending_valid &&
                                     !sdram_wr_full;
    wire        sdram_wr_push      = bg_flush_wr_push;

    /* While armed, ignore residual RD FIFO data from the previous burst. */
    wire        scan_rd_pop = scan_fill_active && !scan_fill_armed && !sdram_rd_empty &&
                              (scan_fill_store_idx + (scan_rd_capture ? 10'd1 : 10'd0) <
                               LINE_WORDS_10) &&
                              !scan_fill_chunk_done;
    wire        sdram_rd_pop = scan_rd_pop;

    /* Keep the background flush-to-FIFO pipeline moving until the WR FIFO high-water mark. */
    wire        flush_can_issue_read =
        bg_flush_stream_active &&
        !flush_generated_sky &&
        (flush_words_issued < flush_pixels_total) &&
        !flush_fetch_inflight &&
        (sdram_wr_use[8:0] < COPY_WR_FIFO_HIGH_WATER) &&
        (!flush_word_pending_valid || bg_flush_wr_push);
    wire        flush_can_issue_sky =
        bg_flush_stream_active &&
        flush_generated_sky &&
        (flush_words_issued < flush_pixels_total) &&
        (sdram_wr_use[8:0] < COPY_WR_FIFO_HIGH_WATER) &&
        (!flush_word_pending_valid || bg_flush_wr_push);

    wire [8:0]  sdram_wr_length_cfg = (bg_flush_stream_active && scanout_write_slack) ?
                                      COPY_BURST_WORDS_9 : 9'd0;
    /*
     * Keep SDRAM reads in 64-word chunks. A full scanline burst can cross the
     * SDRAM row/column boundary; 64-word chunks stay aligned because both the
     * frame line width and SDRAM column size are multiples of 64.
     */
    /*
     * While scan_fill_load_pending is high, the next burst address is being
     * staged but RD_LOAD has not reached Sdram_Control yet. Holding LENGTH at
     * zero for that one cycle prevents the controller from launching a burst
     * at the previous 64-word chunk address.
     */
    wire [8:0]  sdram_rd_length_cfg = (scan_fill_armed && !scan_fill_load_pending) ?
                                     READ_BURST_WORDS_9 : 9'd0;

    /*  Ping-pong cache port muxing  */
    /* Rasterizer read data from active cache */
    wire [15:0] fb_back_rd_data = draw_cache_sel ? fb_B_rd_data : fb_A_rd_data;
    wire [15:0] z_rd_data       = draw_cache_sel ? z_B_rd_data  : z_A_rd_data;
```

```
    wire [15:0] fb_draw_rd_data_e = draw_cache_sel ? fb_B_e_rd_data : fb_A_e_rd_data;
    wire [15:0] fb_draw_rd_data_o = draw_cache_sel ? fb_B_o_rd_data : fb_A_o_rd_data;
    wire [15:0] z_draw_rd_data_e  = draw_cache_sel ? z_B_e_rd_data  : z_A_e_rd_data;
    wire [15:0] z_draw_rd_data_o  = draw_cache_sel ? z_B_o_rd_data  : z_A_o_rd_data;
    /* Flush read data from flush_cache_sel's cache */
    wire [15:0] flush_fb_rd_data = flush_cache_sel ? fb_B_rd_data : fb_A_rd_data;
    wire [15:0] flush_z_rd_data  = flush_cache_sel ? z_B_rd_data  : z_A_rd_data;

    wire signed [15:0] desc_x_min_raw = $signed(desc_words[0][15:0]);
    wire signed [15:0] desc_y_min_raw = $signed(desc_words[0][31:16]);
    wire signed [15:0] desc_x_max_raw = $signed(desc_words[1][15:0]);
    wire signed [15:0] desc_y_max_raw = $signed(desc_words[1][31:16]);
    wire [9:0] desc_x_min = clamp_x(desc_x_min_raw);
    wire [9:0] desc_x_max = clamp_x(desc_x_max_raw);
    wire [8:0] desc_y_min = clamp_y(desc_y_min_raw);
    wire [8:0] desc_y_max = clamp_y(desc_y_max_raw);
    wire [15:0] desc_z0 = desc_words[14][15:0];
    wire signed [15:0] desc_dz_dx = $signed(desc_words[14][31:16]);
    wire signed [15:0] desc_dz_dy = $signed(desc_words[15][15:0]);
    wire [7:0] desc_tex_or_color = desc_words[15][23:16];
    wire [7:0] desc_flags = desc_words[15][31:24];
    wire [8:0] desc_band_base_y = band_base_row(cache_band_index);
    wire [4:0] desc_sky_index =
        desc_tex_or_color[4:0] - PAL_SKY_GRADIENT_BASE[4:0];
    wire       desc_redundant_sky_clear =
        sky_gradient_clear_enabled &&
        (desc_flags == 8'd0) &&
        (desc_tex_or_color >= PAL_SKY_GRADIENT_BASE) &&
        (desc_tex_or_color <= PAL_SKY_GRADIENT_LAST) &&
        (desc_x_min == 10'd0) &&
        (desc_x_max == 10'd639);
    wire       desc_band_primer =
        (desc_flags == 8'd0) &&
        (desc_tex_or_color == 8'd0) &&
        (desc_x_min == 10'd0) &&
        (desc_x_max == 10'd0) &&
        (desc_y_min == desc_band_base_y) &&
        (desc_y_max == desc_band_base_y);
    wire       cache_sky_patch_state =
        (state == ST_FETCH) &&
        (fetch_count == fetch_target_words) &&
        desc_redundant_sky_clear;
    // Perspective-correct UV: 9 Q16.16 plane coefficients packed directly
    // after the base descriptor (words 16..24). One_over_w is guaranteed positive
    // at any pixel in front of the near plane -- software does not emit quads
    // that touch w <= 0.
    wire signed [31:0] desc_uw_0    = $signed(desc_words[16]);
    wire signed [31:0] desc_uw_dx   = $signed(desc_words[17]);
    wire signed [31:0] desc_uw_dy   = $signed(desc_words[18]);
    wire signed [31:0] desc_vw_0    = $signed(desc_words[19]);
    wire signed [31:0] desc_vw_dx   = $signed(desc_words[20]);
    wire signed [31:0] desc_vw_dy   = $signed(desc_words[21]);
    wire signed [31:0] desc_iw_0    = $signed(desc_words[22]);
    wire signed [31:0] desc_iw_dx   = $signed(desc_words[23]);
    wire signed [31:0] desc_iw_dy   = $signed(desc_words[24]);

    wire [9:0] desc_x_start_even = {desc_x_min[9:1], 1'b0};
    wire [9:0] draw_x_start_even = {draw_x_min[9:1], 1'b0};
    wire [9:0] draw_x_next = draw_x_cur + 10'd1;
```

```
    wire signed [63:0] edge_eval0;
    wire signed [63:0] edge_eval1;
    wire signed [63:0] edge_eval2;
    wire signed [63:0] edge_eval3;
    wire signed [47:0] draw_z_start_val;
    wire signed [63:0] draw_uw_start_val;
    wire signed [63:0] draw_vw_start_val;
    wire signed [63:0] draw_iw_start_val;

    voxel_raster_setup raster_setup (
        .draw_x_start_even (draw_x_start_even),
        .draw_y_min        (draw_y_min),
        .draw_x_min        (draw_x_min),
        .draw_z0           (draw_z0),
        .draw_dz_dx        (draw_dz_dx),
        .draw_uw_0         (draw_uw_0),
        .draw_uw_dx        (draw_uw_dx),
        .draw_vw_0         (draw_vw_0),
        .draw_vw_dx        (draw_vw_dx),
        .draw_iw_0         (draw_iw_0),
        .draw_iw_dx        (draw_iw_dx),
        .edge_a0           (edge_A[0]),
        .edge_a1           (edge_A[1]),
        .edge_a2           (edge_A[2]),
        .edge_a3           (edge_A[3]),
        .edge_b0           (edge_B[0]),
        .edge_b1           (edge_B[1]),
        .edge_b2           (edge_B[2]),
        .edge_b3           (edge_B[3]),
        .edge_c0           (edge_C[0]),
        .edge_c1           (edge_C[1]),
        .edge_c2           (edge_C[2]),
        .edge_c3           (edge_C[3]),
        .edge_eval0        (edge_eval0),
        .edge_eval1        (edge_eval1),
        .edge_eval2        (edge_eval2),
        .edge_eval3        (edge_eval3),
        .z_start_val       (draw_z_start_val),
        .uw_start_val      (draw_uw_start_val),
        .vw_start_val      (draw_vw_start_val),
        .iw_start_val      (draw_iw_start_val)
    );

    wire draw_inside = (edge_cur_val[0] >= 0) && (edge_cur_val[1] >= 0) &&
                       (edge_cur_val[2] >= 0) && (edge_cur_val[3] >= 0);
    wire signed [63:0] edge_cur_val_o0;
    wire signed [63:0] edge_cur_val_o1;
    wire signed [63:0] edge_cur_val_o2;
    wire signed [63:0] edge_cur_val_o3;
    wire draw_inside_o_edge = (edge_cur_val_o0 >= 0) &&
                              (edge_cur_val_o1 >= 0) &&
                              (edge_cur_val_o2 >= 0) &&
                              (edge_cur_val_o3 >= 0);
    wire draw_lane0_in_bounds = (draw_x_cur >= draw_x_min) &&
                                (draw_x_cur <= draw_x_max);
    wire draw_lane1_in_bounds = (draw_x_next >= draw_x_min) &&
                                (draw_x_next <= draw_x_max);
    wire draw_inside_lane0 = draw_inside && draw_lane0_in_bounds;
    wire draw_inside_lane1 = draw_inside_o_edge && draw_lane1_in_bounds;
```

```
    wire draw_pair_edge_inside = draw_inside || draw_inside_o_edge;
    wire draw_pair_exited = draw_row_inside && !draw_inside && !draw_inside_o_edge;
    wire draw_pair_last = (draw_x_next >= draw_x_max);
    wire [15:0] draw_z_value = clamp_z(z_cur_val);
    wire signed [63:0] edge_next_pair0;
    wire signed [63:0] edge_next_pair1;
    wire signed [63:0] edge_next_pair2;
    wire signed [63:0] edge_next_pair3;
    wire signed [63:0] edge_next_row0;
    wire signed [63:0] edge_next_row1;
    wire signed [63:0] edge_next_row2;
    wire signed [63:0] edge_next_row3;
    wire signed [47:0] draw_z_lane1_val;
    wire signed [47:0] z_next_pair;
    wire signed [47:0] z_next_row;
    wire signed [63:0] draw_uw_lane1_val;
    wire signed [63:0] draw_vw_lane1_val;
    wire signed [63:0] draw_iw_lane1_val;
    wire signed [63:0] uw_next_pair;
    wire signed [63:0] uw_next_row;
    wire signed [63:0] vw_next_pair;
    wire signed [63:0] vw_next_row;
    wire signed [63:0] iw_next_pair;
    wire signed [63:0] iw_next_row;

    voxel_draw_step draw_step (
        .edge_cur0        (edge_cur_val[0]),
        .edge_cur1        (edge_cur_val[1]),
        .edge_cur2        (edge_cur_val[2]),
        .edge_cur3        (edge_cur_val[3]),
        .edge_row0        (edge_row_val[0]),
        .edge_row1        (edge_row_val[1]),
        .edge_row2        (edge_row_val[2]),
        .edge_row3        (edge_row_val[3]),
        .edge_a0          (edge_A[0]),
        .edge_a1          (edge_A[1]),
        .edge_a2          (edge_A[2]),
        .edge_a3          (edge_A[3]),
        .edge_b0          (edge_B[0]),
        .edge_b1          (edge_B[1]),
        .edge_b2          (edge_B[2]),
        .edge_b3          (edge_B[3]),
        .z_cur            (z_cur_val),
        .z_row            (z_row_val),
        .dz_dx            (draw_dz_dx),
        .dz_dy            (draw_dz_dy),
        .uw_cur           (uw_cur_val),
        .uw_row           (uw_row_val),
        .uw_dx            (draw_uw_dx),
        .uw_dy            (draw_uw_dy),
        .vw_cur           (vw_cur_val),
        .vw_row           (vw_row_val),
        .vw_dx            (draw_vw_dx),
        .vw_dy            (draw_vw_dy),
        .iw_cur           (iw_cur_val),
        .iw_row           (iw_row_val),
        .iw_dx            (draw_iw_dx),
        .iw_dy            (draw_iw_dy),
        .edge_lane1_0     (edge_cur_val_o0),
```

```
        .edge_lane1_1     (edge_cur_val_o1),
        .edge_lane1_2     (edge_cur_val_o2),
        .edge_lane1_3     (edge_cur_val_o3),
        .edge_next_pair0  (edge_next_pair0),
        .edge_next_pair1  (edge_next_pair1),
        .edge_next_pair2  (edge_next_pair2),
        .edge_next_pair3  (edge_next_pair3),
        .edge_next_row0   (edge_next_row0),
        .edge_next_row1   (edge_next_row1),
        .edge_next_row2   (edge_next_row2),
        .edge_next_row3   (edge_next_row3),
        .z_lane1          (draw_z_lane1_val),
        .z_next_pair      (z_next_pair),
        .z_next_row       (z_next_row),
        .uw_lane1         (draw_uw_lane1_val),
        .uw_next_pair     (uw_next_pair),
        .uw_next_row      (uw_next_row),
        .vw_lane1         (draw_vw_lane1_val),
        .vw_next_pair     (vw_next_pair),
        .vw_next_row      (vw_next_row),
        .iw_lane1         (draw_iw_lane1_val),
        .iw_next_pair     (iw_next_pair),
        .iw_next_row      (iw_next_row)
    );

    wire [15:0] draw_z_value_o = clamp_z(draw_z_lane1_val);
    wire [2:0]  draw_band_index = y_to_band(draw_y_cur);
    wire        draw_cache_hit = cache_valid && (cache_band_index == draw_band_index);
    wire [15:0] draw_x_offset = {6'd0, draw_x_cur} - {6'd0, draw_x_start_even};
    wire [15:0] draw_cache_addr = draw_row_base + draw_x_offset;
    wire [15:0] draw_cache_addr_o = draw_cache_addr + 16'd1;
    wire signed [31:0] draw_uw_q = clamp_s32(uw_cur_val);
    wire signed [31:0] draw_vw_q = clamp_s32(vw_cur_val);
    wire [31:0] draw_iw_q = clamp_pos_u32(iw_cur_val);
    wire signed [31:0] draw_uw_q_o = clamp_s32(draw_uw_lane1_val);
    wire signed [31:0] draw_vw_q_o = clamp_s32(draw_vw_lane1_val);
    wire [31:0] draw_iw_q_o = clamp_pos_u32(draw_iw_lane1_val);
    wire [5:0] pipe0_iw_msb;
    wire [31:0] pipe0_iw_norm_q;
    wire [5:0] pipe0_iw_msb_o;
    wire [31:0] pipe0_iw_norm_q_o;

    voxel_iw_normalize iw_norm_lane0 (
        .iw_q      (pipe0_iw_q),
        .iw_msb    (pipe0_iw_msb),
        .iw_norm_q (pipe0_iw_norm_q)
    );

    voxel_iw_normalize iw_norm_lane1 (
        .iw_q      (pipe0_iw_q_o),
        .iw_msb    (pipe0_iw_msb_o),
        .iw_norm_q (pipe0_iw_norm_q_o)
    );

    wire [15:0] recip0_iw_phase = recip0_iw_norm_q[15:0];
    wire [10:0] recip0_iw_lut_idx = {1'b0, recip0_iw_phase[15:6]};
    wire [5:0] recip0_iw_lut_frac = recip0_iw_phase[5:0];
    wire [15:0] recip0_iw_phase_o = recip0_iw_norm_q_o[15:0];
    wire [10:0] recip0_iw_lut_idx_o = {1'b0, recip0_iw_phase_o[15:6]};
```

```
    wire [5:0] recip0_iw_lut_frac_o = recip0_iw_phase_o[5:0];
    wire [31:0] recip1_w_norm_q;
    wire [31:0] recip1_w_norm_q_o;
    wire [31:0] recip2_w_q;
    wire [31:0] recip2_w_q_o;

    voxel_recip_interpolate recip_interp_lane0 (
        .w_norm_lo   (recip1_w_norm_lo),
        .w_norm_hi   (recip1_w_norm_hi),
        .iw_lut_frac (recip1_iw_lut_frac),
        .w_norm_q    (recip1_w_norm_q)
    );

    voxel_recip_interpolate recip_interp_lane1 (
        .w_norm_lo   (recip1_w_norm_lo_o),
        .w_norm_hi   (recip1_w_norm_hi_o),
        .iw_lut_frac (recip1_iw_lut_frac_o),
        .w_norm_q    (recip1_w_norm_q_o)
    );

    voxel_w_denormalize w_denorm_lane0 (
        .iw_zero  (recip2_iw_zero),
        .iw_msb   (recip2_iw_msb),
        .w_norm_q (recip2_w_norm_q),
        .w_q      (recip2_w_q)
    );

    voxel_w_denormalize w_denorm_lane1 (
        .iw_zero  (recip2_iw_zero_o),
        .iw_msb   (recip2_iw_msb_o),
        .w_norm_q (recip2_w_norm_q_o),
        .w_q      (recip2_w_q_o)
    );
    wire signed [63:0] pipe1_u_prod = $signed(pipe1_uw_q) * $signed(pipe1_w_q);
    wire signed [63:0] pipe1_v_prod = $signed(pipe1_vw_q) * $signed(pipe1_w_q);
    wire signed [63:0] pipe1_u_prod_o = $signed(pipe1_uw_q_o) * $signed(pipe1_w_q_o);
    wire signed [63:0] pipe1_v_prod_o = $signed(pipe1_vw_q_o) * $signed(pipe1_w_q_o);
    wire tex0_repeat_uv = tex0_tex_or_color[7];
    wire [3:0] tex0_tex_u = texture_coord(tex0_u_prod, tex0_repeat_uv);
    wire [3:0] tex0_tex_v = texture_coord(tex0_v_prod, tex0_repeat_uv);
    wire [14:0] tex0_tex_addr = tex0_textured ?
                                {tex0_tex_or_color[6:0], tex0_tex_v, tex0_tex_u} :
                                15'd0;
    wire tex0_repeat_uv_o = tex0_tex_or_color_o[7];
    wire [3:0] tex0_tex_u_o = texture_coord(tex0_u_prod_o, tex0_repeat_uv_o);
    wire [3:0] tex0_tex_v_o = texture_coord(tex0_v_prod_o, tex0_repeat_uv_o);
    wire [14:0] tex0_tex_addr_o = tex0_textured_o ?
                                  {tex0_tex_or_color_o[6:0], tex0_tex_v_o, tex0_tex_u_o} :
                                  15'd0;
    wire  [7:0] draw_pipe_raw_color = draw_pipe_textured ? tex_rd_data : draw_pipe_tex_or_color;
    wire  [7:0] draw_pipe_color = apply_light_bank(draw_pipe_raw_color, draw_pipe_light_bank);
    wire  [7:0] draw_pipe_raw_color_o =
        draw_pipe_textured_o ? tex_rd_data_o : draw_pipe_tex_or_color_o;
    wire  [7:0] draw_pipe_color_o =
        apply_light_bank(draw_pipe_raw_color_o, draw_pipe_light_bank_o);
    wire        draw_pipe_generated_sky =
        !draw_pipe_textured &&
        !draw_pipe_ztest &&
        !draw_pipe_fog &&
```

```
        (draw_pipe_alpha == 2'd0) &&
        (draw_pipe_light_bank == 2'd0) &&
        (draw_pipe_raw_color >= PAL_SKY_GRADIENT_BASE) &&
        (draw_pipe_raw_color <= PAL_SKY_GRADIENT_LAST);
    wire        draw_pipe_generated_sky_o =
        !draw_pipe_textured_o &&
        !draw_pipe_ztest_o &&
        !draw_pipe_fog_o &&
        (draw_pipe_alpha_o == 2'd0) &&
        (draw_pipe_light_bank_o == 2'd0) &&
        (draw_pipe_raw_color_o >= PAL_SKY_GRADIENT_BASE) &&
        (draw_pipe_raw_color_o <= PAL_SKY_GRADIENT_LAST);
    wire  [4:0] draw_pipe_sky_index =
        draw_pipe_raw_color[4:0] - PAL_SKY_GRADIENT_BASE[4:0];
    wire  [4:0] draw_pipe_sky_index_o =
        draw_pipe_raw_color_o[4:0] - PAL_SKY_GRADIENT_BASE[4:0];
    wire  [7:0] palette_src_addr = (state == ST_CLEAR) ? 8'd0 : draw_pipe_color;
    wire  [7:0] palette_src_addr_o = (state == ST_CLEAR) ? 8'd0 : draw_pipe_color_o;
    /* Palette reads for the draw pipeline are sampled into plr_* one
     * cycle before fog0 (see the plr_* register block). The
     * rgb888_to_rgb565 conversion is cheap bit slicing, so we leave it
     * as a combinational derivation on the registered palette output
     * and let fog0_src_rgb565 / fog0_fog_rgb565 pick it up on the next
     * cycle. */
    wire [15:0] plr_src_rgb565 = rgb888_to_rgb565(plr_src_rgb);
    wire [15:0] plr_fog_rgb565 = rgb888_to_rgb565(plr_fog_rgb);
    wire [15:0] plr_src_rgb565_o = rgb888_to_rgb565(plr_src_rgb_o);
    wire [15:0] plr_fog_rgb565_o = rgb888_to_rgb565(plr_fog_rgb_o);

    /* Separate combinational reads used by clear/cache-init paths, which write
     * background colors straight to the local band cache and do not go through
     * the draw pipeline. */
    wire [15:0] clear_rgb565 = rgb888_to_rgb565(palette[8'd0]);
    wire        sky_gradient_clear_enabled =
        extmem_ctrl[EXTMEM_SKY_GRADIENT_CLEAR_BIT];
    wire [15:0] flush_clear_rgb565 =
        sky_gradient_clear_enabled ?
        rgb888_to_rgb565(sky_palette[flush_sky_palette]) :
        clear_rgb565;
    wire [15:0] draw_clear_rgb565 =
        sky_gradient_clear_enabled ?
        rgb888_to_rgb565(sky_palette[sky_palette_for_y(recip0_y)]) :
        clear_rgb565;
    wire draw_pipe_transparent = draw_pipe_textured &&
                                 draw_pipe_alpha_key &&
                                 (draw_pipe_raw_color == 8'd0);
    wire draw_pipe_transparent_o = draw_pipe_textured_o &&
                                   draw_pipe_alpha_key_o &&
                                   (draw_pipe_raw_color_o == 8'd0);
    /*
     * Radial distance estimate: starting from the per-pixel w (linear depth
     * along the camera forward axis) we scale by sqrt(1 + r^2/f^2) where r is
     * the pixel offset from screen center and f is the projection depth in
     * pixels. The square-root is approximated with 1 + 3/8 * r^2/f^2, which
     * is within a couple of percent for our field of view. The final distance
     * is produced in Q8.8 so it can be compared directly against the
     * fog_start_dist / fog_end_dist registers.
     */
    wire signed [11:0] draw_pipe_dx_center =
```

```
        $signed({1'b0, draw_pipe_x}) - 12'sd320;
    wire signed [10:0] draw_pipe_dy_center =
        11'sd240 - $signed({1'b0, draw_pipe_y});
    wire [23:0] draw_pipe_dx_sq = draw_pipe_dx_center * draw_pipe_dx_center;
    wire [23:0] draw_pipe_dy_sq = draw_pipe_dy_center * draw_pipe_dy_center;
    wire [24:0] draw_pipe_radius_sq = draw_pipe_dx_sq + draw_pipe_dy_sq;
    wire [40:0] draw_pipe_r2_prod = draw_pipe_radius_sq * fog_inv_proj_sq;
    wire [31:0] draw_pipe_r2_q16 = draw_pipe_r2_prod[31:0];
    wire [33:0] draw_pipe_ray_scale_q16 =
        34'd65536 + (({2'b00, draw_pipe_r2_q16} * 3'd3) >> 3);
    wire signed [11:0] draw_pipe_dx_center_o =
        $signed({1'b0, draw_pipe_x_o}) - 12'sd320;
    wire signed [10:0] draw_pipe_dy_center_o =
        11'sd240 - $signed({1'b0, draw_pipe_y_o});
    wire [23:0] draw_pipe_dx_sq_o = draw_pipe_dx_center_o * draw_pipe_dx_center_o;
    wire [23:0] draw_pipe_dy_sq_o = draw_pipe_dy_center_o * draw_pipe_dy_center_o;
    wire [24:0] draw_pipe_radius_sq_o = draw_pipe_dx_sq_o + draw_pipe_dy_sq_o;
    wire [40:0] draw_pipe_r2_prod_o = draw_pipe_radius_sq_o * fog_inv_proj_sq;
    wire [31:0] draw_pipe_r2_q16_o = draw_pipe_r2_prod_o[31:0];
    wire [33:0] draw_pipe_ray_scale_q16_o =
        34'd65536 + (({2'b00, draw_pipe_r2_q16_o} * 3'd3) >> 3);
    wire [65:0] fog0_radial_prod = fog0_w_q * fog0_ray_scale_q16;
    wire [31:0] fog0_radial_q16 = fog0_radial_prod[47:16];
    wire [15:0] fog0_radial_q8_8 = fog0_radial_q16[23:8];
    wire [65:0] fog0_radial_prod_o = fog0_w_q_o * fog0_ray_scale_q16_o;
    wire [31:0] fog0_radial_q16_o = fog0_radial_prod_o[47:16];
    wire [15:0] fog0_radial_q8_8_o = fog0_radial_q16_o[23:8];

    wire [15:0] fog1_out_rgb565;
    wire [15:0] fog1_out_rgb565_o;

    voxel_fog_blend fog_blend_lane0 (
        .fog_enable     (fog_enable),
        .pixel_fog      (fog1_fog),
        .radial_q8_8    (fog1_radial_q8_8),
        .fog_start_dist (fog_start_dist),
        .fog_end_dist   (fog_end_dist),
        .src_rgb565     (fog1_src_rgb565),
        .dst_rgb565     (fog1_dst_rgb565),
        .fog_rgb565     (fog1_fog_rgb565),
        .alpha          (fog1_alpha),
        .out_rgb565     (fog1_out_rgb565)
    );

    voxel_fog_blend fog_blend_lane1 (
        .fog_enable     (fog_enable),
        .pixel_fog      (fog1_fog_o),
        .radial_q8_8    (fog1_radial_q8_8_o),
        .fog_start_dist (fog_start_dist),
        .fog_end_dist   (fog_end_dist),
        .src_rgb565     (fog1_src_rgb565_o),
        .dst_rgb565     (fog1_dst_rgb565_o),
        .fog_rgb565     (fog1_fog_rgb565_o),
        .alpha          (fog1_alpha_o),
        .out_rgb565     (fog1_out_rgb565_o)
    );
    wire draw_commit_pass = draw_pipe_inside &&
                            !draw_pipe_transparent &&
                            (!draw_pipe_ztest || (draw_pipe_z < draw_pipe_z_ref));
```

```
    wire draw_commit_pass_o = draw_pipe_inside_o &&
                              !draw_pipe_transparent_o &&
                              (!draw_pipe_ztest_o || (draw_pipe_z_o < draw_pipe_z_ref_o));
    wire [31:0] status_word = {
        12'h0,
        {5'h0, fifo_count},
        vsy_latch,
        fifo_empty,
        fifo_full,
        engine_busy
    };

    wire [31:0] control_word = {
        28'h0,
        (clear_pending || (state == ST_CLEAR)),
        ctrl_ien,
        ctrl_flp_pending,
        ctrl_en
    };

    wire [23:0] pixel_rgb = {
        expand5_to_8(scan_rgb565_r[15:11]),
        expand6_to_8(scan_rgb565_r[10:5]),
        expand5_to_8(scan_rgb565_r[4:0])
    };

`include "voxel_raster_helpers.svh"
`include "voxel_color_helpers.svh"

    voxel_vga_counters counters (
        .clk50       (clk),
        .reset       (reset),
        .hcount      (hcount),
        .vcount      (vcount),
        .VGA_CLK     (VGA_CLK),
        .VGA_HS      (VGA_HS),
        .VGA_VS      (VGA_VS),
        .VGA_BLANK_n (VGA_BLANK_n),
        .VGA_SYNC_n  (VGA_SYNC_n)
    );

    Sdram_Control sdram_ctrl (
        .REF_CLK     (clk),
        .RESET_N     (sdram_ctrl_reset_n),
        .CLK         (sdram_ctrl_clk),
        .WR_DATA     (flush_word_pending),
        .WR          (sdram_wr_push),
        .WR_ADDR     (flush_sdram_wr_addr),
        .WR_MAX_ADDR (flush_sdram_wr_max_addr),
        .WR_LENGTH   (sdram_wr_length_cfg),
        .WR_LOAD     (bg_flush_wr_load_req),
        .WR_CLK      (clk),
        .WR_FULL     (sdram_wr_full),
        .WR_USE      (sdram_wr_use),
        .RD_DATA     (sdram_rd_data),
        .RD          (sdram_rd_pop),
        .RD_ADDR     (sdram_rd_addr_cfg),
        .RD_MAX_ADDR (sdram_rd_max_addr_cfg),
        .RD_LENGTH   (sdram_rd_length_cfg),
```

```
        .RD_LOAD     (sdram_rd_load_out),
        .RD_CLK      (clk),
        .RD_EMPTY    (sdram_rd_empty),
        .RD_USE      (sdram_rd_use),
        .SA          (DRAM_ADDR),
        .BA          (DRAM_BA),
        .CS_N        (dram_cs_n_bus),
        .CKE         (DRAM_CKE),
        .RAS_N       (DRAM_RAS_N),
        .CAS_N       (DRAM_CAS_N),
        .WE_N        (DRAM_WE_N),
        .DQ          (DRAM_DQ),
        .DQM         ({DRAM_UDQM, DRAM_LDQM}),
        .SDR_CLK     (DRAM_CLK)
    );

    assign DRAM_CS_N = dram_cs_n_bus[0];

    /* Ping-pong band caches A and B. Even and odd linear addresses land in
     * separate banks so the two raster lanes can read/write independently. */
    voxel_banked_sdp_ram #(
        .DATA_W(16),
        .ADDR_W(16),
        .DEPTH(BAND_PIXELS)
    ) fb_back_ram_A (
        .clk       (clk),
        .rd_addr_e (fb_A_e_rd_addr),
        .rd_data_e (fb_A_e_rd_data),
        .rd_addr_o (fb_A_o_rd_addr),
        .rd_data_o (fb_A_o_rd_data),
        .wr_addr_e (fb_A_e_wr_addr),
        .wr_data_e (fb_A_e_wr_data),
        .wr_en_e   (fb_A_e_wr_en),
        .wr_addr_o (fb_A_o_wr_addr),
        .wr_data_o (fb_A_o_wr_data),
        .wr_en_o   (fb_A_o_wr_en)
    );

    voxel_banked_sdp_ram #(
        .DATA_W(16),
        .ADDR_W(16),
        .DEPTH(BAND_PIXELS)
    ) fb_back_ram_B (
        .clk       (clk),
        .rd_addr_e (fb_B_e_rd_addr),
        .rd_data_e (fb_B_e_rd_data),
        .rd_addr_o (fb_B_o_rd_addr),
        .rd_data_o (fb_B_o_rd_data),
        .wr_addr_e (fb_B_e_wr_addr),
        .wr_data_e (fb_B_e_wr_data),
        .wr_en_e   (fb_B_e_wr_en),
        .wr_addr_o (fb_B_o_wr_addr),
        .wr_data_o (fb_B_o_wr_data),
        .wr_en_o   (fb_B_o_wr_en)
    );

    voxel_banked_sdp_ram #(
        .DATA_W(16),
        .ADDR_W(16),
```

```
        .DEPTH(BAND_PIXELS)
    ) z_ram_A (
        .clk       (clk),
        .rd_addr_e (z_A_e_rd_addr),
        .rd_data_e (z_A_e_rd_data),
        .rd_addr_o (z_A_o_rd_addr),
        .rd_data_o (z_A_o_rd_data),
        .wr_addr_e (z_A_e_wr_addr),
        .wr_data_e (z_A_e_wr_data),
        .wr_en_e   (z_A_e_wr_en),
        .wr_addr_o (z_A_o_wr_addr),
        .wr_data_o (z_A_o_wr_data),
        .wr_en_o   (z_A_o_wr_en)
    );

    voxel_banked_sdp_ram #(
        .DATA_W(16),
        .ADDR_W(16),
        .DEPTH(BAND_PIXELS)
    ) z_ram_B (
        .clk       (clk),
        .rd_addr_e (z_B_e_rd_addr),
        .rd_data_e (z_B_e_rd_data),
        .rd_addr_o (z_B_o_rd_addr),
        .rd_data_o (z_B_o_rd_data),
        .wr_addr_e (z_B_e_wr_addr),
        .wr_data_e (z_B_e_wr_data),
        .wr_en_e   (z_B_e_wr_en),
        .wr_addr_o (z_B_o_wr_addr),
        .wr_data_o (z_B_o_wr_data),
        .wr_en_o   (z_B_o_wr_en)
    );

    /* Rebuild the single-cache read view from the bank selected last cycle. */
    assign fb_A_rd_data = fb_A_rd_sel_q ? fb_A_o_rd_data : fb_A_e_rd_data;
    assign fb_B_rd_data = fb_B_rd_sel_q ? fb_B_o_rd_data : fb_B_e_rd_data;
    assign z_A_rd_data  = z_A_rd_sel_q  ? z_A_o_rd_data  : z_A_e_rd_data;
    assign z_B_rd_data  = z_B_rd_sel_q  ? z_B_o_rd_data  : z_B_e_rd_data;

    /*
     * Texture-atlas ROM. Two explicit 1-cycle read ports keep texels aligned
     * with draw_pipe metadata; tex_or_color[6:0] selects the tile and bit 7
     * controls UV wrapping.
     */
    voxel_texture_rom #(
        .DATA_W(8),
        .ADDR_W(15),
        .DEPTH(TEXTURE_BYTES),
        .INIT_FILE("voxel_gpu/assets/textures.mif")
    ) texture_rom (
        .clk       (clk),
        .rd_addr   (pipe2_tex_addr),
        .rd_data   (tex_rd_data),
        .rd_addr_b (pipe2_tex_addr_o),
        .rd_data_b (tex_rd_data_o)
    );

    integer i;
    integer ei;
```

```
    initial begin
        ctrl_en          = 1'b0;
        ctrl_ien         = 1'b0;
        ctrl_flp_pending = 1'b0;
        clear_pending    = 1'b0;
        frame_count      = 32'h0;
        pal_addr         = 8'h0;
        sky_pal_addr     = 5'd0;
        fog_enable       = 1'b0;
        fog_color        = 8'd0;
        fog_start_dist   = 16'd0;
        fog_end_dist     = 16'd0;
        fog_inv_proj_sq  = 16'd0;
        extmem_ctrl      = DEFAULT_EXTMEM_CTRL;
        extmem_front_base = DEFAULT_EXTMEM_FRONT_BASE;
        extmem_back_base = DEFAULT_EXTMEM_BACK_BASE;
        extmem_stride_bytes = DEFAULT_EXTMEM_STRIDE;
        extmem_tile_cfg  = 32'd0;
        extmem_dma_status = 32'd0;
        vsy_latch        = 1'b0;
        display_sel      = 1'b0;
        display_valid    = 1'b0;
        copy_target_sel  = 1'b1;
        copy_complete_pending = 1'b0;
        state            = ST_IDLE;
        fifo_wr_ptr      = 10'd0;
        fifo_rd_ptr      = 10'd0;
        fifo_count       = 11'd0;
        fetch_count      = 6'd0;
        prefetch_count   = 6'd0;
        prefetch_target_words = BASE_QUAD_WORDS_6;
        prefetch_active  = 1'b0;
        prefetch_valid   = 1'b0;
        draw_flush_count = 4'd0;
        clear_addr       = 16'd0;
        draw_row_base    = 16'd0;
        draw_x_min       = 10'd0;
        draw_x_max       = 10'd0;
        draw_x_cur       = 10'd0;
        draw_y_min       = 9'd0;
        draw_y_max       = 9'd0;
        draw_y_cur       = 9'd0;
        draw_tex_or_color = 8'd0;
        draw_flags       = 8'd0;
        draw_z0          = 16'd0;
        draw_dz_dx       = 16'sd0;
        draw_dz_dy       = 16'sd0;
        draw_uw_0        = 32'sd0;
        draw_uw_dx       = 32'sd0;
        draw_uw_dy       = 32'sd0;
        draw_vw_0        = 32'sd0;
        draw_vw_dx       = 32'sd0;
        draw_vw_dy       = 32'sd0;
        draw_iw_0        = 32'sd0;
        draw_iw_dx       = 32'sd0;
        draw_iw_dy       = 32'sd0;
        z_row_val        = 48'sd0;
        z_cur_val        = 48'sd0;
        uw_row_val       = 64'sd0;
```

```
        uw_cur_val       = 64'sd0;
        vw_row_val       = 64'sd0;
        vw_cur_val       = 64'sd0;
        iw_row_val       = 64'sd0;
        iw_cur_val       = 64'sd0;
        pipe0_valid      = 1'b0;
        pipe0_inside     = 1'b0;
        pipe0_ztest      = 1'b0;
        pipe0_textured   = 1'b0;
        pipe0_alpha_key  = 1'b0;
        pipe0_alpha      = 2'd0;
        pipe0_fog        = 1'b0;
        pipe0_light_bank = 2'd0;
        pipe0_tex_or_color = 8'd0;
        pipe0_addr       = 16'd0;
        pipe0_z          = 16'd0;
        pipe0_x          = 10'd0;
        pipe0_y          = 9'd0;
        pipe0_uw_q       = 32'sd0;
        pipe0_vw_q       = 32'sd0;
        pipe0_iw_q       = 32'd0;
        recip0_valid     = 1'b0;
        recip0_inside    = 1'b0;
        recip0_ztest     = 1'b0;
        recip0_textured  = 1'b0;
        recip0_alpha_key = 1'b0;
        recip0_alpha     = 2'd0;
        recip0_fog       = 1'b0;
        recip0_light_bank = 2'd0;
        recip0_tex_or_color = 8'd0;
        recip0_addr      = 16'd0;
        recip0_z         = 16'd0;
        recip0_x         = 10'd0;
        recip0_y         = 9'd0;
        recip0_uw_q      = 32'sd0;
        recip0_vw_q      = 32'sd0;
        recip0_iw_zero   = 1'b1;
        recip0_iw_msb    = 6'd0;
        recip0_iw_norm_q = 32'd0;
        recip1_valid     = 1'b0;
        recip1_inside    = 1'b0;
        recip1_ztest     = 1'b0;
        recip1_textured  = 1'b0;
        recip1_alpha_key = 1'b0;
        recip1_alpha     = 2'd0;
        recip1_fog       = 1'b0;
        recip1_light_bank = 2'd0;
        recip1_tex_or_color = 8'd0;
        recip1_addr      = 16'd0;
        recip1_z         = 16'd0;
        recip1_z_ref     = 16'd0;
        recip1_dst_rgb565 = 16'h0000;
        recip1_x         = 10'd0;
        recip1_y         = 9'd0;
        recip1_uw_q      = 32'sd0;
        recip1_vw_q      = 32'sd0;
        recip1_iw_zero   = 1'b1;
        recip1_iw_msb    = 6'd0;
        recip1_iw_lut_frac = 6'd0;
```

```
        recip1_w_norm_lo = 32'd0;
        recip1_w_norm_hi = 32'd0;
        recip2_valid     = 1'b0;
        recip2_inside    = 1'b0;
        recip2_ztest     = 1'b0;
        recip2_textured  = 1'b0;
        recip2_alpha_key = 1'b0;
        recip2_alpha     = 2'd0;
        recip2_fog       = 1'b0;
        recip2_light_bank = 2'd0;
        recip2_tex_or_color = 8'd0;
        recip2_addr      = 16'd0;
        recip2_z         = 16'd0;
        recip2_z_ref     = 16'd0;
        recip2_dst_rgb565 = 16'h0000;
        recip2_x         = 10'd0;
        recip2_y         = 9'd0;
        recip2_uw_q      = 32'sd0;
        recip2_vw_q      = 32'sd0;
        recip2_iw_zero   = 1'b1;
        recip2_iw_msb    = 6'd0;
        recip2_w_norm_q  = 32'd0;
        pipe1_valid      = 1'b0;
        pipe1_inside     = 1'b0;
        pipe1_ztest      = 1'b0;
        pipe1_textured   = 1'b0;
        pipe1_alpha_key  = 1'b0;
        pipe1_alpha      = 2'd0;
        pipe1_fog        = 1'b0;
        pipe1_light_bank = 2'd0;
        pipe1_tex_or_color = 8'd0;
        pipe1_addr       = 16'd0;
        pipe1_z          = 16'd0;
        pipe1_z_ref      = 16'd0;
        pipe1_dst_rgb565 = 16'h0000;
        pipe1_x          = 10'd0;
        pipe1_y          = 9'd0;
        pipe1_uw_q       = 32'sd0;
        pipe1_vw_q       = 32'sd0;
        pipe1_w_q        = 32'd0;
        tex0_valid       = 1'b0;
        tex0_inside      = 1'b0;
        tex0_ztest       = 1'b0;
        tex0_textured    = 1'b0;
        tex0_alpha_key   = 1'b0;
        tex0_alpha       = 2'd0;
        tex0_fog         = 1'b0;
        tex0_light_bank  = 2'd0;
        tex0_tex_or_color = 8'd0;
        tex0_addr        = 16'd0;
        tex0_z           = 16'd0;
        tex0_z_ref       = 16'd0;
        tex0_dst_rgb565  = 16'h0000;
        tex0_x           = 10'd0;
        tex0_y           = 9'd0;
        tex0_w_q         = 32'd0;
        tex0_u_prod      = 64'sd0;
        tex0_v_prod      = 64'sd0;
        pipe2_valid      = 1'b0;
```

```
        pipe2_inside     = 1'b0;
        pipe2_ztest      = 1'b0;
        pipe2_textured   = 1'b0;
        pipe2_alpha_key  = 1'b0;
        pipe2_alpha      = 2'd0;
        pipe2_fog        = 1'b0;
        pipe2_light_bank = 2'd0;
        pipe2_tex_or_color = 8'd0;
        pipe2_addr       = 16'd0;
        pipe2_z          = 16'd0;
        pipe2_z_ref      = 16'd0;
        pipe2_dst_rgb565 = 16'h0000;
        pipe2_x          = 10'd0;
        pipe2_y          = 9'd0;
        pipe2_w_q        = 32'd0;
        pipe2_tex_addr   = 15'd0;
        draw_pipe_valid  = 1'b0;
        draw_pipe_inside = 1'b0;
        draw_pipe_ztest  = 1'b0;
        draw_pipe_textured = 1'b0;
        draw_pipe_alpha_key = 1'b0;
        draw_pipe_alpha = 2'd0;
        draw_pipe_fog    = 1'b0;
        draw_pipe_light_bank = 2'd0;
        draw_pipe_tex_or_color = 8'd0;
        draw_pipe_addr   = 16'd0;
        draw_pipe_z      = 16'd0;
        draw_pipe_z_ref  = 16'd0;
        draw_pipe_dst_rgb565 = 16'h0000;
        draw_pipe_x      = 10'd0;
        draw_pipe_y      = 9'd0;
        draw_pipe_w_q    = 32'd0;
        draw_is_band_primer = 1'b0;
        pal_rd_valid     = 1'b0;
        pal_rd_pass      = 1'b0;
        pal_rd_ztest     = 1'b0;
        pal_rd_alpha     = 2'd0;
        pal_rd_fog       = 1'b0;
        pal_rd_addr      = 16'd0;
        pal_rd_z         = 16'd0;
        pal_rd_src_addr  = 8'd0;
        pal_rd_fog_addr  = 8'd0;
        pal_rd_src_sky   = 1'b0;
        pal_rd_sky_index = 5'd0;
        pal_rd_dst_rgb565 = 16'h0000;
        pal_rd_w_q       = 32'd0;
        pal_rd_ray_scale_q16 = 34'd0;
        plr_valid        = 1'b0;
        plr_pass         = 1'b0;
        plr_ztest        = 1'b0;
        plr_alpha        = 2'd0;
        plr_fog          = 1'b0;
        plr_addr         = 16'd0;
        plr_z            = 16'd0;
        plr_src_rgb      = 24'h000000;
        plr_dst_rgb565   = 16'h0000;
        plr_fog_rgb      = 24'h000000;
        plr_w_q          = 32'd0;
        plr_ray_scale_q16 = 34'd0;
```

```
        fog0_valid       = 1'b0;
        fog0_pass        = 1'b0;
        fog0_ztest       = 1'b0;
        fog0_alpha       = 2'd0;
        fog0_fog         = 1'b0;
        fog0_addr        = 16'd0;
        fog0_z           = 16'd0;
        fog0_src_rgb565  = 16'h0000;
        fog0_dst_rgb565  = 16'h0000;
        fog0_fog_rgb565  = 16'h0000;
        fog0_w_q         = 32'd0;
        fog0_ray_scale_q16 = 34'd0;
        fog1_valid       = 1'b0;
        fog1_pass        = 1'b0;
        fog1_ztest       = 1'b0;
        fog1_alpha       = 2'd0;
        fog1_fog         = 1'b0;
        fog1_addr        = 16'd0;
        fog1_z           = 16'd0;
        fog1_src_rgb565  = 16'h0000;
        fog1_dst_rgb565  = 16'h0000;
        fog1_fog_rgb565  = 16'h0000;
        fog1_radial_q8_8 = 16'd0;
        commit_valid     = 1'b0;
        commit_pass      = 1'b0;
        commit_ztest     = 1'b0;
        commit_addr      = 16'd0;
        commit_z         = 16'd0;
        commit_color     = 16'h0000;
        pipe0_valid_o    = 1'b0;
        recip0_valid_o   = 1'b0;
        recip1_valid_o   = 1'b0;
        recip2_valid_o   = 1'b0;
        pipe1_valid_o    = 1'b0;
        tex0_valid_o     = 1'b0;
        pipe2_valid_o    = 1'b0;
        draw_pipe_valid_o = 1'b0;
        pal_rd_valid_o   = 1'b0;
        pal_rd_src_sky_o = 1'b0;
        pal_rd_sky_index_o = 5'd0;
        plr_valid_o      = 1'b0;
        fog0_valid_o     = 1'b0;
        fog1_valid_o     = 1'b0;
        commit_valid_o   = 1'b0;
        scan_rgb565_r    = 16'h0000;
        scan_visible_r   = 1'b0;
        scan_line0_ready = 1'b0;
        scan_line1_ready = 1'b0;
        scan_line2_ready = 1'b0;
        scan_line0_row   = 9'd0;
        scan_line1_row   = 9'd0;
        scan_line2_row   = 9'd0;
        scan_active_bank = 2'd0;
        scan_active_valid = 1'b0;
        scan_active_row  = 9'd0;
        scan_fill_active = 1'b0;
        scan_fill_armed  = 1'b0;
        scan_fill_load_pending = 1'b0;
        scan_fill_bank   = 2'd0;
```

```
        scan_fill_row    = 9'd0;
        scan_fill_base_words = 25'd0;
        scan_fill_store_idx = 10'd0;
        scan_rd_capture = 1'b0;
        sdram_rd_addr_cfg = 25'd0;
        sdram_rd_max_addr_cfg = 25'd0;
        sdram_rd_load_pulse = 1'b0;
        sdram_rd_load_stretch_req = 1'b0;
        sdram_rd_load_hold  = 4'd0;
        cache_band_index = 3'd0;
        cache_target_band = 3'd0;
        band_flush_y_min_cfg = 6'd0;
        band_flush_y_max_cfg = 6'd59;
        cache_flush_y_min = 6'd0;
        cache_window_start = 16'd0;
        cache_window_pixels = 16'd38400;
        cache_window_end = 16'd38400;
        cache_valid = 1'b0;
        cache_dirty = 1'b0;
        cache_draw_dirty = 1'b0;
        cache_maint_addr = 16'd0;
        flush_active = 1'b0;
        flush_maint_addr = 16'd0;
        flush_window_start = 16'd0;
        flush_pixels_total = 16'd0;
        flush_words_issued = 16'd0;
        flush_words_done = 16'd0;
        flush_fetch_inflight = 1'b0;
        flush_word_pending_valid = 1'b0;
        flush_word_pending = 16'd0;
        flush_load_pending = 1'b0;
        flush_drain_count = 8'd0;
        flush_band_index = 3'd0;
        flush_sdram_wr_addr = 25'd0;
        flush_sdram_wr_max_addr = 25'd0;
        flush_cache_sel = 1'b0;
        flush_generated_sky = 1'b0;
        flush_sky_x = 10'd0;
        flush_sky_row_count = 5'd0;
        flush_sky_palette = 5'd0;
        flush_fetch_clear_rgb565 = 16'h0000;
        sdram_powerup_counter = 18'd0;
        sdram_init_wait_counter = 16'd0;
        sdram_ctrl_reset_n = 1'b0;
        sdram_ready = 1'b0;
        vga_vs_d         = 1'b1;

        palette[0]  = 24'h101018;
        palette[1]  = 24'h6BA43A;
        palette[2]  = 24'h8B6341;
        palette[3]  = 24'h6F5737;
        palette[4]  = 24'h7C7C7C;
        palette[5]  = 24'hFFFFFF;
        palette[6]  = 24'hFF4040;
        palette[7]  = 24'h40A0FF;
        palette[8]  = 24'hFFD040;
        palette[9]  = 24'h5C8634;
        palette[10] = 24'h6A4A2C;
        palette[11] = 24'h9D7B4D;
```

```
        palette[12] = 24'h533823;
        palette[13] = 24'h989898;
        palette[14] = 24'h5C5C5C;
        palette[15] = 24'hA77952;
        palette[16] = 24'h59412A;
        palette[17] = 24'h4F782D;
        palette[18] = 24'h84BA57;
        palette[19] = 24'h6F4F32;
        palette[20] = 24'hAA815A;
        palette[21] = 24'h886A44;
        palette[22] = 24'h503B24;
        palette[23] = 24'h636363;
        palette[24] = 24'h9A9A9A;
        for (i = 25; i < 256; i = i + 1)
            palette[i] = {i[7:0], i[7:0], i[7:0]};
        for (i = 0; i < SKY_GRADIENT_COLORS; i = i + 1)
            sky_palette[i] = 24'h78B4F0;

        for (i = 0; i < FIFO_DEPTH; i = i + 1)
            fifo_mem[i] = 32'h0;

        /*
         * The texture atlas is loaded by voxel_texture_rom via
         * altsyncram init_file (voxel_gpu/assets/textures.mif); no $readmemh needed
         * for it here anymore.
         */
        $readmemh("voxel_gpu/assets/recip_lut.hex", recip_lut);

        for (i = 0; i < MAX_DESC_WORDS; i = i + 1)
            desc_words[i] = 32'h0;
        for (i = 0; i < MAX_DESC_WORDS; i = i + 1)
            prefetch_words[i] = 32'h0;

        for (i = 0; i < 4; i = i + 1) begin
            edge_A[i] = 32'sd0;
            edge_B[i] = 32'sd0;
            edge_C[i] = 32'sd0;
            edge_row_val[i] = 64'sd0;
            edge_cur_val[i] = 64'sd0;
        end

        for (i = 0; i < LINE_WORDS; i = i + 1) begin
            scan_linebuf0[i] = 16'h0000;
            scan_linebuf1[i] = 16'h0000;
            scan_linebuf2[i] = 16'h0000;
        end
    end

    always_comb begin
        if (!scan_current_x_valid)
            scan_rgb565_now = 16'h0000;
        else if (scan_active_bank == 2'd1)
            scan_rgb565_now = scan_linebuf1[scan_current_x];
        else if (scan_active_bank == 2'd2)
            scan_rgb565_now = scan_linebuf2[scan_current_x];
        else
            scan_rgb565_now = scan_linebuf0[scan_current_x];

        // VGA_BLANK_n is registered inside voxel_vga_counters (1-cycle late vs
```

```
2153        // hcount). scan_visible_r is then registered again here, putting it
2154        // 2 cycles behind hcount. scan_rgb565_r is only 1 cycle behind. The
2155        // resulting phase mismatch blanks pixel 0 every line (linebuf[0] is
2156        // computed but the visibility flag is still showing last cycle's
2157        // blanking from hcount=1599). Use the combinational visible predicate
2158        // so scan_visible_r and VGA_BLANK_n at the DAC are in phase.
2159        scan_visible_now = scan_current_x_valid && vcount_visible &&
2160                           scan_visible_data_ready;
2161        if (!scan_visible_now)
2162            scan_rgb565_now = 16'h0000;
2163
2164        fb_back_rd_addr = pipe0_addr;
2165        fb_back_rd_addr_o = pipe0_addr_o;
2166        draw_addr = draw_cache_addr;
2167        draw_addr_o = draw_cache_addr_o;
2168        fb_wr_addr = draw_addr;
2169        fb_wr_data = 16'h0000;
2170        fb_back_wr_en = 1'b0;
2171        fb_wr_addr_e = fb_wr_addr;
2172        fb_wr_data_e = fb_wr_data;
2173        fb_back_wr_en_e = fb_back_wr_en && (fb_wr_addr[0] == 1'b0);
2174        fb_wr_addr_o = fb_wr_addr;
2175        fb_wr_data_o = fb_wr_data;
2176        fb_back_wr_en_o = fb_back_wr_en && (fb_wr_addr[0] == 1'b1);
2177        z_rd_addr = pipe0_addr;
2178        z_rd_addr_o = pipe0_addr_o;
2179        z_wr_addr = draw_pipe_addr;
2180        z_wr_data = draw_pipe_z;
2181        z_wr_en   = 1'b0;
2182        z_wr_addr_e = z_wr_addr;
2183        z_wr_data_e = z_wr_data;
2184        z_wr_en_e = z_wr_en && (z_wr_addr[0] == 1'b0);
2185        z_wr_addr_o = z_wr_addr;
2186        z_wr_data_o = z_wr_data;
2187        z_wr_en_o = z_wr_en && (z_wr_addr[0] == 1'b1);
2188
2189        case (state)
2190            ST_CLEAR: begin
2191                fb_wr_addr = clear_addr;
2192                fb_wr_data = clear_rgb565;
2193                z_wr_addr = clear_addr;
2194                z_wr_data = Z_CLEAR_SENTINEL;
2195                z_wr_en   = 1'b1;
2196                fb_back_wr_en = 1'b1;
2197            end
2198
2199            ST_FETCH: begin
2200                if (cache_sky_patch_state) begin
2201                    fb_wr_addr = band_local_addr(desc_x_min, desc_y_min, cache_band_index);
2202                    fb_wr_data = rgb888_to_rgb565(sky_palette[desc_sky_index]);
2203                    fb_back_wr_en = 1'b1;
2204                    z_wr_addr = fb_wr_addr;
2205                    z_wr_data = Z_VALID_FAR;
2206                    z_wr_en = 1'b1;
2207                end
2208            end
2209
2210            ST_CACHE_INIT: begin
2211                /* Lazy color clear: only clear Z, two pixels/cycle. The color
```

```
2212                 * cache is left untouched; Z_CLEAR_SENTINEL means "this pixel
2213                 * was never written this band, synthesize sky/clear color when
2214                 * a later blend or flush reads it." This avoids the LAB-heavy
2215                 * dual color+Z init path that failed fitting. */
2216                z_wr_addr_e = cache_maint_addr;
2217                z_wr_data_e = Z_CLEAR_SENTINEL;
2218                z_wr_en_e = 1'b1;
2219                z_wr_addr_o = cache_maint_addr | 16'd1;
2220                z_wr_data_o = Z_CLEAR_SENTINEL;
2221                z_wr_en_o = 1'b1;
2222            end
2223
2224            ST_DRAW,
2225            ST_DRAW_FLUSH: begin
2226                if (commit_valid && commit_pass) begin
2227                    fb_wr_addr_e = commit_addr;
2228                    fb_wr_data_e = commit_color;
2229                    fb_back_wr_en_e = 1'b1;
2230                end
2231
2232                if (commit_valid && commit_pass) begin
2233                    z_wr_en_e = 1'b1;
2234                    z_wr_addr_e = commit_addr;
2235                    z_wr_data_e = commit_ztest ? commit_z : Z_VALID_FAR;
2236                end
2237
2238                if (commit_valid_o && commit_pass_o) begin
2239                    fb_wr_addr_o = commit_addr_o;
2240                    fb_wr_data_o = commit_color_o;
2241                    fb_back_wr_en_o = 1'b1;
2242                end
2243
2244                if (commit_valid_o && commit_pass_o) begin
2245                    z_wr_en_o = 1'b1;
2246                    z_wr_addr_o = commit_addr_o;
2247                    z_wr_data_o = commit_ztest_o ? commit_z_o : Z_VALID_FAR;
2248                end
2249            end
2250
2251            default: ;
2252        endcase
2253
2254        /* ST_DRAW / ST_DRAW_FLUSH and ST_CACHE_INIT drive the _e/_o write ports
2255         * directly. Other one-pixel maintenance states drive the unsuffixed
2256         * fb_wr_addr / z_wr_en signals and use this fanout block. */
2257        if (!(state == ST_DRAW || state == ST_DRAW_FLUSH ||
2258              state == ST_CACHE_INIT)) begin
2259            fb_wr_addr_e = fb_wr_addr;
2260            fb_wr_data_e = fb_wr_data;
2261            fb_back_wr_en_e = fb_back_wr_en && (fb_wr_addr[0] == 1'b0);
2262            fb_wr_addr_o = fb_wr_addr;
2263            fb_wr_data_o = fb_wr_data;
2264            fb_back_wr_en_o = fb_back_wr_en && (fb_wr_addr[0] == 1'b1);
2265            z_wr_addr_e = z_wr_addr;
2266            z_wr_data_e = z_wr_data;
2267            z_wr_en_e = z_wr_en && (z_wr_addr[0] == 1'b0);
2268            z_wr_addr_o = z_wr_addr;
2269            z_wr_data_o = z_wr_data;
2270            z_wr_en_o = z_wr_en && (z_wr_addr[0] == 1'b1);
```

```
2271        end
2272
2273        /*  Ping-pong port routing  */
2274        /* Rasterizer (draw_cache_sel): owns read+write of active cache.   */
2275        /* Flush controller (flush_cache_sel): reads from its cache.       */
2276
2277        /*  Cache A ports  */
2278        if (draw_cache_sel == 1'b0 && cache_used_by_main) begin
2279            /* A is active: rasterizer read+write. */
2280            fb_A_e_rd_addr = fb_back_rd_addr;
2281            fb_A_o_rd_addr = fb_back_rd_addr_o;
2282            fb_A_e_wr_addr = fb_wr_addr_e;
2283            fb_A_o_wr_addr = fb_wr_addr_o;
2284            fb_A_e_wr_data = fb_wr_data_e;
2285            fb_A_o_wr_data = fb_wr_data_o;
2286            fb_A_e_wr_en   = fb_back_wr_en_e;
2287            fb_A_o_wr_en   = fb_back_wr_en_o;
2288            z_A_e_rd_addr  = z_rd_addr;
2289            z_A_o_rd_addr  = z_rd_addr_o;
2290            z_A_e_wr_addr  = z_wr_addr_e;
2291            z_A_o_wr_addr  = z_wr_addr_o;
2292            z_A_e_wr_data  = z_wr_data_e;
2293            z_A_o_wr_data  = z_wr_data_o;
2294            z_A_e_wr_en    = z_wr_en_e;
2295            z_A_o_wr_en    = z_wr_en_o;
2296        end else if (flush_cache_sel == 1'b0 && flush_active) begin
2297            /* A is being flushed: flush reads. */
2298            fb_A_e_rd_addr = flush_maint_addr;
2299            fb_A_o_rd_addr = flush_maint_addr;
2300            fb_A_e_wr_addr = 16'd0;
2301            fb_A_o_wr_addr = 16'd0;
2302            fb_A_e_wr_data = 16'd0;
2303            fb_A_o_wr_data = 16'd0;
2304            fb_A_e_wr_en   = 1'b0;
2305            fb_A_o_wr_en   = 1'b0;
2306            z_A_e_rd_addr  = flush_maint_addr;
2307            z_A_o_rd_addr  = flush_maint_addr;
2308            z_A_e_wr_addr  = 16'd0;
2309            z_A_o_wr_addr  = 16'd0;
2310            z_A_e_wr_data  = 16'd0;
2311            z_A_o_wr_data  = 16'd0;
2312            z_A_e_wr_en    = 1'b0;
2313            z_A_o_wr_en    = 1'b0;
2314        end else begin
2315            /* A is idle */
2316            fb_A_e_rd_addr = 16'd0;
2317            fb_A_o_rd_addr = 16'd0;
2318            fb_A_e_wr_addr = 16'd0;
2319            fb_A_o_wr_addr = 16'd0;
2320            fb_A_e_wr_data = 16'd0;
2321            fb_A_o_wr_data = 16'd0;
2322            fb_A_e_wr_en   = 1'b0;
2323            fb_A_o_wr_en   = 1'b0;
2324            z_A_e_rd_addr  = 16'd0;
2325            z_A_o_rd_addr  = 16'd0;
2326            z_A_e_wr_addr  = 16'd0;
2327            z_A_o_wr_addr  = 16'd0;
2328            z_A_e_wr_data  = 16'd0;
2329            z_A_o_wr_data  = 16'd0;
```

```
            z_A_e_wr_en    = 1'b0;
            z_A_o_wr_en    = 1'b0;
        end

        /*  Cache B ports  */
        if (draw_cache_sel == 1'b1 && cache_used_by_main) begin
            /* B is active: rasterizer read+write. */
            fb_B_e_rd_addr = fb_back_rd_addr;
            fb_B_o_rd_addr = fb_back_rd_addr_o;
            fb_B_e_wr_addr = fb_wr_addr_e;
            fb_B_o_wr_addr = fb_wr_addr_o;
            fb_B_e_wr_data = fb_wr_data_e;
            fb_B_o_wr_data = fb_wr_data_o;
            fb_B_e_wr_en   = fb_back_wr_en_e;
            fb_B_o_wr_en   = fb_back_wr_en_o;
            z_B_e_rd_addr  = z_rd_addr;
            z_B_o_rd_addr  = z_rd_addr_o;
            z_B_e_wr_addr  = z_wr_addr_e;
            z_B_o_wr_addr  = z_wr_addr_o;
            z_B_e_wr_data  = z_wr_data_e;
            z_B_o_wr_data  = z_wr_data_o;
            z_B_e_wr_en    = z_wr_en_e;
            z_B_o_wr_en    = z_wr_en_o;
        end else if (flush_cache_sel == 1'b1 && flush_active) begin
            /* B is being flushed: flush reads. */
            fb_B_e_rd_addr = flush_maint_addr;
            fb_B_o_rd_addr = flush_maint_addr;
            fb_B_e_wr_addr = 16'd0;
            fb_B_o_wr_addr = 16'd0;
            fb_B_e_wr_data = 16'd0;
            fb_B_o_wr_data = 16'd0;
            fb_B_e_wr_en   = 1'b0;
            fb_B_o_wr_en   = 1'b0;
            z_B_e_rd_addr  = flush_maint_addr;
            z_B_o_rd_addr  = flush_maint_addr;
            z_B_e_wr_addr  = 16'd0;
            z_B_o_wr_addr  = 16'd0;
            z_B_e_wr_data  = 16'd0;
            z_B_o_wr_data  = 16'd0;
            z_B_e_wr_en    = 1'b0;
            z_B_o_wr_en    = 1'b0;
        end else begin
            /* B is idle */
            fb_B_e_rd_addr = 16'd0;
            fb_B_o_rd_addr = 16'd0;
            fb_B_e_wr_addr = 16'd0;
            fb_B_o_wr_addr = 16'd0;
            fb_B_e_wr_data = 16'd0;
            fb_B_o_wr_data = 16'd0;
            fb_B_e_wr_en   = 1'b0;
            fb_B_o_wr_en   = 1'b0;
            z_B_e_rd_addr  = 16'd0;
            z_B_o_rd_addr  = 16'd0;
            z_B_e_wr_addr  = 16'd0;
            z_B_o_wr_addr  = 16'd0;
            z_B_e_wr_data  = 16'd0;
            z_B_o_wr_data  = 16'd0;
            z_B_e_wr_en    = 1'b0;
            z_B_o_wr_en    = 1'b0;
```

```
        end
    end

    always_ff @(posedge clk) begin
        if (reset) begin
            ctrl_en          <= 1'b0;
            ctrl_ien         <= 1'b0;
            ctrl_flp_pending <= 1'b0;
            clear_pending    <= 1'b0;
            frame_count      <= 32'h0;
            pal_addr         <= 8'h0;
            sky_pal_addr     <= 5'd0;
            fog_enable       <= 1'b0;
            fog_color        <= 8'd0;
            fog_start_dist   <= 16'd0;
            fog_end_dist     <= 16'd0;
            fog_inv_proj_sq  <= 16'd0;
            extmem_ctrl      <= DEFAULT_EXTMEM_CTRL;
            extmem_front_base <= DEFAULT_EXTMEM_FRONT_BASE;
            extmem_back_base <= DEFAULT_EXTMEM_BACK_BASE;
            extmem_stride_bytes <= DEFAULT_EXTMEM_STRIDE;
            extmem_tile_cfg  <= 32'd0;
            extmem_dma_status <= 32'd0;
            vsy_latch        <= 1'b0;
            display_sel      <= 1'b0;
            display_valid    <= 1'b0;
            copy_target_sel  <= 1'b1;
            copy_complete_pending <= 1'b0;
            state            <= ST_IDLE;
            fifo_wr_ptr      <= 10'd0;
            fifo_rd_ptr      <= 10'd0;
            fifo_count       <= 11'd0;
            fetch_count      <= 6'd0;
            prefetch_count   <= 6'd0;
            prefetch_target_words <= BASE_QUAD_WORDS_6;
            prefetch_active  <= 1'b0;
            prefetch_valid   <= 1'b0;
            draw_flush_count <= 4'd0;
            clear_addr       <= 16'd0;
            draw_row_base    <= 16'd0;
            draw_x_min       <= 10'd0;
            draw_x_max       <= 10'd0;
            draw_x_cur       <= 10'd0;
            draw_y_min       <= 9'd0;
            draw_y_max       <= 9'd0;
            draw_y_cur       <= 9'd0;
            draw_tex_or_color <= 8'd0;
            draw_flags       <= 8'd0;
            draw_z0          <= 16'd0;
            draw_dz_dx       <= 16'sd0;
            draw_dz_dy       <= 16'sd0;
            draw_uw_0        <= 32'sd0;
            draw_uw_dx       <= 32'sd0;
            draw_uw_dy       <= 32'sd0;
            draw_vw_0        <= 32'sd0;
            draw_vw_dx       <= 32'sd0;
            draw_vw_dy       <= 32'sd0;
            draw_iw_0        <= 32'sd0;
            draw_iw_dx       <= 32'sd0;
```

```
            draw_iw_dy       <= 32'sd0;
            z_row_val        <= 48'sd0;
            z_cur_val        <= 48'sd0;
            uw_row_val       <= 64'sd0;
            uw_cur_val       <= 64'sd0;
            vw_row_val       <= 64'sd0;
            vw_cur_val       <= 64'sd0;
            iw_row_val       <= 64'sd0;
            iw_cur_val       <= 64'sd0;
            pipe0_valid      <= 1'b0;
            pipe0_inside     <= 1'b0;
            pipe0_ztest      <= 1'b0;
            pipe0_textured   <= 1'b0;
            pipe0_alpha_key  <= 1'b0;
            pipe0_alpha      <= 2'd0;
            pipe0_fog        <= 1'b0;
            pipe0_light_bank <= 2'd0;
            pipe0_tex_or_color <= 8'd0;
            pipe0_addr       <= 16'd0;
            pipe0_z          <= 16'd0;
            pipe0_x          <= 10'd0;
            pipe0_y          <= 9'd0;
            pipe0_uw_q       <= 32'sd0;
            pipe0_vw_q       <= 32'sd0;
            pipe0_iw_q       <= 32'd0;
            recip0_valid     <= 1'b0;
            recip0_inside    <= 1'b0;
            recip0_ztest     <= 1'b0;
            recip0_textured  <= 1'b0;
            recip0_alpha_key <= 1'b0;
            recip0_alpha     <= 2'd0;
            recip0_fog       <= 1'b0;
            recip0_light_bank <= 2'd0;
            recip0_tex_or_color <= 8'd0;
            recip0_addr      <= 16'd0;
            recip0_z         <= 16'd0;
            recip0_x         <= 10'd0;
            recip0_y         <= 9'd0;
            recip0_uw_q      <= 32'sd0;
            recip0_vw_q      <= 32'sd0;
            recip0_iw_zero   <= 1'b1;
            recip0_iw_msb    <= 6'd0;
            recip0_iw_norm_q <= 32'd0;
            recip1_valid     <= 1'b0;
            recip1_inside    <= 1'b0;
            recip1_ztest     <= 1'b0;
            recip1_textured  <= 1'b0;
            recip1_alpha_key <= 1'b0;
            recip1_alpha     <= 2'd0;
            recip1_fog       <= 1'b0;
            recip1_light_bank <= 2'd0;
            recip1_tex_or_color <= 8'd0;
            recip1_addr      <= 16'd0;
            recip1_z         <= 16'd0;
            recip1_z_ref     <= 16'd0;
            recip1_dst_rgb565 <= 16'h0000;
            recip1_x         <= 10'd0;
            recip1_y         <= 9'd0;
            recip1_uw_q      <= 32'sd0;
```

```
            recip1_vw_q      <= 32'sd0;
            recip1_iw_zero   <= 1'b1;
            recip1_iw_msb    <= 6'd0;
            recip1_iw_lut_frac <= 6'd0;
            recip1_w_norm_lo <= 32'd0;
            recip1_w_norm_hi <= 32'd0;
            recip2_valid     <= 1'b0;
            recip2_inside    <= 1'b0;
            recip2_ztest     <= 1'b0;
            recip2_textured  <= 1'b0;
            recip2_alpha_key <= 1'b0;
            recip2_alpha     <= 2'd0;
            recip2_fog       <= 1'b0;
            recip2_light_bank <= 2'd0;
            recip2_tex_or_color <= 8'd0;
            recip2_addr      <= 16'd0;
            recip2_z         <= 16'd0;
            recip2_z_ref     <= 16'd0;
            recip2_dst_rgb565 <= 16'h0000;
            recip2_x         <= 10'd0;
            recip2_y         <= 9'd0;
            recip2_uw_q      <= 32'sd0;
            recip2_vw_q      <= 32'sd0;
            recip2_iw_zero   <= 1'b1;
            recip2_iw_msb    <= 6'd0;
            recip2_w_norm_q  <= 32'd0;
            pipe1_valid      <= 1'b0;
            pipe1_inside     <= 1'b0;
            pipe1_ztest      <= 1'b0;
            pipe1_textured   <= 1'b0;
            pipe1_alpha_key  <= 1'b0;
            pipe1_alpha      <= 2'd0;
            pipe1_fog        <= 1'b0;
            pipe1_light_bank <= 2'd0;
            pipe1_tex_or_color <= 8'd0;
            pipe1_addr       <= 16'd0;
            pipe1_z          <= 16'd0;
            pipe1_z_ref      <= 16'd0;
            pipe1_dst_rgb565 <= 16'h0000;
            pipe1_x          <= 10'd0;
            pipe1_y          <= 9'd0;
            pipe1_uw_q       <= 32'sd0;
            pipe1_vw_q       <= 32'sd0;
            pipe1_w_q        <= 32'd0;
            tex0_valid       <= 1'b0;
            tex0_inside      <= 1'b0;
            tex0_ztest       <= 1'b0;
            tex0_textured    <= 1'b0;
            tex0_alpha_key   <= 1'b0;
            tex0_alpha       <= 2'd0;
            tex0_fog         <= 1'b0;
            tex0_light_bank  <= 2'd0;
            tex0_tex_or_color <= 8'd0;
            tex0_addr        <= 16'd0;
            tex0_z           <= 16'd0;
            tex0_z_ref       <= 16'd0;
            tex0_dst_rgb565  <= 16'h0000;
            tex0_x           <= 10'd0;
            tex0_y           <= 9'd0;
```

```
            tex0_w_q         <= 32'd0;
            tex0_u_prod      <= 64'sd0;
            tex0_v_prod      <= 64'sd0;
            pipe2_valid      <= 1'b0;
            pipe2_inside     <= 1'b0;
            pipe2_ztest      <= 1'b0;
            pipe2_textured   <= 1'b0;
            pipe2_alpha_key  <= 1'b0;
            pipe2_alpha      <= 2'd0;
            pipe2_fog        <= 1'b0;
            pipe2_light_bank <= 2'd0;
            pipe2_tex_or_color <= 8'd0;
            pipe2_addr       <= 16'd0;
            pipe2_z          <= 16'd0;
            pipe2_z_ref      <= 16'd0;
            pipe2_dst_rgb565 <= 16'h0000;
            pipe2_x          <= 10'd0;
            pipe2_y          <= 9'd0;
            pipe2_w_q        <= 32'd0;
            pipe2_tex_addr   <= 15'd0;
            draw_pipe_valid  <= 1'b0;
            draw_pipe_inside <= 1'b0;
            draw_pipe_ztest  <= 1'b0;
            draw_pipe_textured <= 1'b0;
            draw_pipe_alpha_key <= 1'b0;
            draw_pipe_alpha <= 2'd0;
            draw_pipe_fog    <= 1'b0;
            draw_pipe_light_bank <= 2'd0;
            draw_pipe_tex_or_color <= 8'd0;
            draw_pipe_addr   <= 16'd0;
            draw_pipe_z      <= 16'd0;
            draw_pipe_z_ref  <= 16'd0;
            draw_pipe_dst_rgb565 <= 16'h0000;
            draw_pipe_x      <= 10'd0;
            draw_pipe_y      <= 9'd0;
            draw_pipe_w_q    <= 32'd0;
            draw_is_band_primer <= 1'b0;
            pal_rd_valid     <= 1'b0;
            pal_rd_pass      <= 1'b0;
            pal_rd_ztest     <= 1'b0;
            pal_rd_alpha     <= 2'd0;
            pal_rd_fog       <= 1'b0;
            pal_rd_addr      <= 16'd0;
            pal_rd_z         <= 16'd0;
            pal_rd_src_addr  <= 8'd0;
            pal_rd_fog_addr  <= 8'd0;
            pal_rd_src_sky   <= 1'b0;
            pal_rd_sky_index <= 5'd0;
            pal_rd_dst_rgb565 <= 16'h0000;
            pal_rd_w_q       <= 32'd0;
            pal_rd_ray_scale_q16 <= 34'd0;
            plr_valid        <= 1'b0;
            plr_pass         <= 1'b0;
            plr_ztest        <= 1'b0;
            plr_alpha        <= 2'd0;
            plr_fog          <= 1'b0;
            plr_addr         <= 16'd0;
            plr_z            <= 16'd0;
            plr_src_rgb      <= 24'h000000;
```

```
            plr_dst_rgb565   <= 16'h0000;
            plr_fog_rgb      <= 24'h000000;
            plr_w_q          <= 32'd0;
            plr_ray_scale_q16 <= 34'd0;
            fog0_valid       <= 1'b0;
            fog0_pass        <= 1'b0;
            fog0_ztest       <= 1'b0;
            fog0_alpha       <= 2'd0;
            fog0_fog         <= 1'b0;
            fog0_addr        <= 16'd0;
            fog0_z           <= 16'd0;
            fog0_src_rgb565  <= 16'h0000;
            fog0_dst_rgb565  <= 16'h0000;
            fog0_fog_rgb565  <= 16'h0000;
            fog0_w_q         <= 32'd0;
            fog0_ray_scale_q16 <= 34'd0;
            fog1_valid       <= 1'b0;
            fog1_pass        <= 1'b0;
            fog1_ztest       <= 1'b0;
            fog1_alpha       <= 2'd0;
            fog1_fog         <= 1'b0;
            fog1_addr        <= 16'd0;
            fog1_z           <= 16'd0;
            fog1_src_rgb565  <= 16'h0000;
            fog1_dst_rgb565  <= 16'h0000;
            fog1_fog_rgb565  <= 16'h0000;
            fog1_radial_q8_8 <= 16'd0;
            commit_valid     <= 1'b0;
            commit_pass      <= 1'b0;
            commit_ztest     <= 1'b0;
            commit_addr      <= 16'd0;
            commit_z         <= 16'd0;
            commit_color     <= 16'h0000;
            pipe0_valid_o    <= 1'b0;
            recip0_valid_o   <= 1'b0;
            recip1_valid_o   <= 1'b0;
            recip2_valid_o   <= 1'b0;
            pipe1_valid_o    <= 1'b0;
            tex0_valid_o     <= 1'b0;
            pipe2_valid_o    <= 1'b0;
            draw_pipe_valid_o <= 1'b0;
            pal_rd_valid_o   <= 1'b0;
            pal_rd_src_sky_o <= 1'b0;
            pal_rd_sky_index_o <= 5'd0;
            plr_valid_o      <= 1'b0;
            fog0_valid_o     <= 1'b0;
            fog1_valid_o     <= 1'b0;
            commit_valid_o   <= 1'b0;
            scan_rgb565_r    <= 16'h0000;
            scan_line0_ready <= 1'b0;
            scan_line1_ready <= 1'b0;
            scan_line2_ready <= 1'b0;
            scan_line0_row   <= 9'd0;
            scan_line1_row   <= 9'd0;
            scan_line2_row   <= 9'd0;
            scan_active_bank <= 2'd0;
            scan_active_valid <= 1'b0;
            scan_active_row  <= 9'd0;
            scan_fill_active <= 1'b0;
```

```
            scan_fill_armed  <= 1'b0;
            scan_fill_load_pending <= 1'b0;
            scan_fill_bank   <= 2'd0;
            scan_fill_row    <= 9'd0;
            scan_fill_base_words <= 25'd0;
            scan_fill_store_idx <= 10'd0;
            scan_rd_capture <= 1'b0;
            scan_late_count <= 16'd0;
            sdram_rd_addr_cfg <= 25'd0;
            sdram_rd_max_addr_cfg <= 25'd0;
            sdram_rd_load_pulse <= 1'b0;
            sdram_rd_load_stretch_req <= 1'b0;
            sdram_rd_load_hold  <= 4'd0;
            cache_band_index <= 3'd0;
            cache_target_band <= 3'd0;
            band_index_cfg <= 3'd0;
            band_flush_y_min_cfg <= 6'd0;
            band_flush_y_max_cfg <= 6'd59;
            cache_flush_y_min <= 6'd0;
            cache_window_start <= 16'd0;
            cache_window_pixels <= 16'd38400;
            cache_window_end <= 16'd38400;
            band_begin_pending <= 1'b0;
            band_flush_pending <= 1'b0;
            cache_valid <= 1'b0;
            cache_dirty <= 1'b0;
            cache_draw_dirty <= 1'b0;
            cache_maint_addr <= 16'd0;
            /* Ping-pong cache reset */
            draw_cache_sel <= 1'b0;
            fb_A_rd_sel_q <= 1'b0;
            fb_B_rd_sel_q <= 1'b0;
            z_A_rd_sel_q <= 1'b0;
            z_B_rd_sel_q <= 1'b0;
            flush_active <= 1'b0;
            flush_maint_addr <= 16'd0;
            flush_window_start <= 16'd0;
            flush_pixels_total <= 16'd0;
            flush_words_issued <= 16'd0;
            flush_words_done <= 16'd0;
            flush_fetch_inflight <= 1'b0;
            flush_word_pending_valid <= 1'b0;
            flush_word_pending <= 16'd0;
            flush_load_pending <= 1'b0;
            flush_drain_count <= 8'd0;
            flush_band_index <= 3'd0;
            flush_sdram_wr_addr <= 25'd0;
            flush_sdram_wr_max_addr <= 25'd0;
            flush_cache_sel <= 1'b0;
            flush_generated_sky <= 1'b0;
            flush_sky_x <= 10'd0;
            flush_sky_row_count <= 5'd0;
            flush_sky_palette <= 5'd0;
            flush_fetch_clear_rgb565 <= 16'h0000;
            draw_row_inside <= 1'b0;
            sdram_powerup_counter <= 18'd0;
            sdram_init_wait_counter <= 16'd0;
            sdram_ctrl_reset_n <= 1'b0;
            sdram_ready <= 1'b0;
```

```
            vga_vs_d <= 1'b1;
            scan_visible_r <= 1'b0;
            for (ei = 0; ei < 4; ei = ei + 1) begin
                edge_A[ei] <= 32'sd0;
                edge_B[ei] <= 32'sd0;
                edge_C[ei] <= 32'sd0;
                edge_row_val[ei] <= 64'sd0;
                edge_cur_val[ei] <= 64'sd0;
            end
        end else begin
            vga_vs_d <= VGA_VS;
            scan_rgb565_r <= scan_rgb565_now;
            scan_visible_r <= scan_visible_now;
            sdram_rd_load_pulse <= 1'b0;
            sdram_rd_load_stretch_req <= 1'b0;
            scan_rd_capture <= scan_rd_pop;
            fb_A_rd_sel_q <= fb_A_e_rd_addr[0];
            fb_B_rd_sel_q <= fb_B_e_rd_addr[0];
            z_A_rd_sel_q <= z_A_e_rd_addr[0];
            z_B_rd_sel_q <= z_B_e_rd_addr[0];

            /*
             * Whenever a stretched pulse is requested (sampled here as the
             * previous-cycle scheduler result), prime the hold counter so
             * RD_LOAD stays high for 3 additional REF_CLK cycles (4 total).
             */
            if (sdram_rd_load_pulse && sdram_rd_load_stretch_req) begin
                sdram_rd_load_hold <= 4'd3;
            end else if (sdram_rd_load_hold != 4'd0) begin
                sdram_rd_load_hold <= sdram_rd_load_hold - 4'd1;
            end

            if (!sdram_ctrl_reset_n) begin
                if (sdram_powerup_counter == SDRAM_POWERUP_HOLD_LAST) begin
                    sdram_ctrl_reset_n <= 1'b1;
                end else begin
                    sdram_powerup_counter <= sdram_powerup_counter + 18'd1;
                end
            end else if (!sdram_ready) begin
                if (sdram_init_wait_counter == SDRAM_INIT_WAIT_LAST) begin
                    sdram_ready <= 1'b1;
                end else begin
                    sdram_init_wait_counter <= sdram_init_wait_counter + 16'd1;
                end
            end

            if (vsync_pulse) begin
                frame_count <= frame_count + 32'd1;
                scan_line0_ready <= 1'b0;
                scan_line1_ready <= 1'b0;
                scan_line2_ready <= 1'b0;
                scan_line0_row <= 9'd0;
                scan_line1_row <= 9'd0;
                scan_line2_row <= 9'd0;
                scan_active_bank <= 2'd0;
                scan_active_valid <= 1'b0;
                scan_active_row <= 9'd0;
                scan_fill_active <= 1'b0;
                scan_fill_armed <= 1'b0;
```

```
                scan_fill_load_pending <= 1'b0;
                scan_fill_bank <= 2'd0;
                scan_fill_row <= 9'd0;
                scan_fill_base_words <= 25'd0;
                scan_fill_store_idx <= 10'd0;
                scan_rd_capture <= 1'b0;

                if (copy_complete_pending) begin
                    display_sel <= copy_target_sel;
                    display_valid <= 1'b1;
                    ctrl_flp_pending <= 1'b0;
                    copy_complete_pending <= 1'b0;
                    vsy_latch <= 1'b1;
                    if (sdram_ready) begin
                        scan_fill_active <= 1'b1;
                        scan_fill_armed <= 1'b1;
                        scan_fill_load_pending <= 1'b0;
                        scan_fill_bank <= 2'd0;
                        scan_fill_row <= 9'd0;
                        scan_fill_base_words <= copy_target_base_words;
                        scan_fill_store_idx <= 10'd0;
                        sdram_rd_addr_cfg <= copy_target_base_words;
                        sdram_rd_max_addr_cfg <= copy_target_base_words + READ_BURST_WORDS_25;
                        sdram_rd_load_pulse <= 1'b1;
                        sdram_rd_load_stretch_req <= 1'b1;
                    end
                end else begin
                    vsy_latch <= !ctrl_flp_pending;
                    if (display_valid && sdram_ready) begin
                        scan_fill_active <= 1'b1;
                        scan_fill_armed <= 1'b1;
                        scan_fill_load_pending <= 1'b0;
                        scan_fill_bank <= 2'd0;
                        scan_fill_row <= 9'd0;
                        scan_fill_base_words <= display_base_words;
                        scan_fill_store_idx <= 10'd0;
                        sdram_rd_addr_cfg <= display_base_words;
                        sdram_rd_max_addr_cfg <= display_base_words + READ_BURST_WORDS_25;
                        sdram_rd_load_pulse <= 1'b1;
                        sdram_rd_load_stretch_req <= 1'b1;
                    end
                end
            end

            if (wr) begin
                case (address)
                    ADDR_CONTROL: begin
                        ctrl_en  <= writedata[0];
                        ctrl_ien <= writedata[2];
                        if (writedata[1]) begin
                            ctrl_flp_pending <= 1'b1;
                            vsy_latch        <= 1'b0;
                        end
                        if (writedata[3]) begin
                            clear_pending <= 1'b1;
                        end
                    end
                    ADDR_PAL_ADDR: pal_addr <= writedata[7:0];
                    ADDR_PAL_DATA: begin
```

```
                    palette[pal_addr] <= writedata[23:0];
                    pal_addr <= pal_addr + 8'd1;
                end
                ADDR_SKY_PAL_ADDR: sky_pal_addr <= writedata[4:0];
                ADDR_SKY_PAL_DATA: begin
                    if (sky_pal_addr <= SKY_GRADIENT_LAST_INDEX)
                        sky_palette[sky_pal_addr] <= writedata[23:0];
                    sky_pal_addr <= (sky_pal_addr == SKY_GRADIENT_LAST_INDEX) ?
                                    5'd0 : sky_pal_addr + 5'd1;
                end
                ADDR_FOG_RANGE: begin
                    fog_start_dist <= writedata[15:0];
                    fog_end_dist   <= writedata[31:16];
                end
                ADDR_FOG_CTRL: begin
                    fog_color       <= writedata[7:0];
                    fog_enable      <= writedata[8];
                    fog_inv_proj_sq <= writedata[31:16];
                end
                ADDR_EXTMEM_CTRL: extmem_ctrl <= writedata;
                ADDR_EXTMEM_FRONT: extmem_front_base <= writedata;
                ADDR_EXTMEM_BACK: extmem_back_base <= writedata;
                ADDR_EXTMEM_STRIDE: extmem_stride_bytes <= writedata;
                ADDR_EXTMEM_TILE: extmem_tile_cfg <= writedata;
                ADDR_BAND_INDEX: begin
                    band_index_cfg <= writedata[2:0];
                end
                ADDR_BAND_WINDOW: begin
                    band_flush_y_min_cfg <= writedata[5:0];
                    band_flush_y_max_cfg <= writedata[13:8];
                end
                ADDR_BAND_CTRL: begin
                    if (writedata[0])
                        band_begin_pending <= 1'b1;
                    if (writedata[1])
                        band_flush_pending <= 1'b1;
                end
                default: ;
            endcase
        end

        if (fifo_push_req)
            fifo_mem[fifo_wr_ptr] <= writedata;
        if (fetch_pop_req)
            desc_words[fetch_count[4:0]] <= fifo_head;
        else if (prefetch_pop_req)
            prefetch_words[prefetch_count[4:0]] <= fifo_head;

        if (prefetch_can_start) begin
            prefetch_active <= 1'b1;
            prefetch_valid <= 1'b0;
            prefetch_count <= 6'd0;
            prefetch_target_words <= BASE_QUAD_WORDS_6;
        end else if (prefetch_pop_req) begin
            prefetch_count <= prefetch_count + 6'd1;
            prefetch_target_words <= prefetch_capture_target_words;
            if (prefetch_finishes_on_pop) begin
                prefetch_active <= 1'b0;
                prefetch_valid <= 1'b1;
```

```
            end
        end

        case ({fifo_push_req, fifo_pop_req})
            2'b10: begin
                fifo_wr_ptr <= fifo_wr_ptr + 10'd1;
                fifo_count  <= fifo_count + 11'd1;
            end
            2'b01: begin
                fifo_rd_ptr <= fifo_rd_ptr + 10'd1;
                fifo_count  <= fifo_count - 11'd1;
            end
            2'b11: begin
                fifo_wr_ptr <= fifo_wr_ptr + 10'd1;
                fifo_rd_ptr <= fifo_rd_ptr + 10'd1;
            end
            default: ;
        endcase

        if (!vsync_pulse) begin
            if (scan_hblank_start && display_valid && scan_active_valid &&
                scan_target_valid && !scan_target_line_ready &&
                (scan_late_count != 16'hffff))
                scan_late_count <= scan_late_count + 16'd1;

            if (scan_fill_load_pending) begin
                scan_fill_load_pending <= 1'b0;
                sdram_rd_addr_cfg <= scan_fill_base_words +
                                     {15'd0, scan_fill_store_idx};
                sdram_rd_max_addr_cfg <= scan_fill_base_words +
                                         {15'd0, scan_fill_store_idx} +
                                         READ_BURST_WORDS_25;
                sdram_rd_load_pulse <= 1'b1;
            end else if (scan_fill_armed && !sdram_rd_load_out && !sdram_rd_empty) begin
                scan_fill_armed <= 1'b0;
            end

            if (scan_rd_capture) begin
                case (scan_fill_bank)
                    2'd0: scan_linebuf0[scan_fill_store_idx] <= sdram_rd_data;
                    2'd1: scan_linebuf1[scan_fill_store_idx] <= sdram_rd_data;
                    default: scan_linebuf2[scan_fill_store_idx] <= sdram_rd_data;
                endcase

                if (scan_fill_line_done) begin
                    /*
                     * If the SDRAM RD FIFO output register coughs up a
                     * stale word on the first pop after RD_LOAD, it
                     * lands in linebuf[0] and looks like the rightmost
                     * column wrapping to the left edge. Once the full
                     * line is resident, copy column 1 over column 0 as
                     * a one-pixel edge guard. This does not shift the
                     * scanout timing or touch the actual framebuffer.
                     */
                    case (scan_fill_bank)
                        2'd0: scan_linebuf0[10'd0] <= scan_linebuf0[10'd1];
                        2'd1: scan_linebuf1[10'd0] <= scan_linebuf1[10'd1];
                        default: scan_linebuf2[10'd0] <= scan_linebuf2[10'd1];
                    endcase
```

```
                        scan_fill_active <= 1'b0;
                        scan_fill_armed <= 1'b0;
                        case (scan_fill_bank)
                            2'd0: begin
                                scan_line0_ready <= 1'b1;
                                scan_line0_row <= scan_fill_row;
                            end
                            2'd1: begin
                                scan_line1_ready <= 1'b1;
                                scan_line1_row <= scan_fill_row;
                            end
                            default: begin
                                scan_line2_ready <= 1'b1;
                                scan_line2_row <= scan_fill_row;
                            end
                        endcase

                        if (!scan_active_valid) begin
                            scan_active_valid <= 1'b1;
                            scan_active_bank <= scan_fill_bank;
                            scan_active_row <= scan_fill_row;
                        end
                    end else begin
                        scan_fill_store_idx <= scan_fill_store_idx + 10'd1;
                        if (scan_fill_chunk_done) begin
                            scan_fill_armed <= 1'b1;
                            scan_fill_load_pending <= 1'b1;
                        end
                    end
                end

                /* Only change the displayed linebuffer in horizontal blank.
                 * If a line is late, keep showing the last complete line for
                 * this scanline instead of switching partway across the row. */
                if (scan_hblank_window && display_valid && scan_active_valid &&
                    (scan_active_row != scan_target_row)) begin
                    if (scan_line0_ready && (scan_line0_row == scan_target_row)) begin
                        scan_active_bank <= 2'd0;
                        scan_active_row <= scan_target_row;
                    end else if (scan_line1_ready && (scan_line1_row == scan_target_row)) begin
                        scan_active_bank <= 2'd1;
                        scan_active_row <= scan_target_row;
                    end else if (scan_line2_ready && (scan_line2_row == scan_target_row)) begin
                        scan_active_bank <= 2'd2;
                        scan_active_row <= scan_target_row;
                    end else if ((scan_active_row != scan_current_row) &&
                                 scan_line0_ready && (scan_line0_row == scan_current_row)) begin
                        scan_active_bank <= 2'd0;
                        scan_active_row <= scan_current_row;
                    end else if ((scan_active_row != scan_current_row) &&
                                 scan_line1_ready && (scan_line1_row == scan_current_row)) begin
                        scan_active_bank <= 2'd1;
                        scan_active_row <= scan_current_row;
                    end else if ((scan_active_row != scan_current_row) &&
                                 scan_line2_ready && (scan_line2_row == scan_current_row)) begin
                        scan_active_bank <= 2'd2;
                        scan_active_row <= scan_current_row;
                    end else if (scan_line0_ready && (scan_line0_row == scan_active_next_row))
    begin
```

```
                        scan_active_bank <= 2'd0;
                        scan_active_row <= scan_active_next_row;
                    end else if (scan_line1_ready && (scan_line1_row == scan_active_next_row)) begin
                        scan_active_bank <= 2'd1;
                        scan_active_row <= scan_active_next_row;
                    end else if (scan_line2_ready && (scan_line2_row == scan_active_next_row)) begin
                        scan_active_bank <= 2'd2;
                        scan_active_row <= scan_active_next_row;
                    end
                end

                if (scan_prefetch_req) begin
                    case (scan_prefetch_bank)
                    2'd0: begin
                        scan_fill_active <= 1'b1;
                        scan_fill_armed <= 1'b1;
                        scan_fill_load_pending <= 1'b0;
                        scan_fill_bank <= 2'd0;
                        scan_fill_row <= scan_prefetch_row;
                        scan_fill_base_words <= scan_prefetch_base_words;
                        scan_fill_store_idx <= 10'd0;
                        scan_line0_ready <= 1'b0;
                        sdram_rd_addr_cfg <= scan_prefetch_base_words;
                        sdram_rd_max_addr_cfg <= scan_prefetch_base_words + READ_BURST_WORDS_25;
                        sdram_rd_load_pulse <= 1'b1;
                        sdram_rd_load_stretch_req <= 1'b1;
                    end
                    2'd1: begin
                        scan_fill_active <= 1'b1;
                        scan_fill_armed <= 1'b1;
                        scan_fill_load_pending <= 1'b0;
                        scan_fill_bank <= 2'd1;
                        scan_fill_row <= scan_prefetch_row;
                        scan_fill_base_words <= scan_prefetch_base_words;
                        scan_fill_store_idx <= 10'd0;
                        scan_line1_ready <= 1'b0;
                        sdram_rd_addr_cfg <= scan_prefetch_base_words;
                        sdram_rd_max_addr_cfg <= scan_prefetch_base_words + READ_BURST_WORDS_25;
                        sdram_rd_load_pulse <= 1'b1;
                        sdram_rd_load_stretch_req <= 1'b1;
                    end
                    default: begin
                        scan_fill_active <= 1'b1;
                        scan_fill_armed <= 1'b1;
                        scan_fill_load_pending <= 1'b0;
                        scan_fill_bank <= 2'd2;
                        scan_fill_row <= scan_prefetch_row;
                        scan_fill_base_words <= scan_prefetch_base_words;
                        scan_fill_store_idx <= 10'd0;
                        scan_line2_ready <= 1'b0;
                        sdram_rd_addr_cfg <= scan_prefetch_base_words;
                        sdram_rd_max_addr_cfg <= scan_prefetch_base_words + READ_BURST_WORDS_25;
                        sdram_rd_load_pulse <= 1'b1;
                        sdram_rd_load_stretch_req <= 1'b1;
                    end
                    endcase
                end
```

```
3094            end
3095
3096            /*  Background flush controller  */
3097            /* Runs independently of the main FSM. Reads from the inactive
3098             * cache and streams pixels to the SDRAM write FIFO. */
3099            if (bg_flush_wr_push) begin
3100                flush_word_pending_valid <= 1'b0;
3101                flush_words_done <= flush_words_done + 16'd1;
3102            end
3103
3104            if (flush_active) begin
3105                /* Clear load pending once WR_LOAD has safely reset the FIFO. */
3106                if (bg_flush_wr_load_req)
3107                    flush_load_pending <= 1'b0;
3108
3109                /* Capture read data after one-cycle latency */
3110                if (!flush_generated_sky && flush_fetch_inflight &&
3111                    (!flush_word_pending_valid || bg_flush_wr_push)) begin
3112                    flush_word_pending <=
3113                        (flush_z_rd_data == Z_CLEAR_SENTINEL) ?
3114                        flush_fetch_clear_rgb565 : flush_fb_rd_data;
3115                    flush_word_pending_valid <= 1'b1;
3116                    flush_fetch_inflight <= 1'b0;
3117                end
3118
3119                /* Issue next read from inactive cache */
3120                if (!flush_generated_sky && flush_can_issue_read) begin
3121                    flush_maint_addr <= flush_window_start + flush_words_issued;
3122                    flush_words_issued <= flush_words_issued + 16'd1;
3123                    flush_fetch_inflight <= 1'b1;
3124                    flush_fetch_clear_rgb565 <= flush_clear_rgb565;
3125                end
3126
3127                /*
3128                 * Sky-only bands do not need the local cache as a flush source:
3129                 * no real draw committed over the lazy-cleared Z sentinels, so
3130                 * generate the SDRAM stream directly and leave both local color
3131                 * caches out of the path.
3132                 */
3133                if (flush_can_issue_sky) begin
3134                    flush_word_pending <= flush_clear_rgb565;
3135                    flush_word_pending_valid <= 1'b1;
3136                    flush_words_issued <= flush_words_issued + 16'd1;
3137                end
3138
3139                if ((!flush_generated_sky && flush_can_issue_read) ||
3140                    flush_can_issue_sky) begin
3141                    if (flush_sky_x == 10'd639) begin
3142                        flush_sky_x <= 10'd0;
3143                        if (flush_sky_row_count == 5'd19) begin
3144                            flush_sky_row_count <= 5'd0;
3145                            if (flush_sky_palette != SKY_GRADIENT_LAST_INDEX)
3146                                flush_sky_palette <= flush_sky_palette + 5'd1;
3147                        end else begin
3148                            flush_sky_row_count <= flush_sky_row_count + 5'd1;
3149                        end
3150                    end else begin
3151                        flush_sky_x <= flush_sky_x + 10'd1;
3152                    end
```

```
            end

            /* Flush complete: all words pushed to SDRAM, FIFO drained,
             * and the SDRAM controller has had time to finish the final
             * burst it already pulled out of the FIFO. */
            if ((flush_words_done == flush_pixels_total) &&
                !flush_word_pending_valid &&
                !flush_fetch_inflight &&
                (sdram_wr_use[8:0] == 9'd0)) begin
                if (flush_drain_count == COPY_DRAIN_CYCLES) begin
                    flush_active <= 1'b0;
                    flush_generated_sky <= 1'b0;
                    flush_drain_count <= 8'd0;
                end else begin
                    flush_drain_count <= flush_drain_count + 8'd1;
                end
            end else begin
                flush_drain_count <= 8'd0;
            end
        end

        // draw_pipe -> pal_rd: register palette/fog addresses.
        pal_rd_valid       <= draw_pipe_valid;
        pal_rd_pass        <= draw_commit_pass;
        pal_rd_ztest       <= draw_pipe_ztest;
        pal_rd_alpha       <= draw_pipe_alpha;
        pal_rd_fog         <= draw_pipe_fog;
        pal_rd_addr        <= draw_pipe_addr;
        pal_rd_z           <= draw_pipe_z;
        pal_rd_src_addr    <= palette_src_addr;
        pal_rd_fog_addr    <= fog_color;
        pal_rd_src_sky     <= draw_pipe_generated_sky;
        pal_rd_sky_index   <= draw_pipe_sky_index;
        pal_rd_dst_rgb565  <= draw_pipe_dst_rgb565;
        pal_rd_w_q         <= draw_pipe_w_q;
        pal_rd_ray_scale_q16 <= draw_pipe_ray_scale_q16;
        pal_rd_valid_o       <= draw_pipe_valid_o;
        pal_rd_pass_o        <= draw_commit_pass_o;
        pal_rd_ztest_o       <= draw_pipe_ztest_o;
        pal_rd_alpha_o       <= draw_pipe_alpha_o;
        pal_rd_fog_o         <= draw_pipe_fog_o;
        pal_rd_addr_o        <= draw_pipe_addr_o;
        pal_rd_z_o           <= draw_pipe_z_o;
        pal_rd_src_addr_o    <= palette_src_addr_o;
        pal_rd_fog_addr_o    <= fog_color;
        pal_rd_src_sky_o     <= draw_pipe_generated_sky_o;
        pal_rd_sky_index_o   <= draw_pipe_sky_index_o;
        pal_rd_dst_rgb565_o  <= draw_pipe_dst_rgb565_o;
        pal_rd_w_q_o         <= draw_pipe_w_q_o;
        pal_rd_ray_scale_q16_o <= draw_pipe_ray_scale_q16_o;

        // pal_rd -> plr: capture palette RGB values.
        plr_valid          <= pal_rd_valid;
        plr_pass           <= pal_rd_pass;
        plr_ztest          <= pal_rd_ztest;
        plr_alpha          <= pal_rd_alpha;
        plr_fog            <= pal_rd_fog;
        plr_addr           <= pal_rd_addr;
        plr_z              <= pal_rd_z;
```

```
            plr_src_rgb        <= pal_rd_src_sky ?
                                  sky_palette[pal_rd_sky_index] :
                                  palette[pal_rd_src_addr];
            plr_dst_rgb565     <= pal_rd_dst_rgb565;
            plr_fog_rgb        <= palette[pal_rd_fog_addr];
            plr_w_q            <= pal_rd_w_q;
            plr_ray_scale_q16  <= pal_rd_ray_scale_q16;
            plr_valid_o          <= pal_rd_valid_o;
            plr_pass_o           <= pal_rd_pass_o;
            plr_ztest_o          <= pal_rd_ztest_o;
            plr_alpha_o          <= pal_rd_alpha_o;
            plr_fog_o            <= pal_rd_fog_o;
            plr_addr_o           <= pal_rd_addr_o;
            plr_z_o              <= pal_rd_z_o;
            plr_src_rgb_o        <= pal_rd_src_sky_o ?
                                    sky_palette[pal_rd_sky_index_o] :
                                    palette[pal_rd_src_addr_o];
            plr_dst_rgb565_o     <= pal_rd_dst_rgb565_o;
            plr_fog_rgb_o        <= palette[pal_rd_fog_addr_o];
            plr_w_q_o            <= pal_rd_w_q_o;
            plr_ray_scale_q16_o  <= pal_rd_ray_scale_q16_o;

            // plr -> fog0: convert source/fog colors to RGB565.
            fog0_valid <= plr_valid;
            fog0_pass <= plr_pass;
            fog0_ztest <= plr_ztest;
            fog0_alpha <= plr_alpha;
            fog0_fog <= plr_fog;
            fog0_addr <= plr_addr;
            fog0_z <= plr_z;
            fog0_src_rgb565 <= plr_src_rgb565;
            fog0_dst_rgb565 <= plr_dst_rgb565;
            fog0_fog_rgb565 <= plr_fog_rgb565;
            fog0_w_q <= plr_w_q;
            fog0_ray_scale_q16 <= plr_ray_scale_q16;
            fog0_valid_o <= plr_valid_o;
            fog0_pass_o <= plr_pass_o;
            fog0_ztest_o <= plr_ztest_o;
            fog0_alpha_o <= plr_alpha_o;
            fog0_fog_o <= plr_fog_o;
            fog0_addr_o <= plr_addr_o;
            fog0_z_o <= plr_z_o;
            fog0_src_rgb565_o <= plr_src_rgb565_o;
            fog0_dst_rgb565_o <= plr_dst_rgb565_o;
            fog0_fog_rgb565_o <= plr_fog_rgb565_o;
            fog0_w_q_o <= plr_w_q_o;
            fog0_ray_scale_q16_o <= plr_ray_scale_q16_o;

            // fog0 -> fog1: finish radial distance in Q8.8.
            fog1_valid <= fog0_valid;
            fog1_pass <= fog0_pass;
            fog1_ztest <= fog0_ztest;
            fog1_alpha <= fog0_alpha;
            fog1_fog <= fog0_fog;
            fog1_addr <= fog0_addr;
            fog1_z <= fog0_z;
            fog1_src_rgb565 <= fog0_src_rgb565;
            fog1_dst_rgb565 <= fog0_dst_rgb565;
            fog1_fog_rgb565 <= fog0_fog_rgb565;
```

```
            fog1_radial_q8_8 <= fog0_radial_q8_8;
            fog1_valid_o <= fog0_valid_o;
            fog1_pass_o <= fog0_pass_o;
            fog1_ztest_o <= fog0_ztest_o;
            fog1_alpha_o <= fog0_alpha_o;
            fog1_fog_o <= fog0_fog_o;
            fog1_addr_o <= fog0_addr_o;
            fog1_z_o <= fog0_z_o;
            fog1_src_rgb565_o <= fog0_src_rgb565_o;
            fog1_dst_rgb565_o <= fog0_dst_rgb565_o;
            fog1_fog_rgb565_o <= fog0_fog_rgb565_o;
            fog1_radial_q8_8_o <= fog0_radial_q8_8_o;

            // fog1 -> commit: final fog/alpha-blended color.
            commit_valid <= fog1_valid;
            commit_pass <= fog1_pass;
            commit_ztest <= fog1_ztest;
            commit_addr <= fog1_addr;
            commit_z <= fog1_z;
            commit_color <= fog1_out_rgb565;
            commit_valid_o <= fog1_valid_o;
            commit_pass_o <= fog1_pass_o;
            commit_ztest_o <= fog1_ztest_o;
            commit_addr_o <= fog1_addr_o;
            commit_z_o <= fog1_z_o;
            commit_color_o <= fog1_out_rgb565_o;
            if ((state == ST_DRAW || state == ST_DRAW_FLUSH) &&
                ((commit_valid && commit_pass) ||
                 (commit_valid_o && commit_pass_o))) begin
                cache_dirty <= 1'b1;
                if (!draw_is_band_primer)
                    cache_draw_dirty <= 1'b1;
            end

            case (state)
                ST_IDLE: begin
                    fetch_count <= 6'd0;
                    draw_flush_count <= 4'd0;
                    pipe0_valid <= 1'b0;
                    recip0_valid <= 1'b0;
                    recip1_valid <= 1'b0;
                    recip2_valid <= 1'b0;
                    pipe1_valid <= 1'b0;
                    tex0_valid <= 1'b0;
                    pipe2_valid <= 1'b0;
                    draw_pipe_valid <= 1'b0;
                    pal_rd_valid <= 1'b0;
                    plr_valid <= 1'b0;
                    fog0_valid <= 1'b0;
                    fog1_valid <= 1'b0;
                    commit_valid <= 1'b0;
                    pipe0_valid_o <= 1'b0;
                    recip0_valid_o <= 1'b0;
                    recip1_valid_o <= 1'b0;
                    recip2_valid_o <= 1'b0;
                    pipe1_valid_o <= 1'b0;
                    tex0_valid_o <= 1'b0;
                    pipe2_valid_o <= 1'b0;
                    draw_pipe_valid_o <= 1'b0;
```

```
                    pal_rd_valid_o <= 1'b0;
                    plr_valid_o <= 1'b0;
                    fog0_valid_o <= 1'b0;
                    fog1_valid_o <= 1'b0;
                    commit_valid_o <= 1'b0;
                    if (clear_pending) begin
                        state         <= ST_CLEAR;
                        clear_pending <= 1'b0;
                        clear_addr    <= 16'd0;
                    end else if (band_begin_pending && band_begin_cache_available) begin
                        /* Only two local caches exist; wait here rather than
                         * toggling back into a cache still being flushed. */
                        band_begin_pending <= 1'b0;
                        cache_target_band <= band_index_cfg;
                        cache_band_index <= band_index_cfg;
                        cache_flush_y_min <= band_flush_y_min_cfg;
                        cache_window_start <= band_local_row_offset(band_flush_y_min_cfg);
                        cache_window_pixels <= band_row_window_count(
                            band_flush_y_min_cfg, band_flush_y_max_cfg);
                        cache_window_end <= band_local_row_offset(band_flush_y_min_cfg) +
                                            band_row_window_count(
                                                band_flush_y_min_cfg,
                                                band_flush_y_max_cfg);
                        cache_maint_addr <= band_local_row_offset(band_flush_y_min_cfg);
                        cache_valid <= 1'b0;
                        cache_dirty <= 1'b0;
                        cache_draw_dirty <= 1'b0;
                        /* Toggle cache selector: draw into the other cache */
                        draw_cache_sel <= ~draw_cache_sel;
                        state <= ST_CACHE_INIT;
                    end else if (band_flush_pending && !flush_active) begin
                        /* Do not require fifo_count==0: pipelined next-band
                         * descriptors may already be queued behind this flush. */
                        band_flush_pending <= 1'b0;
                        if (cache_valid && cache_dirty) begin
                            /* Flush this cache while the next band uses the other. */
                            flush_active <= 1'b1;
                            flush_band_index <= cache_band_index;
                            flush_pixels_total <= cache_window_pixels;
                            flush_window_start <= cache_window_start;
                            flush_maint_addr <= cache_window_start;
                            flush_words_issued <= 16'd0;
                            flush_words_done <= 16'd0;
                            flush_fetch_inflight <= 1'b0;
                            flush_word_pending_valid <= 1'b0;
                            flush_load_pending <= 1'b1;
                            flush_drain_count <= 8'd0;
                            flush_cache_sel <= draw_cache_sel;
                            flush_generated_sky <= sky_gradient_clear_enabled &&
                                                   !cache_draw_dirty;
                            flush_sky_x <= 10'd0;
                            flush_sky_row_count <=
                                sky_row_count_for_local_y(cache_flush_y_min);
                            flush_sky_palette <=
                                sky_clear_index_for_local_y(cache_band_index,
                                                            cache_flush_y_min);
                            flush_sdram_wr_addr <= copy_target_base_words +
                                                   band_word_offset(cache_band_index) +
                                                   {9'd0, cache_window_start};
```

```
                    flush_sdram_wr_max_addr <= copy_target_base_words +
                                               band_word_offset(cache_band_index) +
                                               {9'd0, cache_window_start} +
                                               {9'd0, cache_window_pixels};
                    cache_valid <= 1'b0;
                    cache_dirty <= 1'b0;
                    cache_draw_dirty <= 1'b0;
                end
                /* Background flush owns SDRAM writeback; stay in
                 * ST_IDLE so the next BEGIN_BAND can proceed. */
            end else if (ctrl_flp_pending && sdram_ready &&
                         !copy_complete_pending && !flush_active) begin
                copy_complete_pending <= 1'b1;
            end else if (prefetch_valid && !band_flush_pending) begin
                for (ei = 0; ei < MAX_DESC_WORDS; ei = ei + 1)
                    desc_words[ei] <= prefetch_words[ei];
                fetch_count <= prefetch_target_words;
                prefetch_valid <= 1'b0;
                state <= ST_FETCH;
            end else if (!prefetch_active && ctrl_en && (fifo_count >= 11'd16) &&
                         !band_flush_pending) begin
                /* Wait until the queued flush/begin pair advances the
                 * resident cache before fetching next-band descriptors. */
                state       <= ST_FETCH;
                fetch_count <= 6'd0;
            end
        end

        ST_CLEAR: begin
            copy_target_sel <= ~display_sel;
            cache_valid <= 1'b0;
            cache_dirty <= 1'b0;
            cache_draw_dirty <= 1'b0;
            cache_flush_y_min <= 6'd0;
            cache_window_start <= 16'd0;
            cache_window_pixels <= 16'd38400;
            cache_window_end <= 16'd38400;
            band_begin_pending <= 1'b0;
            band_flush_pending <= 1'b0;
            prefetch_count <= 6'd0;
            prefetch_target_words <= BASE_QUAD_WORDS_6;
            prefetch_active <= 1'b0;
            prefetch_valid <= 1'b0;
            copy_complete_pending <= 1'b0;
            draw_cache_sel <= 1'b0;
            /* Do not kill flush_active here -- see the comment in
             * ctrl_clear_write. Let the background flush drain
             * naturally so the SDRAM framebuffer is fully written
             * before scanout displays it. */
            state <= ST_IDLE;
        end

        ST_FETCH: begin
            if (fetch_count == fetch_target_words) begin
                draw_x_min <= desc_x_min;
                draw_x_max <= desc_x_max;
                draw_y_min <= desc_y_min;
                draw_y_max <= desc_y_max;
                draw_row_base <= band_local_addr(desc_x_start_even, desc_y_min,
```

```
                                                cache_band_index);
                draw_x_cur <= desc_x_start_even;
                draw_y_cur <= desc_y_min;
                draw_row_inside <= 1'b0;
                draw_tex_or_color <= desc_tex_or_color;
                draw_flags <= desc_flags;
                draw_is_band_primer <= desc_band_primer;
                draw_z0    <= desc_z0;
                draw_dz_dx <= desc_dz_dx;
                draw_dz_dy <= desc_dz_dy;
                if (desc_has_uv) begin
                    draw_uw_0  <= desc_uw_0;
                    draw_uw_dx <= desc_uw_dx;
                    draw_uw_dy <= desc_uw_dy;
                    draw_vw_0  <= desc_vw_0;
                    draw_vw_dx <= desc_vw_dx;
                    draw_vw_dy <= desc_vw_dy;
                    draw_iw_0  <= desc_iw_0;
                    draw_iw_dx <= desc_iw_dx;
                    draw_iw_dy <= desc_iw_dy;
                end else begin
                    draw_uw_0  <= 32'sd0;
                    draw_uw_dx <= 32'sd0;
                    draw_uw_dy <= 32'sd0;
                    draw_vw_0  <= 32'sd0;
                    draw_vw_dx <= 32'sd0;
                    draw_vw_dy <= 32'sd0;
                    draw_iw_0  <= 32'sd0;
                    draw_iw_dx <= 32'sd0;
                    draw_iw_dy <= 32'sd0;
                end
                pipe0_valid <= 1'b0;
                recip0_valid <= 1'b0;
                recip1_valid <= 1'b0;
                recip2_valid <= 1'b0;
                pipe1_valid <= 1'b0;
                tex0_valid <= 1'b0;
                pipe2_valid <= 1'b0;
                draw_pipe_valid <= 1'b0;
                pal_rd_valid <= 1'b0;
                plr_valid <= 1'b0;
                fog0_valid <= 1'b0;
                fog1_valid <= 1'b0;
                commit_valid <= 1'b0;
                pipe0_valid_o <= 1'b0;
                recip0_valid_o <= 1'b0;
                recip1_valid_o <= 1'b0;
                recip2_valid_o <= 1'b0;
                pipe1_valid_o <= 1'b0;
                tex0_valid_o <= 1'b0;
                pipe2_valid_o <= 1'b0;
                draw_pipe_valid_o <= 1'b0;
                pal_rd_valid_o <= 1'b0;
                plr_valid_o <= 1'b0;
                fog0_valid_o <= 1'b0;
                fog1_valid_o <= 1'b0;
                commit_valid_o <= 1'b0;
                for (ei = 0; ei < 4; ei = ei + 1) begin
                    edge_A[ei] <= $signed(desc_words[2 + ei * 3]);
```

```
                    edge_B[ei] <= $signed(desc_words[3 + ei * 3]);
                    edge_C[ei] <= $signed(desc_words[4 + ei * 3]);
                end

                if ((desc_x_min > desc_x_max) || (desc_y_min > desc_y_max) ||
                    desc_redundant_sky_clear)
                    state <= ST_IDLE;
                else
                    state <= ST_SETUP;
            end else if (fifo_pop_req) begin
                fetch_count <= fetch_count + 6'd1;
            end
        end

        ST_SETUP: begin
            edge_row_val[0] <= edge_eval0;
            edge_cur_val[0] <= edge_eval0;
            edge_row_val[1] <= edge_eval1;
            edge_cur_val[1] <= edge_eval1;
            edge_row_val[2] <= edge_eval2;
            edge_cur_val[2] <= edge_eval2;
            edge_row_val[3] <= edge_eval3;
            edge_cur_val[3] <= edge_eval3;

            z_row_val <= draw_z_start_val;
            z_cur_val <= draw_z_start_val;
            uw_row_val <= draw_uw_start_val;
            uw_cur_val <= draw_uw_start_val;
            vw_row_val <= draw_vw_start_val;
            vw_cur_val <= draw_vw_start_val;
            iw_row_val <= draw_iw_start_val;
            iw_cur_val <= draw_iw_start_val;

            state <= ST_DRAW;
        end

        ST_DRAW,
        ST_DRAW_FLUSH: begin
            // pipe0 -> recip0: normalize 1/w; all raster metadata carries.
            recip0_valid <= pipe0_valid;
            recip0_inside <= pipe0_inside;
            recip0_ztest <= pipe0_ztest;
            recip0_textured <= pipe0_textured;
            recip0_alpha_key <= pipe0_alpha_key;
            recip0_alpha <= pipe0_alpha;
            recip0_fog <= pipe0_fog;
            recip0_light_bank <= pipe0_light_bank;
            recip0_tex_or_color <= pipe0_tex_or_color;
            recip0_addr <= pipe0_addr;
            recip0_z <= pipe0_z;
            recip0_x <= pipe0_x;
            recip0_y <= pipe0_y;
            recip0_uw_q <= pipe0_uw_q;
            recip0_vw_q <= pipe0_vw_q;
            recip0_iw_zero <= (pipe0_iw_q == 32'd0);
            recip0_iw_msb <= pipe0_iw_msb;
            recip0_iw_norm_q <= pipe0_iw_norm_q;
            recip0_valid_o <= pipe0_valid_o;
            recip0_inside_o <= pipe0_inside_o;
```

```
                    recip0_ztest_o <= pipe0_ztest_o;
                    recip0_textured_o <= pipe0_textured_o;
                    recip0_alpha_key_o <= pipe0_alpha_key_o;
                    recip0_alpha_o <= pipe0_alpha_o;
                    recip0_fog_o <= pipe0_fog_o;
                    recip0_light_bank_o <= pipe0_light_bank_o;
                    recip0_tex_or_color_o <= pipe0_tex_or_color_o;
                    recip0_addr_o <= pipe0_addr_o;
                    recip0_z_o <= pipe0_z_o;
                    recip0_x_o <= pipe0_x_o;
                    recip0_y_o <= pipe0_y_o;
                    recip0_uw_q_o <= pipe0_uw_q_o;
                    recip0_vw_q_o <= pipe0_vw_q_o;
                    recip0_iw_zero_o <= (pipe0_iw_q_o == 32'd0);
                    recip0_iw_msb_o <= pipe0_iw_msb_o;
                    recip0_iw_norm_q_o <= pipe0_iw_norm_q_o;

                    // recip0 -> recip1: read reciprocal LUT and destination cache.
                    recip1_valid <= recip0_valid;
                    recip1_inside <= recip0_inside;
                    recip1_ztest <= recip0_ztest;
                    recip1_textured <= recip0_textured;
                    recip1_alpha_key <= recip0_alpha_key;
                    recip1_alpha <= recip0_alpha;
                    recip1_fog <= recip0_fog;
                    recip1_light_bank <= recip0_light_bank;
                    recip1_tex_or_color <= recip0_tex_or_color;
                    recip1_addr <= recip0_addr;
                    recip1_z <= recip0_z;
                    recip1_z_ref <= z_draw_rd_data_e;
                    recip1_dst_rgb565 <=
                        (z_draw_rd_data_e == Z_CLEAR_SENTINEL) ?
                        draw_clear_rgb565 : fb_draw_rd_data_e;
                    recip1_x <= recip0_x;
                    recip1_y <= recip0_y;
                    recip1_uw_q <= recip0_uw_q;
                    recip1_vw_q <= recip0_vw_q;
                    recip1_iw_zero <= recip0_iw_zero;
                    recip1_iw_msb <= recip0_iw_msb;
                    recip1_iw_lut_frac <= recip0_iw_lut_frac;
                    recip1_w_norm_lo <= recip_lut[recip0_iw_lut_idx];
                    recip1_w_norm_hi <= recip_lut[recip0_iw_lut_idx + 11'd1];
                    recip1_valid_o <= recip0_valid_o;
                    recip1_inside_o <= recip0_inside_o;
                    recip1_ztest_o <= recip0_ztest_o;
                    recip1_textured_o <= recip0_textured_o;
                    recip1_alpha_key_o <= recip0_alpha_key_o;
                    recip1_alpha_o <= recip0_alpha_o;
                    recip1_fog_o <= recip0_fog_o;
                    recip1_light_bank_o <= recip0_light_bank_o;
                    recip1_tex_or_color_o <= recip0_tex_or_color_o;
                    recip1_addr_o <= recip0_addr_o;
                    recip1_z_o <= recip0_z_o;
                    recip1_z_ref_o <= z_draw_rd_data_o;
                    recip1_dst_rgb565_o <=
                        (z_draw_rd_data_o == Z_CLEAR_SENTINEL) ?
                        draw_clear_rgb565 : fb_draw_rd_data_o;
                    recip1_x_o <= recip0_x_o;
                    recip1_y_o <= recip0_y_o;
```

```
                    recip1_uw_q_o <= recip0_uw_q_o;
                    recip1_vw_q_o <= recip0_vw_q_o;
                    recip1_iw_zero_o <= recip0_iw_zero_o;
                    recip1_iw_msb_o <= recip0_iw_msb_o;
                    recip1_iw_lut_frac_o <= recip0_iw_lut_frac_o;
                    recip1_w_norm_lo_o <= recip_lut[recip0_iw_lut_idx_o];
                    recip1_w_norm_hi_o <= recip_lut[recip0_iw_lut_idx_o + 11'd1];

                    // recip1 -> recip2: interpolate reciprocal.
                    recip2_valid <= recip1_valid;
                    recip2_inside <= recip1_inside;
                    recip2_ztest <= recip1_ztest;
                    recip2_textured <= recip1_textured;
                    recip2_alpha_key <= recip1_alpha_key;
                    recip2_alpha <= recip1_alpha;
                    recip2_fog <= recip1_fog;
                    recip2_light_bank <= recip1_light_bank;
                    recip2_tex_or_color <= recip1_tex_or_color;
                    recip2_addr <= recip1_addr;
                    recip2_z <= recip1_z;
                    recip2_z_ref <= recip1_z_ref;
                    recip2_dst_rgb565 <= recip1_dst_rgb565;
                    recip2_x <= recip1_x;
                    recip2_y <= recip1_y;
                    recip2_uw_q <= recip1_uw_q;
                    recip2_vw_q <= recip1_vw_q;
                    recip2_iw_zero <= recip1_iw_zero;
                    recip2_iw_msb <= recip1_iw_msb;
                    recip2_w_norm_q <= recip1_w_norm_q;
                    recip2_valid_o <= recip1_valid_o;
                    recip2_inside_o <= recip1_inside_o;
                    recip2_ztest_o <= recip1_ztest_o;
                    recip2_textured_o <= recip1_textured_o;
                    recip2_alpha_key_o <= recip1_alpha_key_o;
                    recip2_alpha_o <= recip1_alpha_o;
                    recip2_fog_o <= recip1_fog_o;
                    recip2_light_bank_o <= recip1_light_bank_o;
                    recip2_tex_or_color_o <= recip1_tex_or_color_o;
                    recip2_addr_o <= recip1_addr_o;
                    recip2_z_o <= recip1_z_o;
                    recip2_z_ref_o <= recip1_z_ref_o;
                    recip2_dst_rgb565_o <= recip1_dst_rgb565_o;
                    recip2_x_o <= recip1_x_o;
                    recip2_y_o <= recip1_y_o;
                    recip2_uw_q_o <= recip1_uw_q_o;
                    recip2_vw_q_o <= recip1_vw_q_o;
                    recip2_iw_zero_o <= recip1_iw_zero_o;
                    recip2_iw_msb_o <= recip1_iw_msb_o;
                    recip2_w_norm_q_o <= recip1_w_norm_q_o;

                    // recip2 -> pipe1: denormalize w.
                    pipe1_valid <= recip2_valid;
                    pipe1_inside <= recip2_inside;
                    pipe1_ztest <= recip2_ztest;
                    pipe1_textured <= recip2_textured;
                    pipe1_alpha_key <= recip2_alpha_key;
                    pipe1_alpha <= recip2_alpha;
                    pipe1_fog <= recip2_fog;
                    pipe1_light_bank <= recip2_light_bank;
```

```
                    pipe1_tex_or_color <= recip2_tex_or_color;
                    pipe1_addr <= recip2_addr;
                    pipe1_z <= recip2_z;
                    pipe1_z_ref <= recip2_z_ref;
                    pipe1_dst_rgb565 <= recip2_dst_rgb565;
                    pipe1_x <= recip2_x;
                    pipe1_y <= recip2_y;
                    pipe1_uw_q <= recip2_uw_q;
                    pipe1_vw_q <= recip2_vw_q;
                    pipe1_w_q <= recip2_w_q;
                    pipe1_valid_o <= recip2_valid_o;
                    pipe1_inside_o <= recip2_inside_o;
                    pipe1_ztest_o <= recip2_ztest_o;
                    pipe1_textured_o <= recip2_textured_o;
                    pipe1_alpha_key_o <= recip2_alpha_key_o;
                    pipe1_alpha_o <= recip2_alpha_o;
                    pipe1_fog_o <= recip2_fog_o;
                    pipe1_light_bank_o <= recip2_light_bank_o;
                    pipe1_tex_or_color_o <= recip2_tex_or_color_o;
                    pipe1_addr_o <= recip2_addr_o;
                    pipe1_z_o <= recip2_z_o;
                    pipe1_z_ref_o <= recip2_z_ref_o;
                    pipe1_dst_rgb565_o <= recip2_dst_rgb565_o;
                    pipe1_x_o <= recip2_x_o;
                    pipe1_y_o <= recip2_y_o;
                    pipe1_uw_q_o <= recip2_uw_q_o;
                    pipe1_vw_q_o <= recip2_vw_q_o;
                    pipe1_w_q_o <= recip2_w_q_o;

                    // pipe1 -> tex0: multiply (u/w, v/w) by w.
                    tex0_valid <= pipe1_valid;
                    tex0_inside <= pipe1_inside;
                    tex0_ztest <= pipe1_ztest;
                    tex0_textured <= pipe1_textured;
                    tex0_alpha_key <= pipe1_alpha_key;
                    tex0_alpha <= pipe1_alpha;
                    tex0_fog <= pipe1_fog;
                    tex0_light_bank <= pipe1_light_bank;
                    tex0_tex_or_color <= pipe1_tex_or_color;
                    tex0_addr <= pipe1_addr;
                    tex0_z <= pipe1_z;
                    tex0_z_ref <= pipe1_z_ref;
                    tex0_dst_rgb565 <= pipe1_dst_rgb565;
                    tex0_x <= pipe1_x;
                    tex0_y <= pipe1_y;
                    tex0_w_q <= pipe1_w_q;
                    tex0_u_prod <= pipe1_u_prod;
                    tex0_v_prod <= pipe1_v_prod;
                    tex0_valid_o <= pipe1_valid_o;
                    tex0_inside_o <= pipe1_inside_o;
                    tex0_ztest_o <= pipe1_ztest_o;
                    tex0_textured_o <= pipe1_textured_o;
                    tex0_alpha_key_o <= pipe1_alpha_key_o;
                    tex0_alpha_o <= pipe1_alpha_o;
                    tex0_fog_o <= pipe1_fog_o;
                    tex0_light_bank_o <= pipe1_light_bank_o;
                    tex0_tex_or_color_o <= pipe1_tex_or_color_o;
                    tex0_addr_o <= pipe1_addr_o;
                    tex0_z_o <= pipe1_z_o;
```

```
                    tex0_z_ref_o <= pipe1_z_ref_o;
                    tex0_dst_rgb565_o <= pipe1_dst_rgb565_o;
                    tex0_x_o <= pipe1_x_o;
                    tex0_y_o <= pipe1_y_o;
                    tex0_w_q_o <= pipe1_w_q_o;
                    tex0_u_prod_o <= pipe1_u_prod_o;
                    tex0_v_prod_o <= pipe1_v_prod_o;

                    // tex0 -> pipe2: build texture atlas address.
                    pipe2_valid <= tex0_valid;
                    pipe2_inside <= tex0_inside;
                    pipe2_ztest <= tex0_ztest;
                    pipe2_textured <= tex0_textured;
                    pipe2_alpha_key <= tex0_alpha_key;
                    pipe2_alpha <= tex0_alpha;
                    pipe2_fog <= tex0_fog;
                    pipe2_light_bank <= tex0_light_bank;
                    pipe2_tex_or_color <= tex0_tex_or_color;
                    pipe2_addr <= tex0_addr;
                    pipe2_z <= tex0_z;
                    pipe2_z_ref <= tex0_z_ref;
                    pipe2_dst_rgb565 <= tex0_dst_rgb565;
                    pipe2_x <= tex0_x;
                    pipe2_y <= tex0_y;
                    pipe2_w_q <= tex0_w_q;
                    pipe2_tex_addr <= tex0_tex_addr;
                    pipe2_valid_o <= tex0_valid_o;
                    pipe2_inside_o <= tex0_inside_o;
                    pipe2_ztest_o <= tex0_ztest_o;
                    pipe2_textured_o <= tex0_textured_o;
                    pipe2_alpha_key_o <= tex0_alpha_key_o;
                    pipe2_alpha_o <= tex0_alpha_o;
                    pipe2_fog_o <= tex0_fog_o;
                    pipe2_light_bank_o <= tex0_light_bank_o;
                    pipe2_tex_or_color_o <= tex0_tex_or_color_o;
                    pipe2_addr_o <= tex0_addr_o;
                    pipe2_z_o <= tex0_z_o;
                    pipe2_z_ref_o <= tex0_z_ref_o;
                    pipe2_dst_rgb565_o <= tex0_dst_rgb565_o;
                    pipe2_x_o <= tex0_x_o;
                    pipe2_y_o <= tex0_y_o;
                    pipe2_w_q_o <= tex0_w_q_o;
                    pipe2_tex_addr_o <= tex0_tex_addr_o;

                    // pipe2 -> draw_pipe: texture ROM output is now aligned.
                    draw_pipe_valid <= pipe2_valid;
                    draw_pipe_inside <= pipe2_inside;
                    draw_pipe_ztest <= pipe2_ztest;
                    draw_pipe_textured <= pipe2_textured;
                    draw_pipe_alpha_key <= pipe2_alpha_key;
                    draw_pipe_alpha <= pipe2_alpha;
                    draw_pipe_fog <= pipe2_fog;
                    draw_pipe_light_bank <= pipe2_light_bank;
                    draw_pipe_tex_or_color <= pipe2_tex_or_color;
                    draw_pipe_addr <= pipe2_addr;
                    draw_pipe_z <= pipe2_z;
                    draw_pipe_z_ref <= pipe2_z_ref;
                    draw_pipe_dst_rgb565 <= pipe2_dst_rgb565;
                    draw_pipe_x <= pipe2_x;
```

```
                draw_pipe_y <= pipe2_y;
                draw_pipe_w_q <= pipe2_w_q;
                draw_pipe_valid_o <= pipe2_valid_o;
                draw_pipe_inside_o <= pipe2_inside_o;
                draw_pipe_ztest_o <= pipe2_ztest_o;
                draw_pipe_textured_o <= pipe2_textured_o;
                draw_pipe_alpha_key_o <= pipe2_alpha_key_o;
                draw_pipe_alpha_o <= pipe2_alpha_o;
                draw_pipe_fog_o <= pipe2_fog_o;
                draw_pipe_light_bank_o <= pipe2_light_bank_o;
                draw_pipe_tex_or_color_o <= pipe2_tex_or_color_o;
                draw_pipe_addr_o <= pipe2_addr_o;
                draw_pipe_z_o <= pipe2_z_o;
                draw_pipe_z_ref_o <= pipe2_z_ref_o;
                draw_pipe_dst_rgb565_o <= pipe2_dst_rgb565_o;
                draw_pipe_x_o <= pipe2_x_o;
                draw_pipe_y_o <= pipe2_y_o;
                draw_pipe_w_q_o <= pipe2_w_q_o;

                if (state == ST_DRAW && draw_cache_hit) begin
                    pipe0_valid <= 1'b1;
                    pipe0_inside <= draw_inside_lane0;
                    pipe0_ztest <= draw_flags[FLAG_ZTEST_BIT];
                    pipe0_textured <= draw_flags[FLAG_TEX_BIT];
                    pipe0_alpha_key <= draw_flags[FLAG_ALPHA_KEY_BIT];
                    pipe0_alpha <= draw_flags[FLAG_ALPHA_MSB:FLAG_ALPHA_LSB];
                    pipe0_fog <= draw_flags[FLAG_FOG_BIT];
                    pipe0_light_bank <= draw_flags[FLAG_LIGHT_MSB:FLAG_LIGHT_LSB];
                    pipe0_tex_or_color <= draw_tex_or_color;
                    pipe0_addr <= draw_addr;
                    pipe0_z <= draw_z_value;
                    pipe0_x <= draw_x_cur;
                    pipe0_y <= draw_y_cur;
                    pipe0_uw_q <= draw_uw_q;
                    pipe0_vw_q <= draw_vw_q;
                    pipe0_iw_q <= draw_iw_q;
                    pipe0_valid_o <= 1'b1;
                    pipe0_inside_o <= draw_inside_lane1;
                    pipe0_ztest_o <= draw_flags[FLAG_ZTEST_BIT];
                    pipe0_textured_o <= draw_flags[FLAG_TEX_BIT];
                    pipe0_alpha_key_o <= draw_flags[FLAG_ALPHA_KEY_BIT];
                    pipe0_alpha_o <= draw_flags[FLAG_ALPHA_MSB:FLAG_ALPHA_LSB];
                    pipe0_fog_o <= draw_flags[FLAG_FOG_BIT];
                    pipe0_light_bank_o <= draw_flags[FLAG_LIGHT_MSB:FLAG_LIGHT_LSB];
                    pipe0_tex_or_color_o <= draw_tex_or_color;
                    pipe0_addr_o <= draw_addr_o;
                    pipe0_z_o <= draw_z_value_o;
                    pipe0_x_o <= draw_x_next;
                    pipe0_y_o <= draw_y_cur;
                    pipe0_uw_q_o <= draw_uw_q_o;
                    pipe0_vw_q_o <= draw_vw_q_o;
                    pipe0_iw_q_o <= draw_iw_q_o;

                    /* Track inside-to-outside transition for early row exit */
                    if (draw_pair_edge_inside)
                        draw_row_inside <= 1'b1;

                    /* Early scanline exit: if we were inside the quad on
                     * this row and now we're outside, all remaining pixels
```

```
                     * on this row are also outside (convex quad property).
                     * Skip directly to the next row. */
                    if (draw_pair_last || draw_pair_exited) begin
                        if (draw_y_cur == draw_y_max) begin
                            state <= ST_DRAW_FLUSH;
                            draw_flush_count <= DRAW_FLUSH_CYCLES;
                        end else begin
                            draw_row_base <= draw_row_base + 16'd640;
                            draw_x_cur <= draw_x_start_even;
                            draw_y_cur <= draw_y_cur + 9'd1;
                            draw_row_inside <= 1'b0;
                            edge_row_val[0] <= edge_next_row0;
                            edge_cur_val[0] <= edge_next_row0;
                            edge_row_val[1] <= edge_next_row1;
                            edge_cur_val[1] <= edge_next_row1;
                            edge_row_val[2] <= edge_next_row2;
                            edge_cur_val[2] <= edge_next_row2;
                            edge_row_val[3] <= edge_next_row3;
                            edge_cur_val[3] <= edge_next_row3;
                            z_row_val <= z_next_row;
                            z_cur_val <= z_next_row;
                            uw_row_val <= uw_next_row;
                            uw_cur_val <= uw_next_row;
                            vw_row_val <= vw_next_row;
                            vw_cur_val <= vw_next_row;
                            iw_row_val <= iw_next_row;
                            iw_cur_val <= iw_next_row;
                        end
                    end else begin
                        draw_x_cur <= draw_x_cur + 10'd2;
                        edge_cur_val[0] <= edge_next_pair0;
                        edge_cur_val[1] <= edge_next_pair1;
                        edge_cur_val[2] <= edge_next_pair2;
                        edge_cur_val[3] <= edge_next_pair3;
                        z_cur_val <= z_next_pair;
                        uw_cur_val <= uw_next_pair;
                        vw_cur_val <= vw_next_pair;
                        iw_cur_val <= iw_next_pair;
                    end
                end else if (state == ST_DRAW) begin
                    pipe0_valid <= 1'b0;
                    pipe0_valid_o <= 1'b0;
                    /*
                     * Software has already binned this descriptor into each
                     * overlapping band pass. Pixels outside the resident
                     * band are ignored here and will be drawn during their
                     * own BEGIN_BAND/END_BAND pass.
                     */
                    /* Early scanline exit for cache-miss path too */
                    if (draw_pair_edge_inside)
                        draw_row_inside <= 1'b1;

                    if (draw_pair_last || draw_pair_exited) begin
                        if (draw_y_cur == draw_y_max) begin
                            state <= ST_DRAW_FLUSH;
                            draw_flush_count <= DRAW_FLUSH_CYCLES;
                        end else begin
                            draw_row_base <= draw_row_base + 16'd640;
                            draw_x_cur <= draw_x_start_even;
```

```
                draw_y_cur <= draw_y_cur + 9'd1;
                draw_row_inside <= 1'b0;
                edge_row_val[0] <= edge_next_row0;
                edge_cur_val[0] <= edge_next_row0;
                edge_row_val[1] <= edge_next_row1;
                edge_cur_val[1] <= edge_next_row1;
                edge_row_val[2] <= edge_next_row2;
                edge_cur_val[2] <= edge_next_row2;
                edge_row_val[3] <= edge_next_row3;
                edge_cur_val[3] <= edge_next_row3;
                z_row_val <= z_next_row;
                z_cur_val <= z_next_row;
                uw_row_val <= uw_next_row;
                uw_cur_val <= uw_next_row;
                vw_row_val <= vw_next_row;
                vw_cur_val <= vw_next_row;
                iw_row_val <= iw_next_row;
                iw_cur_val <= iw_next_row;
            end
        end else begin
            draw_x_cur <= draw_x_cur + 10'd2;
            edge_cur_val[0] <= edge_next_pair0;
            edge_cur_val[1] <= edge_next_pair1;
            edge_cur_val[2] <= edge_next_pair2;
            edge_cur_val[3] <= edge_next_pair3;
            z_cur_val <= z_next_pair;
            uw_cur_val <= uw_next_pair;
            vw_cur_val <= vw_next_pair;
            iw_cur_val <= iw_next_pair;
        end
    end else begin
        pipe0_valid <= 1'b0;
        pipe0_inside <= 1'b0;
        pipe0_ztest <= 1'b0;
        pipe0_textured <= 1'b0;
        pipe0_alpha_key <= 1'b0;
        pipe0_alpha <= 2'd0;
        pipe0_fog <= 1'b0;
        pipe0_light_bank <= 2'd0;
        pipe0_tex_or_color <= 8'd0;
        pipe0_addr <= 16'd0;
        pipe0_z <= 16'd0;
        pipe0_x <= 10'd0;
        pipe0_y <= 9'd0;
        pipe0_uw_q <= 32'sd0;
        pipe0_vw_q <= 32'sd0;
        pipe0_iw_q <= 32'd0;
        pipe0_valid_o <= 1'b0;
        pipe0_inside_o <= 1'b0;
        pipe0_ztest_o <= 1'b0;
        pipe0_textured_o <= 1'b0;
        pipe0_alpha_key_o <= 1'b0;
        pipe0_alpha_o <= 2'd0;
        pipe0_fog_o <= 1'b0;
        pipe0_light_bank_o <= 2'd0;
        pipe0_tex_or_color_o <= 8'd0;
        pipe0_addr_o <= 16'd0;
        pipe0_z_o <= 16'd0;
        pipe0_x_o <= 10'd0;
```

```
                    pipe0_y_o <= 9'd0;
                    pipe0_uw_q_o <= 32'sd0;
                    pipe0_vw_q_o <= 32'sd0;
                    pipe0_iw_q_o <= 32'd0;

                    if (draw_flush_count == 4'd1) begin
                        draw_flush_count <= 4'd0;
                        if (prefetch_valid) begin
                            for (ei = 0; ei < MAX_DESC_WORDS; ei = ei + 1)
                                desc_words[ei] <= prefetch_words[ei];
                            fetch_count <= prefetch_target_words;
                            prefetch_valid <= 1'b0;
                            state <= ST_FETCH;
                        end else if (prefetch_active) begin
                            draw_flush_count <= 4'd1;
                            state <= ST_DRAW_FLUSH;
                        end else begin
                            state <= ST_IDLE;
                        end
                    end else begin
                        draw_flush_count <= draw_flush_count - 4'd1;
                    end
                end
            end

            ST_CACHE_INIT: begin
                if (cache_maint_addr == cache_window_end - 16'd2) begin
                    cache_valid <= 1'b1;
                    cache_dirty <= 1'b1;
                    cache_draw_dirty <= 1'b0;
                    cache_band_index <= cache_target_band;
                    cache_maint_addr <= cache_window_start;
                    state <= ST_IDLE;
                end else begin
                    cache_maint_addr <= cache_maint_addr + 16'd2;
                end
            end

            default: state <= ST_IDLE;
        endcase

        /*
         * CLR is a frame-level abort/restart request from the driver. Do
         * not wait for ST_IDLE: cache maintenance can legitimately take a
         * long time at 640x480, and if a previous frame wedges we need the
         * next CLEAR_FRAME ioctl to recover the engine instead of timing
         * out behind stale BUSY.
         */
        if (ctrl_clear_write) begin
            state <= ST_IDLE;
            clear_pending <= 1'b1;
            ctrl_flp_pending <= 1'b0;
            copy_complete_pending <= 1'b0;
            copy_target_sel <= ~display_sel;
            cache_valid <= 1'b0;
            cache_dirty <= 1'b0;
            cache_draw_dirty <= 1'b0;
            cache_flush_y_min <= 6'd0;
            cache_window_start <= 16'd0;
```

```
                cache_window_pixels <= 16'd38400;
                cache_window_end <= 16'd38400;
                band_begin_pending <= 1'b0;
                band_flush_pending <= 1'b0;
                draw_cache_sel <= 1'b0;
                /* CLEAR_FRAME must not kill an in-flight background flush. */
                cache_maint_addr <= 16'd0;
                scan_late_count <= 16'd0;
                /* Leave scanout RD state alone; CLEAR_FRAME can arrive mid-line. */
                fifo_wr_ptr <= 10'd0;
                fifo_rd_ptr <= 10'd0;
                fifo_count <= 11'd0;
                fetch_count <= 6'd0;
                prefetch_count <= 6'd0;
                prefetch_target_words <= BASE_QUAD_WORDS_6;
                prefetch_active <= 1'b0;
                prefetch_valid <= 1'b0;
                draw_flush_count <= 4'd0;
                pipe0_valid <= 1'b0;
                recip0_valid <= 1'b0;
                recip1_valid <= 1'b0;
                recip2_valid <= 1'b0;
                pipe1_valid <= 1'b0;
                tex0_valid <= 1'b0;
                pipe2_valid <= 1'b0;
                draw_pipe_valid <= 1'b0;
                pal_rd_valid <= 1'b0;
                plr_valid <= 1'b0;
                fog0_valid <= 1'b0;
                fog1_valid <= 1'b0;
                commit_valid <= 1'b0;
                pipe0_valid_o <= 1'b0;
                recip0_valid_o <= 1'b0;
                recip1_valid_o <= 1'b0;
                recip2_valid_o <= 1'b0;
                pipe1_valid_o <= 1'b0;
                tex0_valid_o <= 1'b0;
                pipe2_valid_o <= 1'b0;
                draw_pipe_valid_o <= 1'b0;
                pal_rd_valid_o <= 1'b0;
                plr_valid_o <= 1'b0;
                fog0_valid_o <= 1'b0;
                fog1_valid_o <= 1'b0;
                commit_valid_o <= 1'b0;
            end

            extmem_dma_status <= {
                scan_late_count,
                cache_band_index,
                display_sel,
                copy_target_sel,
                copy_complete_pending,
                cache_init_state,
                flush_active,
                scan_fill_active,
                display_valid,
                sdram_ready,
                scan_active_bank[0],
                scan_fill_bank[0],
```

```
                scan_line2_ready,
                scan_line1_ready,
                scan_line0_ready
            };
        end
    end

    wire perf_flip_write = wr && (address == ADDR_CONTROL) && writedata[1];
    wire perf_in_draw    = (state == ST_DRAW) || (state == ST_DRAW_FLUSH);
    wire perf_draw_commit = commit_valid || commit_valid_o;
    wire perf_flush_push = bg_flush_wr_push;

    /*
     * Per-frame perf counters accumulate while a frame is in flight and reset
     * on the FLIP write that ends the frame. Counter categories can overlap
     * because background flush, cache init, and draw can run concurrently.
     */
    voxel_perf_counters #(
        .COPY_WR_FIFO_HIGH_WATER(COPY_WR_FIFO_HIGH_WATER)
    ) perf_counters (
        .clk                    (clk),
        .reset                  (reset),
        .flip_write             (perf_flip_write),
        .in_draw                (perf_in_draw),
        .draw_commit            (perf_draw_commit),
        .in_cache_init          (cache_init_state),
        .flush_active           (flush_active),
        .flush_push             (perf_flush_push),
        .flush_load_pending     (flush_load_pending),
        .sdram_wr_full          (sdram_wr_full),
        .sdram_wr_use           (sdram_wr_use),
        .flush_words_done       (flush_words_done),
        .flush_pixels_total     (flush_pixels_total),
        .flush_word_pending_valid (flush_word_pending_valid),
        .flush_fetch_inflight   (flush_fetch_inflight),
        .perf_draw_active       (perf_draw_active),
        .perf_draw_idle         (perf_draw_idle),
        .perf_flush_active      (perf_flush_active),
        .perf_flush_stall       (perf_flush_stall),
        .perf_init              (perf_init),
        .perf_load              (perf_load),
        .perf_flush_wait_load   (perf_flush_wait_load),
        .perf_flush_wait_fifo   (perf_flush_wait_fifo),
        .perf_flush_wait_data   (perf_flush_wait_data),
        .perf_flush_wait_drain  (perf_flush_wait_drain)
    );

    always_comb begin
        case (address)
            ADDR_CONTROL : readdata = control_word;
            ADDR_STATUS  : readdata = status_word;
            ADDR_FRAMECNT: readdata = frame_count;
            ADDR_PAL_ADDR: readdata = {24'h0, pal_addr};
            ADDR_SKY_PAL_ADDR: readdata = {27'h0, sky_pal_addr};
            ADDR_FOG_RANGE: readdata = {fog_end_dist, fog_start_dist};
            ADDR_FOG_CTRL: readdata = {fog_inv_proj_sq, 7'h0, fog_enable, fog_color};
            ADDR_EXTMEM_CTRL: readdata = extmem_ctrl;
            ADDR_EXTMEM_FRONT: readdata = extmem_front_base;
            ADDR_EXTMEM_BACK: readdata = extmem_back_base;
```

```
            ADDR_EXTMEM_STRIDE: readdata = extmem_stride_bytes;
            ADDR_EXTMEM_TILE: readdata = extmem_tile_cfg;
            ADDR_EXTMEM_STAT: readdata = extmem_dma_status;
            ADDR_BAND_INDEX: readdata = {29'h0, band_index_cfg};
            ADDR_BAND_CTRL: readdata = {30'h0, band_flush_pending, band_begin_pending};
            ADDR_BAND_WINDOW: readdata = {18'd0, band_flush_y_max_cfg,
                                          2'd0, band_flush_y_min_cfg};
            ADDR_PERF_DRAW_ACT : readdata = perf_draw_active;
            ADDR_PERF_DRAW_IDLE: readdata = perf_draw_idle;
            ADDR_PERF_FLUSH_ACT: readdata = perf_flush_active;
            ADDR_PERF_FLUSH_STL: readdata = perf_flush_stall;
            ADDR_PERF_INIT     : readdata = perf_init;
            ADDR_PERF_LOAD     : readdata = perf_load;
            ADDR_PERF_FLUSH_LOAD : readdata = perf_flush_wait_load;
            ADDR_PERF_FLUSH_FIFO : readdata = perf_flush_wait_fifo;
            ADDR_PERF_FLUSH_DATA : readdata = perf_flush_wait_data;
            ADDR_PERF_FLUSH_DRAIN: readdata = perf_flush_wait_drain;
            default      : readdata = 32'h0;  /* palette readback not needed by driver */
        endcase
    end

    always_comb begin
        if (scan_visible_r) begin
            VGA_R = pixel_rgb[23:16];
            VGA_G = pixel_rgb[15:8];
            VGA_B = pixel_rgb[7:0];
        end else begin
            VGA_R = 8'h00;
            VGA_G = 8'h00;
            VGA_B = 8'h00;
        end
    end

endmodule
```

Complete `voxel_math`$_u$*tils.svRTLlisting*

Listing 4: Complete `voxel_math`$_u$*tils.svRTLsource*.

```
// Pure combinational math helpers used by voxel_gpu.sv.
//
// Keep tiny stateless utility modules together here so the RTL directory
// reflects real ownership boundaries without scattering one-screen files.

module voxel_raster_setup (
    input  logic        [9:0]  draw_x_start_even,
    input  logic        [8:0]  draw_y_min,
    input  logic        [9:0]  draw_x_min,
    input  logic        [15:0] draw_z0,
    input  logic signed [15:0] draw_dz_dx,
    input  logic signed [31:0] draw_uw_0,
    input  logic signed [31:0] draw_uw_dx,
    input  logic signed [31:0] draw_vw_0,
    input  logic signed [31:0] draw_vw_dx,
    input  logic signed [31:0] draw_iw_0,
    input  logic signed [31:0] draw_iw_dx,
    input  logic signed [31:0] edge_a0,
```

```
    input  logic signed [31:0] edge_a1,
    input  logic signed [31:0] edge_a2,
    input  logic signed [31:0] edge_a3,
    input  logic signed [31:0] edge_b0,
    input  logic signed [31:0] edge_b1,
    input  logic signed [31:0] edge_b2,
    input  logic signed [31:0] edge_b3,
    input  logic signed [31:0] edge_c0,
    input  logic signed [31:0] edge_c1,
    input  logic signed [31:0] edge_c2,
    input  logic signed [31:0] edge_c3,
    output logic signed [63:0] edge_eval0,
    output logic signed [63:0] edge_eval1,
    output logic signed [63:0] edge_eval2,
    output logic signed [63:0] edge_eval3,
    output logic signed [47:0] z_start_val,
    output logic signed [63:0] uw_start_val,
    output logic signed [63:0] vw_start_val,
    output logic signed [63:0] iw_start_val
);
    function automatic signed [63:0] sext32_to_s64(input logic signed [31:0] value);
        begin
            sext32_to_s64 = {{32{value[31]}}, value};
        end
    endfunction

    function automatic signed [47:0] sext16_to_s48(input logic signed [15:0] value);
        begin
            sext16_to_s48 = {{32{value[15]}}, value};
        end
    endfunction

    wire signed [10:0] setup_start_x = $signed({1'b0, draw_x_start_even});
    wire signed  [9:0] setup_start_y = $signed({1'b0, draw_y_min});

    wire signed [63:0] edge_ax0 = $signed(edge_a0) * setup_start_x;
    wire signed [63:0] edge_ax1 = $signed(edge_a1) * setup_start_x;
    wire signed [63:0] edge_ax2 = $signed(edge_a2) * setup_start_x;
    wire signed [63:0] edge_ax3 = $signed(edge_a3) * setup_start_x;
    wire signed [63:0] edge_by0 = $signed(edge_b0) * setup_start_y;
    wire signed [63:0] edge_by1 = $signed(edge_b1) * setup_start_y;
    wire signed [63:0] edge_by2 = $signed(edge_b2) * setup_start_y;
    wire signed [63:0] edge_by3 = $signed(edge_b3) * setup_start_y;

    wire signed [47:0] draw_dz_dx_ext = sext16_to_s48(draw_dz_dx);
    wire signed [63:0] draw_uw_dx_ext = sext32_to_s64(draw_uw_dx);
    wire signed [63:0] draw_vw_dx_ext = sext32_to_s64(draw_vw_dx);
    wire signed [63:0] draw_iw_dx_ext = sext32_to_s64(draw_iw_dx);

    always_comb begin
        edge_eval0 = edge_ax0 + edge_by0 + sext32_to_s64(edge_c0);
        edge_eval1 = edge_ax1 + edge_by1 + sext32_to_s64(edge_c1);
        edge_eval2 = edge_ax2 + edge_by2 + sext32_to_s64(edge_c2);
        edge_eval3 = edge_ax3 + edge_by3 + sext32_to_s64(edge_c3);

        z_start_val =
            $signed({32'd0, draw_z0}) -
            (draw_x_min[0] ? draw_dz_dx_ext : 48'sd0);
        uw_start_val =
```

```
            sext32_to_s64(draw_uw_0) -
            (draw_x_min[0] ? draw_uw_dx_ext : 64'sd0);
        vw_start_val =
            sext32_to_s64(draw_vw_0) -
            (draw_x_min[0] ? draw_vw_dx_ext : 64'sd0);
        iw_start_val =
            sext32_to_s64(draw_iw_0) -
            (draw_x_min[0] ? draw_iw_dx_ext : 64'sd0);
    end
endmodule

module voxel_draw_step (
    input  logic signed [63:0] edge_cur0,
    input  logic signed [63:0] edge_cur1,
    input  logic signed [63:0] edge_cur2,
    input  logic signed [63:0] edge_cur3,
    input  logic signed [63:0] edge_row0,
    input  logic signed [63:0] edge_row1,
    input  logic signed [63:0] edge_row2,
    input  logic signed [63:0] edge_row3,
    input  logic signed [31:0] edge_a0,
    input  logic signed [31:0] edge_a1,
    input  logic signed [31:0] edge_a2,
    input  logic signed [31:0] edge_a3,
    input  logic signed [31:0] edge_b0,
    input  logic signed [31:0] edge_b1,
    input  logic signed [31:0] edge_b2,
    input  logic signed [31:0] edge_b3,
    input  logic signed [47:0] z_cur,
    input  logic signed [47:0] z_row,
    input  logic signed [15:0] dz_dx,
    input  logic signed [15:0] dz_dy,
    input  logic signed [63:0] uw_cur,
    input  logic signed [63:0] uw_row,
    input  logic signed [31:0] uw_dx,
    input  logic signed [31:0] uw_dy,
    input  logic signed [63:0] vw_cur,
    input  logic signed [63:0] vw_row,
    input  logic signed [31:0] vw_dx,
    input  logic signed [31:0] vw_dy,
    input  logic signed [63:0] iw_cur,
    input  logic signed [63:0] iw_row,
    input  logic signed [31:0] iw_dx,
    input  logic signed [31:0] iw_dy,
    output logic signed [63:0] edge_lane1_0,
    output logic signed [63:0] edge_lane1_1,
    output logic signed [63:0] edge_lane1_2,
    output logic signed [63:0] edge_lane1_3,
    output logic signed [63:0] edge_next_pair0,
    output logic signed [63:0] edge_next_pair1,
    output logic signed [63:0] edge_next_pair2,
    output logic signed [63:0] edge_next_pair3,
    output logic signed [63:0] edge_next_row0,
    output logic signed [63:0] edge_next_row1,
    output logic signed [63:0] edge_next_row2,
    output logic signed [63:0] edge_next_row3,
    output logic signed [47:0] z_lane1,
    output logic signed [47:0] z_next_pair,
    output logic signed [47:0] z_next_row,
```

```
    output logic signed [63:0] uw_lane1,
    output logic signed [63:0] uw_next_pair,
    output logic signed [63:0] uw_next_row,
    output logic signed [63:0] vw_lane1,
    output logic signed [63:0] vw_next_pair,
    output logic signed [63:0] vw_next_row,
    output logic signed [63:0] iw_lane1,
    output logic signed [63:0] iw_next_pair,
    output logic signed [63:0] iw_next_row
);
    function automatic signed [63:0] sext32_to_s64(input logic signed [31:0] value);
        begin
            sext32_to_s64 = {{32{value[31]}}, value};
        end
    endfunction

    function automatic signed [47:0] sext16_to_s48(input logic signed [15:0] value);
        begin
            sext16_to_s48 = {{32{value[15]}}, value};
        end
    endfunction

    wire signed [63:0] edge_a0_ext = sext32_to_s64(edge_a0);
    wire signed [63:0] edge_a1_ext = sext32_to_s64(edge_a1);
    wire signed [63:0] edge_a2_ext = sext32_to_s64(edge_a2);
    wire signed [63:0] edge_a3_ext = sext32_to_s64(edge_a3);
    wire signed [63:0] edge_b0_ext = sext32_to_s64(edge_b0);
    wire signed [63:0] edge_b1_ext = sext32_to_s64(edge_b1);
    wire signed [63:0] edge_b2_ext = sext32_to_s64(edge_b2);
    wire signed [63:0] edge_b3_ext = sext32_to_s64(edge_b3);
    wire signed [47:0] dz_dx_ext = sext16_to_s48(dz_dx);
    wire signed [47:0] dz_dy_ext = sext16_to_s48(dz_dy);
    wire signed [63:0] uw_dx_ext = sext32_to_s64(uw_dx);
    wire signed [63:0] uw_dy_ext = sext32_to_s64(uw_dy);
    wire signed [63:0] vw_dx_ext = sext32_to_s64(vw_dx);
    wire signed [63:0] vw_dy_ext = sext32_to_s64(vw_dy);
    wire signed [63:0] iw_dx_ext = sext32_to_s64(iw_dx);
    wire signed [63:0] iw_dy_ext = sext32_to_s64(iw_dy);

    always_comb begin
        edge_lane1_0 = edge_cur0 + edge_a0_ext;
        edge_lane1_1 = edge_cur1 + edge_a1_ext;
        edge_lane1_2 = edge_cur2 + edge_a2_ext;
        edge_lane1_3 = edge_cur3 + edge_a3_ext;

        edge_next_pair0 = edge_cur0 + edge_a0_ext + edge_a0_ext;
        edge_next_pair1 = edge_cur1 + edge_a1_ext + edge_a1_ext;
        edge_next_pair2 = edge_cur2 + edge_a2_ext + edge_a2_ext;
        edge_next_pair3 = edge_cur3 + edge_a3_ext + edge_a3_ext;

        edge_next_row0 = edge_row0 + edge_b0_ext;
        edge_next_row1 = edge_row1 + edge_b1_ext;
        edge_next_row2 = edge_row2 + edge_b2_ext;
        edge_next_row3 = edge_row3 + edge_b3_ext;

        z_lane1 = z_cur + dz_dx_ext;
        z_next_pair = z_cur + dz_dx_ext + dz_dx_ext;
        z_next_row = z_row + dz_dy_ext;

```

```
        uw_lane1 = uw_cur + uw_dx_ext;
        uw_next_pair = uw_cur + uw_dx_ext + uw_dx_ext;
        uw_next_row = uw_row + uw_dy_ext;

        vw_lane1 = vw_cur + vw_dx_ext;
        vw_next_pair = vw_cur + vw_dx_ext + vw_dx_ext;
        vw_next_row = vw_row + vw_dy_ext;

        iw_lane1 = iw_cur + iw_dx_ext;
        iw_next_pair = iw_cur + iw_dx_ext + iw_dx_ext;
        iw_next_row = iw_row + iw_dy_ext;
    end
endmodule

module voxel_iw_normalize (
    input  logic [31:0] iw_q,
    output logic [5:0]  iw_msb,
    output logic [31:0] iw_norm_q
);
    function automatic [5:0] msb_index32(input logic [31:0] value);
        integer bit_idx;
        logic found;
        begin
            msb_index32 = 6'd0;
            found = 1'b0;
            for (bit_idx = 31; bit_idx >= 0; bit_idx = bit_idx - 1) begin
                if (!found && value[bit_idx]) begin
                    msb_index32 = bit_idx[5:0];
                    found = 1'b1;
                end
            end
        end
    endfunction

    wire [5:0] iw_msb_calc = msb_index32(iw_q);

    always_comb begin
        iw_msb = iw_msb_calc;
        iw_norm_q =
            (iw_q == 32'd0) ? 32'd0 :
            (iw_msb_calc >= 6'd16) ?
            (iw_q >> (iw_msb_calc - 6'd16)) :
            (iw_q << (6'd16 - iw_msb_calc));
    end
endmodule

module voxel_recip_interpolate (
    input  logic [31:0] w_norm_lo,
    input  logic [31:0] w_norm_hi,
    input  logic [5:0]  iw_lut_frac,
    output logic [31:0] w_norm_q
);
    wire [31:0] w_norm_delta = w_norm_lo - w_norm_hi;
    wire [37:0] w_interp_prod = w_norm_delta * iw_lut_frac;
    wire [37:0] w_interp_step_ext = (w_interp_prod + 38'd32) >> 6;
    wire [31:0] w_interp_step = w_interp_step_ext[31:0];

    assign w_norm_q = w_norm_lo - w_interp_step;
endmodule
```

```
module voxel_w_denormalize (
    input  logic        iw_zero,
    input  logic [5:0]  iw_msb,
    input  logic [31:0] w_norm_q,
    output logic [31:0] w_q
);
    always_comb begin
        w_q =
            iw_zero ? 32'd0 :
            (iw_msb >= 6'd16) ?
            (w_norm_q >> (iw_msb - 6'd16)) :
            (w_norm_q << (6'd16 - iw_msb));
    end
endmodule

module voxel_fog_blend (
    input  logic        fog_enable,
    input  logic        pixel_fog,
    input  logic [15:0] radial_q8_8,
    input  logic [15:0] fog_start_dist,
    input  logic [15:0] fog_end_dist,
    input  logic [15:0] src_rgb565,
    input  logic [15:0] dst_rgb565,
    input  logic [15:0] fog_rgb565,
    input  logic [1:0]  alpha,
    output logic [15:0] out_rgb565
);
`include "voxel_color_helpers.svh"

    wire fog_active =
        fog_enable && pixel_fog && (fog_end_dist > fog_start_dist) &&
        (radial_q8_8 > fog_start_dist);
    wire fog_full = fog_active && (radial_q8_8 >= fog_end_dist);
    wire [15:0] fog_range = fog_end_dist - fog_start_dist;
    wire [15:0] fog_q1 = fog_start_dist + {2'b00, fog_range[15:2]};
    wire [15:0] fog_q2 = fog_start_dist + {1'b0, fog_range[15:1]};
    wire [1:0] fog_alpha =
        !fog_active ? 2'd0 :
        fog_full    ? 2'd0 :
        (radial_q8_8 < fog_q1) ? 2'd1 :
        (radial_q8_8 < fog_q2) ? 2'd2 : 2'd3;
    wire [15:0] fog_blended = blend_rgb565(src_rgb565, fog_rgb565, fog_alpha);
    wire [15:0] fogged_rgb565 = fog_full ? fog_rgb565 : fog_blended;

    assign out_rgb565 = blend_rgb565(fogged_rgb565, dst_rgb565, alpha);
endmodule
```

Complete `voxel_raster`$_h$*elpers.svRTLlisting*

Listing 5: Complete `voxel_raster`$_h$*elpers.svRTLsource.*

```
    function automatic [9:0] clamp_x(input logic signed [15:0] value);
        begin
            if (value < 0)
                clamp_x = 10'd0;
            else if (value > 16'sd639)
```

```
            clamp_x = 10'd639;
        else
            clamp_x = value[9:0];
    end
endfunction

function automatic [8:0] clamp_y(input logic signed [15:0] value);
    begin
        if (value < 0)
            clamp_y = 9'd0;
        else if (value > 16'sd479)
            clamp_y = 9'd479;
        else
            clamp_y = value[8:0];
    end
endfunction

function automatic [15:0] band_local_addr(input logic [9:0] x,
                                          input logic [8:0] y,
                                          input logic [2:0] band);
    logic [8:0] band_base_y;
    logic [8:0] local_y;
    begin
        band_base_y = band_base_row(band);
        local_y = (y >= band_base_y) ? (y - band_base_y) : 9'd0;
        band_local_addr = local_y * 16'd640 + {6'd0, x};
    end
endfunction

function automatic [15:0] band_pixel_count(input logic [2:0] band);
    logic [9:0]  base_row;
    logic [9:0]  band_end;
    logic [15:0] visible_rows;
    begin
        base_row = {1'b0, band_base_row(band)};
        band_end = base_row + 10'd60;
        if (base_row >= 10'd480)
            band_pixel_count = 16'd0;
        else if (band_end > 10'd480) begin
            visible_rows = {6'd0, (10'd480 - base_row)};
            band_pixel_count = visible_rows * 16'd640;
        end else begin
            band_pixel_count = 16'd38400;
        end
    end
endfunction

function automatic [15:0] band_local_row_offset(input logic [5:0] local_y);
    begin
        band_local_row_offset = {local_y, 9'd0} + {3'd0, local_y, 7'd0};
    end
endfunction

function automatic [15:0] band_row_window_count(input logic [5:0] y_min,
                                                input logic [5:0] y_max);
    logic [6:0] rows;
    begin
        rows = {1'b0, y_max} - {1'b0, y_min} + 7'd1;
        band_row_window_count = {rows, 9'd0} + {2'd0, rows, 7'd0};
```

```
        end
    endfunction

    function automatic [24:0] band_word_count(input logic [2:0] band);
        begin
            band_word_count = {9'd0, band_pixel_count(band)};
        end
    endfunction

    function automatic [24:0] band_word_offset(input logic [2:0] band);
        begin
            band_word_offset = {7'd0, band, 15'd0} +
                               {10'd0, band, 12'd0} +
                               {12'd0, band, 10'd0} +
                               {13'd0, band, 9'd0};
        end
    endfunction

    function automatic [8:0] band_base_row(input logic [2:0] band);
        begin
            band_base_row = {band, 6'd0} - {4'd0, band, 2'd0};
        end
    endfunction

    function automatic [4:0] sky_clear_start_index(input logic [2:0] band);
        begin
            case (band)
                3'd0: sky_clear_start_index = 5'd0;
                3'd1: sky_clear_start_index = 5'd3;
                3'd2: sky_clear_start_index = 5'd6;
                3'd3: sky_clear_start_index = 5'd9;
                3'd4: sky_clear_start_index = 5'd12;
                3'd5: sky_clear_start_index = 5'd15;
                3'd6: sky_clear_start_index = 5'd18;
                default: sky_clear_start_index = 5'd21;
            endcase
        end
    endfunction

    function automatic [4:0] sky_row_count_for_local_y(input logic [5:0] local_y);
        logic [4:0] local_mod;
        begin
            local_mod = local_y[4:0];
            if (local_y >= 6'd40)
                sky_row_count_for_local_y = local_mod - 5'd8;
            else if (local_y >= 6'd32)
                sky_row_count_for_local_y = local_mod + 5'd12;
            else if (local_y >= 6'd20)
                sky_row_count_for_local_y = local_mod - 5'd20;
            else
                sky_row_count_for_local_y = local_mod;
        end
    endfunction

    function automatic [4:0] sky_clear_index_for_local_y(input logic [2:0] band,
                                                         input logic [5:0] local_y);
        logic [5:0] pal;
        begin
            pal = {1'b0, sky_clear_start_index(band)};
```

```
            if (local_y >= 6'd40)
                pal = pal + 6'd2;
            else if (local_y >= 6'd20)
                pal = pal + 6'd1;
            sky_clear_index_for_local_y =
                (pal >= SKY_GRADIENT_COLORS_6) ?
                SKY_GRADIENT_LAST_INDEX : pal[4:0];
        end
    endfunction

    function automatic [4:0] sky_palette_for_y(input logic [8:0] y);
        logic [2:0] band;
        logic [8:0] local_y_wide;
        logic [5:0] local_y;
        logic [6:0] row_sum;
        logic [1:0] pal_offset;
        logic [5:0] pal;
        begin
            band = y_to_band(y);
            local_y_wide = y - band_base_row(band);
            local_y = local_y_wide[5:0];
            row_sum = {1'b0, local_y};
            if (row_sum >= 7'd60)
                pal_offset = 2'd3;
            else if (row_sum >= 7'd40)
                pal_offset = 2'd2;
            else if (row_sum >= 7'd20)
                pal_offset = 2'd1;
            else
                pal_offset = 2'd0;

            pal = {1'b0, sky_clear_start_index(band)} +
                  {4'b0, pal_offset};
            sky_palette_for_y = (pal >= SKY_GRADIENT_COLORS_6) ?
                                SKY_GRADIENT_LAST_INDEX : pal[4:0];
        end
    endfunction

    function automatic [2:0] y_to_band(input logic [8:0] y);
        logic [9:0] y_adjusted;
        begin
            y_adjusted = {1'b0, y} + {5'd0, y[8:5], 1'b0} + 10'd2;
            y_to_band = y_adjusted[8:6];
        end
    endfunction

    function automatic [15:0] clamp_z(input logic signed [47:0] value);
        begin
            if (value < 0)
                clamp_z = 16'h0000;
            else if (value > 48'sd65534)
                clamp_z = Z_VALID_FAR;
            else
                clamp_z = value[15:0];
        end
    endfunction

    function automatic signed [31:0] clamp_s32(input logic signed [63:0] value);
        begin
```

```
            if (value > 64'sh7FFF_FFFF)
                clamp_s32 = 32'sh7FFF_FFFF;
            else if (value < -64'sh8000_0000)
                clamp_s32 = -32'sh8000_0000;
            else
                clamp_s32 = value[31:0];
        end
    endfunction

    function automatic [31:0] clamp_pos_u32(input logic signed [63:0] value);
        begin
            if (value <= 0)
                clamp_pos_u32 = 32'd0;
            else if (value > 64'sh7FFF_FFFF)
                clamp_pos_u32 = 32'h7FFF_FFFF;
            else
                clamp_pos_u32 = value[31:0];
        end
    endfunction

    function automatic [3:0] texture_coord(input logic signed [63:0] value,
                                           input logic repeat_uv);
        begin
            if (repeat_uv)
                texture_coord = value[35:32];
            else if (value <= 64'sd0)
                texture_coord = 4'd0;
            else if (value >= 64'sh0000_0010_0000_0000)
                texture_coord = 4'd15;
            else
                texture_coord = value[35:32];
        end
    endfunction
```