

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 | 3 |
| 2 | Overview | 4 |
| 2.1 | What the player sees | 4 |
| 2.2 | The hardware/software split | 6 |
| 2.3 | Memory architecture | 6 |
| 2.4 | What is in the source tree | 7 |
| 3 | Algorithms | 7 |
| 3.1 | Quadrilateral rasterization | 7 |
| 3.2 | Perspective-correct texturing | 8 |
| 3.3 | Z-buffering | 8 |
| 3.4 | Banded rendering | 8 |
| 3.5 | Lighting, fog, and alpha | 8 |
| 3.6 | Chunk streaming and meshing | 9 |
| 4 | Hardware Design | 10 |
| 4.1 | The voxel_gpu peripheral | 10 |
| 4.2 | Main FSM | 10 |
| 4.3 | Two-lane pixel pipeline | 11 |
| 4.4 | Band caches | 12 |
| 4.5 | Background flush | 13 |
| 4.6 | VGA scanout | 14 |
| 4.7 | Texture ROM and palette | 14 |
| 4.8 | Performance counters | 15 |
| 5 | Software Design | 15 |
| 5.1 | Linux kernel driver | 15 |
| 5.2 | Renderer | 15 |
| 5.3 | GPU transport | 16 |
| 5.4 | World, lighting, fluids, redstone | 17 |
| 5.5 | Virtual hardware | 18 |
| 5.6 | Game loop and input | 18 |

| | | |
|----------|--|-----------|
| 6 | Hardware/Software Interface | 18 |
| 6.1 | Register map | 18 |
| 6.2 | IOCTLs | 19 |
| 6.3 | Descriptor layout | 20 |
| 7 | Implementation Results | 20 |
| 7.1 | Resource utilization | 20 |
| 7.2 | Timing | 21 |
| 7.3 | Verification | 22 |
| 7.4 | Performance | 22 |
| 8 | Who Did What, Lessons Learned, Advice | 22 |
| 8.1 | Division of work | 22 |
| 8.2 | Lessons learned | 23 |
| 8.3 | Advice for future teams | 23 |
| 9 | File Listings | 23 |
| 9.1 | Source inventory | 23 |
| 9.2 | Selected listings | 26 |

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

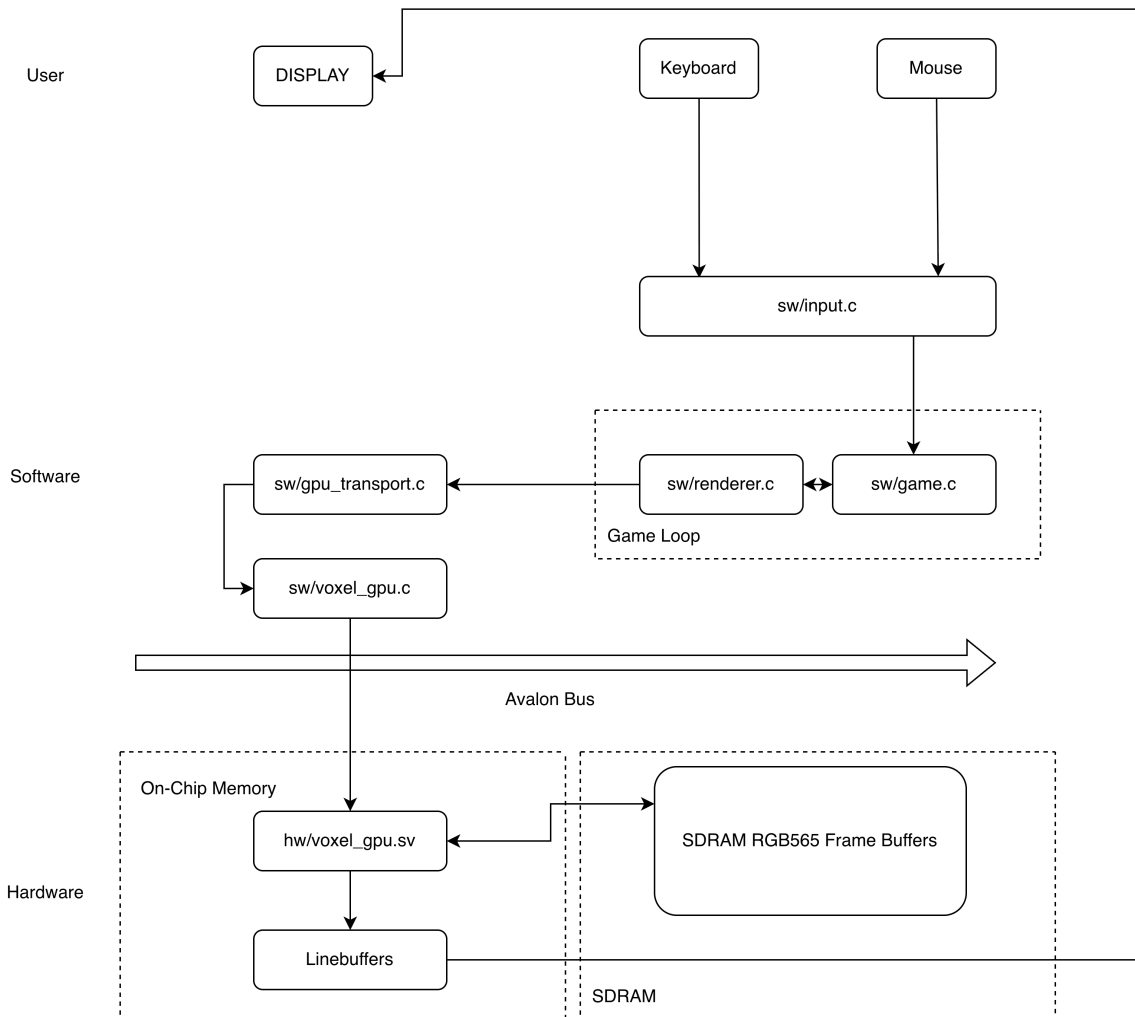


Figure 1: High-level system block diagram. The dashed regions separate the user, software, on-chip hardware, and SDRAM portions of the system. Game logic, world simulation, the renderer, and the kernel driver all live on the HPS. The FPGA receives a packed descriptor stream over the lightweight HPS-to-FPGA 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.

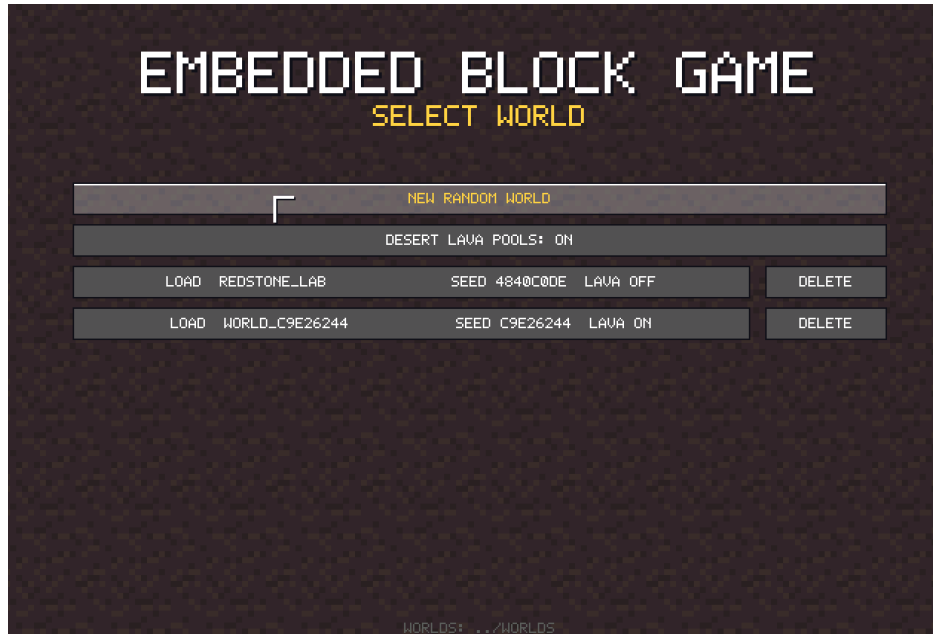


Figure 2: Block Game Main Menu

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 $320 \times 240 \times 8$ 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 $640 \times 60 \times 16$ 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:

$$\begin{aligned}(u/w)(x, y) &= (u/w)_0 + (u/w)_x \cdot \Delta x + (u/w)_y \cdot \Delta y \\(v/w)(x, y) &= (v/w)_0 + (v/w)_x \cdot \Delta x + (v/w)_y \cdot \Delta y \\(1/w)(x, y) &= (1/w)_0 + (1/w)_x \cdot \Delta x + (1/w)_y \cdot \Delta y\end{aligned}$$

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 $640 \times 60 \times 16$ 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 $y = 14$.

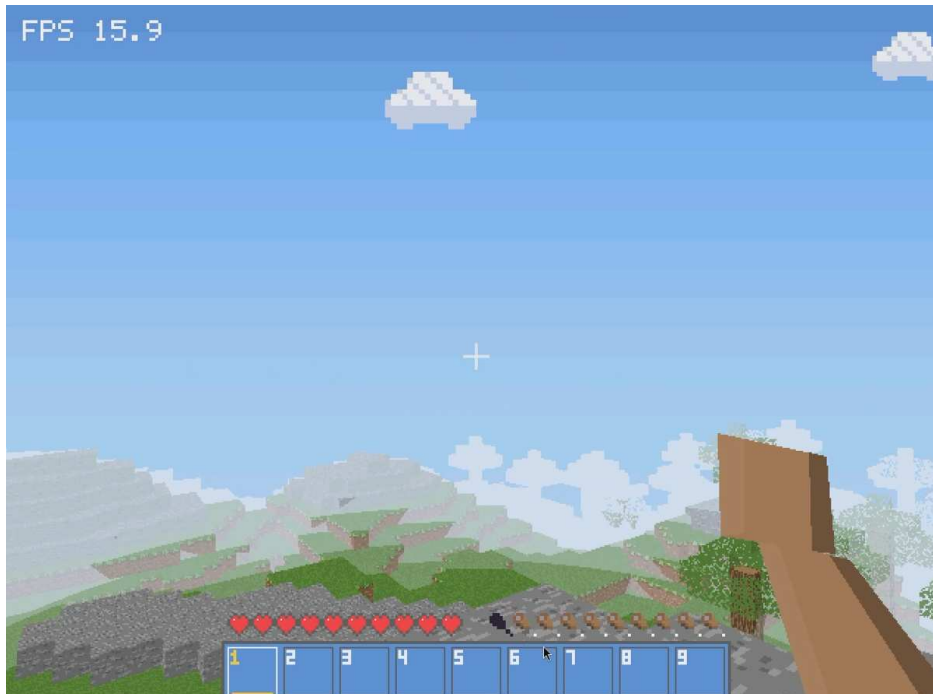


Figure 4: Randomly generated world terrain

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.

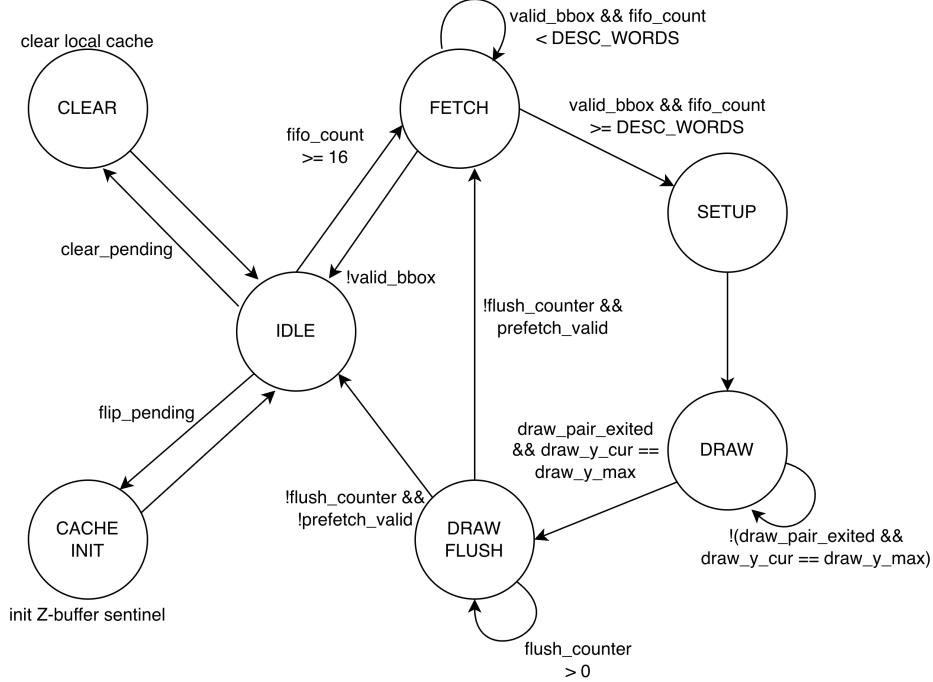


Figure 5: The voxel_gpu main FSM. `ST_IDLE` arbitrates among descriptor work, band maintenance, and clear/restart requests. `ST_FETCH` pulls 16 base words and (if the texture flag is set) 9 UV words from the FIFO into the descriptor scratch. `ST_SETUP` computes the starting edge, depth, and UV plane 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

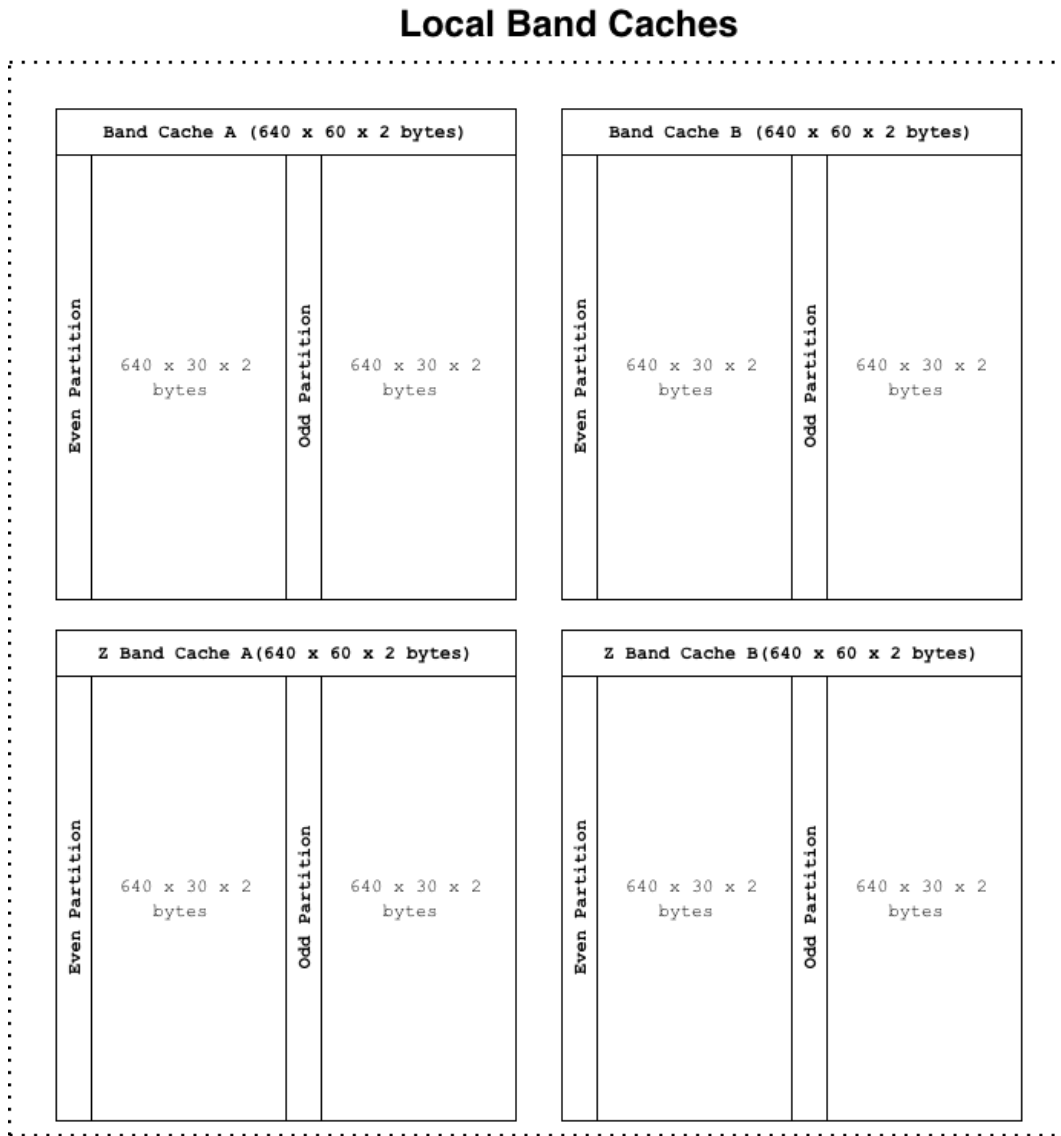


Figure 6: Local band caches. Two color caches (A/B) and two Z caches (A/B) each hold one 640×60 working set. The draw path writes to the active caches; the background flush controller reads from the inactive ones and pushes RGB565 words to SDRAM. Each cache is implemented as a banked simple dual-port 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 $640 \times 60 \times 16$ 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

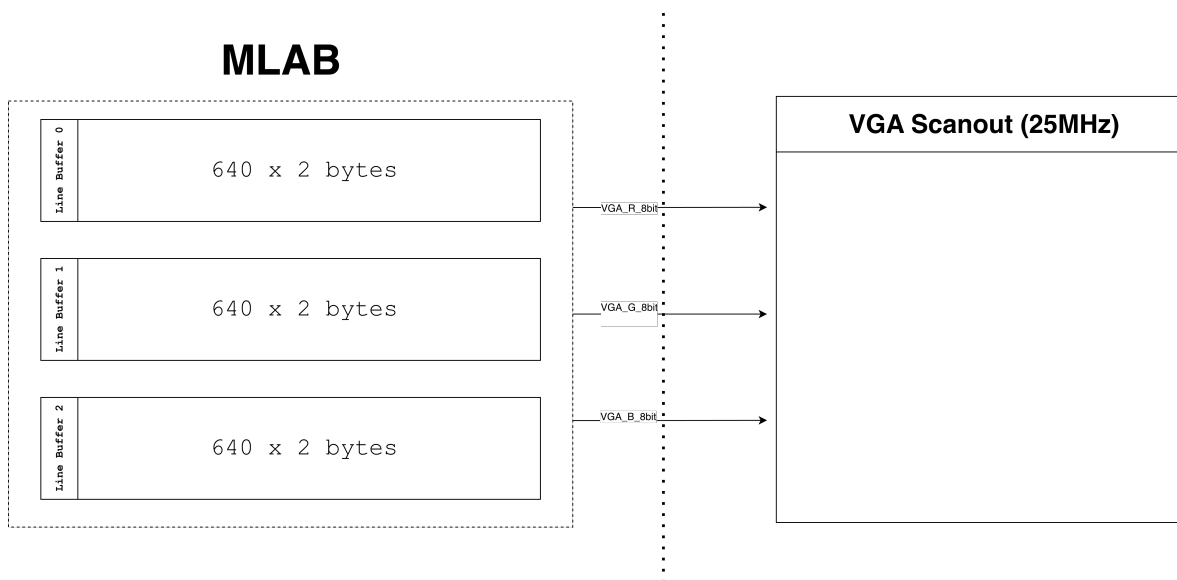


Figure 7: VGA scanout path. The 640×480 timing comes from the `voxel_vga_counters.sv` module, which divides the 50 MHz logic clock by 2 to get the 25 MHz VGA pixel clock (`VGA_CLK = ~hcount[0]`). Three line buffers triple-buffer the scanout: one is being displayed, one is being filled from SDRAM, one 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 $32 \mu\text{s}$), 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_reps` 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 `ioctl`s. `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 `ioctl`s 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 `ioctl`s 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_asserts` 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 <code>QUAD_FLAG_TEX</code> set |
| 19–21 | v/w plane | Q16.16, present if <code>QUAD_FLAG_TEX</code> set |
| 22–24 | 1/w plane | Q16.16, present if <code>QUAD_FLAG_TEX</code> 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 | <code>soc_system.fit.summary</code> |
| Flow status | Successful, 2026-05-12 17:22:27 | <code>soc_system.flow.rpt</code> |
| Logic utilization in ALMs | 28 687 / 32 070 (89%) | <code>fit.summary</code> |
| Total LABs partially or completely used | 3 207 / 3 207 (100%) | <code>fit.rpt</code> |
| Combinational ALUT usage for logic | 31 792 | <code>fit.rpt</code> |
| Total registers | 46 525 | <code>fit.summary</code> |
| Pins | 362 / 457 (79%) | <code>fit.summary</code> |
| Block memory bits | 3 079 296 / 4 065 280 (76%) | <code>fit.summary</code> |
| M10K blocks | 397 / 397 (100%) | <code>fit.rpt</code> |
| DSP blocks | 44 / 87 (51%) | <code>fit.summary</code> |
| PLLs and DLLs | 2 / 6 PLLs, 1 / 4 DLLs | <code>fit.summary</code> |

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 decisions 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.

```

1 /*
2  * voxel_gpu.c - MVP device driver for the FPGA voxel GPU peripheral.
3  *
4  * Responsibilities (per the MVP spec):
5  *   (A) Map the FPGA registers exposed through the lightweight HPS bridge.
6  *   (B) Expose /dev/voxel_gpu via the misc subsystem.
7  *   (C) Stream user-space command bytes into the on-chip FIFO via write().
8  *   (D) Provide the control ioctls: CLEAR_FRAME, FLIP, SET_PALETTE,
9  *       GET_STATUS, GET_FRAME_COUNT, SET_FOG, SET_EXTMEM, GET_EXTMEM,
10 *       BEGIN_BAND, END_BAND.
11 *   (E) Use polling on STATUS for synchronization (no interrupts).
12 *
13 * The driver is intentionally thin: it does not parse, validate, or
14 * schedule descriptors. It is a pass-through pipe between user space
15 * and the FIFO.
16 *
17 * Bind via the device tree compatible string "csee4840,voxel_gpu-1.0".
18 */
19
20 #include <linux/module.h>

```

```

21 #include <linux/init.h>
22 #include <linux/errno.h>
23 #include <linux/kernel.h>
24 #include <linux/platform_device.h>
25 #include <linux/miscdevice.h>
26 #include <linux/io.h>
27 #include <linux/of.h>
28 #include <linux/of_address.h>
29 #include <linux/fs.h>
30 #include <linux/uaccess.h>
31 #include <linux/slab.h>
32 #include <linux/mutex.h>
33 #include <linux/delay.h>
34 #include <linux/jiffies.h>
35 #include <linux/ktime.h>
36 #include <linux/math64.h>
37 #include <linux/string.h>
38 #include <linux/version.h>
39
40 #include "voxel_gpu.h"
41
42 #define DRIVER_NAME "voxel_gpu"
43
44 /* write() staging buffer; FIFO writes still back-pressure on STATUS.FIFO_COUNT. */
45 #define VOXEL_BOUNCE_WORDS 2048 /* 8 KB per chunk */
46 #define VOXEL_BOUNCE_BYTES (VOXEL_BOUNCE_WORDS * 4)
47
48 /* Generous polling timeout for a 60 Hz display path. */
49 #define VOXEL_POLL_TIMEOUT_MS 250
50 #define VOXEL_POLL_DELAY_US 1
51 #define VOXEL_FIFO_MIN_BURST_WORDS 64
52
53 static bool voxel_diag_upload;
54 module_param_named(diag_upload, voxel_diag_upload, bool, 0644);
55 MODULE_PARM_DESC(diag_upload,
56 "Print descriptor upload timing split: FIFO-space wait vs MMIO writes");
57
58 struct voxel_gpu_dev {
59     struct resource res;
60     void __iomem *base;
61     struct mutex lock; /* serializes register access */
62     u32 *bounce; /* reusable write() staging buffer */
63 };
64
65 static struct voxel_gpu_dev voxdev;
66
67 struct voxel_upload_diag {
68     u64 bytes;
69     u64 calls;
70     u64 bursts;
71     u64 wait_ns;
72     u64 mmio_ns;
73 };
74
75 static struct voxel_upload_diag upload_diag_accum;
76 static unsigned long upload_diag_next_report;
77
78 /* ----- low-level register helpers ----- */
79

```

```

80 static inline u32 voxel_rd(u32 off)
81 {
82     return ioread32(voxdev.base + off);
83 }
84
85 static inline void voxel_wr(u32 off, u32 val)
86 {
87     iowrite32(val, voxdev.base + off);
88 }
89
90 static inline u32 voxel_status(void)
91 {
92     return voxel_rd(VOXEL_REG_STATUS);
93 }
94
95 static inline u32 voxel_fifo_count(void)
96 {
97     return (voxel_status() >> VOXEL_STAT_FIFO_SHIFT) &
98           VOXEL_STAT_FIFO_MASK;
99 }
100
101 /*
102  * Wait until (status & mask) == expect, polling every VOXEL_POLL_DELAY_US.
103  * Returns 0 on success, -ETIMEDOUT if the condition never holds.
104  */
105 static int voxel_poll_status(u32 mask, u32 expect, unsigned int timeout_ms)
106 {
107     unsigned long deadline = jiffies + msecs_to_jiffies(timeout_ms);
108
109     while (time_before(jiffies, deadline)) {
110         if ((voxel_status() & mask) == expect)
111             return 0;
112         udelay(VOXEL_POLL_DELAY_US);
113         cond_resched();
114     }
115     return ((voxel_status() & mask) == expect) ? 0 : -ETIMEDOUT;
116 }
117
118 /* ----- FIFO streaming ----- */
119
120 static int voxel_fifo_wait_space(size_t *space_words, size_t min_req)
121 {
122     unsigned long deadline = jiffies + msecs_to_jiffies(VOXEL_POLL_TIMEOUT_MS);
123     unsigned int short_waits = 0;
124
125     for (;;) {
126         u32 used = voxel_fifo_count();
127         u32 space = used < VOXEL_FIFO_WORDS ? VOXEL_FIFO_WORDS - used : 0;
128
129         if (space >= min_req) {
130             *space_words = space;
131             return 0;
132         }
133         if (space > 0 && ++short_waits >= 8) {
134             *space_words = space;
135             return 0;
136         }
137         if (time_after(jiffies, deadline))
138             return -ETIMEDOUT;

```

```

139         usleep_range(2, 5);
140     }
141 }
142
143 static void voxel_diag_record_upload(size_t bytes, u64 wait_ns, u64 mmio_ns,
144                                     u64 bursts)
145 {
146     u64 total_ns;
147     u64 total_kib_s;
148     u64 mmio_kib_s;
149     u64 wait_pct;
150     u64 mmio_pct;
151
152     if (!voxel_diag_upload)
153         return;
154
155     upload_diag_accum.bytes += bytes;
156     upload_diag_accum.calls++;
157     upload_diag_accum.bursts += bursts;
158     upload_diag_accum.wait_ns += wait_ns;
159     upload_diag_accum.mmio_ns += mmio_ns;
160
161     if (!upload_diag_next_report ||
162         time_after_eq(jiffies, upload_diag_next_report)) {
163         total_ns = upload_diag_accum.wait_ns + upload_diag_accum.mmio_ns;
164         total_kib_s = total_ns ?
165             div64_u64(upload_diag_accum.bytes * 1000000000ULL,
166                       total_ns * 1024ULL) : 0;
167         mmio_kib_s = upload_diag_accum.mmio_ns ?
168             div64_u64(upload_diag_accum.bytes * 1000000000ULL,
169                       upload_diag_accum.mmio_ns * 1024ULL) : 0;
170         wait_pct = total_ns ?
171             div64_u64(upload_diag_accum.wait_ns * 100ULL,
172                       total_ns) : 0;
173         mmio_pct = total_ns ?
174             div64_u64(upload_diag_accum.mmio_ns * 100ULL,
175                       total_ns) : 0;
176
177         pr_info(DRIVER_NAME
178                ": upload diag: calls=%llu bytes=%llu bursts=%llu "
179                "wait=%lluus(%llu%) mmio=%lluus(%llu%) "
180                "rate_total=%lluKiB/s rate_mmio=%lluKiB/s\n",
181                upload_diag_accum.calls,
182                upload_diag_accum.bytes,
183                upload_diag_accum.bursts,
184                div64_u64(upload_diag_accum.wait_ns, 1000),
185                wait_pct,
186                div64_u64(upload_diag_accum.mmio_ns, 1000),
187                mmio_pct,
188                total_kib_s,
189                mmio_kib_s);
190
191         memset(&upload_diag_accum, 0, sizeof(upload_diag_accum));
192         upload_diag_next_report = jiffies + msecs_to_jiffies(1000);
193     }
194 }
195
196 /* ----- file ops ----- */
197

```

```

198 static int voxel_open(struct inode *inode, struct file *filp)
199 {
200     filp->private_data = &voxdev;
201     return 0;
202 }
203
204 static int voxel_release(struct inode *inode, struct file *filp)
205 {
206     return 0;
207 }
208
209 /*
210  * write(): treat the byte stream as a packed sequence of 32-bit FIFO
211  * words. The driver does not interpret descriptor boundaries; user
212  * space is responsible for handing us properly-aligned, multiple-of-4
213  * payloads (typically whole quads = 16 words = 64 bytes).
214  */
215 static ssize_t voxel_write(struct file *filp, const char __user *buf,
216                          size_t count, loff_t *ppos)
217 {
218     u32 *bounce;
219     size_t total = 0;
220     int ret = 0;
221
222     if (count == 0)
223         return 0;
224     if (count & 0x3)
225         return -EINVAL;          /* must be 32-bit aligned */
226
227     bounce = voxdev.bounce;
228     if (!bounce)
229         return -ENODEV;
230
231     if (mutex_lock_interruptible(&voxdev.lock)) {
232         return -ERESTARTSYS;
233     }
234
235     while (total < count) {
236         size_t chunk = min_t(size_t, count - total, VOXEL_BOUNCE_BYTES);
237         size_t words;
238         size_t written_words = 0;
239         u64 write_wait_ns = 0;
240         u64 write_mmio_ns = 0;
241         u64 write_bursts = 0;
242
243         if (copy_from_user(bounce, buf + total, chunk)) {
244             ret = -EFAULT;
245             break;
246         }
247
248         words = chunk / 4;
249         while (written_words < words) {
250             size_t space_words;
251             size_t burst_words;
252             size_t rem_words = words - written_words;
253             size_t min_req = min_t(size_t, VOXEL_FIFO_MIN_BURST_WORDS,
254                                   rem_words);
255             u64 t0;

```

```

257         t0 = ktime_get_ns();
258         ret = voxel_fifo_wait_space(&space_words, min_req);
259         write_wait_ns += ktime_get_ns() - t0;
260         if (ret)
261             goto out;
262
263         burst_words = min(space_words, rem_words);
264         t0 = ktime_get_ns();
265         iowrite32_rep(voxdev.base + VOXEL_FIFO_BASE,
266                     &bounce[written_words],
267                     burst_words);
268         write_mmio_ns += ktime_get_ns() - t0;
269         write_bursts++;
270
271         written_words += burst_words;
272     }
273     total += chunk;
274     voxel_diag_record_upload(chunk, write_wait_ns, write_mmio_ns,
275                             write_bursts);
276 }
277
278 out:
279     mutex_unlock(&voxdev.lock);
280
281     if (total > 0)
282         return total;
283     return ret;
284 }
285
286 /* ----- ioctl handlers ----- */
287
288 static long voxel_ioc_clear(void)
289 {
290     u32 ctrl;
291     int ret;
292
293     mutex_lock(&voxdev.lock);
294
295     /* Pulse CLR with EN held high; hardware self-clears CLR. */
296     ctrl = voxel_rd(VOXEL_REG_CONTROL) | VOXEL_CTRL_EN | VOXEL_CTRL_CLR;
297     voxel_wr(VOXEL_REG_CONTROL, ctrl);
298
299     ret = voxel_poll_status(VOXEL_STAT_BSY, 0, VOXEL_POLL_TIMEOUT_MS);
300
301     mutex_unlock(&voxdev.lock);
302     return ret;
303 }
304
305 static long voxel_ioc_flip_common(bool wait_for_vsync)
306 {
307     u32 ctrl;
308     int ret;
309
310     mutex_lock(&voxdev.lock);
311
312     /*
313      * Drain the FIFO before requesting a flip -- the rasterizer must
314      * have consumed all pending quads before we swap buffers.
315      */

```

```

316     ret = voxel_poll_status(VOXEL_STAT_FEM, VOXEL_STAT_FEM,
317                             VOXEL_POLL_TIMEOUT_MS);
318     if (ret)
319         goto out;
320
321     ret = voxel_poll_status(VOXEL_STAT_BSY, 0, VOXEL_POLL_TIMEOUT_MS);
322     if (ret)
323         goto out;
324
325     ctrl = voxel_rd(VOXEL_REG_CONTROL) | VOXEL_CTRL_EN | VOXEL_CTRL_FLP;
326     voxel_wr(VOXEL_REG_CONTROL, ctrl);
327
328     if (!wait_for_vsync)
329         goto out;
330
331     /* Block until the next vsync pulse latches. */
332     ret = voxel_poll_status(VOXEL_STAT_VSY, VOXEL_STAT_VSY,
333                             VOXEL_POLL_TIMEOUT_MS);
334
335 out:
336     mutex_unlock(&voxdev.lock);
337     return ret;
338 }
339
340 static long voxel_ioc_flip(void)
341 {
342     return voxel_ioc_flip_common(true);
343 }
344
345 static long voxel_ioc_flip_async(void)
346 {
347     return voxel_ioc_flip_common(false);
348 }
349
350 static long voxel_ioc_wait_flip(void)
351 {
352     int ret;
353
354     mutex_lock(&voxdev.lock);
355     ret = voxel_poll_status(VOXEL_STAT_VSY, VOXEL_STAT_VSY,
356                             VOXEL_POLL_TIMEOUT_MS);
357     mutex_unlock(&voxdev.lock);
358     return ret;
359 }
360
361 static long voxel_ioc_begin_band(void __user *uarg)
362 {
363     struct voxel_band_state band;
364     int ret;
365
366     if (copy_from_user(&band, uarg, sizeof(band)))
367         return -EFAULT;
368     if (band.band_index > 7)
369         return -EINVAL;
370     if (band.flush_y_min >= VOXEL_BAND_CACHE_HEIGHT ||
371         band.flush_y_max >= VOXEL_BAND_CACHE_HEIGHT ||
372         band.flush_y_min > band.flush_y_max)
373         return -EINVAL;
374

```

```

375     mutex_lock(&voxdev.lock);
376
377     ret = voxel_poll_status(VOXEL_STAT_FEM, VOXEL_STAT_FEM,
378                           VOXEL_POLL_TIMEOUT_MS);
379     if (ret)
380         goto out;
381     ret = voxel_poll_status(VOXEL_STAT_BSY, 0, VOXEL_POLL_TIMEOUT_MS);
382     if (ret)
383         goto out;
384
385     voxel_wr(VOXEL_REG_BAND_INDEX, band.band_index);
386     voxel_wr(VOXEL_REG_BAND_WINDOW,
387             (band.flush_y_min & 0x3fu) |
388             ((band.flush_y_max & 0x3fu) << 8));
389     voxel_wr(VOXEL_REG_BAND_CTRL, VOXEL_BAND_CTRL_BEGIN);
390     /* Keep BEGIN_BAND pipelined; the readback orders CSR writes before write(). */
391     voxel_rd(VOXEL_REG_BAND_CTRL);
392
393 out:
394     mutex_unlock(&voxdev.lock);
395     return ret;
396 }
397
398 static long voxel_ioc_end_band(void)
399 {
400     int ret;
401
402     mutex_lock(&voxdev.lock);
403
404     ret = voxel_poll_status(VOXEL_STAT_FEM, VOXEL_STAT_FEM,
405                           VOXEL_POLL_TIMEOUT_MS);
406     if (ret)
407         goto out;
408     ret = voxel_poll_status(VOXEL_STAT_BSY, 0, VOXEL_POLL_TIMEOUT_MS);
409     if (ret)
410         goto out;
411
412     voxel_wr(VOXEL_REG_BAND_CTRL, VOXEL_BAND_CTRL_FLUSH);
413     /* The RTL orders this background flush before the next band begins. */
414
415 out:
416     mutex_unlock(&voxdev.lock);
417     return ret;
418 }
419
420 static long voxel_ioc_set_palette(void __user *uarg)
421 {
422     struct voxel_palette_entry e;
423     u32 data;
424
425     if (copy_from_user(&e, uarg, sizeof(e)))
426         return -EFAULT;
427
428     data = ((u32)e.r << 16) | ((u32)e.g << 8) | (u32)e.b;
429
430     mutex_lock(&voxdev.lock);
431     voxel_wr(VOXEL_REG_PALETTE_ADDR, e.index);
432     voxel_wr(VOXEL_REG_PALETTE_DATA, data);
433     mutex_unlock(&voxdev.lock);

```

```

434     return 0;
435 }
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) |

```

```

493         ((fog.enabled ? 1u : 0u) << 8) |
494         (u32)fog.color_index;
495
496     mutex_lock(&voxdev.lock);
497     voxel_wr(VOXEL_REG_FOG_RANGE, range_word);
498     voxel_wr(VOXEL_REG_FOG_CTRL, ctrl_word);
499     mutex_unlock(&voxdev.lock);
500
501     return 0;
502 }
503
504 static long voxel_ioc_set_extmem(void __user *uarg)
505 {
506     struct voxel_extmem_state ext;
507
508     if (copy_from_user(&ext, uarg, sizeof(ext)))
509         return -EFAULT;
510
511     mutex_lock(&voxdev.lock);
512     voxel_wr(VOXEL_REG_EXTMEM_CTRL, ext.ctrl);
513     voxel_wr(VOXEL_REG_EXTMEM_FRONT, ext.front_base);
514     voxel_wr(VOXEL_REG_EXTMEM_BACK, ext.back_base);
515     voxel_wr(VOXEL_REG_EXTMEM_STRIDE, ext.stride_bytes);
516     voxel_wr(VOXEL_REG_EXTMEM_TILE, ext.tile_cfg);
517     mutex_unlock(&voxdev.lock);
518
519     return 0;
520 }
521
522 static long voxel_ioc_get_extmem(void __user *uarg)
523 {
524     struct voxel_extmem_state ext;
525
526     mutex_lock(&voxdev.lock);
527     ext.ctrl = voxel_rd(VOXEL_REG_EXTMEM_CTRL);
528     ext.front_base = voxel_rd(VOXEL_REG_EXTMEM_FRONT);
529     ext.back_base = voxel_rd(VOXEL_REG_EXTMEM_BACK);
530     ext.stride_bytes = voxel_rd(VOXEL_REG_EXTMEM_STRIDE);
531     ext.tile_cfg = voxel_rd(VOXEL_REG_EXTMEM_TILE);
532     ext.dma_status = voxel_rd(VOXEL_REG_EXTMEM_STAT);
533     mutex_unlock(&voxdev.lock);
534
535     if (copy_to_user(uarg, &ext, sizeof(ext)))
536         return -EFAULT;
537     return 0;
538 }
539
540 static long voxel_ioc_get_perf(void __user *uarg)
541 {
542     struct voxel_perf_counters p;
543
544     mutex_lock(&voxdev.lock);
545     p.draw_active = voxel_rd(VOXEL_REG_PERF_DRAW_ACT);
546     p.draw_idle = voxel_rd(VOXEL_REG_PERF_DRAW_IDLE);
547     p.flush_active = voxel_rd(VOXEL_REG_PERF_FLUSH_ACT);
548     p.flush_stall = voxel_rd(VOXEL_REG_PERF_FLUSH_STL);
549     p.init = voxel_rd(VOXEL_REG_PERF_INIT);
550     p.load = voxel_rd(VOXEL_REG_PERF_LOAD);
551     mutex_unlock(&voxdev.lock);

```

```

552     if (copy_to_user(uarg, &p, sizeof(p)))
553         return -EFAULT;
554     return 0;
555 }
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:

```

```

611         return voxel_ioc_begin_band((void __user *)arg);
612 case VOXEL_IOC_END_BAND:
613         return voxel_ioc_end_band();
614 case VOXEL_IOC_GET_PERF:
615         return voxel_ioc_get_perf((void __user *)arg);
616 case VOXEL_IOC_GET_PERF2:
617         return voxel_ioc_get_perf2((void __user *)arg);
618 default:
619         return -ENOTTY;
620     }
621 }
622
623 static const struct file_operations voxel_fops = {
624     .owner          = THIS_MODULE,
625     .open           = voxel_open,
626     .release        = voxel_release,
627     .write          = voxel_write,
628     .unlocked_ioctl = voxel_ioctl,
629     .llseek         = noop_llseek,
630 };
631
632 static struct miscdevice voxel_miscdev = {
633     .minor = MISC_DYNAMIC_MINOR,
634     .name  = DRIVER_NAME,
635     .fops  = &voxel_fops,
636     .mode  = 0666,
637 };
638
639 /* ----- platform driver glue ----- */
640
641 static int voxel_probe(struct platform_device *pdev)
642 {
643     int ret;
644
645     mutex_init(&voxdev.lock);
646     voxdev.bounce = kmalloc(VOXEL_BOUNCE_BYTES, GFP_KERNEL);
647     if (!voxdev.bounce)
648         return -ENOMEM;
649
650     ret = misc_register(&voxel_miscdev);
651     if (ret) {
652         dev_err(&pdev->dev, "misc_register failed: %d\n", ret);
653         goto err_bounce;
654     }
655
656     ret = of_address_to_resource(pdev->dev.of_node, 0, &voxdev.res);
657     if (ret) {
658         dev_err(&pdev->dev, "of_address_to_resource failed\n");
659         ret = -ENOENT;
660         goto err_misc;
661     }
662
663     if (!request_mem_region(voxdev.res.start, resource_size(&voxdev.res),
664                             DRIVER_NAME)) {
665         dev_err(&pdev->dev, "request_mem_region failed\n");
666         ret = -EBUSY;
667         goto err_misc;
668     }
669

```

```

670     voxdev.base = of_iomap(pdev->dev.of_node, 0);
671     if (!voxdev.base) {
672         dev_err(&pdev->dev, "of_iomap failed\n");
673         ret = -ENOMEM;
674         goto err_release;
675     }
676
677     /* Bring the engine up: enable, no interrupts, no pending pulses. */
678     voxel_wr(VOXEL_REG_CONTROL, VOXEL_CTRL_EN);
679
680     dev_info(&pdev->dev,
681             "voxel_gpu probed @ %pa, %llu bytes; status=0x%08x\n",
682             &voxdev.res.start,
683             (unsigned long long)resource_size(&voxdev.res),
684             voxel_status());
685     return 0;
686
687 err_release:
688     release_mem_region(voxdev.res.start, resource_size(&voxdev.res));
689 err_misc:
690     misc_deregister(&voxel_miscdev);
691 err_bounce:
692     kfree(voxdev.bounce);
693     voxdev.bounce = NULL;
694     return ret;
695 }
696
697 #if LINUX_VERSION_CODE >= KERNEL_VERSION(6, 11, 0)
698 static void voxel_remove(struct platform_device *pdev)
699 #else
700 static int voxel_remove(struct platform_device *pdev)
701 #endif
702 {
703     (void)pdev;
704
705     /* Disable the engine before tearing down. */
706     voxel_wr(VOXEL_REG_CONTROL, 0);
707
708     iounmap(voxdev.base);
709     release_mem_region(voxdev.res.start, resource_size(&voxdev.res));
710     misc_deregister(&voxel_miscdev);
711     kfree(voxdev.bounce);
712     voxdev.bounce = NULL;
713
714 #if LINUX_VERSION_CODE < KERNEL_VERSION(6, 11, 0)
715     return 0;
716 #endif
717 }
718
719 #ifndef CONFIG_OF
720 static const struct of_device_id voxel_of_match[] = {
721     { .compatible = "csee4840,voxel_gpu-1.0" },
722     {}},
723 };
724 MODULE_DEVICE_TABLE(of, voxel_of_match);
725 #endif
726
727 static struct platform_driver voxel_driver = {
728     .driver = {

```

```

729         .name          = DRIVER_NAME,
730         .owner         = THIS_MODULE,
731         .of_match_table = of_match_ptr(voxel_of_match),
732     },
733     .remove = __exit_p(voxel_remove),
734 };
735
736 static int __init voxel_init(void)
737 {
738     pr_info(DRIVER_NAME ": init\n");
739     return platform_driver_probe(&voxel_driver, voxel_probe);
740 }
741
742 static void __exit voxel_exit(void)
743 {
744     platform_driver_unregister(&voxel_driver);
745     pr_info(DRIVER_NAME ": exit\n");
746 }
747
748 module_init(voxel_init);
749 module_exit(voxel_exit);
750
751 MODULE_LICENSE("GPL");
752 MODULE_AUTHOR("the gang and twin");
753 MODULE_DESCRIPTION("Voxel GPU MVP driver -- FIFO streaming + control ioctls");

```

Complete renderer.c Implementation

Listing 2: Complete renderer.c Implementation.

```

1  #include "renderer.h"
2  #include "world.h"
3  #include "gpu_transport.h"
4  #include "voxel_gpu.h"
5  #include "env_util.h"
6  #include <stdlib.h>
7  #include <stdio.h>
8  #include <math.h>
9  #include <string.h>
10
11 #define NEAR_PLANE 0.1f
12 /*
13  * Projected screen coordinates live on pixel-box edges: the visible viewport
14  * spans [0, SCREEN_WIDTH] x [0, SCREEN_HEIGHT], while raster samples sit at
15  * pixel centers (x + 0.5, y + 0.5). Keep clipping in edge space, then bake the
16  * 0.5 center offset into edge/depth setup before descriptors hit the FPGA.
17  */
18 #define VIEW_X_MIN 0.0f
19 #define VIEW_Y_MIN 0.0f
20 #define VIEW_X_MAX SCREEN_WIDTH
21 #define VIEW_Y_MAX SCREEN_HEIGHT
22 #define MAX_VIEW_CLIP_VERTS 12
23 #define PI_F 3.14159265358979323846f
24 #define SKY_DAY_LENGTH_SECONDS 180.0f
25 #define SKY_DOME_DISTANCE 512.0f
26 #define SKY_GRADIENT_BANDS 24
27 #define SKY_STAR_COUNT 144

```

```

28 #define OCCLUSION_TILE_SIZE 4
29 #define OCCLUSION_TILES_X ((VOXEL_RENDER_WIDTH + OCCLUSION_TILE_SIZE - 1u) / \
30     OCCLUSION_TILE_SIZE)
31 #define OCCLUSION_TILES_Y ((VOXEL_RENDER_HEIGHT + OCCLUSION_TILE_SIZE - 1u) / \
32     OCCLUSION_TILE_SIZE)
33 #define OCCLUSION_TILE_COUNT (OCCLUSION_TILES_X * OCCLUSION_TILES_Y)
34 #define OCCLUSION_DEPTH_EPSILON (2.0f / 32768.0f)
35 #define REDSTONE_DEVICE_PLATE_INSET 0.09f
36 #define REDSTONE_DEVICE_PLATE_HALF (0.5f - REDSTONE_DEVICE_PLATE_INSET)
37 #define REDSTONE_DEVICE_PLATE_TEXEL (REDSTONE_DEVICE_PLATE_HALF / 8.0f)
38 #define REDSTONE_DEVICE_POST_HALF 0.060f
39
40 /* Quantize sky-gradient palette updates so tiny time deltas do not churn the
41  * hardware band-reuse epoch every frame. */
42 #define DEFAULT_SKY_PALETTE_TIME_STEP_SECONDS 1.0f
43 #define MIN_SKY_PALETTE_TIME_STEP_SECONDS 0.1f
44 #define MAX_SKY_PALETTE_TIME_STEP_SECONDS 10.0f
45
46 enum {
47     PAL_SKY_HIGH = 25,
48     PAL_SKY_MID = 26,
49     PAL_SKY_HORIZON = 27,
50     PAL_CLOUD = 28,
51     PAL_CLOUD_SHADOW = 29,
52     PAL_SUN_CORE = 30,
53     PAL_SUN_GLOW = 31,
54     PAL_MOON = 32,
55     PAL_MOON_SHADOW = 33,
56     PAL_STAR = 34,
57     PAL_GLASS = 35,
58     PAL_GLASS_EDGE = 36,
59     PAL_GLASS_HIGHLIGHT = 37,
60     PAL_LAMP_GLOW = 38,
61     PAL_LAMP_FRAME = 39,
62     PAL_SKY_GRADIENT_BASE = 40,
63 };
64
65 typedef struct {
66     int x, y, z;
67     uint8_t type;
68     int occupied;
69 } LookupEntry;
70
71 typedef struct {
72     LookupEntry *entries;
73     int capacity;
74     int mask;
75 } BlockLookup;
76
77 typedef struct {
78     uint8_t r, g, b;
79 } RGB24;
80
81 typedef struct {
82     RGB24 zenith;
83     RGB24 high;
84     RGB24 mid;
85     RGB24 horizon;
86     RGB24 cloud;

```

```

87     RGB24 cloud_shadow;
88     RGB24 sun_core;
89     RGB24 sun_glow;
90     RGB24 moon;
91     RGB24 moon_shadow;
92     RGB24 star;
93 } SkyPalette;
94
95 static Vec3 sun_direction_for_time(float time_seconds);
96
97 static void upload_default_palette(GPUPTransport *transport)
98 {
99     static const struct voxel_palette_entry entries[] = {
100     { 0, 0x10, 0x10, 0x18 }, /* background */
101     { 1, 0x6b, 0xa4, 0x3a }, /* grass top */
102     { 2, 0x8b, 0x63, 0x41 }, /* dirt side */
103     { 3, 0x6f, 0x57, 0x37 }, /* wood side */
104     { 4, 0x7c, 0x7c, 0x7c }, /* stone side */
105     { 5, 0xff, 0xff, 0xff }, /* debug white */
106     { 6, 0xff, 0x40, 0x40 }, /* debug red */
107     { 7, 0x40, 0xa0, 0xff }, /* debug blue */
108     { 8, 0xff, 0xd0, 0x40 }, /* debug yellow */
109     { 9, 0x5c, 0x86, 0x34 }, /* grass side */
110     { 10, 0x6a, 0x4a, 0x2c }, /* grass bottom / dark dirt */
111     { 11, 0x9d, 0x7b, 0x4d }, /* wood top */
112     { 12, 0x53, 0x38, 0x23 }, /* wood bottom */
113     { 13, 0x98, 0x98, 0x98 }, /* stone top */
114     { 14, 0x5c, 0x5c, 0x5c }, /* stone bottom */
115     { 15, 0xa7, 0x79, 0x52 }, /* dirt top */
116     { 16, 0x59, 0x41, 0x2a }, /* dirt bottom */
117     { 17, 0x4f, 0x78, 0x2d }, /* grass dark */
118     { 18, 0x84, 0xba, 0x57 }, /* grass highlight */
119     { 19, 0x6f, 0x4f, 0x32 }, /* dirt dark */
120     { 20, 0xaa, 0x81, 0x5a }, /* dirt light */
121     { 21, 0x88, 0x6a, 0x44 }, /* wood grain */
122     { 22, 0x50, 0x3b, 0x24 }, /* wood bark dark */
123     { 23, 0x63, 0x63, 0x63 }, /* stone dark */
124     { 24, 0x9a, 0x9a, 0x9a }, /* stone light */
125     { 25, 0x78, 0xb4, 0xf0 }, /* sky high */
126     { 26, 0xb0, 0xd8, 0xff }, /* sky mid */
127     { 27, 0xe2, 0xef, 0xff }, /* sky horizon */
128     { 28, 0xf5, 0xfa, 0xff }, /* cloud */
129     { 29, 0xcb, 0xd8, 0xec }, /* cloud shadow */
130     { 30, 0xff, 0xee, 0xaa }, /* sun core */
131     { 31, 0xff, 0xbb, 0x55 }, /* sun glow */
132     { 32, 0xe7, 0xeb, 0xf8 }, /* moon */
133     { 33, 0x9c, 0xa4, 0xc0 }, /* moon shadow */
134     { 34, 0xff, 0xff, 0xff }, /* stars */
135     { 35, 0xb8, 0xe4, 0xff }, /* glass body */
136     { 36, 0x5e, 0x7c, 0x98 }, /* glass edge / frame */
137     { 37, 0xff, 0xff, 0xff }, /* glass highlight */
138     { 38, 0xff, 0xd7, 0x79 }, /* lamp glow */
139     { 39, 0x6d, 0x53, 0x30 }, /* lamp frame */
140     { 40, 0x2a, 0x52, 0x9c }, /* banked water deep */
141     { 41, 0x3a, 0x6c, 0xc4 }, /* banked water mid */
142     { 42, 0x6f, 0x9d, 0xe4 }, /* banked water highlight */
143     { 43, 0x7a, 0x20, 0x10 }, /* lava dark */
144     { 44, 0xe8, 0x5c, 0x18 }, /* lava orange */
145     { 45, 0xff, 0xd8, 0x5a }, /* lava hot */

```

```

146     { 46, 0xd8, 0xc0, 0x74 }, /* sand */
147     { 47, 0xa8, 0x90, 0x50 }, /* sand dark */
148     { 48, 0xb8, 0x4d, 0x3f }, /* brick */
149     { 49, 0x6e, 0x2a, 0x27 }, /* brick dark */
150     { 50, 0x20, 0x1b, 0x2c }, /* obsidian */
151     { 51, 0x43, 0x36, 0x58 }, /* obsidian edge */
152     { 52, 0x72, 0x82, 0x91 }, /* clay */
153     { 53, 0xed, 0xdc, 0x9c }, /* sand light */
154     { 54, 0xc3, 0xac, 0x67 }, /* sand shadow */
155     { 55, 0xd0, 0x68, 0x55 }, /* brick light */
156     { 56, 0x8b, 0x56, 0x4e }, /* brick mortar */
157 };
158
159 for (size_t i = 0; i < sizeof(entries) / sizeof(entries[0]); i++) {
160     if (gpu_transport_set_palette(transport, &entries[i]) < 0)
161         break;
162 }
163 }
164
165 typedef struct {
166     const Chunk *chunk;
167     const ChunkFace *face;
168     float view_z;
169     uint32_t stable_order;
170 } TranslucentFaceRef;
171
172 typedef enum {
173     OCCLUSION_PASS_DISABLED = 0,
174     OCCLUSION_PASS_OPAQUE_WORLD,
175 } OcclusionPass;
176
177 struct RenderContext {
178     GPUTransport *transport;
179     Camera current_camera;
180     Vec3 light_dir;
181     float world_daylight;
182
183     float cos_yaw, sin_yaw;
184     float cos_pitch, sin_pitch;
185
186     uint8_t *submit_buffer;
187     size_t submit_capacity;
188     size_t submit_bytes;
189     LookupEntry *lookup_entries;
190     int lookup_capacity;
191     uint16_t palette_cache[256];
192     uint8_t palette_valid[256];
193     struct voxel_palette_entry palette_pending[256];
194     uint8_t palette_dirty[256];
195     uint16_t sky_palette_cache[SKY_GRADIENT_BANDS];
196     uint8_t sky_palette_valid[SKY_GRADIENT_BANDS];
197     struct voxel_sky_palette_entry sky_palette_pending[SKY_GRADIENT_BANDS];
198     uint8_t sky_palette_dirty[SKY_GRADIENT_BANDS];
199     struct voxel_fog_state fog_cache;
200     struct voxel_fog_state fog_pending;
201     uint8_t fog_valid;
202     uint8_t fog_dirty;
203     uint8_t light_palette_key;
204     uint8_t light_palette_key_valid;

```

```

205     uint8_t face_light_flags[NUM_FACES];
206     Vec3 last_sun_dir;
207     uint8_t last_sun_dir_valid;
208     int n_quads;
209     uint8_t occlusion_enabled;
210     uint8_t occlusion_diag;
211     OcclusionPass occlusion_pass;
212     uint8_t occlusion_covered[OCCLUSION_TILE_COUNT];
213     float occlusion_depth[OCCLUSION_TILE_COUNT];
214     uint32_t occlusion_tested;
215     uint32_t occlusion_culled;
216     uint32_t occlusion_tiles_written;
217     uint32_t occlusion_frame_count;
218
219     /* Scratch buffer reused across frames to back-to-front sort translucent
220      * faces. Grown on demand; memory only leaves if the scene peaks higher
221      * than it has before. */
222     TranslucentFaceRef *translucent_scratch;
223     int translucent_scratch_capacity;
224
225 };
226
227 static bool stage_prepared_quad(RenderContext *ctx, RenderQuad quad);
228 static bool stage_projected_quad_no_clip(RenderContext *ctx,
229                                         const RenderQuad *quad);
230 static bool stage_projected_polygon(RenderContext *ctx,
231                                   const RenderQuad *quad,
232                                   const Vertex2D *verts,
233                                   int count);
234
235 typedef struct {
236     float x, y, z;
237     float u, v;
238 } CameraVertex;
239
240 static RGB24 rgb24(uint8_t r, uint8_t g, uint8_t b)
241 {
242     RGB24 color = { r, g, b };
243     return color;
244 }
245
246 static RGB24 default_palette_color(uint8_t index)
247 {
248     switch (index) {
249     case 0: return rgb24(0x10, 0x10, 0x18);
250     case 1: return rgb24(0x6b, 0xa4, 0x3a);
251     case 2: return rgb24(0x8b, 0x63, 0x41);
252     case 3: return rgb24(0x6f, 0x57, 0x37);
253     case 4: return rgb24(0x7c, 0x7c, 0x7c);
254     case 5: return rgb24(0xff, 0xff, 0xff);
255     case 6: return rgb24(0xff, 0x40, 0x40);
256     case 7: return rgb24(0x40, 0xa0, 0xff);
257     case 8: return rgb24(0xff, 0xd0, 0x40);
258     case 9: return rgb24(0x5c, 0x86, 0x34);
259     case 10: return rgb24(0x6a, 0x4a, 0x2c);
260     case 11: return rgb24(0x9d, 0x7b, 0x4d);
261     case 12: return rgb24(0x53, 0x38, 0x23);
262     case 13: return rgb24(0x98, 0x98, 0x98);
263     case 14: return rgb24(0x5c, 0x5c, 0x5c);

```

```

264     case 15: return rgb24(0xa7, 0x79, 0x52);
265     case 16: return rgb24(0x59, 0x41, 0x2a);
266     case 17: return rgb24(0x4f, 0x78, 0x2d);
267     case 18: return rgb24(0x84, 0xba, 0x57);
268     case 19: return rgb24(0x6f, 0x4f, 0x32);
269     case 20: return rgb24(0xaa, 0x81, 0x5a);
270     case 21: return rgb24(0x88, 0x6a, 0x44);
271     case 22: return rgb24(0x50, 0x3b, 0x24);
272     case 23: return rgb24(0x63, 0x63, 0x63);
273     case 24: return rgb24(0x9a, 0x9a, 0x9a);
274     case 25: return rgb24(0x78, 0xb4, 0xf0);
275     case 26: return rgb24(0xb0, 0xd8, 0xff);
276     case 27: return rgb24(0xe2, 0xef, 0xff);
277     case 28: return rgb24(0xf5, 0xfa, 0xff);
278     case 29: return rgb24(0xcb, 0xd8, 0xec);
279     case 30: return rgb24(0xff, 0xee, 0xaa);
280     case 31: return rgb24(0xff, 0xbb, 0x55);
281     case 32: return rgb24(0xe7, 0xeb, 0xf8);
282     case 33: return rgb24(0x9c, 0xa4, 0xc0);
283     case 34: return rgb24(0xff, 0xff, 0xff);
284     case 35: return rgb24(0xb8, 0xe4, 0xff);
285     case 36: return rgb24(0x5e, 0x7c, 0x98);
286     case 37: return rgb24(0xff, 0xff, 0xff);
287     case 38: return rgb24(0xff, 0xd7, 0x79);
288     case 39: return rgb24(0x6d, 0x53, 0x30);
289     case 40: return rgb24(0x2a, 0x52, 0x9c);
290     case 41: return rgb24(0x3a, 0x6c, 0xc4);
291     case 42: return rgb24(0x6f, 0x9d, 0xe4);
292     case 43: return rgb24(0x7a, 0x20, 0x10);
293     case 44: return rgb24(0xe8, 0x5c, 0x18);
294     case 45: return rgb24(0xff, 0xd8, 0x5a);
295     case 46: return rgb24(0xd8, 0xc0, 0x74);
296     case 47: return rgb24(0xa8, 0x90, 0x50);
297     case 48: return rgb24(0xb8, 0x4d, 0x3f);
298     case 49: return rgb24(0x6e, 0x2a, 0x27);
299     case 50: return rgb24(0x20, 0x1b, 0x2c);
300     case 51: return rgb24(0x43, 0x36, 0x58);
301     case 52: return rgb24(0x72, 0x82, 0x91);
302     case 53: return rgb24(0xed, 0xdc, 0x9c);
303     case 54: return rgb24(0xc3, 0xac, 0x67);
304     case 55: return rgb24(0xd0, 0x68, 0x55);
305     case 56: return rgb24(0x8b, 0x56, 0x4e);
306     default: return rgb24(0x00, 0x00, 0x00);
307 }
308 }
309
310 static float clamp01(float value)
311 {
312     if (value < 0.0f)
313         return 0.0f;
314     if (value > 1.0f)
315         return 1.0f;
316     return value;
317 }
318
319 static float smoothstepf(float edge0, float edge1, float x)
320 {
321     if (edge0 == edge1)
322         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)

```

```

382     return false;
383
384     packed = rgb24_to_rgb565(color);
385     if (ctx->sky_palette_valid[index] &&
386         ctx->sky_palette_cache[index] == packed)
387         return true;
388
389     ctx->sky_palette_valid[index] = 1;
390     ctx->sky_palette_cache[index] = packed;
391     ctx->sky_palette_pending[index] = (struct voxel_sky_palette_entry){
392         .index = index,
393         .r = color.r,
394         .g = color.g,
395         .b = color.b,
396     };
397     ctx->sky_palette_dirty[index] = 1;
398     return true;
399 }
400
401 static bool same_fog_state(const struct voxel_fog_state *a,
402                          const struct voxel_fog_state *b)
403 {
404     return a->start_dist == b->start_dist &&
405           a->end_dist == b->end_dist &&
406           a->color_index == b->color_index &&
407           a->enabled == b->enabled &&
408           a->inv_proj_sq == b->inv_proj_sq;
409 }
410
411 static bool renderer_set_fog_state(RenderContext *ctx,
412                                   const struct voxel_fog_state *fog)
413 {
414     if (ctx->fog_valid && same_fog_state(&ctx->fog_cache, fog))
415         return true;
416
417     ctx->fog_cache = *fog;
418     ctx->fog_pending = *fog;
419     ctx->fog_valid = 1;
420     ctx->fog_dirty = 1;
421     return true;
422 }
423
424 static bool renderer_flush_gpu_state(RenderContext *ctx)
425 {
426     for (int i = 0; i < SKY_GRADIENT_BANDS; i++) {
427         if (!ctx->sky_palette_dirty[i])
428             continue;
429         if (gpu_transport_set_sky_palette(ctx->transport,
430                                         &ctx->sky_palette_pending[i]) < 0)
431             return false;
432         ctx->sky_palette_dirty[i] = 0;
433     }
434
435     for (int i = 0; i < 256; i++) {
436         if (!ctx->palette_dirty[i])
437             continue;
438         if (gpu_transport_set_palette(ctx->transport,
439                                     &ctx->palette_pending[i]) < 0)
440             return false;

```

```

441     ctx->palette_dirty[i] = 0;
442 }
443
444 if (ctx->fog_dirty) {
445     if (gpu_transport_set_fog(ctx->transport, &ctx->fog_pending) < 0)
446         return false;
447     ctx->fog_dirty = 0;
448 }
449
450 return true;
451 }
452
453 static void upload_light_palette_banks(RenderContext *ctx, float daylight)
454 {
455     static const float bank_scale_day[4] = { 1.00f, 0.82f, 0.64f, 0.50f };
456     static const float bank_scale_night[4] = { 1.00f, 0.34f, 0.20f, 0.10f };
457     int daylight_key;
458
459     if (!ctx)
460         return;
461
462     daylight = clamp01(daylight);
463     daylight_key = (int)lroundf(daylight * 31.0f);
464     if (ctx->light_palette_key_valid &&
465         ctx->light_palette_key == (uint8_t)daylight_key)
466         return;
467
468     daylight = (float)daylight_key / 31.0f;
469
470     for (int bank = 1; bank < 4; bank++) {
471         float bank_scale = bank_scale_night[bank] +
472             (bank_scale_day[bank] - bank_scale_night[bank]) *
473             daylight;
474
475         for (int index = 1; index < 64; index++) {
476             RGB24 color = scale_rgb24(default_palette_color((uint8_t)index),
477                                     bank_scale);
478
479             if (!renderer_set_palette_rgb(ctx,
480                                         (uint8_t)(bank * 64 + index),
481                                         color))
482                 return;
483         }
484     }
485
486     ctx->light_palette_key = (uint8_t)daylight_key;
487     ctx->light_palette_key_valid = 1;
488 }
489
490 static Vec3 normalize_vec3(Vec3 v)
491 {
492     float len_sq = v.x * v.x + v.y * v.y + v.z * v.z;
493
494     if (len_sq <= 0.0f)
495         return (Vec3){ 0.0f, 0.0f, 1.0f };
496
497     float inv_len = 1.0f / sqrtf(len_sq);
498     return (Vec3){ v.x * inv_len, v.y * inv_len, v.z * inv_len };
499 }

```

```

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,

```

```

559         BlockFace face,
560         uint8_t sky_light)
561 {
562     float ndot1;
563     float facing;
564     float intensity;
565     int level;
566
567     if (!ctx || face < 0 || face >= NUM_FACES || sky_light == 0)
568         return 0;
569
570     ndot1 = face_normals[face].x * ctx->light_dir.x +
571           face_normals[face].y * ctx->light_dir.y +
572           face_normals[face].z * ctx->light_dir.z;
573     facing = clamp01(0.5f + 0.5f * ndot1);
574     intensity = 0.12f + ctx->world_daylight * (0.25f + 0.63f * facing);
575     level = (int)lroundf((float)sky_light * clamp01(intensity));
576     if (level < 0)
577         level = 0;
578     if (level > 15)
579         level = 15;
580     return (uint8_t)level;
581 }
582
583 static uint8_t choose_chunk_face_light_flags(const RenderContext *ctx,
584                                             BlockID type, BlockFace face,
585                                             uint8_t sky_light,
586                                             uint8_t block_light)
587 {
588     uint8_t effective_level = block_light;
589     uint8_t sky_level = sky_light_level_for_face(ctx, face, sky_light);
590
591     if (block_is_self_lit(type))
592         effective_level = 15;
593     if (sky_level > effective_level)
594         effective_level = sky_level;
595
596     return QUAD_LIGHT_LEVEL(light_bank_for_level(effective_level));
597 }
598
599 static float daylight_strength_for_sun_direction(Vec3 sun_dir)
600 {
601     return smoothstepf(-0.18f, 0.08f, sun_dir.y);
602 }
603
604 static Vec3 terrain_light_direction_for_sun(Vec3 sun_dir)
605 {
606     if (sun_dir.y < 0.18f)
607         sun_dir.y = 0.18f;
608     return normalize_vec3(sun_dir);
609 }
610
611 static void update_world_light_from_sun(RenderContext *ctx, Vec3 sun_dir)
612 {
613     if (!ctx)
614         return;
615
616     if (ctx->last_sun_dir_valid &&
617         ctx->last_sun_dir.x == sun_dir.x &&

```

```

618     ctx->last_sun_dir.y == sun_dir.y &&
619     ctx->last_sun_dir.z == sun_dir.z)
620     return;
621
622     ctx->world_daylight = daylight_strength_for_sun_direction(sun_dir);
623     ctx->light_dir = terrain_light_direction_for_sun(sun_dir);
624     update_face_light_flags(ctx);
625     upload_light_palette_banks(ctx, ctx->world_daylight);
626     ctx->last_sun_dir = sun_dir;
627     ctx->last_sun_dir_valid = 1;
628 }
629
630 static float palette_time_for(float time_seconds)
631 {
632     static int initialized;
633     static float step_seconds = DEFAULT_SKY_PALETTE_TIME_STEP_SECONDS;
634
635     if (!initialized) {
636         step_seconds =
637             env_float_or_default("VOXEL_SKY_PALETTE_STEP_SECONDS",
638                                 DEFAULT_SKY_PALETTE_TIME_STEP_SECONDS,
639                                 MIN_SKY_PALETTE_TIME_STEP_SECONDS,
640                                 MAX_SKY_PALETTE_TIME_STEP_SECONDS);
641         initialized = 1;
642     }
643
644     return floorf(time_seconds / step_seconds) * step_seconds;
645 }
646
647 static void update_world_light_state(RenderContext *ctx, float time_seconds)
648 {
649     update_world_light_from_sun(ctx,
650                                 sun_direction_for_time(palette_time_for(time_seconds)));
651 }
652
653 static void world_to_camera(RenderContext *ctx, Vec3 world, CameraVertex *out)
654 {
655     float dx = world.x - ctx->current_camera.position.x;
656     float dy = world.y - ctx->current_camera.position.y;
657     float dz = world.z - ctx->current_camera.position.z;
658
659     float x_yaw = dx * ctx->cos_yaw - dz * ctx->sin_yaw;
660     float z_yaw = dx * ctx->sin_yaw + dz * ctx->cos_yaw;
661     float y_yaw = dy;
662
663     out->x = x_yaw;
664     out->y = y_yaw * ctx->cos_pitch + z_yaw * ctx->sin_pitch;
665     out->z = -y_yaw * ctx->sin_pitch + z_yaw * ctx->cos_pitch;
666 }
667
668 /* Project a camera-space point into screen space.
669  * Returns false if the point is behind the near plane. */
670 static bool project_camera_vertex(RenderContext *ctx, const CameraVertex *in, Vertex2D *out)
671 {
672     if (in->z < NEAR_PLANE)
673         return false;
674
675     /*
676     * Use an inverse-depth mapping so the per-quad depth plane stays affine

```

```

677     * in screen space and fits inside the descriptor's Q1.15 range.
678     * Smaller values remain closer, matching the hardware z compare.
679     */
680     out->z = 1.0f - (NEAR_PLANE / in->z);
681
682     float depth = ctx->current_camera.depth;
683     float inv_w = 1.0f / in->z;
684     out->x = in->x * inv_w * depth + SCREEN_WIDTH / 2.0f;
685     out->y = SCREEN_HEIGHT / 2.0f - in->y * inv_w * depth;
686     /*
687     * Store u/w, v/w, 1/w instead of u, v. These three quantities are linear
688     * in screen space for any planar world-space quad, so viewport clipping
689     * can lerp them correctly and the HW can recover true u, v by dividing
690     * per pixel. This eliminates the affine-texture swim under perspective.
691     */
692     out->u_over_w = in->u * inv_w;
693     out->v_over_w = in->v * inv_w;
694     out->one_over_w = inv_w;
695     return true;
696 }
697
698 static bool camera_quad_outside_view(RenderContext *ctx, const CameraVertex v[4])
699 {
700     float x_slope = (SCREEN_WIDTH * 0.5f) / ctx->current_camera.depth;
701     float y_slope = (SCREEN_HEIGHT * 0.5f) / ctx->current_camera.depth;
702     bool outside_near = true;
703     bool outside_left = true;
704     bool outside_right = true;
705     bool outside_top = true;
706     bool outside_bottom = true;
707
708     /*
709     * The side-plane tests (x + x_slope*z < 0, etc.) are valid only for
710     * vertices in front of the near plane. For z <= 0, x_slope*z is non-positive,
711     * so the test can flip sign and false-positive -- a vertex behind the camera
712     * with any x looks "to the left of left" by this math, which used to cull
713     * any face that had a vertex behind the player. Gate side-plane votes on
714     * z >= NEAR_PLANE; a partially-behind face only gets culled if every vertex
715     * is behind the near plane.
716     */
717     for (int i = 0; i < 4; i++) {
718         bool z_in_front = (v[i].z >= NEAR_PLANE);
719         outside_near &= !z_in_front;
720         outside_left &= z_in_front && (v[i].x + x_slope * v[i].z < 0.0f);
721         outside_right &= z_in_front && (-v[i].x + x_slope * v[i].z < 0.0f);
722         outside_top &= z_in_front && (y_slope * v[i].z - v[i].y < 0.0f);
723         outside_bottom &= z_in_front && (v[i].y + y_slope * v[i].z < 0.0f);
724     }
725
726     return outside_near || outside_left || outside_right ||
727            outside_top || outside_bottom;
728 }
729
730 /* Dot-product backface cull: skip face if normal points away from camera. */
731 static bool is_face_visible(Vec3 block_pos, Vec3 normal, Vec3 cam_pos)
732 {
733     Vec3 v = {
734         (block_pos.x + 0.5f + normal.x * 0.5f) - cam_pos.x,
735         (block_pos.y + 0.5f + normal.y * 0.5f) - cam_pos.y,

```

```

736     (block_pos.z + 0.5f + normal.z * 0.5f) - cam_pos.z,
737 };
738 return (v.x*normal.x + v.y*normal.y + v.z*normal.z) < 0.0f;
739 }
740
741 static float height_eighths_to_world(uint8_t height_eighths)
742 {
743     if (height_eighths < 1)
744         height_eighths = 8;
745     if (height_eighths > 8)
746         height_eighths = 8;
747     return (float)height_eighths * (1.0f / 8.0f);
748 }
749
750 static Vec3 merged_face_center(Vec3 block_pos, BlockFace face,
751                               int u_size, int v_size,
752                               uint8_t height_eighths)
753 {
754     float h = height_eighths_to_world(height_eighths);
755     float u = (float)(u_size > 0 ? u_size : 1);
756     float v = (float)(v_size > 0 ? v_size : 1);
757     float vertical_extent = (v - 1.0f) + h;
758
759     switch (face) {
760     case FACE_TOP:
761         return (Vec3){ block_pos.x + 0.5f * u,
762                       block_pos.y + h,
763                       block_pos.z + 0.5f * v };
764     case FACE_BOTTOM:
765         return (Vec3){ block_pos.x + 0.5f * u,
766                       block_pos.y,
767                       block_pos.z + 0.5f * v };
768     case FACE_LEFT:
769         return (Vec3){ block_pos.x,
770                       block_pos.y + 0.5f * vertical_extent,
771                       block_pos.z + 0.5f * u };
772     case FACE_RIGHT:
773         return (Vec3){ block_pos.x + 1.0f,
774                       block_pos.y + 0.5f * vertical_extent,
775                       block_pos.z + 0.5f * u };
776     case FACE_FRONT:
777         return (Vec3){ block_pos.x + 0.5f * u,
778                       block_pos.y + 0.5f * vertical_extent,
779                       block_pos.z };
780     case FACE_BACK:
781         return (Vec3){ block_pos.x + 0.5f * u,
782                       block_pos.y + 0.5f * vertical_extent,
783                       block_pos.z + 1.0f };
784     default:
785         return block_pos;
786     }
787 }
788
789 static bool merged_face_visible(Vec3 block_pos, BlockFace face,
790                                int u_size, int v_size,
791                                uint8_t height_eighths, Vec3 cam_pos)
792 {
793     if (face < 0 || face >= NUM_FACES)
794         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)

```

```

854         out[out_count++] = cur;
855
856         prev = cur;
857         prev_inside = cur_inside;
858     }
859
860     return out_count;
861 }
862
863 typedef enum {
864     CLIP_LEFT = 0,
865     CLIP_RIGHT,
866     CLIP_TOP,
867     CLIP_BOTTOM,
868 } ClipBoundary;
869
870 static bool vertex_inside_boundary(const Vertex2D *v, ClipBoundary boundary)
871 {
872     switch (boundary) {
873     case CLIP_LEFT:
874         return v->x >= VIEW_X_MIN;
875     case CLIP_RIGHT:
876         return v->x <= VIEW_X_MAX;
877     case CLIP_TOP:
878         return v->y >= VIEW_Y_MIN;
879     case CLIP_BOTTOM:
880         return v->y <= VIEW_Y_MAX;
881     default:
882         return false;
883     }
884 }
885
886 static float boundary_value(ClipBoundary boundary)
887 {
888     switch (boundary) {
889     case CLIP_LEFT:
890         return VIEW_X_MIN;
891     case CLIP_RIGHT:
892         return VIEW_X_MAX;
893     case CLIP_TOP:
894         return VIEW_Y_MIN;
895     case CLIP_BOTTOM:
896         return VIEW_Y_MAX;
897     default:
898         return 0.0f;
899     }
900 }
901
902 static int clip_polygon_against_boundary(const Vertex2D *in, int count,
903                                         Vertex2D *out, ClipBoundary boundary)
904 {
905     if (count <= 0)
906         return 0;
907
908     int out_count = 0;
909     Vertex2D prev = in[count - 1];
910     bool prev_inside = vertex_inside_boundary(&prev, boundary);
911     float bound = boundary_value(boundary);
912

```

```

913     for (int i = 0; i < count; i++) {
914         Vertex2D cur = in[i];
915         bool cur_inside = vertex_inside_boundary(&cur, boundary);
916
917         if (prev_inside != cur_inside) {
918             float denom;
919             float t;
920             Vertex2D clipped;
921
922             if (boundary == CLIP_LEFT || boundary == CLIP_RIGHT)
923                 denom = cur.x - prev.x;
924             else
925                 denom = cur.y - prev.y;
926
927             if (fabsf(denom) < 1e-6f)
928                 t = 0.0f;
929             else if (boundary == CLIP_LEFT || boundary == CLIP_RIGHT)
930                 t = (bound - prev.x) / denom;
931             else
932                 t = (bound - prev.y) / denom;
933
934             clipped = lerp_vertex2d(&prev, &cur, t);
935             if (boundary == CLIP_LEFT || boundary == CLIP_RIGHT)
936                 clipped.x = bound;
937             else
938                 clipped.y = bound;
939             if (out_count >= MAX_VIEW_CLIP_VERTS)
940                 return 0;
941             out[out_count++] = clipped;
942         }
943
944         if (cur_inside) {
945             if (out_count >= MAX_VIEW_CLIP_VERTS)
946                 return 0;
947             out[out_count++] = cur;
948         }
949
950         prev = cur;
951         prev_inside = cur_inside;
952     }
953
954     return out_count;
955 }
956
957 static int clip_polygon_against_y_bound(const Vertex2D *in, int count,
958                                       Vertex2D *out, float bound,
959                                       bool keep_greater_equal)
960 {
961     if (count <= 0)
962         return 0;
963
964     int out_count = 0;
965     Vertex2D prev = in[count - 1];
966     bool prev_inside = keep_greater_equal ? (prev.y >= bound) :
967                                     (prev.y <= bound);
968
969     for (int i = 0; i < count; i++) {
970         Vertex2D cur = in[i];
971         bool cur_inside = keep_greater_equal ? (cur.y >= bound) :

```

```

972         (cur.y <= bound);
973
974     if (prev_inside != cur_inside) {
975         float denom = cur.y - prev.y;
976         float t = fabsf(denom) < 1e-6f ? 0.0f : (bound - prev.y) / denom;
977         Vertex2D clipped = lerp_vertex2d(&prev, &cur, t);
978
979         clipped.y = bound;
980         if (out_count >= MAX_VIEW_CLIP_VERTS)
981             return 0;
982         out[out_count++] = clipped;
983     }
984
985     if (cur_inside) {
986         if (out_count >= MAX_VIEW_CLIP_VERTS)
987             return 0;
988         out[out_count++] = cur;
989     }
990
991     prev = cur;
992     prev_inside = cur_inside;
993 }
994
995 return out_count;
996 }
997
998 static int clip_polygon_to_y_range(const Vertex2D *in, int count,
999                                 Vertex2D *out, float y_min, float y_max)
1000 {
1001     Vertex2D tmp[MAX_VIEW_CLIP_VERTS];
1002
1003     count = clip_polygon_against_y_bound(in, count, tmp, y_min, true);
1004     if (count == 0)
1005         return 0;
1006     count = clip_polygon_against_y_bound(tmp, count, out, y_max, false);
1007     return count;
1008 }
1009
1010 static int clip_polygon_to_viewport(const Vertex2D in[4], Vertex2D out[MAX_VIEW_CLIP_VERTS])
1011 {
1012     Vertex2D buf_a[MAX_VIEW_CLIP_VERTS];
1013     Vertex2D buf_b[MAX_VIEW_CLIP_VERTS];
1014     int count = 4;
1015
1016     memcpy(buf_a, in, 4 * sizeof(Vertex2D));
1017
1018     count = clip_polygon_against_boundary(buf_a, count, buf_b, CLIP_LEFT);
1019     if (count == 0)
1020         return 0;
1021     count = clip_polygon_against_boundary(buf_b, count, buf_a, CLIP_RIGHT);
1022     if (count == 0)
1023         return 0;
1024     count = clip_polygon_against_boundary(buf_a, count, buf_b, CLIP_TOP);
1025     if (count == 0)
1026         return 0;
1027     count = clip_polygon_against_boundary(buf_b, count, buf_a, CLIP_BOTTOM);
1028     if (count == 0)
1029         return 0;
1030

```

```

1031     memcpy(out, buf_a, (size_t)count * sizeof(Vertex2D));
1032     return count;
1033 }
1034
1035 static uint32_t hash_grid_coord(int x, int y, int z)
1036 {
1037     return ((uint32_t)x * 73856093u ^
1038            ((uint32_t)y * 19349663u ^
1039            ((uint32_t)z * 83492791u));
1040 }
1041
1042 static bool ensure_lookup_capacity(RenderContext *ctx, int num_blocks)
1043 {
1044     int needed = 16;
1045
1046     while (needed < num_blocks * 4)
1047         needed <<= 1;
1048
1049     if (needed <= ctx->lookup_capacity)
1050         return true;
1051
1052     LookupEntry *entries = realloc(ctx->lookup_entries,
1053                                   (size_t)needed * sizeof(*entries));
1054     if (!entries)
1055         return false;
1056
1057     ctx->lookup_entries = entries;
1058     ctx->lookup_capacity = needed;
1059     return true;
1060 }
1061
1062 static bool build_block_lookup(RenderContext *ctx, const Block *blocks, int num_blocks,
1063                               BlockLookup *lookup)
1064 {
1065     if (!ensure_lookup_capacity(ctx, num_blocks))
1066         return false;
1067
1068     memset(ctx->lookup_entries, 0,
1069            (size_t)ctx->lookup_capacity * sizeof(*ctx->lookup_entries));
1070
1071     lookup->entries = ctx->lookup_entries;
1072     lookup->capacity = ctx->lookup_capacity;
1073     lookup->mask = ctx->lookup_capacity - 1;
1074
1075     for (int i = 0; i < num_blocks; i++) {
1076         if (blocks[i].type == BLOCK_AIR)
1077             continue;
1078
1079         int x = (int)lroundf(blocks[i].position.x);
1080         int y = (int)lroundf(blocks[i].position.y);
1081         int z = (int)lroundf(blocks[i].position.z);
1082         uint32_t idx = hash_grid_coord(x, y, z) & (uint32_t)lookup->mask;
1083
1084         while (lookup->entries[idx].occupied) {
1085             if (lookup->entries[idx].x == x &&
1086                 lookup->entries[idx].y == y &&
1087                 lookup->entries[idx].z == z)
1088                 break;
1089             idx = (idx + 1u) & (uint32_t)lookup->mask;

```

```

1090     }
1091
1092     lookup->entries[idx].x = x;
1093     lookup->entries[idx].y = y;
1094     lookup->entries[idx].z = z;
1095     lookup->entries[idx].type = (uint8_t)blocks[i].type;
1096     lookup->entries[idx].occupied = 1;
1097 }
1098
1099     return true;
1100 }
1101
1102 static BlockID lookup_block_at(const BlockLookup *lookup, Vec3 pos)
1103 {
1104     int x = (int)lroundf(pos.x);
1105     int y = (int)lroundf(pos.y);
1106     int z = (int)lroundf(pos.z);
1107     uint32_t idx = hash_grid_coord(x, y, z) & (uint32_t)lookup->mask;
1108
1109     while (lookup->entries[idx].occupied) {
1110         if (lookup->entries[idx].x == x &&
1111             lookup->entries[idx].y == y &&
1112             lookup->entries[idx].z == z)
1113             return (BlockID)lookup->entries[idx].type;
1114         idx = (idx + 1u) & (uint32_t)lookup->mask;
1115     }
1116
1117     return BLOCK_AIR;
1118 }
1119
1120 static bool is_face_exposed(const Block *block, BlockFace face,
1121                             const BlockLookup *lookup)
1122 {
1123     Vec3 neighbor = {
1124         block->position.x + face_normals[face].x,
1125         block->position.y + face_normals[face].y,
1126         block->position.z + face_normals[face].z,
1127     };
1128
1129     return block_face_should_render(block->type, lookup_block_at(lookup, neighbor));
1130 }
1131
1132 /* Pack a float edge coefficient into Q24.8 (multiply by 256, round). */
1133 static inline int32_t to_q24_8(float v)
1134 {
1135     float scaled = v * 256.0f;
1136     return (int32_t)(scaled >= 0.0f ? scaled + 0.5f : scaled - 0.5f);
1137 }
1138
1139 static inline float snap_q24_8(float v)
1140 {
1141     return (float)to_q24_8(v) / 256.0f;
1142 }
1143
1144 static inline int floor_div_256_i32(int32_t v)
1145 {
1146     int q = v / 256;
1147     int r = v % 256;
1148

```

```

1149     return (r && v < 0) ? q - 1 : q;
1150 }
1151
1152 static inline int ceil_div_256_i32(int32_t v)
1153 {
1154     int q = v / 256;
1155     int r = v % 256;
1156
1157     return (r && v > 0) ? q + 1 : q;
1158 }
1159
1160 static bool same_screen_xy(const Vertex2D *a, const Vertex2D *b)
1161 {
1162     return a->x == b->x && a->y == b->y;
1163 }
1164
1165 static int snap_and_compact_polygon(Vertex2D *verts, int count)
1166 {
1167     Vertex2D compact[MAX_VIEW_CLIP_VERTS];
1168     int out_count = 0;
1169
1170     for (int i = 0; i < count; i++) {
1171         verts[i].x = snap_q24_8(verts[i].x);
1172         verts[i].y = snap_q24_8(verts[i].y);
1173
1174         if (out_count > 0 && same_screen_xy(&compact[out_count - 1], &verts[i]))
1175             continue;
1176
1177         compact[out_count++] = verts[i];
1178     }
1179
1180     if (out_count > 1 && same_screen_xy(&compact[0], &compact[out_count - 1]))
1181         out_count--;
1182
1183     memcpy(verts, compact, (size_t)out_count * sizeof(*verts));
1184     return out_count;
1185 }
1186
1187 static bool split_quads_at_hw_band_edges(void)
1188 {
1189     static int cached = -1;
1190
1191     if (cached < 0)
1192         cached = env_flag("VOXEL_SPLIT_BAND_QUADS", true) ? 1 : 0;
1193     return cached != 0;
1194 }
1195
1196 static bool split_translucent_quads_at_hw_band_edges(void)
1197 {
1198     static int cached = -1;
1199
1200     if (cached < 0)
1201         cached = env_flag("VOXEL_SPLIT_TRANSLUCENT_QUADS", false) ? 1 : 0;
1202     return cached != 0;
1203 }
1204
1205 static int textured_world_slice_height_px(void)
1206 {
1207     static int cached = -1;

```

```

1208
1209     if (cached < 0) {
1210         int slice_px = env_int_or_default("VOXEL_TEXTURE_SLICE_PX", 30, 0,
1211                                         (int)VOXEL_BAND_CACHE_HEIGHT);
1212         if (slice_px > 0 && slice_px < 4)
1213             slice_px = 4;
1214         cached = slice_px;
1215     }
1216     return cached;
1217 }
1218
1219 static bool stage_polygon_as_quads(RenderContext *ctx,
1220                                   const RenderQuad *src,
1221                                   const Vertex2D *verts,
1222                                   int count)
1223 {
1224     Vertex2D compact[MAX_VIEW_CLIP_VERTS];
1225
1226     if (count > MAX_VIEW_CLIP_VERTS)
1227         return false;
1228     memcpy(compact, verts, (size_t)count * sizeof(*compact));
1229     count = snap_and_compact_polygon(compact, count);
1230     if (count < 3)
1231         return false;
1232
1233     if (count == 3) {
1234         RenderQuad tri = {
1235             .vertices = { compact[0], compact[1], compact[2], compact[2] },
1236             .texture_id = src->texture_id,
1237             .color_tint = src->color_tint,
1238             .flags = src->flags,
1239         };
1240         return stage_prepared_quad(ctx, tri);
1241     }
1242
1243     if (count == 4) {
1244         RenderQuad quad = {
1245             .vertices = { compact[0], compact[1], compact[2], compact[3] },
1246             .texture_id = src->texture_id,
1247             .color_tint = src->color_tint,
1248             .flags = src->flags,
1249         };
1250         return stage_prepared_quad(ctx, quad);
1251     }
1252
1253     bool emitted = false;
1254     for (int i = 1; i + 1 < count; i++) {
1255         RenderQuad tri = {
1256             .vertices = { compact[0], compact[i], compact[i + 1], compact[i + 1] },
1257             .texture_id = src->texture_id,
1258             .color_tint = src->color_tint,
1259             .flags = src->flags,
1260         };
1261         if (stage_prepared_quad(ctx, tri))
1262             emitted = true;
1263     }
1264     return emitted;
1265 }
1266

```

```

1267 static bool stage_projected_polygon(RenderContext *ctx,
1268                                   const RenderQuad *quad,
1269                                   const Vertex2D *verts,
1270                                   int count)
1271 {
1272     if (count < 3)
1273         return false;
1274
1275     float y_min = verts[0].y;
1276     float y_max = verts[0].y;
1277     for (int i = 1; i < count; i++) {
1278         if (verts[i].y < y_min) y_min = verts[i].y;
1279         if (verts[i].y > y_max) y_max = verts[i].y;
1280     }
1281
1282     if (!split_quads_at_hw_band_edges())
1283         return stage_polygon_as_quads(ctx, quad, verts, count);
1284     if (!(quad->flags & QUAD_FLAG_TEX))
1285         return stage_polygon_as_quads(ctx, quad, verts, count);
1286
1287     const bool alpha_blended =
1288         (quad->flags & QUAD_ALPHA_MASK) != QUAD_ALPHA_OPAQUE;
1289     if (alpha_blended && !split_translucent_quads_at_hw_band_edges())
1290         return stage_polygon_as_quads(ctx, quad, verts, count);
1291
1292     /*
1293      * The hardware renderer consumes one 60px-tall resident band at a time.
1294      * Clip in float space before descriptor packing so each per-band descriptor
1295      * gets its own freshly solved UV/depth planes. This avoids stitching bands
1296      * together by adding dy to already-quantized Q16.16/Q1.15 starts, which can
1297      * show up as texture shimmer or a horizontal slice on tall faces. Z-tested
1298      * textured world geometry can also be split more finely inside each band
1299      * with VOXEL_TEXTURE_SLICE_PX to reduce fixed-point edge/UV drift.
1300      */
1301     const float band_h = (float)VOXEL_BAND_CACHE_HEIGHT;
1302     const bool ztested_textured = (quad->flags & QUAD_FLAG_ZTEST) != 0;
1303     const int slice_px = ztested_textured ? textured_world_slice_height_px() : 0;
1304     const float slice_h = (slice_px > 0) ? (float)slice_px : band_h;
1305     int first_band = (int)floorf(y_min / band_h);
1306     int last_band = (int)floorf(y_max / band_h);
1307     if (first_band < 0) first_band = 0;
1308     if (last_band < 0) last_band = 0;
1309     if (first_band >= (int)VOXEL_BAND_COUNT)
1310         first_band = (int)VOXEL_BAND_COUNT - 1;
1311     if (last_band >= (int)VOXEL_BAND_COUNT)
1312         last_band = (int)VOXEL_BAND_COUNT - 1;
1313
1314     float bucket_y_max = y_max - 1e-4f;
1315     if (bucket_y_max < y_min)
1316         bucket_y_max = y_min;
1317     int first_slice = slice_px > 0 ? (int)floorf(y_min / slice_h) : 0;
1318     int last_slice = slice_px > 0 ? (int)floorf(bucket_y_max / slice_h) : 0;
1319
1320     if (first_band == last_band && (slice_px <= 0 || first_slice == last_slice))
1321         return stage_polygon_as_quads(ctx, quad, verts, count);
1322
1323     bool emitted = false;
1324     float emit_y_min = fmaxf(y_min, 0.0f);
1325     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)

```

```

1385     return edge;
1386
1387     if (!(dy < 0.0f || (fabsf(dy) < 1e-6f && dx > 0.0f)))
1388         edge.C -= 1;
1389
1390     return edge;
1391 }
1392
1393 /* Pack a float depth value into Q1.15 unsigned (clamp to [0,2]). */
1394 static inline uint16_t to_q1_15u(float v)
1395 {
1396     if (v < 0.0f) v = 0.0f;
1397     if (v > 1.99f) v = 1.99f;
1398     return (uint16_t)(v * 32768.0f + 0.5f);
1399 }
1400
1401 /* Pack a float depth gradient into Q1.15 signed (clamp to [-1,1]). */
1402 static inline int16_t to_q1_15s(float v)
1403 {
1404     if (v < -1.0f) v = -1.0f;
1405     if (v > 0.999f) v = 0.999f;
1406     {
1407         float scaled = v * 32768.0f;
1408         return (int16_t)(scaled >= 0.0f ? scaled + 0.5f : scaled - 0.5f);
1409     }
1410 }
1411
1412 static inline uint16_t to_q8_8u(float v)
1413 {
1414     if (v < 0.0f) v = 0.0f;
1415     if (v > 255.996f) v = 255.996f;
1416     return (uint16_t)(v * 256.0f + 0.5f);
1417 }
1418
1419 static inline uint16_t to_q0_16u(float v)
1420 {
1421     if (v < 0.0f) v = 0.0f;
1422     if (v > 0.9999847f) v = 0.9999847f;
1423     return (uint16_t)(v * 65536.0f + 0.5f);
1424 }
1425
1426 static inline int32_t to_q16_16(float v)
1427 {
1428     float scaled = v * 65536.0f;
1429     return (int32_t)(scaled >= 0.0f ? scaled + 0.5f : scaled - 0.5f);
1430 }
1431
1432 static float quad_signed_area(const Vertex2D v[4])
1433 {
1434     float area = 0.0f;
1435
1436     for (int i = 0; i < 4; i++) {
1437         const Vertex2D *a = &v[i];
1438         const Vertex2D *b = &v[(i + 1) % 4];
1439         area += a->x * b->y - b->x * a->y;
1440     }
1441
1442     return 0.5f * area;
1443 }

```

```

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  $z(x,y) = z_0 + dz\_dx*x + dz\_dy*y$  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,

```

```

1562         v[basis->b].u_over_w,
1563         v[basis->c].u_over_w,
1564         &duw_dx, &duw_dy);
1565     solve_attr_with_basis(basis,
1566         v[basis->a].v_over_w,
1567         v[basis->b].v_over_w,
1568         v[basis->c].v_over_w,
1569         &dvw_dx, &dvw_dy);
1570     solve_attr_with_basis(basis,
1571         v[basis->a].one_over_w,
1572         v[basis->b].one_over_w,
1573         v[basis->c].one_over_w,
1574         &diw_dx, &diw_dy);
1575
1576     float uw_origin = v[a].u_over_w - duw_dx * v[a].x - duw_dy * v[a].y;
1577     float vw_origin = v[a].v_over_w - dvw_dx * v[a].x - dvw_dy * v[a].y;
1578     float iw_origin = v[a].one_over_w - diw_dx * v[a].x - diw_dy * v[a].y;
1579     float uw_sample = uw_origin + duw_dx * sample_x + duw_dy * sample_y;
1580     float vw_sample = vw_origin + dvw_dx * sample_x + dvw_dy * sample_y;
1581     float iw_sample = iw_origin + diw_dx * sample_x + diw_dy * sample_y;
1582
1583     uv->u_over_w_0 = to_q16_16(uw_sample);
1584     uv->u_over_w_dx = to_q16_16(duw_dx);
1585     uv->u_over_w_dy = to_q16_16(duw_dy);
1586     uv->v_over_w_0 = to_q16_16(vw_sample);
1587     uv->v_over_w_dx = to_q16_16(dvw_dx);
1588     uv->v_over_w_dy = to_q16_16(dvw_dy);
1589     uv->one_over_w_0 = to_q16_16(iw_sample);
1590     uv->one_over_w_dx = to_q16_16(diw_dx);
1591     uv->one_over_w_dy = to_q16_16(diw_dy);
1592 }
1593
1594 /* Conservative software occlusion:
1595  * a tile is considered covered only when one opaque, z-tested quad contains
1596  * all four extreme pixel centers. A later quad is culled only when every tile
1597  * in its bbox is covered by an occluder whose farthest vertex is still closer
1598  * than the candidate's nearest vertex. */
1599 static void occlusion_reset(RenderContext *ctx)
1600 {
1601     memset(ctx->occlusion_covered, 0, sizeof(ctx->occlusion_covered));
1602     ctx->occlusion_pass = OCCLUSION_PASS_DISABLED;
1603     ctx->occlusion_tested = 0;
1604     ctx->occlusion_culled = 0;
1605     ctx->occlusion_tiles_written = 0;
1606 }
1607
1608 static bool occlusion_quad_eligible(uint8_t flags)
1609 {
1610     return (flags & QUAD_FLAG_ZTEST) &&
1611         !(flags & QUAD_FLAG_ALPHA_KEY) &&
1612         ((flags & QUAD_ALPHA_MASK) == QUAD_ALPHA_OPAQUE);
1613 }
1614
1615 static int occlusion_tile_index(int tx, int ty)
1616 {
1617     return ty * (int)OCCLUSION_TILES_X + tx;
1618 }
1619
1620 static bool point_inside_convex_quad(const Vertex2D v[4], float px, float py)

```

```

1621 {
1622     bool saw_positive = false;
1623     bool saw_negative = false;
1624
1625     for (int i = 0; i < 4; i++) {
1626         const Vertex2D *a = &v[i];
1627         const Vertex2D *b = &v[(i + 1) & 3];
1628         float dx = b->x - a->x;
1629         float dy = b->y - a->y;
1630         float cross;
1631
1632         if (fabsf(dx) < 1e-5f && fabsf(dy) < 1e-5f)
1633             continue;
1634
1635         cross = dx * (py - a->y) - dy * (px - a->x);
1636         if (fabsf(cross) <= 1e-4f)
1637             return false;
1638         if (cross > 0.0f)
1639             saw_positive = true;
1640         else
1641             saw_negative = true;
1642
1643         if (saw_positive && saw_negative)
1644             return false;
1645     }
1646
1647     return true;
1648 }
1649
1650 static float quad_min_depth(const Vertex2D v[4])
1651 {
1652     float z = v[0].z;
1653
1654     for (int i = 1; i < 4; i++) {
1655         if (v[i].z < z)
1656             z = v[i].z;
1657     }
1658     return z;
1659 }
1660
1661 static float quad_max_depth(const Vertex2D v[4])
1662 {
1663     float z = v[0].z;
1664
1665     for (int i = 1; i < 4; i++) {
1666         if (v[i].z > z)
1667             z = v[i].z;
1668     }
1669     return z;
1670 }
1671
1672 static void occlusion_tile_range(int x_min, int y_min, int x_max, int y_max,
1673                                int *tx_min, int *ty_min,
1674                                int *tx_max, int *ty_max)
1675 {
1676     *tx_min = x_min / OCCLUSION_TILE_SIZE;
1677     *ty_min = y_min / OCCLUSION_TILE_SIZE;
1678     *tx_max = x_max / OCCLUSION_TILE_SIZE;
1679     *ty_max = y_max / OCCLUSION_TILE_SIZE;

```

```

1680
1681     if (*tx_min < 0) *tx_min = 0;
1682     if (*ty_min < 0) *ty_min = 0;
1683     if (*tx_max >= (int)OCCLUSION_TILES_X)
1684         *tx_max = (int)OCCLUSION_TILES_X - 1;
1685     if (*ty_max >= (int)OCCLUSION_TILES_Y)
1686         *ty_max = (int)OCCLUSION_TILES_Y - 1;
1687 }
1688
1689 static bool occlusion_quad_hidden(RenderContext *ctx, const Vertex2D v[4],
1690                                 int x_min, int y_min, int x_max, int y_max)
1691 {
1692     int tx_min, ty_min, tx_max, ty_max;
1693     float nearest_z = quad_min_depth(v);
1694
1695     occlusion_tile_range(x_min, y_min, x_max, y_max,
1696                        &tx_min, &ty_min, &tx_max, &ty_max);
1697
1698     for (int ty = ty_min; ty <= ty_max; ty++) {
1699         for (int tx = tx_min; tx <= tx_max; tx++) {
1700             int idx = occlusion_tile_index(tx, ty);
1701
1702             if (!ctx->occlusion_covered[idx])
1703                 return false;
1704             if (ctx->occlusion_depth[idx] + OCCLUSION_DEPTH_EPSILON >= nearest_z)
1705                 return false;
1706         }
1707     }
1708
1709     return true;
1710 }
1711
1712 static bool occlusion_quad_covers_tile(const Vertex2D v[4], int tx, int ty)
1713 {
1714     float x0 = (float)(tx * OCCLUSION_TILE_SIZE) + 0.5f;
1715     float y0 = (float)(ty * OCCLUSION_TILE_SIZE) + 0.5f;
1716     float x1 = (float)((tx + 1) * OCCLUSION_TILE_SIZE) - 0.5f;
1717     float y1 = (float)((ty + 1) * OCCLUSION_TILE_SIZE) - 0.5f;
1718
1719     if (x1 > SCREEN_WIDTH - 0.5f)
1720         x1 = SCREEN_WIDTH - 0.5f;
1721     if (y1 > SCREEN_HEIGHT - 0.5f)
1722         y1 = SCREEN_HEIGHT - 0.5f;
1723
1724     return point_inside_convex_quad(v, x0, y0) &&
1725            point_inside_convex_quad(v, x1, y0) &&
1726            point_inside_convex_quad(v, x0, y1) &&
1727            point_inside_convex_quad(v, x1, y1);
1728 }
1729
1730 static void occlusion_record_quad(RenderContext *ctx, const Vertex2D v[4],
1731                                 int x_min, int y_min, int x_max, int y_max)
1732 {
1733     int tx_min, ty_min, tx_max, ty_max;
1734     float farthest_z = quad_max_depth(v);
1735
1736     occlusion_tile_range(x_min, y_min, x_max, y_max,
1737                        &tx_min, &ty_min, &tx_max, &ty_max);
1738

```

```

1739     for (int ty = ty_min; ty <= ty_max; ty++) {
1740         for (int tx = tx_min; tx <= tx_max; tx++) {
1741             int idx;
1742
1743             if (!occlusion_quad_covers_tile(v, tx, ty))
1744                 continue;
1745
1746             idx = occlusion_tile_index(tx, ty);
1747             if (!ctx->occlusion_covered[idx] ||
1748                 farthest_z < ctx->occlusion_depth[idx]) {
1749                 ctx->occlusion_covered[idx] = 1;
1750                 ctx->occlusion_depth[idx] = farthest_z;
1751                 ctx->occlusion_tiles_written++;
1752             }
1753         }
1754     }
1755 }
1756
1757 static bool projected_quad_fully_inside_viewport(const Vertex2D v[4])
1758 {
1759     for (int i = 0; i < 4; i++) {
1760         if (v[i].x < VIEW_X_MIN || v[i].x > VIEW_X_MAX ||
1761             v[i].y < VIEW_Y_MIN || v[i].y > VIEW_Y_MAX)
1762             return false;
1763     }
1764     return true;
1765 }
1766
1767 static bool stage_projected_quad_no_clip(RenderContext *ctx,
1768                                         const RenderQuad *quad)
1769 {
1770     Vertex2D verts[4];
1771
1772     memcpy(verts, quad->vertices, sizeof(verts));
1773     return stage_projected_polygon(ctx, quad, verts, 4);
1774 }
1775
1776 static bool emit_camera_quad(RenderContext *ctx, const CameraVertex verts_cam[4],
1777                             uint8_t texture_id, uint8_t extra_flags)
1778 {
1779     Vertex2D verts[4];
1780     uint8_t flags = QUAD_FLAG_ZTEST | QUAD_FLAG_TEX | extra_flags;
1781
1782     for (int i = 0; i < 4; i++) {
1783         if (!project_camera_vertex(ctx, &verts_cam[i], &verts[i]))
1784             return false;
1785     }
1786
1787     /* Most world quads are already on-screen; avoid the general clipper. */
1788     if (projected_quad_fully_inside_viewport(verts)) {
1789         RenderQuad q = {0};
1790
1791         memcpy(q.vertices, verts, sizeof(verts));
1792         q.texture_id = texture_id;
1793         q.flags = flags;
1794         return stage_projected_quad_no_clip(ctx, &q);
1795     }
1796
1797     RenderQuad q = {0};

```

```

1798     memcpy(q.vertices, verts, sizeof(verts));
1799     q.texture_id = texture_id;
1800     q.flags = flags;
1801     return renderer_push_quad(ctx, &q);
1802 }
1803
1804 static bool emit_camera_quad_color(RenderContext *ctx,
1805                                   const CameraVertex verts_cam[4],
1806                                   uint8_t color,
1807                                   uint8_t extra_flags)
1808 {
1809     Vertex2D verts[4];
1810     uint8_t flags = QUAD_FLAG_ZTEST | extra_flags;
1811
1812     for (int i = 0; i < 4; i++) {
1813         if (!project_camera_vertex(ctx, &verts_cam[i], &verts[i]))
1814             return false;
1815     }
1816
1817     if (projected_quad_fully_inside_viewport(verts)) {
1818         RenderQuad q = {0};
1819
1820         memcpy(q.vertices, verts, sizeof(verts));
1821         q.color_tint = color;
1822         q.flags = flags;
1823         return stage_projected_quad_no_clip(ctx, &q);
1824     }
1825
1826     RenderQuad q = {0};
1827     memcpy(q.vertices, verts, sizeof(verts));
1828     q.color_tint = color;
1829     q.flags = flags;
1830     return renderer_push_quad(ctx, &q);
1831 }
1832
1833 static void emit_clipped_face(RenderContext *ctx, const CameraVertex *poly,
1834                              int count, uint8_t texture_id, uint8_t extra_flags)
1835 {
1836     if (count < 3)
1837         return;
1838
1839     if (count == 3) {
1840         CameraVertex tri[4] = { poly[0], poly[1], poly[2], poly[2] };
1841         emit_camera_quad(ctx, tri, texture_id, extra_flags);
1842         return;
1843     }
1844
1845     if (count == 4) {
1846         CameraVertex quad[4] = { poly[0], poly[1], poly[2], poly[3] };
1847         emit_camera_quad(ctx, quad, texture_id, extra_flags);
1848         return;
1849     }
1850
1851     for (int i = 1; i + 1 < count; i++) {
1852         CameraVertex tri[4] = { poly[0], poly[i], poly[i + 1], poly[i + 1] };
1853         emit_camera_quad(ctx, tri, texture_id, extra_flags);
1854     }
1855 }
1856

```

```

1857 static void emit_clipped_face_color(RenderContext *ctx,
1858                                 const CameraVertex *poly,
1859                                 int count,
1860                                 uint8_t color,
1861                                 uint8_t extra_flags)
1862 {
1863     if (count < 3)
1864         return;
1865
1866     if (count == 3) {
1867         CameraVertex tri[4] = { poly[0], poly[1], poly[2], poly[2] };
1868         emit_camera_quad_color(ctx, tri, color, extra_flags);
1869         return;
1870     }
1871
1872     if (count == 4) {
1873         CameraVertex quad[4] = { poly[0], poly[1], poly[2], poly[3] };
1874         emit_camera_quad_color(ctx, quad, color, extra_flags);
1875         return;
1876     }
1877
1878     for (int i = 1; i + 1 < count; i++) {
1879         CameraVertex tri[4] = { poly[0], poly[i], poly[i + 1], poly[i + 1] };
1880         emit_camera_quad_color(ctx, tri, color, extra_flags);
1881     }
1882 }
1883
1884 static uint8_t choose_face_texture_lod(const RenderContext *ctx,
1885                                     uint8_t base_tile,
1886                                     const CameraVertex face_cam[4])
1887 {
1888     (void)ctx;
1889     static int texture_lod_enabled = -1;
1890
1891     if (texture_lod_enabled < 0)
1892         texture_lod_enabled = env_flag("VOXEL_TEXTURE_LOD", true) ? 1 : 0;
1893     if (!texture_lod_enabled)
1894         return base_tile;
1895
1896     float nearest_z = face_cam[0].z;
1897     int lod = 0;
1898
1899     for (int i = 1; i < 4; i++) {
1900         if (face_cam[i].z < nearest_z)
1901             nearest_z = face_cam[i].z;
1902     }
1903
1904     /*
1905     * Use a conservative forward-distance rule instead of projected-span
1906     * math. The old span-based selector was more expensive and could choose
1907     * low-res tiles for nearby but highly oblique faces because one projected
1908     * axis collapsed. These thresholds push mip usage farther out so nearby
1909     * surfaces stay crisp and runtime overhead stays tiny.
1910     */
1911     if (nearest_z > 44.0f)
1912         lod = 2;
1913     else if (nearest_z > 28.0f)
1914         lod = 1;
1915

```

```

1916     return texture_lod_tile_id(base_tile, lod);
1917 }
1918
1919 static uint8_t choose_face_light_flags(const RenderContext *ctx, BlockFace face)
1920 {
1921     return ctx->face_light_flags[face];
1922 }
1923
1924 static void merged_face_vertices(Vec3 block_pos, BlockFace face,
1925                                 int u_size, int v_size,
1926                                 uint8_t height_eighths, Vec3 out[4])
1927 {
1928     float u = (float)u_size;
1929     float v = (float)v_size;
1930     float x = block_pos.x;
1931     float y = block_pos.y;
1932     float z = block_pos.z;
1933     /* "Top" Y of the face. For full-height blocks (height_eighths == 8)
1934      * this is exactly y + v, which keeps the existing greedy-merge math
1935      * untouched. For partial-height water (height_eighths < 8) v_size is
1936      * guaranteed to be 1 (translucent never merges), so the partial top
1937      * sits at y + height/8. */
1938     float top_y = y + (v - 1.0f) + (float)height_eighths * (1.0f / 8.0f);
1939     float top_y_only = y + (float)height_eighths * (1.0f / 8.0f);
1940
1941     switch (face) {
1942     case FACE_TOP:
1943         out[0] = (Vec3){ x,    top_y_only, z + v };
1944         out[1] = (Vec3){ x + u, top_y_only, z + v };
1945         out[2] = (Vec3){ x + u, top_y_only, z    };
1946         out[3] = (Vec3){ x,    top_y_only, z    };
1947         break;
1948     case FACE_BOTTOM:
1949         out[0] = (Vec3){ x,    y, z    };
1950         out[1] = (Vec3){ x + u, y, z    };
1951         out[2] = (Vec3){ x + u, y, z + v };
1952         out[3] = (Vec3){ x,    y, z + v };
1953         break;
1954     case FACE_LEFT:
1955         out[0] = (Vec3){ x, y,    z    };
1956         out[1] = (Vec3){ x, y,    z + u };
1957         out[2] = (Vec3){ x, top_y, z + u };
1958         out[3] = (Vec3){ x, top_y, z    };
1959         break;
1960     case FACE_RIGHT:
1961         out[0] = (Vec3){ x + 1, y,    z + u };
1962         out[1] = (Vec3){ x + 1, y,    z    };
1963         out[2] = (Vec3){ x + 1, top_y, z    };
1964         out[3] = (Vec3){ x + 1, top_y, z + u };
1965         break;
1966     case FACE_FRONT:
1967         out[0] = (Vec3){ x,    y,    z };
1968         out[1] = (Vec3){ x + u, y,    z };
1969         out[2] = (Vec3){ x + u, top_y, z };
1970         out[3] = (Vec3){ x,    top_y, z };
1971         break;
1972     case FACE_BACK:
1973         out[0] = (Vec3){ x + u, y,    z + 1 };
1974         out[1] = (Vec3){ x,    y,    z + 1 };

```

```

1975     out[2] = (Vec3){ x,    top_y, z + 1 };
1976     out[3] = (Vec3){ x + u, top_y, z + 1 };
1977     break;
1978     default:
1979         for (int i = 0; i < 4; i++)
1980             out[i] = block_pos;
1981     break;
1982 }
1983 }
1984
1985 /*
1986  * Runtime toggle for merged quad emission with QUAD_TEX_REPEAT_UV. Default ON:
1987  * far chunks were greedily meshed specifically to keep the descriptor stream
1988  * small enough for steady input/frame pacing. Set VOXEL_MERGE_FAR_QUADS=0 to
1989  * fall back to unit-quad expansion for visual A/B testing.
1990  */
1991 static bool merged_emit_repeat_uv_enabled(void)
1992 {
1993     static int cached = -1;
1994
1995     if (cached < 0)
1996         cached = env_flag_fallback("VOXEL_MERGE_FAR_QUADS",
1997                                   "BLOCK_GAME_MERGE_FAR_QUADS",
1998                                   true) ? 1 : 0;
1999     return cached != 0;
2000 }
2001
2002 static Vec3 merged_face_sub_origin(Vec3 block_pos, BlockFace face,
2003                                   int du, int dv)
2004 {
2005     switch (face) {
2006     case FACE_TOP:
2007     case FACE_BOTTOM:
2008         block_pos.x += (float)du;
2009         block_pos.z += (float)dv;
2010         break;
2011     case FACE_LEFT:
2012     case FACE_RIGHT:
2013         block_pos.z += (float)du;
2014         block_pos.y += (float)dv;
2015         break;
2016     case FACE_FRONT:
2017     case FACE_BACK:
2018         block_pos.x += (float)du;
2019         block_pos.y += (float)dv;
2020         break;
2021     default:
2022         break;
2023     }
2024     return block_pos;
2025 }
2026
2027 static void emit_merged_block_face_lit(RenderContext *ctx, BlockID type,
2028                                       Vec3 block_pos, BlockFace face,
2029                                       int u_size, int v_size,
2030                                       uint8_t height_eighths,
2031                                       uint8_t light_flags);
2032
2033 static void expand_merged_face_to_unit_quads(RenderContext *ctx, BlockID type,

```

```

2034         Vec3 block_pos, BlockFace face,
2035         int u_size, int v_size,
2036         uint8_t light_flags)
2037     {
2038         for (int dv = 0; dv < v_size; dv++) {
2039             for (int du = 0; du < u_size; du++) {
2040                 Vec3 unit = merged_face_sub_origin(block_pos, face, du, dv);
2041                 emit_merged_block_face_lit(ctx, type, unit, face,
2042                     1, 1, 8, light_flags);
2043                 if (ctx->n_quads >= MAX_QUADS_IN_FLIGHT)
2044                     return;
2045             }
2046         }
2047     }
2048
2049     static void emit_merged_block_face_lit(RenderContext *ctx, BlockID type,
2050         Vec3 block_pos, BlockFace face,
2051         int u_size, int v_size,
2052         uint8_t height_eighths,
2053         uint8_t light_flags)
2054     {
2055         static const float tile_span = 16.0f;
2056         Vec3 face_world[4];
2057         CameraVertex face_cam[4];
2058         uint8_t face_flags;
2059         uint8_t texture_id;
2060         uint8_t base_tile;
2061
2062         if (type == BLOCK_AIR)
2063             return;
2064         if (face < 0 || face >= NUM_FACES)
2065             return;
2066
2067         /*
2068          * A/B fallback: decomposing a merged run avoids QUAD_TEX_REPEAT_UV entirely,
2069          * but it is intentionally not the default because it can multiply the far
2070          * terrain descriptor count and make input pacing noticeably worse.
2071          */
2072         if ((u_size > 1 || v_size > 1) && !merged_emit_repeat_uv_enabled()) {
2073             /*
2074              * Unit emits below re-run is_face_visible() with their own centers,
2075              * so we don't pre-cull the anchor here: a merged run can straddle a
2076              * face whose center is behind the camera while individual cells are
2077              * not, and vice versa. The per-unit check is the same one used in
2078              * the non-merged path.
2079              */
2080             expand_merged_face_to_unit_quads(ctx, type, block_pos, face,
2081                 u_size, v_size, light_flags);
2082             return;
2083         }
2084
2085         if (!merged_face_visible(block_pos, face, u_size, v_size,
2086             height_eighths, ctx->current_camera.position))
2087             return;
2088
2089         merged_face_vertices(block_pos, face, u_size, v_size, height_eighths, face_world);
2090         for (int i = 0; i < 4; i++) {
2091             world_to_camera(ctx, face_world[i], &face_cam[i]);
2092             face_cam[i].u = (i == 1 || i == 2) ? tile_span * (float)u_size : 0.0f;

```

```

2093     /*
2094     * Atlas row 0 sits at the top of each tile image, and the side faces'
2095     * vertex 0 is world-bottom. Invert V so world-top maps to texture-top.
2096     */
2097     face_cam[i].v = (i == 2 || i == 3) ? 0.0f : tile_span * (float)v_size;
2098 }
2099 if (camera_quad_outside_view(ctx, face_cam))
2100     return;
2101
2102 base_tile = block_face_texture_id(type, face);
2103 texture_id = choose_face_texture_lod(ctx, base_tile, face_cam);
2104 if (u_size > 1 || v_size > 1)
2105     texture_id |= QUAD_TEX_REPEAT_UV;
2106 face_flags = light_flags | QUAD_FLAG_FOG;
2107 /*
2108 * Translucent blocks (glass) ride the hardware alpha-blend path. 50% src
2109 * / 50% dst gives an obviously see-through look while still leaving the
2110 * glass texture itself clearly readable. We keep ZTEST on so the glass
2111 * occludes things behind it at the correct depth; because chunks are
2112 * drawn nearest-first the destination pixel will usually already hold an
2113 * opaque fragment that the blend can mix against.
2114 */
2115 if (block_is_translucent(type))
2116     face_flags |= QUAD_ALPHA_50;
2117 /*
2118 * Cutout textures (leaves, flowers, doors, torches) carry palette index 0
2119 * for their holes. Without ALPHA_KEY those texels render as the palette-0
2120 * background color and look like solid sky-tinted blobs instead of holes.
2121 */
2122 if (block_is_alpha_keyed(type))
2123     face_flags |= QUAD_FLAG_ALPHA_KEY;
2124
2125 CameraVertex clipped[6];
2126 int clipped_count = clip_face_to_near_plane(face_cam, clipped);
2127 emit_clipped_face(ctx, clipped, clipped_count, texture_id, face_flags);
2128 }
2129
2130 static void cross_face_vertices(Vec3 block_pos, uint8_t face,
2131                               Vec3 out[4])
2132 {
2133     const float inset = 0.08f;
2134     float x = block_pos.x;
2135     float y = block_pos.y;
2136     float z = block_pos.z;
2137
2138     if (face == CHUNK_FACE_CROSS_B) {
2139         out[0] = (Vec3){ x + 1.0f - inset, y,          z + inset };
2140         out[1] = (Vec3){ x + inset,      y,          z + 1.0f - inset };
2141         out[2] = (Vec3){ x + inset,      y + 1.0f,  z + 1.0f - inset };
2142         out[3] = (Vec3){ x + 1.0f - inset, y + 1.0f,  z + inset };
2143         return;
2144     }
2145
2146     out[0] = (Vec3){ x + inset,          y,          z + inset };
2147     out[1] = (Vec3){ x + 1.0f - inset, y,          z + 1.0f - inset };
2148     out[2] = (Vec3){ x + 1.0f - inset, y + 1.0f,  z + 1.0f - inset };
2149     out[3] = (Vec3){ x + inset,          y + 1.0f,  z + inset };
2150 }
2151

```

```

2152 static void emit_cross_block_face_lit(RenderContext *ctx, BlockID type,
2153                                     Vec3 block_pos, uint8_t face,
2154                                     uint8_t light_flags)
2155 {
2156     static const float tile_span = 16.0f;
2157     Vec3 face_world[4];
2158     CameraVertex face_cam[4];
2159     uint8_t texture_id;
2160     uint8_t face_flags = light_flags | QUAD_FLAG_FOG | QUAD_FLAG_ALPHA_KEY;
2161
2162     if (type == BLOCK_AIR)
2163         return;
2164
2165     cross_face_vertices(block_pos, face, face_world);
2166     for (int i = 0; i < 4; i++) {
2167         world_to_camera(ctx, face_world[i], &face_cam[i]);
2168         face_cam[i].u = (i == 1 || i == 2) ? tile_span : 0.0f;
2169         face_cam[i].v = (i == 2 || i == 3) ? 0.0f : tile_span;
2170     }
2171     if (camera_quad_outside_view(ctx, face_cam))
2172         return;
2173
2174     texture_id = choose_face_texture_lod(ctx,
2175                                         block_face_texture_id(type, FACE_FRONT),
2176                                         face_cam);
2177
2178     CameraVertex clipped[6];
2179     int clipped_count = clip_face_to_near_plane(face_cam, clipped);
2180     emit_clipped_face(ctx, clipped, clipped_count, texture_id, face_flags);
2181 }
2182
2183 static void emit_model_quad_lit(RenderContext *ctx,
2184                                Vec3 face_world[4], uint8_t texture_tile,
2185                                uint8_t light_flags);
2186 static void emit_model_quad_lit_rotated(RenderContext *ctx,
2187                                         Vec3 face_world[4],
2188                                         uint8_t texture_tile,
2189                                         uint8_t light_flags,
2190                                         int uv_rotation);
2191
2192 static void flat_face_vertices(Vec3 block_pos,
2193                               BlockID type,
2194                               Vec3 out[4])
2195 {
2196     float inset = 0.05f;
2197     float y = block_pos.y + 0.025f;
2198
2199     if (type == BLOCK_BUTTON) {
2200         inset = 0.24f;
2201         y = block_pos.y + 0.085f;
2202     } else if (type == BLOCK_BUTTON_PRESSED) {
2203         inset = 0.29f;
2204         y = block_pos.y + 0.025f;
2205     } else if (block_is_pressure_plate(type) &&
2206              !block_pressure_plate_powered(type)) {
2207         inset = 0.08f;
2208         y = block_pos.y + 0.055f;
2209     } else if (block_pressure_plate_powered(type)) {
2210         inset = 0.08f;

```

```

2211     y = block_pos.y + 0.015f;
2212 }
2213
2214 out[0] = (Vec3){ block_pos.x + inset,      y,
2215                block_pos.z + inset };
2216 out[1] = (Vec3){ block_pos.x + 1.0f - inset, y,
2217                block_pos.z + inset };
2218 out[2] = (Vec3){ block_pos.x + 1.0f - inset, y,
2219                block_pos.z + 1.0f - inset };
2220 out[3] = (Vec3){ block_pos.x + inset,      y,
2221                block_pos.z + 1.0f - inset };
2222 }
2223
2224 enum {
2225     REDSTONE_WIRE_N = 1u << 0,
2226     REDSTONE_WIRE_E = 1u << 1,
2227     REDSTONE_WIRE_S = 1u << 2,
2228     REDSTONE_WIRE_W = 1u << 3,
2229 };
2230
2231 static bool render_block_is_redstone_wire(BlockID id)
2232 {
2233     return id == BLOCK_REDSTONE_WIRE_UNCONNECTED ||
2234            id == BLOCK_REDSTONE_WIRE_OFF ||
2235            id == BLOCK_REDSTONE_WIRE_ON;
2236 }
2237
2238 static bool render_block_is_redstone_component(BlockID id)
2239 {
2240     return render_block_is_redstone_wire(id) ||
2241            id == BLOCK_REDSTONE_TORCH_OFF ||
2242            id == BLOCK_REDSTONE_TORCH_ON ||
2243            block_is_repeater(id) ||
2244            block_is_comparator(id) ||
2245            block_is_lever(id) ||
2246            block_is_pressure_plate(id) ||
2247            id == BLOCK_LAMP_OFF ||
2248            id == BLOCK_LAMP ||
2249            id == BLOCK_REDSTONE_BLOCK ||
2250            id == BLOCK_BUTTON ||
2251            id == BLOCK_BUTTON_PRESSED;
2252 }
2253
2254 static bool render_redstone_wire_connects_to(BlockID id)
2255 {
2256     if (render_block_is_redstone_component(id))
2257         return true;
2258     return id != BLOCK_AIR &&
2259            block_render_model(id) == BLOCK_RENDER_CUBE &&
2260            !block_is_passable(id);
2261 }
2262
2263 static uint8_t render_redstone_wire_mask(const VoxelWorld *world,
2264                                         int wx,
2265                                         int wy,
2266                                         int wz)
2267 {
2268     uint8_t mask = 0;
2269

```

```

2270     if (render_redstone_wire_connects_to(world_get_block(world, wx, wy, wz - 1)))
2271         mask |= REDSTONE_WIRE_N;
2272     if (render_redstone_wire_connects_to(world_get_block(world, wx + 1, wy, wz)))
2273         mask |= REDSTONE_WIRE_E;
2274     if (render_redstone_wire_connects_to(world_get_block(world, wx, wy, wz + 1)))
2275         mask |= REDSTONE_WIRE_S;
2276     if (render_redstone_wire_connects_to(world_get_block(world, wx - 1, wy, wz)))
2277         mask |= REDSTONE_WIRE_W;
2278     return mask;
2279 }
2280
2281 static int redstone_mask_bit_count(uint8_t mask)
2282 {
2283     int count = 0;
2284
2285     for (int bit = 0; bit < 4; bit++) {
2286         if (mask & (1u << bit))
2287             count++;
2288     }
2289     return count;
2290 }
2291
2292 static uint8_t redstone_wire_texture_for_mask(BlockID type,
2293                                             uint8_t mask,
2294                                             int *uv_rotation_out)
2295 {
2296     bool powered = type == BLOCK_REDSTONE_WIRE_ON;
2297     int count = redstone_mask_bit_count(mask);
2298
2299     if (uv_rotation_out)
2300         *uv_rotation_out = 0;
2301
2302     if (count == 0)
2303         return powered ? TEX_TILE_REDSTONE_WIRE_DOT_ON :
2304             TEX_TILE_REDSTONE_WIRE_UNCONNECTED;
2305
2306     if (count == 1 ||
2307         mask == (REDSTONE_WIRE_E | REDSTONE_WIRE_W) ||
2308         mask == (REDSTONE_WIRE_N | REDSTONE_WIRE_S)) {
2309         if ((mask & (REDSTONE_WIRE_N | REDSTONE_WIRE_S)) &&
2310             !(mask & (REDSTONE_WIRE_E | REDSTONE_WIRE_W)) &&
2311             uv_rotation_out)
2312             *uv_rotation_out = 1;
2313         return powered ? TEX_TILE_REDSTONE_WIRE_ON :
2314             TEX_TILE_REDSTONE_WIRE_OFF;
2315     }
2316
2317     if (count == 2) {
2318         if (uv_rotation_out) {
2319             /*
2320              * Flat top-face UVs map texture north to world south at rotation
2321              * zero, so the canonical texture N+E corner appears as S+E until
2322              * rotated.
2323              */
2324             if (mask == (REDSTONE_WIRE_N | REDSTONE_WIRE_E))
2325                 *uv_rotation_out = 1;
2326             else if (mask == (REDSTONE_WIRE_W | REDSTONE_WIRE_N))
2327                 *uv_rotation_out = 2;
2328             else if (mask == (REDSTONE_WIRE_S | REDSTONE_WIRE_W))

```

```

2329         *uv_rotation_out = 3;
2330     }
2331     return powered ? TEX_TILE_REDSTONE_WIRE_CORNER_ON :
2332         TEX_TILE_REDSTONE_WIRE_CORNER_OFF;
2333 }
2334
2335 if (count == 3) {
2336     if (uv_rotation_out) {
2337         uint8_t missing = (uint8_t)(~mask & 0x0f);
2338
2339         if (missing & REDSTONE_WIRE_W)
2340             *uv_rotation_out = 1;
2341         else if (missing & REDSTONE_WIRE_S)
2342             *uv_rotation_out = 2;
2343         else if (missing & REDSTONE_WIRE_E)
2344             *uv_rotation_out = 3;
2345     }
2346     return powered ? TEX_TILE_REDSTONE_WIRE_T_ON :
2347         TEX_TILE_REDSTONE_WIRE_T_OFF;
2348 }
2349
2350 return powered ? TEX_TILE_REDSTONE_WIRE_CROSS_ON :
2351     TEX_TILE_REDSTONE_WIRE_CROSS_OFF;
2352 }
2353
2354 static int redstone_directional_uv_rotation(BlockID type)
2355 {
2356     /*
2357     * Directional redstone textures in the atlas point toward their bottom edge.
2358     * Repeater source textures are rotated into that convention by the atlas
2359     * generator so repeaters and comparators can share this map.
2360     */
2361     switch (block_redstone_facing(type)) {
2362     case BLOCK_DOOR_FACING_EAST:
2363         return 3;
2364     case BLOCK_DOOR_FACING_SOUTH:
2365         return 2;
2366     case BLOCK_DOOR_FACING_WEST:
2367         return 1;
2368     case BLOCK_DOOR_FACING_NORTH:
2369     default:
2370         return 0;
2371     }
2372 }
2373
2374 static Vec3 redstone_device_local_point(Vec3 block_pos,
2375                                         BlockDoorFacing facing,
2376                                         float side,
2377                                         float forward,
2378                                         float y)
2379 {
2380     int fdx;
2381     int fdz;
2382     float rdx;
2383     float rdz;
2384
2385     switch (facing) {
2386     case BLOCK_DOOR_FACING_EAST:
2387         fdx = 1;

```

```

2388     fdz = 0;
2389     break;
2390 case BLOCK_DOOR_FACING_SOUTH:
2391     fdx = 0;
2392     fdz = 1;
2393     break;
2394 case BLOCK_DOOR_FACING_WEST:
2395     fdx = -1;
2396     fdz = 0;
2397     break;
2398 case BLOCK_DOOR_FACING_NORTH:
2399 default:
2400     fdx = 0;
2401     fdz = -1;
2402     break;
2403 }
2404
2405 rdx = (float)-fdz;
2406 rdz = (float)fdx;
2407 return (Vec3){
2408     block_pos.x + 0.5f + rdx * side + (float)fdx * forward,
2409     block_pos.y + y,
2410     block_pos.z + 0.5f + rdz * side + (float)fdz * forward,
2411 };
2412 }
2413
2414 static void redstone_device_plate_top_vertices(Vec3 block_pos, Vec3 out[4])
2415 {
2416     const float inset = REDSTONE_DEVICE_PLATE_INSET;
2417     const float y = 0.080f;
2418
2419     out[0] = (Vec3){ block_pos.x + inset,          block_pos.y + y,
2420                    block_pos.z + inset };
2421     out[1] = (Vec3){ block_pos.x + 1.0f - inset, block_pos.y + y,
2422                    block_pos.z + inset };
2423     out[2] = (Vec3){ block_pos.x + 1.0f - inset, block_pos.y + y,
2424                    block_pos.z + 1.0f - inset };
2425     out[3] = (Vec3){ block_pos.x + inset,          block_pos.y + y,
2426                    block_pos.z + 1.0f - inset };
2427 }
2428
2429 static void emit_redstone_device_plate_side(RenderContext *ctx,
2430                                             Vec3 block_pos,
2431                                             BlockDoorFacing facing,
2432                                             int side_index,
2433                                             uint8_t light_flags)
2434 {
2435     const float half_w = REDSTONE_DEVICE_PLATE_HALF;
2436     const float half_l = REDSTONE_DEVICE_PLATE_HALF;
2437     const float bottom = 0.018f;
2438     const float top = 0.080f;
2439     Vec3 face_world[4];
2440
2441     switch (side_index) {
2442     case 0:
2443         face_world[0] = redstone_device_local_point(block_pos, facing,
2444                                                    -half_w, -half_l, bottom);
2445         face_world[1] = redstone_device_local_point(block_pos, facing,
2446                                                    half_w, -half_l, bottom);

```

```

2447     face_world[2] = redstone_device_local_point(block_pos, facing,
2448                                                  half_w, -half_l, top);
2449     face_world[3] = redstone_device_local_point(block_pos, facing,
2450                                                  -half_w, -half_l, top);
2451     break;
2452 case 1:
2453     face_world[0] = redstone_device_local_point(block_pos, facing,
2454                                                  half_w, -half_l, bottom);
2455     face_world[1] = redstone_device_local_point(block_pos, facing,
2456                                                  half_w, half_l, bottom);
2457     face_world[2] = redstone_device_local_point(block_pos, facing,
2458                                                  half_w, half_l, top);
2459     face_world[3] = redstone_device_local_point(block_pos, facing,
2460                                                  half_w, -half_l, top);
2461     break;
2462 case 2:
2463     face_world[0] = redstone_device_local_point(block_pos, facing,
2464                                                  half_w, half_l, bottom);
2465     face_world[1] = redstone_device_local_point(block_pos, facing,
2466                                                  -half_w, half_l, bottom);
2467     face_world[2] = redstone_device_local_point(block_pos, facing,
2468                                                  -half_w, half_l, top);
2469     face_world[3] = redstone_device_local_point(block_pos, facing,
2470                                                  half_w, half_l, top);
2471     break;
2472 default:
2473     face_world[0] = redstone_device_local_point(block_pos, facing,
2474                                                  -half_w, half_l, bottom);
2475     face_world[1] = redstone_device_local_point(block_pos, facing,
2476                                                  -half_w, -half_l, bottom);
2477     face_world[2] = redstone_device_local_point(block_pos, facing,
2478                                                  -half_w, -half_l, top);
2479     face_world[3] = redstone_device_local_point(block_pos, facing,
2480                                                  -half_w, half_l, top);
2481     break;
2482 }
2483
2484 emit_model_quad_lit(ctx, face_world, TEX_TILE_STONE, light_flags);
2485 }
2486
2487 static void emit_redstone_device_post(RenderContext *ctx,
2488                                       Vec3 block_pos,
2489                                       BlockDoorFacing facing,
2490                                       float side,
2491                                       float forward,
2492                                       uint8_t texture_tile,
2493                                       uint8_t light_flags)
2494 {
2495     const float half = REDSTONE_DEVICE_POST_HALF;
2496     const float bottom = 0.080f;
2497     const float top = 0.430f;
2498     Vec3 face_world[4];
2499
2500     face_world[0] = redstone_device_local_point(block_pos, facing,
2501                                                  side - half, forward, bottom);
2502     face_world[1] = redstone_device_local_point(block_pos, facing,
2503                                                  side + half, forward, bottom);
2504     face_world[2] = redstone_device_local_point(block_pos, facing,
2505                                                  side + half, forward, top);

```

```

2506     face_world[3] = redstone_device_local_point(block_pos, facing,
2507                                                  side - half, forward, top);
2508     emit_model_quad_lit(ctx, face_world, texture_tile, light_flags);
2509
2510     face_world[0] = redstone_device_local_point(block_pos, facing,
2511                                                  side, forward - half, bottom);
2512     face_world[1] = redstone_device_local_point(block_pos, facing,
2513                                                  side, forward + half, bottom);
2514     face_world[2] = redstone_device_local_point(block_pos, facing,
2515                                                  side, forward + half, top);
2516     face_world[3] = redstone_device_local_point(block_pos, facing,
2517                                                  side, forward - half, top);
2518     emit_model_quad_lit(ctx, face_world, texture_tile, light_flags);
2519 }
2520
2521 static float repeater_moving_post_forward(uint8_t delay_ticks)
2522 {
2523     static const float forward_texels_by_delay[4] = {
2524         4.0f, 2.0f, 0.0f, -2.0f,
2525     };
2526
2527     if (delay_ticks < 1)
2528         delay_ticks = 1;
2529     if (delay_ticks > 4)
2530         delay_ticks = 4;
2531
2532     /*
2533      * Move the adjustable torch two texture texels toward the input side per
2534      * delay step. Delay 1 should sit close to the fixed torch, then walk back
2535      * along the inset red strip as the delay increases.
2536      */
2537     return forward_texels_by_delay[delay_ticks - 1u] *
2538            REDSTONE_DEVICE_PLATE_TEXEL;
2539 }
2540
2541 static void emit_redstone_device_block_lit(RenderContext *ctx,
2542                                           BlockID type,
2543                                           Vec3 block_pos,
2544                                           uint8_t visual_state,
2545                                           uint8_t light_flags)
2546 {
2547     Vec3 face_world[4];
2548     BlockDoorFacing facing = block_redstone_facing(type);
2549     uint8_t top_tile = block_face_texture_id(type, FACE_TOP);
2550     uint8_t post_tile = block_redstone_directional_powered(type) ?
2551         TEX_TILE_REDSTONE_TORCH_ON :
2552         TEX_TILE_REDSTONE_TORCH_OFF;
2553     int uv_rotation = redstone_directional_uv_rotation(type);
2554
2555     redstone_device_plate_top_vertices(block_pos, face_world);
2556     emit_model_quad_lit_rotated(ctx, face_world, top_tile, light_flags,
2557                                uv_rotation);
2558     for (int side = 0; side < 4; side++)
2559         emit_redstone_device_plate_side(ctx, block_pos, facing, side,
2560                                       light_flags);
2561
2562     if (block_is_comparator(type)) {
2563         emit_redstone_device_post(ctx, block_pos, facing,
2564                                  -0.18f, -0.16f, post_tile, light_flags);

```

```

2565     emit_redstone_device_post(ctx, block_pos, facing,
2566                               0.18f, -0.16f, post_tile, light_flags);
2567     emit_redstone_device_post(ctx, block_pos, facing,
2568                               0.0f, 0.20f, post_tile, light_flags);
2569 } else {
2570     float moving_post_forward;
2571
2572     moving_post_forward = repeater_moving_post_forward(visual_state);
2573     emit_redstone_device_post(ctx, block_pos, facing,
2574                               0.0f, 5.0f * REDSTONE_DEVICE_PLATE_TEXEL,
2575                               post_tile, light_flags);
2576     emit_redstone_device_post(ctx, block_pos, facing,
2577                               0.0f, moving_post_forward,
2578                               post_tile, light_flags);
2579 }
2580 }
2581
2582 static void emit_flat_block_face_lit(RenderContext *ctx, BlockID type,
2583                                     const VoxelWorld *world,
2584                                     Vec3 block_pos,
2585                                     int wx,
2586                                     int wy,
2587                                     int wz,
2588                                     uint8_t visual_state,
2589                                     uint8_t light_flags)
2590 {
2591     Vec3 face_world[4];
2592     uint8_t texture_tile;
2593     int uv_rotation = 0;
2594
2595     if (type == BLOCK_AIR)
2596         return;
2597
2598     if (block_is_redstone_directional(type)) {
2599         if (block_is_repeater(type) && world) {
2600             uint8_t live_delay = world_repeater_delay_ticks(world, wx, wy, wz);
2601
2602             if (live_delay)
2603                 visual_state = live_delay;
2604         }
2605         emit_redstone_device_block_lit(ctx, type, block_pos, visual_state,
2606                                       light_flags);
2607         return;
2608     }
2609
2610     flat_face_vertices(block_pos, type, face_world);
2611     texture_tile = block_face_texture_id(type, FACE_TOP);
2612     if (render_block_is_redstone_wire(type)) {
2613         uint8_t mask = render_redstone_wire_mask(world, wx, wy, wz);
2614
2615         texture_tile = redstone_wire_texture_for_mask(type, mask,
2616                                                       &uv_rotation);
2617     }
2618
2619     emit_model_quad_lit_rotated(ctx, face_world, texture_tile, light_flags,
2620                                uv_rotation);
2621 }
2622
2623 static void torch_axis_for_support(Vec3 block_pos,

```

```

2624         uint8_t support_face,
2625         Vec3 *base_out,
2626         Vec3 *tip_out)
2627 {
2628     const float wall_base = 0.08f;
2629     const float wall_tip = 0.42f;
2630     Vec3 base = { block_pos.x + 0.5f,
2631                 block_pos.y,
2632                 block_pos.z + 0.5f };
2633     Vec3 tip = { block_pos.x + 0.5f,
2634                block_pos.y + 0.86f,
2635                block_pos.z + 0.5f };
2636
2637     switch (support_face) {
2638     case FACE_LEFT:
2639         base.x = block_pos.x + wall_base;
2640         base.y = block_pos.y + 0.22f;
2641         tip.x = block_pos.x + wall_tip;
2642         break;
2643     case FACE_RIGHT:
2644         base.x = block_pos.x + 1.0f - wall_base;
2645         base.y = block_pos.y + 0.22f;
2646         tip.x = block_pos.x + 1.0f - wall_tip;
2647         break;
2648     case FACE_FRONT:
2649         base.z = block_pos.z + wall_base;
2650         base.y = block_pos.y + 0.22f;
2651         tip.z = block_pos.z + wall_tip;
2652         break;
2653     case FACE_BACK:
2654         base.z = block_pos.z + 1.0f - wall_base;
2655         base.y = block_pos.y + 0.22f;
2656         tip.z = block_pos.z + 1.0f - wall_tip;
2657         break;
2658     default:
2659         break;
2660     }
2661
2662     if (base_out)
2663         *base_out = base;
2664     if (tip_out)
2665         *tip_out = tip;
2666 }
2667
2668 static void torch_face_vertices(Vec3 block_pos, uint8_t face,
2669                                uint8_t support_face, Vec3 out[4])
2670 {
2671     const bool side_support =
2672         support_face >= FACE_LEFT && support_face <= FACE_BACK;
2673     const float half = side_support ? 0.18f : 0.24f;
2674     Vec3 base;
2675     Vec3 tip;
2676     float ax;
2677     float az;
2678
2679     torch_axis_for_support(block_pos, support_face, &base, &tip);
2680
2681     if (face == CHUNK_FACE_CROSS_B) {
2682         ax = half;

```

```

2683     az = -half;
2684     out[0] = (Vec3){ base.x + ax, base.y, base.z + az };
2685     out[1] = (Vec3){ base.x - ax, base.y, base.z - az };
2686     out[2] = (Vec3){ tip.x - ax, tip.y, tip.z - az };
2687     out[3] = (Vec3){ tip.x + ax, tip.y, tip.z + az };
2688     return;
2689 }
2690
2691 ax = half;
2692 az = half;
2693 out[0] = (Vec3){ base.x - ax, base.y, base.z - az };
2694 out[1] = (Vec3){ base.x + ax, base.y, base.z + az };
2695 out[2] = (Vec3){ tip.x + ax, tip.y, tip.z + az };
2696 out[3] = (Vec3){ tip.x - ax, tip.y, tip.z - az };
2697 }
2698
2699 static void door_slab_planes(BlockID type,
2700                             bool *span_x_out,
2701                             float *outer_plane_out,
2702                             float *inner_plane_out)
2703 {
2704     const float thickness = 0.10f;
2705     const float near_edge = 0.05f;
2706     const float far_edge = 0.95f;
2707     BlockDoorFacing facing = block_door_facing(type);
2708     bool open = block_is_door_open(type);
2709     bool span_x = true;
2710     float plane = near_edge;
2711
2712     if (!open) {
2713         switch (facing) {
2714             case BLOCK_DOOR_FACING_EAST:
2715                 span_x = false;
2716                 plane = far_edge;
2717                 break;
2718             case BLOCK_DOOR_FACING_SOUTH:
2719                 span_x = true;
2720                 plane = far_edge;
2721                 break;
2722             case BLOCK_DOOR_FACING_WEST:
2723                 span_x = false;
2724                 plane = near_edge;
2725                 break;
2726             case BLOCK_DOOR_FACING_NORTH:
2727             default:
2728                 span_x = true;
2729                 plane = near_edge;
2730                 break;
2731         }
2732     } else {
2733         switch (facing) {
2734             case BLOCK_DOOR_FACING_EAST:
2735                 span_x = true;
2736                 plane = near_edge;
2737                 break;
2738             case BLOCK_DOOR_FACING_SOUTH:
2739                 span_x = false;
2740                 plane = far_edge;
2741                 break;

```

```

2742     case BLOCK_DOOR_FACING_WEST:
2743         span_x = true;
2744         plane = far_edge;
2745         break;
2746     case BLOCK_DOOR_FACING_NORTH:
2747     default:
2748         span_x = false;
2749         plane = near_edge;
2750         break;
2751     }
2752 }
2753
2754 if (span_x_out)
2755     *span_x_out = span_x;
2756 if (outer_plane_out)
2757     *outer_plane_out = plane;
2758 if (inner_plane_out)
2759     *inner_plane_out = plane + (plane < 0.5f ? thickness : -thickness);
2760 }
2761
2762 static void door_face_vertices(Vec3 block_pos, BlockID type, uint8_t face,
2763                               Vec3 out[4])
2764 {
2765     bool span_x;
2766     float outer_plane;
2767     float inner_plane;
2768     float plane;
2769
2770     door_slab_planes(type, &span_x, &outer_plane, &inner_plane);
2771     plane = face == CHUNK_FACE_CROSS_B ? inner_plane : outer_plane;
2772
2773     if (span_x) {
2774         float z = block_pos.z + plane;
2775
2776         out[0] = (Vec3){ block_pos.x,      block_pos.y,      z };
2777         out[1] = (Vec3){ block_pos.x + 1.0f, block_pos.y,      z };
2778         out[2] = (Vec3){ block_pos.x + 1.0f, block_pos.y + 1.0f, z };
2779         out[3] = (Vec3){ block_pos.x,      block_pos.y + 1.0f, z };
2780         return;
2781     }
2782
2783     {
2784         float x = block_pos.x + plane;
2785
2786         out[0] = (Vec3){ x, block_pos.y,      block_pos.z + 1.0f };
2787         out[1] = (Vec3){ x, block_pos.y,      block_pos.z };
2788         out[2] = (Vec3){ x, block_pos.y + 1.0f, block_pos.z };
2789         out[3] = (Vec3){ x, block_pos.y + 1.0f, block_pos.z + 1.0f };
2790     }
2791 }
2792
2793 static void door_edge_vertices(Vec3 block_pos, BlockID type, int edge,
2794                               Vec3 out[4])
2795 {
2796     bool span_x;
2797     float outer_plane;
2798     float inner_plane;
2799     float min_plane;
2800     float max_plane;

```

```

2801 float x0 = block_pos.x;
2802 float x1 = block_pos.x + 1.0f;
2803 float y0 = block_pos.y;
2804 float y1 = block_pos.y + 1.0f;
2805 float z0 = block_pos.z;
2806 float z1 = block_pos.z + 1.0f;
2807
2808 door_slab_planes(type, &span_x, &outer_plane, &inner_plane);
2809 min_plane = outer_plane < inner_plane ? outer_plane : inner_plane;
2810 max_plane = outer_plane < inner_plane ? inner_plane : outer_plane;
2811
2812 if (span_x) {
2813     float za = block_pos.z + min_plane;
2814     float zb = block_pos.z + max_plane;
2815
2816     switch (edge) {
2817     case 0:
2818         out[0] = (Vec3){ x0, y0, za };
2819         out[1] = (Vec3){ x0, y0, zb };
2820         out[2] = (Vec3){ x0, y1, zb };
2821         out[3] = (Vec3){ x0, y1, za };
2822         return;
2823     case 1:
2824         out[0] = (Vec3){ x1, y0, zb };
2825         out[1] = (Vec3){ x1, y0, za };
2826         out[2] = (Vec3){ x1, y1, za };
2827         out[3] = (Vec3){ x1, y1, zb };
2828         return;
2829     case 2:
2830         out[0] = (Vec3){ x0, y0, za };
2831         out[1] = (Vec3){ x1, y0, za };
2832         out[2] = (Vec3){ x1, y0, zb };
2833         out[3] = (Vec3){ x0, y0, zb };
2834         return;
2835     default:
2836         out[0] = (Vec3){ x0, y1, zb };
2837         out[1] = (Vec3){ x1, y1, zb };
2838         out[2] = (Vec3){ x1, y1, za };
2839         out[3] = (Vec3){ x0, y1, za };
2840         return;
2841     }
2842 }
2843
2844 {
2845     float xa = block_pos.x + min_plane;
2846     float xb = block_pos.x + max_plane;
2847
2848     switch (edge) {
2849     case 0:
2850         out[0] = (Vec3){ xa, y0, z0 };
2851         out[1] = (Vec3){ xb, y0, z0 };
2852         out[2] = (Vec3){ xb, y1, z0 };
2853         out[3] = (Vec3){ xa, y1, z0 };
2854         return;
2855     case 1:
2856         out[0] = (Vec3){ xb, y0, z1 };
2857         out[1] = (Vec3){ xa, y0, z1 };
2858         out[2] = (Vec3){ xa, y1, z1 };
2859         out[3] = (Vec3){ xb, y1, z1 };

```

```

2860         return;
2861     case 2:
2862         out[0] = (Vec3){ xa, y0, z0 };
2863         out[1] = (Vec3){ xb, y0, z0 };
2864         out[2] = (Vec3){ xb, y0, z1 };
2865         out[3] = (Vec3){ xa, y0, z1 };
2866         return;
2867     default:
2868         out[0] = (Vec3){ xa, y1, z1 };
2869         out[1] = (Vec3){ xb, y1, z1 };
2870         out[2] = (Vec3){ xb, y1, z0 };
2871         out[3] = (Vec3){ xa, y1, z0 };
2872         return;
2873     }
2874 }
2875 }
2876
2877 static void emit_model_quad_lit(RenderContext *ctx,
2878                               Vec3 face_world[4], uint8_t texture_tile,
2879                               uint8_t light_flags)
2880 {
2881     emit_model_quad_lit_rotated(ctx, face_world, texture_tile, light_flags, 0);
2882 }
2883
2884 static void emit_model_quad_lit_rotated(RenderContext *ctx,
2885                                       Vec3 face_world[4],
2886                                       uint8_t texture_tile,
2887                                       uint8_t light_flags,
2888                                       int uv_rotation)
2889 {
2890     static const float base_uv[4][2] = {
2891         { 0.0f, 16.0f },
2892         { 16.0f, 16.0f },
2893         { 16.0f, 0.0f },
2894         { 0.0f, 0.0f },
2895     };
2896     CameraVertex face_cam[4];
2897     uint8_t texture_id;
2898     uint8_t face_flags = light_flags | QUAD_FLAG_FOG | QUAD_FLAG_ALPHA_KEY;
2899
2900     uv_rotation %= 3;
2901     for (int i = 0; i < 4; i++) {
2902         int uv_index = (i + uv_rotation) & 3;
2903
2904         world_to_camera(ctx, face_world[i], &face_cam[i]);
2905         face_cam[i].u = base_uv[uv_index][0];
2906         face_cam[i].v = base_uv[uv_index][1];
2907     }
2908     if (camera_quad_outside_view(ctx, face_cam))
2909         return;
2910
2911     texture_id = choose_face_texture_lod(ctx, texture_tile, face_cam);
2912
2913     CameraVertex clipped[6];
2914     int clipped_count = clip_face_to_near_plane(face_cam, clipped);
2915     emit_clipped_face(ctx, clipped, clipped_count, texture_id, face_flags);
2916 }
2917
2918 static void emit_torch_block_face_lit(RenderContext *ctx, BlockID type,

```

```

2919         Vec3 block_pos, uint8_t face,
2920         uint8_t support_face,
2921         uint8_t light_flags)
2922 {
2923     Vec3 face_world[4];
2924
2925     if (type == BLOCK_AIR)
2926         return;
2927
2928     torch_face_vertices(block_pos, face, support_face, face_world);
2929     emit_model_quad_lit(ctx, face_world,
2930                        block_face_texture_id(type, FACE_FRONT),
2931                        light_flags);
2932 }
2933
2934 static void emit_door_block_face_lit(RenderContext *ctx, BlockID type,
2935                                     Vec3 block_pos, uint8_t face,
2936                                     uint8_t light_flags)
2937 {
2938     Vec3 face_world[4];
2939
2940     if (!block_is_door(type))
2941         return;
2942
2943     door_face_vertices(block_pos, type, face, face_world);
2944     emit_model_quad_lit(ctx, face_world,
2945                        block_face_texture_id(type, FACE_FRONT),
2946                        light_flags);
2947
2948     if (face == CHUNK_FACE_CROSS_A) {
2949         for (int edge = 0; edge < 4; edge++) {
2950             door_edge_vertices(block_pos, type, edge, face_world);
2951             emit_model_quad_lit(ctx, face_world, TEX_TILE_WOOD_PLANK,
2952                                light_flags);
2953         }
2954     }
2955 }
2956
2957 static void emit_merged_block_face(RenderContext *ctx, BlockID type,
2958                                   Vec3 block_pos, BlockFace face,
2959                                   int u_size, int v_size)
2960 {
2961     uint8_t light_flags = block_is_self_lit(type)
2962                         ? QUAD_LIGHT_LEVEL(0)
2963                         : choose_face_light_flags(ctx, face);
2964
2965     emit_merged_block_face_lit(ctx, type, block_pos, face, u_size, v_size,
2966                               8, light_flags);
2967 }
2968
2969 static void emit_block_face(RenderContext *ctx, BlockID type,
2970                             Vec3 block_pos, BlockFace face)
2971 {
2972     emit_merged_block_face(ctx, type, block_pos, face, 1, 1);
2973 }
2974
2975 typedef struct {
2976     uint8_t unlock_stage;
2977     uint8_t x;

```

```

2978     uint8_t y;
2979     uint8_t w;
2980     uint8_t h;
2981 } BreakCrackPatch;
2982
2983 /* 16x16 grid-space crack pieces. Hard rectangular runs keep the break overlay
2984  * pixelated and shard-like instead of looking hand-drawn. */
2985 static const BreakCrackPatch BREAK_CRACK_PATCHES[] = {
2986     { 0, 7, 7, 2, 2 },
2987     { 1, 7, 5, 1, 2 },
2988     { 1, 6, 7, 1, 1 },
2989     { 1, 9, 8, 1, 1 },
2990     { 2, 5, 6, 2, 1 },
2991     { 2, 4, 5, 1, 1 },
2992     { 2, 9, 6, 2, 1 },
2993     { 2, 11, 5, 1, 1 },
2994     { 3, 6, 9, 1, 2 },
2995     { 3, 5, 11, 1, 1 },
2996     { 3, 9, 9, 2, 1 },
2997     { 3, 11, 10, 1, 1 },
2998     { 4, 3, 4, 1, 2 },
2999     { 4, 2, 3, 1, 1 },
3000     { 4, 12, 4, 1, 2 },
3001     { 4, 13, 3, 1, 1 },
3002     { 5, 4, 12, 1, 2 },
3003     { 5, 3, 14, 1, 1 },
3004     { 5, 12, 10, 1, 2 },
3005     { 5, 13, 12, 1, 1 },
3006     { 6, 7, 3, 1, 2 },
3007     { 6, 7, 1, 1, 2 },
3008     { 6, 10, 2, 2, 1 },
3009     { 6, 12, 1, 1, 1 },
3010     { 7, 1, 2, 1, 1 },
3011     { 7, 0, 1, 1, 1 },
3012     { 7, 14, 2, 1, 1 },
3013     { 7, 15, 1, 1, 1 },
3014     { 7, 14, 13, 1, 2 },
3015     { 8, 2, 14, 1, 2 },
3016     { 8, 0, 15, 2, 1 },
3017     { 8, 14, 14, 2, 1 },
3018     { 8, 15, 15, 1, 1 },
3019     { 9, 0, 0, 1, 1 },
3020     { 9, 15, 0, 1, 1 },
3021     { 9, 0, 8, 2, 1 },
3022     { 9, 14, 8, 2, 1 },
3023     { 9, 6, 6, 1, 1 },
3024     { 9, 9, 7, 1, 1 },
3025     { 9, 8, 9, 1, 1 },
3026 };
3027
3028 static Vec3 vec3_add(Vec3 a, Vec3 b)
3029 {
3030     return (Vec3){ a.x + b.x, a.y + b.y, a.z + b.z };
3031 }
3032
3033 static Vec3 vec3_sub(Vec3 a, Vec3 b)
3034 {
3035     return (Vec3){ a.x - b.x, a.y - b.y, a.z - b.z };
3036 }

```

```

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

```

```

3096     progress = clamp01(progress);
3097     if (progress <= 0.0f)
3098         return 0;
3099
3100     before = ctx->n_quads;
3101     stage = (int)floorf(progress * 10.0f);
3102     if (stage < 0)
3103         stage = 0;
3104     if (stage > 9)
3105         stage = 9;
3106     color = 14;
3107     alpha = QUAD_ALPHA_75;
3108     block_pos = (Vec3){ (float)block_x, (float)block_y, (float)block_z };
3109
3110     for (int face = 0; face < NUM_FACES; face++) {
3111         Vec3 corners[4];
3112         Vec3 normal = face_normals[face];
3113
3114         if (!is_face_visible(block_pos, normal, ctx->current_camera.position))
3115             continue;
3116
3117         merged_face_vertices(block_pos, (BlockFace)face, 1, 1, 8, corners);
3118         for (int i = 0; i < 4; i++) {
3119             corners[i].x += normal.x * face_epsilon;
3120             corners[i].y += normal.y * face_epsilon;
3121             corners[i].z += normal.z * face_epsilon;
3122         }
3123
3124         for (size_t i = 0;
3125              i < sizeof(BREAK_CRACK_PATCHES) / sizeof(BREAK_CRACK_PATCHES[0]);
3126              i++) {
3127             if (BREAK_CRACK_PATCHES[i].unlock_stage > stage)
3128                 continue;
3129             emit_break_crack_patch(ctx, corners, &BREAK_CRACK_PATCHES[i],
3130                                   color, alpha);
3131             if (ctx->n_quads >= MAX_QUADS_IN_FLIGHT)
3132                 return ctx->n_quads - before;
3133         }
3134     }
3135
3136     return ctx->n_quads - before;
3137 }
3138
3139 static float distance_to_interval(float value, float min, float max)
3140 {
3141     if (value < min)
3142         return min - value;
3143     if (value > max)
3144         return value - max;
3145     return 0.0f;
3146 }
3147
3148 typedef struct {
3149     CameraVertex center;
3150     float ex;
3151     float ey;
3152     float ez;
3153 } ChunkCameraBounds;
3154

```

```

3155 static void chunk_camera_bounds(RenderContext *ctx, const Chunk *chunk,
3156                               ChunkCameraBounds *bounds)
3157 {
3158     float half_x = WORLD_CHUNK_SIZE * 0.5f;
3159     float half_y = WORLD_CHUNK_HEIGHT * 0.5f;
3160     float half_z = WORLD_CHUNK_SIZE * 0.5f;
3161     float cy = ctx->cos_yaw;
3162     float sy = ctx->sin_yaw;
3163     float cp = ctx->cos_pitch;
3164     float sp = ctx->sin_pitch;
3165     Vec3 chunk_center = {
3166         (float)(chunk->chunk_x * WORLD_CHUNK_SIZE) + half_x,
3167         half_y,
3168         (float)(chunk->chunk_z * WORLD_CHUNK_SIZE) + half_z,
3169     };
3170
3171     world_to_camera(ctx, chunk_center, &bounds->center);
3172
3173     /* Project world-space half-extents into camera space using abs(R) * e. */
3174     bounds->ex = fabsf(cy) * half_x + fabsf(sy) * half_z;
3175     bounds->ey = fabsf(sy * sp) * half_x +
3176                fabsf(cp) * half_y +
3177                fabsf(cy * sp) * half_z;
3178     bounds->ez = fabsf(sy * cp) * half_x +
3179                fabsf(sp) * half_y +
3180                fabsf(cy * cp) * half_z;
3181 }
3182
3183 static float chunk_distance_sq_to_camera(RenderContext *ctx, const Chunk *chunk);
3184
3185 static bool ensure_translucent_scratch(RenderContext *ctx, int needed)
3186 {
3187     int cap;
3188     TranslucentFaceRef *buf;
3189
3190     if (needed <= ctx->translucent_scratch_capacity)
3191         return true;
3192
3193     cap = ctx->translucent_scratch_capacity ? ctx->translucent_scratch_capacity : 64;
3194     while (cap < needed)
3195         cap *= 2;
3196
3197     buf = realloc(ctx->translucent_scratch, (size_t)cap * sizeof(*buf));
3198     if (!buf)
3199         return false;
3200
3201     ctx->translucent_scratch = buf;
3202     ctx->translucent_scratch_capacity = cap;
3203     return true;
3204 }
3205
3206 static uint32_t translucent_face_stable_order(const Chunk *chunk,
3207                                              const ChunkFace *face)
3208 {
3209     uint32_t h = 2166136261u;
3210
3211 #define MIX_U32(v) do { h ^= (uint32_t)(v); h *= 16777619u; } while (0)
3212     MIX_U32(chunk->chunk_x);
3213     MIX_U32(chunk->chunk_z);

```

```

3214     MIX_U32(face->x);
3215     MIX_U32(face->y);
3216     MIX_U32(face->z);
3217     MIX_U32(face->face);
3218 #undef MIX_U32
3219
3220     return h;
3221 }
3222
3223 static int compare_translucent_back_to_front(const void *a, const void *b)
3224 {
3225     const TranslucentFaceRef *ra = a;
3226     const TranslucentFaceRef *rb = b;
3227     float dz = ra->view_z - rb->view_z;
3228
3229     /* Larger view_z (farther) comes first. */
3230     if (dz > 1e-4f) return -1;
3231     if (dz < -1e-4f) return 1;
3232     if (ra->stable_order < rb->stable_order) return -1;
3233     if (ra->stable_order > rb->stable_order) return 1;
3234     return 0;
3235 }
3236
3237 static bool chunk_within_render_distance(RenderContext *ctx,
3238                                         const VoxelWorld *world,
3239                                         const Chunk *chunk)
3240 {
3241     float max_distance;
3242     float max_distance_sq;
3243
3244     if (world->render_distance_chunks <= 0)
3245         return true;
3246
3247     max_distance = (float)(world->render_distance_chunks * WORLD_CHUNK_SIZE);
3248     max_distance_sq = max_distance * max_distance;
3249     return chunk_distance_sq_to_camera(ctx, chunk) <= max_distance_sq;
3250 }
3251
3252 static void configure_world_fog(RenderContext *ctx, const VoxelWorld *world)
3253 {
3254     struct voxel_fog_state fog = {0};
3255
3256     if (!ctx || !world || world->render_distance_chunks <= 0) {
3257         if (ctx)
3258             renderer_set_fog_state(ctx, &fog);
3259         return;
3260     }
3261
3262     float end_distance = (float)(world->render_distance_chunks * WORLD_CHUNK_SIZE);
3263     float fade_span = (float)WORLD_CHUNK_SIZE;
3264     float start_distance;
3265     float inv_proj_sq;
3266
3267     start_distance = end_distance - fade_span;
3268     if (start_distance < NEAR_PLANE)
3269         start_distance = NEAR_PLANE;
3270
3271     inv_proj_sq = 1.0f / (ctx->current_camera.depth * ctx->current_camera.depth);
3272

```

```

3273     fog.start_dist = to_q8_8u(start_distance);
3274     fog.end_dist = to_q8_8u(end_distance);
3275     fog.color_index = PAL_SKY_HORIZON;
3276     fog.enabled = fog.end_dist > fog.start_dist;
3277     fog.inv_proj_sq = to_q0_16u(inv_proj_sq);
3278     renderer_set_fog_state(ctx, &fog);
3279 }
3280
3281 static bool chunk_intersects_frustum(RenderContext *ctx, const Chunk *chunk)
3282 {
3283     float x_slope = (SCREEN_WIDTH * 0.5f) / ctx->current_camera.depth;
3284     float y_slope = (SCREEN_HEIGHT * 0.5f) / ctx->current_camera.depth;
3285     ChunkCameraBounds bounds;
3286
3287     chunk_camera_bounds(ctx, chunk, &bounds);
3288
3289     if (bounds.center.z + bounds.ez < NEAR_PLANE)
3290         return false;
3291     if (bounds.center.x + x_slope * bounds.center.z +
3292         (bounds.ex + x_slope * bounds.ez) < 0.0f)
3293         return false;
3294     if (-bounds.center.x + x_slope * bounds.center.z +
3295         (bounds.ex + x_slope * bounds.ez) < 0.0f)
3296         return false;
3297     if (bounds.center.y + y_slope * bounds.center.z +
3298         (bounds.ey + y_slope * bounds.ez) < 0.0f)
3299         return false;
3300     if (-bounds.center.y + y_slope * bounds.center.z +
3301         (bounds.ey + y_slope * bounds.ez) < 0.0f)
3302         return false;
3303
3304     return true;
3305 }
3306
3307 typedef struct {
3308     const Chunk *chunk;
3309     /* Cached at the start of draw_world so the entire frame iterates a
3310      * stable mesh pointer while workers publish replacements. */
3311     const ChunkMesh *mesh;
3312     float distance_sq;
3313 } ChunkDrawCandidate;
3314
3315 static float chunk_distance_sq_to_camera(RenderContext *ctx, const Chunk *chunk)
3316 {
3317     float min_x = (float)(chunk->chunk_x * WORLD_CHUNK_SIZE);
3318     float max_x = min_x + (float)WORLD_CHUNK_SIZE;
3319     float min_y = 0.0f;
3320     float max_y = (float)WORLD_CHUNK_HEIGHT;
3321     float min_z = (float)(chunk->chunk_z * WORLD_CHUNK_SIZE);
3322     float max_z = min_z + (float)WORLD_CHUNK_SIZE;
3323     float dx = distance_to_interval(ctx->current_camera.position.x, min_x, max_x);
3324     float dy = distance_to_interval(ctx->current_camera.position.y, min_y, max_y);
3325     float dz = distance_to_interval(ctx->current_camera.position.z, min_z, max_z);
3326
3327     return dx * dx + dy * dy + dz * dz;
3328 }
3329
3330 /* --- Public API --- */
3331

```

```

3332 RenderContext *renderer_init(void)
3333 {
3334     RenderContext *ctx = calloc(1, sizeof(*ctx));
3335     if (!ctx) return NULL;
3336
3337     ctx->transport = gpu_transport_open();
3338     if (!ctx->transport) {
3339         free(ctx);
3340         return NULL;
3341     }
3342
3343     ctx->submit_capacity = MAX_QUADS_IN_FLIGHT *
3344         (sizeof(struct quad_desc) + sizeof(struct quad_desc_uv));
3345     ctx->submit_buffer = malloc(ctx->submit_capacity);
3346     ctx->lookup_entries = NULL;
3347     ctx->lookup_capacity = 0;
3348     ctx->translucent_scratch = NULL;
3349     ctx->translucent_scratch_capacity = 0;
3350     ctx->occlusion_enabled = env_flag("VOXEL_OCCLUSION_CULL", false) ? 1 : 0;
3351     ctx->occlusion_diag = env_flag("VOXEL_DIAG_OCCLUSION", false) ? 1 : 0;
3352     if (!ctx->submit_buffer) {
3353         free(ctx->submit_buffer);
3354         gpu_transport_close(ctx->transport);
3355         free(ctx);
3356         return NULL;
3357     }
3358
3359     ctx->light_dir = normalize_vec3((Vec3){ 0.35f, 0.92f, 0.20f });
3360     ctx->world_daylight = 1.0f;
3361     update_face_light_flags(ctx);
3362     upload_default_palette(ctx->transport);
3363     upload_light_palette_banks(ctx, ctx->world_daylight);
3364     renderer_set_fog_state(ctx, &(struct voxel_fog_state){0});
3365     return ctx;
3366 }
3367
3368 void renderer_shutdown(RenderContext *ctx)
3369 {
3370     if (!ctx) return;
3371     free(ctx->translucent_scratch);
3372     free(ctx->lookup_entries);
3373     free(ctx->submit_buffer);
3374     gpu_transport_close(ctx->transport);
3375     free(ctx);
3376 }
3377
3378 void renderer_begin_frame(RenderContext *ctx)
3379 {
3380     ctx->n_quads = 0;
3381     ctx->submit_bytes = 0;
3382     occlusion_reset(ctx);
3383     /* Front-load the per-quad band binning into emit time: stage_prepared_quad
3384      * will hand each finished descriptor to gpu_transport_bin_descriptor as
3385      * soon as it is built (descriptor still hot in L1), and the corresponding
3386      * second-pass walk inside submit_hw_binned is skipped. */
3387     gpu_transport_begin_descriptors(ctx->transport);
3388 }
3389
3390 void renderer_end_frame(RenderContext *ctx)

```

```

3391 {
3392     if (ctx->occlusion_diag && ((ctx->occlusion_frame_count++ % 60u) == 0u)) {
3393         fprintf(stderr,
3394             "renderer: occlusion tested=%u culled=%u tiles=%u enabled=%u\n",
3395             ctx->occlusion_tested,
3396             ctx->occlusion_culled,
3397             ctx->occlusion_tiles_written,
3398             (unsigned)ctx->occlusion_enabled);
3399     }
3400
3401     if (gpu_transport_clear(ctx->transport) < 0)
3402         return;
3403     if (!renderer_flush_gpu_state(ctx))
3404         return;
3405
3406     if (ctx->submit_bytes == 0) {
3407         gpu_transport_flip(ctx->transport);
3408         return;
3409     }
3410
3411     if (gpu_transport_submit_descriptors(ctx->transport,
3412                                         ctx->submit_buffer,
3413                                         ctx->submit_bytes) < 0)
3414         return;
3415
3416     gpu_transport_flip(ctx->transport);
3417 }
3418
3419 void renderer_set_camera(RenderContext *ctx, const Camera *camera)
3420 {
3421     ctx->current_camera = *camera;
3422     ctx->cos_yaw = cosf(camera->yaw);
3423     ctx->sin_yaw = sinf(camera->yaw);
3424     ctx->cos_pitch = cosf(camera->pitch);
3425     ctx->sin_pitch = sinf(camera->pitch);
3426 }
3427
3428 static bool stage_prepared_quad(RenderContext *ctx, RenderQuad quad)
3429 {
3430     const bool input_textured = (quad.flags & QUAD_FLAG_TEX) != 0;
3431     const size_t textured_descriptor_bytes =
3432         gpu_transport_textured_descriptor_size(ctx->transport);
3433     const size_t max_descriptor_bytes =
3434         input_textured ? textured_descriptor_bytes : sizeof(struct quad_desc);
3435
3436     if (ctx->n_quads >= MAX_QUADS_IN_FLIGHT)
3437         return false;
3438     if (ctx->submit_bytes + max_descriptor_bytes > ctx->submit_capacity)
3439         return false;
3440
3441     /*
3442      * The rasterizer only sees Q24.8 edge math. Snap screen-space vertices to
3443      * that same grid before bbox, edge, and depth setup so all three stages
3444      * agree on which pixels a thin quad should cover.
3445      */
3446     for (int i = 0; i < 4; i++) {
3447         quad.vertices[i].x = snap_q24_8(quad.vertices[i].x);
3448         quad.vertices[i].y = snap_q24_8(quad.vertices[i].y);
3449     }

```

```

3450
3451     ensure_clockwise_winding(quad.vertices);
3452
3453     const Vertex2D *v = quad.vertices;
3454     PlaneBasis basis = choose_plane_basis(v);
3455
3456     /* Inclusive integer bbox over pixel-center samples. */
3457     int32_t qxmin = to_q24_8(v[0].x), qxmax = qxmin;
3458     int32_t qymin = to_q24_8(v[0].y), qymax = qymin;
3459     for (int i = 1; i < 4; i++) {
3460         int32_t qx = to_q24_8(v[i].x);
3461         int32_t qy = to_q24_8(v[i].y);
3462
3463         if (qx < qxmin) qxmin = qx;
3464         if (qx > qxmax) qxmax = qx;
3465         if (qy < qymin) qymin = qy;
3466         if (qy > qymax) qymax = qy;
3467     }
3468     int x_min = ceil_div_256_i32(qxmin - 128); if (x_min < 0) x_min = 0;
3469     int y_min = ceil_div_256_i32(qymin - 128); if (y_min < 0) y_min = 0;
3470     int x_max = floor_div_256_i32(qxmax - 128);
3471     int y_max = floor_div_256_i32(qymax - 128);
3472     if (x_max > (int)VOXEL_RENDER_WIDTH - 1)
3473         x_max = (int)VOXEL_RENDER_WIDTH - 1;
3474     if (y_max > (int)VOXEL_RENDER_HEIGHT - 1)
3475         y_max = (int)VOXEL_RENDER_HEIGHT - 1;
3476
3477     if (x_min > x_max || y_min > y_max)
3478         return false; /* degenerate / off-screen */
3479
3480     if (ctx->occlusion_enabled &&
3481         ctx->occlusion_pass == OCCLUSION_PASS_OPAQUE_WORLD &&
3482         occlusion_quad_eligible(quad.flags)) {
3483         ctx->occlusion_tested++;
3484         if (occlusion_quad_hidden(ctx, v, x_min, y_min, x_max, y_max)) {
3485             ctx->occlusion_culled++;
3486             return false;
3487         }
3488     }
3489
3490     struct quad_desc *d = (struct quad_desc *) (ctx->submit_buffer + ctx->submit_bytes);
3491     memset(d, 0, sizeof(*d));
3492
3493     d->x_min = (int16_t)x_min;
3494     d->y_min = (int16_t)y_min;
3495     d->x_max = (int16_t)x_max;
3496     d->y_max = (int16_t)y_max;
3497
3498     for (int i = 0; i < 4; i++) {
3499         float x0 = v[i].x, y0 = v[i].y;
3500         float x1 = v[(i+1)%4].x, y1 = v[(i+1)%4].y;
3501         d->edges[i] = make_edge_coef(x0, y0, x1, y1);
3502     }
3503
3504     /* Compute into locals first to avoid taking addresses of packed members. */
3505     uint16_t z0;
3506     int16_t dz_dx;
3507     int16_t dz_dy;
3508     fit_depth_plane(v, &basis, (float)x_min + 0.5f, (float)y_min + 0.5f,

```

```

3509         &z0, &dz_dx, &dz_dy);
3510     d->z0 = z0;
3511     d->dz_dx = dz_dx;
3512     d->dz_dy = dz_dy;
3513
3514     d->tex_or_color = (quad.flags & QUAD_FLAG_TEX) ? quad.texture_id : quad.color_tint;
3515     d->flags = quad.flags;
3516
3517 #ifdef DEBUG_FLAT_COLOR
3518 #ifndef DEBUG_FLAT_COLOR_INDEX
3519 #define DEBUG_FLAT_COLOR_INDEX 24
3520 #endif
3521     d->flags = (uint16_t)(quad.flags & ~(QUAD_FLAG_TEX));
3522     d->tex_or_color = (uint8_t)DEBUG_FLAT_COLOR_INDEX;
3523 #endif
3524
3525     size_t emitted_size;
3526     if (d->flags & QUAD_FLAG_TEX) {
3527         struct quad_desc_uv *uv = (struct quad_desc_uv *)
3528             (ctx->submit_buffer + ctx->submit_bytes + sizeof(*d));
3529         memset(uv, 0, sizeof(*uv));
3530         fit_uv_plane(v, &basis, (float)x_min + 0.5f, (float)y_min + 0.5f, uv);
3531         emitted_size = textured_descriptor_bytes;
3532     } else {
3533         emitted_size = sizeof(*d);
3534     }
3535     ctx->submit_bytes += emitted_size;
3536     ctx->n_quads++;
3537
3538     /* Bin-during-emit: route this just-built descriptor into the per-band
3539     * bins now while it is still in L1, instead of having submit_hw_binned
3540     * walk the contiguous stream a second time. No-op when the transport is
3541     * not in HW mode. */
3542     gpu_transport_bin_descriptor(ctx->transport, d, emitted_size);
3543
3544     if (ctx->occlusion_enabled &&
3545         ctx->occlusion_pass == OCCLUSION_PASS_OPAQUE_WORLD &&
3546         occlusion_quad_eligible(d->flags)) {
3547         occlusion_record_quad(ctx, v, x_min, y_min, x_max, y_max);
3548     }
3549
3550     return true;
3551 }
3552
3553 bool renderer_push_quad(RenderContext *ctx, const RenderQuad *quad)
3554 {
3555     Vertex2D clipped[MAX_VIEW_CLIP_VERTS];
3556     int count = clip_polygon_to_viewport(quad->vertices, clipped);
3557
3558     return stage_projected_polygon(ctx, quad, clipped, count);
3559 }
3560
3561 static void renderer_draw_block_faces(RenderContext *ctx, const Block *block,
3562                                     const BlockLookup *lookup)
3563 {
3564     if (block->type == BLOCK_AIR) return;
3565
3566     for (int f = 0; f < NUM_FACES; f++) {
3567         if (!is_face_exposed(block, (BlockFace)f, lookup))

```

```

3568         continue;
3569         emit_block_face(ctx, block->type, block->position, (BlockFace)f);
3570     }
3571 }
3572
3573 int renderer_draw_chunk(RenderContext *ctx, const Block *blocks, int num_blocks)
3574 {
3575     BlockLookup lookup;
3576     int before = ctx->n_quads;
3577
3578     if (!build_block_lookup(ctx, blocks, num_blocks, &lookup))
3579         return 0;
3580
3581     for (int i = 0; i < num_blocks; i++)
3582         renderer_draw_block_faces(ctx, &blocks[i], &lookup);
3583
3584     return ctx->n_quads - before;
3585 }
3586
3587 static bool falling_block_hides_mesh_cell(const VoxelWorld *world,
3588                                           int wx, int wy, int wz)
3589 {
3590     int seen = 0;
3591
3592     if (!world || world->falling_block_count <= 0)
3593         return false;
3594
3595     for (int i = 0; i < WORLD_MAX_FALLING_BLOCKS &&
3596          seen < world->falling_block_count; i++) {
3597         const FallingBlock *falling = &world->falling_blocks[i];
3598
3599         if (!falling->active)
3600             continue;
3601         seen++;
3602         if (falling->wx == wx &&
3603             falling->origin_y == wy &&
3604             falling->wz == wz)
3605             return true;
3606     }
3607
3608     return false;
3609 }
3610
3611 static int renderer_draw_falling_blocks(RenderContext *ctx,
3612                                         const VoxelWorld *world)
3613 {
3614     int before = ctx->n_quads;
3615     int seen = 0;
3616
3617     if (!world || world->falling_block_count <= 0)
3618         return 0;
3619
3620     for (int i = 0; i < WORLD_MAX_FALLING_BLOCKS &&
3621          seen < world->falling_block_count; i++) {
3622         const FallingBlock *falling = &world->falling_blocks[i];
3623         if (!falling->active)
3624             continue;
3625         seen++;
3626     }

```

```

3627     Vec3 block_pos = {
3628         (float)falling->wx,
3629         falling->y,
3630         (float)falling->wz,
3631     };
3632     for (int f = 0; f < NUM_FACES; f++) {
3633         emit_block_face(ctx, falling->type, block_pos, (BlockFace)f);
3634         if (ctx->n_quads >= MAX_QUADS_IN_FLIGHT)
3635             return ctx->n_quads - before;
3636     }
3637 }
3638
3639 return ctx->n_quads - before;
3640 }
3641
3642 static bool cube_face_back_facing(BlockFace face,
3643     float wx,
3644     float wy,
3645     float wz,
3646     Vec3 cam_pos)
3647 {
3648     switch (face) {
3649     case FACE_TOP:    return cam_pos.y < wy + 1.0f;
3650     case FACE_BOTTOM: return cam_pos.y > wy;
3651     case FACE_RIGHT:  return cam_pos.x < wx + 1.0f;
3652     case FACE_LEFT:   return cam_pos.x > wx;
3653     case FACE_BACK:   return cam_pos.z < wz + 1.0f;
3654     case FACE_FRONT:  return cam_pos.z > wz;
3655     default:          return false;
3656     }
3657 }
3658
3659 static bool fast_cube_backface_cull_enabled(void)
3660 {
3661     static int cached = -1;
3662
3663     if (cached < 0)
3664         cached = env_flag("VOXEL_FAST_BACKFACE_CULL", true) ? 1 : 0;
3665     return cached != 0;
3666 }
3667
3668 int renderer_draw_world(RenderContext *ctx, const VoxelWorld *world,
3669     float time_seconds)
3670 {
3671     int before = ctx->n_quads;
3672     int candidate_count = 0;
3673
3674     if (!world || world->chunk_count <= 0)
3675         return 0;
3676
3677     update_world_light_state(ctx, time_seconds);
3678     configure_world_fog(ctx, world);
3679
3680     ChunkDrawCandidate candidates[world->chunk_count];
3681
3682     for (int i = 0; i < world->chunk_count; i++) {
3683         const Chunk *chunk = &world->chunks[i];
3684         const ChunkMesh *mesh;
3685

```

```

3686     if (!(chunk->flags & CHUNK_FLAG_LOADED))
3687         continue;
3688     mesh = atomic_load_explicit(&chunk->live_mesh, memory_order_acquire);
3689     if (!mesh || mesh->face_count <= 0)
3690         continue;
3691     if (!chunk_within_render_distance(ctx, world, chunk))
3692         continue;
3693     if (!chunk_intersects_frustum(ctx, chunk))
3694         continue;
3695
3696     candidates[candidate_count++] = (ChunkDrawCandidate){
3697         .chunk = chunk,
3698         .mesh = mesh,
3699         .distance_sq = chunk_distance_sq_to_camera(ctx, chunk),
3700     };
3701 }
3702
3703 for (int i = 1; i < candidate_count; i++) {
3704     ChunkDrawCandidate key = candidates[i];
3705     int j = i - 1;
3706
3707     while (j >= 0 && candidates[j].distance_sq > key.distance_sq) {
3708         candidates[j + 1] = candidates[j];
3709         j--;
3710     }
3711     candidates[j + 1] = key;
3712 }
3713
3714 /*
3715  * Opaque pass: emit all non-translucent faces first, chunks nearest
3716  * first. Glass/translucent faces also write Z through QUAD_FLAG_ZTEST,
3717  * so emitting them first would cause opaque geometry behind the glass
3718  * to be Z-rejected and we'd blend glass against sky instead of against
3719  * the stone behind it.
3720  *
3721  * Optional early back-face cull: face normals are axis-aligned (exactly one
3722  * component 1, rest zero). The full dot-product in is_face_visible
3723  * reduces to a single comparison per face direction:
3724  *   sign > 0 axis: skip if (block_pos[axis] + 1.0) - cam[axis] >= 0
3725  *   sign < 0 axis: skip if   block_pos[axis]           - cam[axis] <= 0
3726  * Doing this before choose_chunk_face_light_flags saves the light
3727  * lookup + function call for all back-facing faces (~50% of faces). */
3728 const Vec3 cam_pos = ctx->current_camera.position;
3729 ctx->occlusion_pass = ctx->occlusion_enabled ?
3730     OCCLUSION_PASS_OPAQUE_WORLD :
3731     OCCLUSION_PASS_DISABLED;
3732 for (int i = 0; i < candidate_count; i++) {
3733     const Chunk *chunk = candidates[i].chunk;
3734     const ChunkMesh *mesh = candidates[i].mesh;
3735     const int opaque_cube_count = mesh->opaque_cube_face_count;
3736     const int opaque_face_count = mesh->opaque_face_count;
3737
3738     for (int face_index = 0; face_index < opaque_cube_count; face_index++) {
3739         const ChunkFace *face = &mesh->faces[face_index];
3740         BlockID id = (BlockID)face->type;
3741         int wxi = chunk->chunk_x * WORLD_CHUNK_SIZE + (int)face->x;
3742         int wyi = (int)face->y;
3743         int wzi = chunk->chunk_z * WORLD_CHUNK_SIZE + (int)face->z;
3744         if (falling_block_hides_mesh_cell(world, wxi, wyi, wzi))

```

```

3745         continue;
3746
3747         float wx = (float)wxi;
3748         float wy = (float>wyi;
3749         float wz = (float>wzi;
3750
3751         if (fast_cube_backface_cull_enabled() &&
3752             cube_face_back_facing((BlockFace)face->face,
3753                                     wx, wy, wz, cam_pos))
3754             continue;
3755
3756         Vec3 block_pos = { wx, wy, wz };
3757         uint8_t light_flags = choose_chunk_face_light_flags(ctx, id,
3758                                                             (BlockFace)face->face,
3759                                                             face->sky_light,
3760                                                             face->block_light);
3761
3762         emit_merged_block_face_lit(ctx, id, block_pos,
3763                                     (BlockFace)face->face,
3764                                     face->u_size ? face->u_size : 1,
3765                                     face->v_size ? face->v_size : 1,
3766                                     face->height ? face->height : 8,
3767                                     light_flags);
3768         if (ctx->n_quads >= MAX_QUADS_IN_FLIGHT) {
3769             ctx->occlusion_pass = OCCLUSION_PASS_DISABLED;
3770             return ctx->n_quads - before;
3771         }
3772     }
3773
3774     for (int face_index = opaque_cube_count;
3775          face_index < opaque_face_count;
3776          face_index++) {
3777         const ChunkFace *face = &mesh->faces[face_index];
3778         BlockID id = (BlockID)face->type;
3779         int wxi = chunk->chunk_x * WORLD_CHUNK_SIZE + (int)face->x;
3780         int wyi = (int)face->y;
3781         int wzi = chunk->chunk_z * WORLD_CHUNK_SIZE + (int)face->z;
3782         if (falling_block_hides_mesh_cell(world, wxi, wyi, wzi))
3783             continue;
3784
3785         Vec3 block_pos = {
3786             (float)wxi,
3787             (float>wyi,
3788             (float>wzi,
3789         };
3790         BlockRenderModel model = block_render_model(id);
3791         uint8_t light_flags = choose_chunk_face_light_flags(ctx, id,
3792                                                             FACE_TOP,
3793                                                             face->sky_light,
3794                                                             face->block_light);
3795
3796         if (model == BLOCK_RENDER_CROSS)
3797             emit_cross_block_face_lit(ctx, id, block_pos,
3798                                     face->face, light_flags);
3799         else if (model == BLOCK_RENDER_TORCH)
3800             emit_torch_block_face_lit(ctx, id, block_pos,
3801                                     face->face, face->height,
3802                                     light_flags);
3803         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,

```

```

3863         v_size, height);
3864     CameraVertex cv;
3865     world_to_camera(ctx, face_center, &cv);
3866
3867     ctx->translucent_scratch[w++] = (TranslucentFaceRef){
3868         .chunk = chunk,
3869         .face = face,
3870         .view_z = cv.z,
3871         .stable_order = translucent_face_stable_order(chunk, face),
3872     };
3873 }
3874 }
3875
3876 if (w > 0) {
3877     if (w > 1) {
3878         qsort(ctx->translucent_scratch, (size_t)w,
3879             sizeof(*ctx->translucent_scratch),
3880             compare_translucent_back_to_front);
3881     }
3882
3883     for (int i = 0; i < w; i++) {
3884         const TranslucentFaceRef *ref = &ctx->translucent_scratch[i];
3885         Vec3 block_pos = {
3886             (float)(ref->chunk->chunk_x * WORLD_CHUNK_SIZE + ref->face->x),
3887             (float)ref->face->y,
3888             (float)(ref->chunk->chunk_z * WORLD_CHUNK_SIZE + ref->face->z),
3889         };
3890         uint8_t light_flags = choose_chunk_face_light_flags(
3891             ctx,
3892             (BlockID)ref->face->type,
3893             (BlockFace)ref->face->face,
3894             ref->face->sky_light,
3895             ref->face->block_light);
3896
3897         emit_merged_block_face_lit(ctx, (BlockID)ref->face->type,
3898             block_pos, (BlockFace)ref->face->face,
3899             ref->face->u_size ? ref->face->u_size : 1,
3900             ref->face->v_size ? ref->face->v_size : 1,
3901             ref->face->height ? ref->face->height : 8,
3902             light_flags);
3903         if (ctx->n_quads >= MAX_QUADS_IN_FLIGHT)
3904             return ctx->n_quads - before;
3905     }
3906 }
3907
3908 return ctx->n_quads - before;
3909 }
3910
3911 static bool push_screen_textured_quad(RenderContext *ctx,
3912     float x0, float y0,
3913     float x1, float y1,
3914     float u0, float v0,
3915     float u1, float v1,
3916     uint8_t texture_id,
3917     uint8_t extra_flags)
3918 {
3919     RenderQuad quad = {0};
3920
3921     quad.texture_id = texture_id;

```

```

3922     quad.flags = QUAD_FLAG_TEX | extra_flags;
3923     quad.vertices[0] = (Vertex2D){ x0, y0, 0.0f, u0, v0, 1.0f };
3924     quad.vertices[1] = (Vertex2D){ x1, y0, 0.0f, u1, v0, 1.0f };
3925     quad.vertices[2] = (Vertex2D){ x1, y1, 0.0f, u1, v1, 1.0f };
3926     quad.vertices[3] = (Vertex2D){ x0, y1, 0.0f, u0, v1, 1.0f };
3927     if (projected_quad_fully_inside_viewport(quad.vertices))
3928         return stage_projected_quad_no_clip(ctx, &quad);
3929     return renderer_push_quad(ctx, &quad);
3930 }
3931
3932 static bool push_screen_flat_quad(RenderContext *ctx,
3933                                 float x0, float y0,
3934                                 float x1, float y1,
3935                                 uint8_t palette_index)
3936 {
3937     return renderer_fill_rect(ctx, x0, y0, x1, y1, palette_index, 0);
3938 }
3939
3940 bool renderer_draw_screen_tile(RenderContext *ctx,
3941                               float x0, float y0, float x1, float y1,
3942                               uint8_t texture_id, uint8_t extra_flags)
3943 {
3944     return push_screen_textured_quad(ctx,
3945                                     x0, y0, x1, y1,
3946                                     0.0f, 0.0f, 16.0f, 16.0f,
3947                                     texture_id,
3948                                     extra_flags);
3949 }
3950
3951 bool renderer_draw_custom_screen_quad(RenderContext *ctx,
3952                                       float x0, float y0,
3953                                       float x1, float y1,
3954                                       float x2, float y2,
3955                                       float x3, float y3,
3956                                       uint8_t texture_id, uint8_t extra_flags)
3957 {
3958     RenderQuad quad = {0};
3959
3960     quad.texture_id = texture_id;
3961     quad.flags = QUAD_FLAG_TEX | extra_flags;
3962     quad.vertices[0] = (Vertex2D){ x0, y0, 0.0f, 0.0f, 0.0f, 1.0f };
3963     quad.vertices[1] = (Vertex2D){ x1, y1, 0.0f, 16.0f, 0.0f, 1.0f };
3964     quad.vertices[2] = (Vertex2D){ x2, y2, 0.0f, 16.0f, 16.0f, 1.0f };
3965     quad.vertices[3] = (Vertex2D){ x3, y3, 0.0f, 0.0f, 16.0f, 1.0f };
3966     if (projected_quad_fully_inside_viewport(quad.vertices))
3967         return stage_projected_quad_no_clip(ctx, &quad);
3968     return renderer_push_quad(ctx, &quad);
3969 }
3970
3971 bool renderer_draw_world_billboard_tile(RenderContext *ctx,
3972                                         Vec3 center,
3973                                         float size_world,
3974                                         uint8_t texture_id,
3975                                         uint8_t extra_flags)
3976 {
3977     CameraVertex camera;
3978     Vertex2D projected;
3979     RenderQuad quad = {0};
3980     float inv_w;

```

```

3981     float half_px;
3982
3983     if (!ctx || size_world <= 0.0f)
3984         return false;
3985
3986     world_to_camera(ctx, center, &camera);
3987     if (!project_camera_vertex(ctx, &camera, &projected))
3988         return false;
3989
3990     inv_w = 1.0f / camera.z;
3991     half_px = ctx->current_camera.depth * size_world * 0.5f * inv_w;
3992     if (half_px < 1.0f)
3993         half_px = 1.0f;
3994
3995     quad.texture_id = texture_id;
3996     quad.flags = QUAD_FLAG_TEX | extra_flags;
3997     quad.vertices[0] = (Vertex2D){
3998         projected.x - half_px, projected.y - half_px, projected.z,
3999         0.0f * inv_w, 0.0f * inv_w, inv_w,
4000     };
4001     quad.vertices[1] = (Vertex2D){
4002         projected.x + half_px, projected.y - half_px, projected.z,
4003         16.0f * inv_w, 0.0f * inv_w, inv_w,
4004     };
4005     quad.vertices[2] = (Vertex2D){
4006         projected.x + half_px, projected.y + half_px, projected.z,
4007         16.0f * inv_w, 16.0f * inv_w, inv_w,
4008     };
4009     quad.vertices[3] = (Vertex2D){
4010         projected.x - half_px, projected.y + half_px, projected.z,
4011         0.0f * inv_w, 16.0f * inv_w, inv_w,
4012     };
4013
4014     if (projected_quad_fully_inside_viewport(quad.vertices))
4015         return stage_projected_quad_no_clip(ctx, &quad);
4016     return renderer_push_quad(ctx, &quad);
4017 }
4018
4019 bool renderer_fill_rect(RenderContext *ctx,
4020                       float x0, float y0, float x1, float y1,
4021                       uint8_t palette_index, uint8_t extra_flags)
4022 {
4023     RenderQuad quad = {0};
4024
4025     quad.color_tint = palette_index;
4026     quad.flags = extra_flags;
4027     quad.vertices[0] = (Vertex2D){ x0, y0, 0.0f, 0.0f, 0.0f, 1.0f };
4028     quad.vertices[1] = (Vertex2D){ x1, y0, 0.0f, 0.0f, 0.0f, 1.0f };
4029     quad.vertices[2] = (Vertex2D){ x1, y1, 0.0f, 0.0f, 0.0f, 1.0f };
4030     quad.vertices[3] = (Vertex2D){ x0, y1, 0.0f, 0.0f, 0.0f, 1.0f };
4031     if (projected_quad_fully_inside_viewport(quad.vertices))
4032         return stage_projected_quad_no_clip(ctx, &quad);
4033     return renderer_push_quad(ctx, &quad);
4034 }
4035
4036 static bool project_sky_direction(RenderContext *ctx, Vec3 dir, float *screen_x, float *screen_y)
4037 {
4038     Vec3 world = {
4039         ctx->current_camera.position.x + dir.x * SKY_DOME_DISTANCE,

```

```

4040         ctx->current_camera.position.y + dir.y * SKY_DOME_DISTANCE,
4041         ctx->current_camera.position.z + dir.z * SKY_DOME_DISTANCE,
4042     };
4043     CameraVertex camera = {0};
4044     Vertex2D projected;
4045
4046     world_to_camera(ctx, world, &camera);
4047     if (!project_camera_vertex(ctx, &camera, &projected))
4048         return false;
4049
4050     *screen_x = projected.x;
4051     *screen_y = projected.y;
4052     return true;
4053 }
4054
4055 static uint32_t sky_hash_u32(uint32_t value)
4056 {
4057     value ^= value >> 16;
4058     value *= 0x7feb352du;
4059     value ^= value >> 15;
4060     value *= 0x846ca68bu;
4061     value ^= value >> 16;
4062     return value;
4063 }
4064
4065 static float sky_hash01(uint32_t value)
4066 {
4067     return (float)(sky_hash_u32(value) & 0x0fffffffu) *
4068         (1.0f / 16777215.0f);
4069 }
4070
4071 static bool draw_sky_sprite_rotated(RenderContext *ctx,
4072                                     Vec3 dir,
4073                                     float width_px,
4074                                     float height_px,
4075                                     float rotation,
4076                                     uint8_t texture_id,
4077                                     uint8_t extra_flags)
4078 {
4079     float center_x;
4080     float center_y;
4081     float half_w = width_px * 0.5f;
4082     float half_h = height_px * 0.5f;
4083     float cr = cosf(rotation);
4084     float sr = sinf(rotation);
4085     float rx = cr * half_w;
4086     float ry = sr * half_w;
4087     float dx = -sr * half_h;
4088     float dy = cr * half_h;
4089
4090     if (!project_sky_direction(ctx, dir, &center_x, &center_y))
4091         return false;
4092
4093     return renderer_draw_custom_screen_quad(ctx,
4094                                             center_x - rx - dx,
4095                                             center_y - ry - dy,
4096                                             center_x + rx - dx,
4097                                             center_y + ry - dy,
4098                                             center_x + rx + dx,

```

```

4099         center_y + ry + dy,
4100         center_x - rx + dx,
4101         center_y - ry + dy,
4102         texture_id,
4103         extra_flags);
4104     }
4105
4106     static bool draw_sky_sprite(RenderContext *ctx, Vec3 dir, float size_px, uint8_t texture_id)
4107     {
4108         float center_x;
4109         float center_y;
4110         float half = size_px * 0.5f;
4111
4112         if (!project_sky_direction(ctx, dir, &center_x, &center_y))
4113             return false;
4114
4115         return push_screen_textured_quad(ctx,
4116                                         center_x - half, center_y - half,
4117                                         center_x + half, center_y + half,
4118                                         0.0f, 0.0f, 16.0f, 16.0f,
4119                                         texture_id,
4120                                         QUAD_FLAG_ALPHA_KEY);
4121     }
4122
4123     static void draw_randomized_stars(RenderContext *ctx)
4124     {
4125         for (uint32_t i = 0; i < SKY_STAR_COUNT; i++) {
4126             float azimuth = sky_hash01(i * 13u + 0x51a7u) * (2.0f * PI_F);
4127             float elevation_t = sky_hash01(i * 17u + 0x932bu);
4128             float size_t = sky_hash01(i * 19u + 0xb31du);
4129             float aspect_t = sky_hash01(i * 23u + 0xc0deu);
4130             float alpha_t = sky_hash01(i * 29u + 0x57a1u);
4131             float elevation = asinf(0.10f + 0.86f * elevation_t);
4132             float size_px = 10.0f + 8.0f * size_t;
4133             float width_px = size_px * (0.86f + 0.28f * aspect_t);
4134             float height_px = size_px * (1.14f - 0.28f * aspect_t);
4135             float rotation = sky_hash01(i * 31u + 0x6f3du) * (2.0f * PI_F);
4136             uint8_t alpha = QUAD_ALPHA_OPAQUE;
4137
4138             if (alpha_t < 0.18f)
4139                 alpha = QUAD_ALPHA_50;
4140             else if (alpha_t < 0.62f)
4141                 alpha = QUAD_ALPHA_75;
4142
4143             draw_sky_sprite_rotated(ctx,
4144                                     direction_from_azimuth_elevation(azimuth,
4145                                                                     elevation),
4146                                     width_px,
4147                                     height_px,
4148                                     rotation,
4149                                     TEX_TILE_STARS,
4150                                     QUAD_FLAG_ALPHA_KEY | alpha);
4151         }
4152     }
4153
4154     static RGB24 sample_sky_gradient(const SkyPalette *palette, float t)
4155     {
4156         if (t <= 0.22f)
4157             return lerp_rgb24(palette->zenith, palette->high, smoothstepf(0.0f, 0.22f, t));

```

```

4158     if (t <= 0.62f)
4159         return lerp_rgb24(palette->high, palette->mid, smoothstepf(0.22f, 0.62f, t));
4160     return lerp_rgb24(palette->mid, palette->horizon, smoothstepf(0.62f, 1.0f, t));
4161 }
4162
4163 static void draw_sky_gradient(RenderContext *ctx, const SkyPalette *palette)
4164 {
4165     for (int band = 0; band < SKY_GRADIENT_BANDS; band++) {
4166         float y0 = SCREEN_HEIGHT * (float)band / (float)SKY_GRADIENT_BANDS;
4167         float y1 = SCREEN_HEIGHT * (float)(band + 1) / (float)SKY_GRADIENT_BANDS;
4168         float t = ((float)band + 0.5f) / (float)SKY_GRADIENT_BANDS;
4169         uint8_t palette_index = (uint8_t)(PAL_SKY_GRADIENT_BASE + band);
4170         RGB24 color = sample_sky_gradient(palette, t);
4171
4172         renderer_set_sky_palette_rgb(ctx, (uint8_t)band, color);
4173         push_screen_flat_quad(ctx, 0.0f, y0, SCREEN_WIDTH, y1, palette_index);
4174     }
4175 }
4176
4177 static Vec3 sun_direction_for_time(float time_seconds)
4178 {
4179     float cycle = fmodf(time_seconds / SKY_DAY_LENGTH_SECONDS + 0.18f, 1.0f);
4180
4181     if (cycle < 0.0f)
4182         cycle += 1.0f;
4183
4184     float orbit = cycle * (2.0f * PI_F);
4185     return normalize_vec3((Vec3){
4186         cosf(orbit),
4187         0.92f * sinf(orbit),
4188         0.35f,
4189     });
4190 }
4191
4192 static SkyPalette make_sky_palette(Vec3 sun_dir)
4193 {
4194     float daylight = smoothstepf(-0.18f, 0.08f, sun_dir.y);
4195     float twilight = clamp01(1.0f - fabsf(sun_dir.y) * 4.5f);
4196     float night = 1.0f - daylight;
4197     SkyPalette palette = {
4198         .zenith = lerp_rgb24(rgb24(0x08, 0x0a, 0x16), rgb24(0x58, 0x96, 0xdb), daylight),
4199         .high = lerp_rgb24(rgb24(0x15, 0x18, 0x31), rgb24(0x7f, 0xbd, 0xff), daylight),
4200         .mid = lerp_rgb24(rgb24(0x22, 0x24, 0x46), rgb24(0xad, 0xd8, 0xff), daylight),
4201         .horizon = lerp_rgb24(rgb24(0x2c, 0x1f, 0x2d), rgb24(0xdf, 0xee, 0xff), daylight),
4202         .cloud = lerp_rgb24(rgb24(0x54, 0x5c, 0x79), rgb24(0xf5, 0xfa, 0xff), daylight),
4203         .cloud_shadow = lerp_rgb24(rgb24(0x34, 0x3b, 0x53), rgb24(0xcb, 0xd8, 0xec), daylight),
4204         .sun_core = rgb24(0xff, 0xee, 0xaa),
4205         .sun_glow = rgb24(0xff, 0xbd, 0x58),
4206         .moon = lerp_rgb24(rgb24(0xb9, 0xc3, 0xe0), rgb24(0xe7, 0xeb, 0xf8), night),
4207         .moon_shadow = lerp_rgb24(rgb24(0x5d, 0x67, 0x85), rgb24(0x9c, 0xa4, 0xc0), night),
4208     };
4209     RGB24 sunset = rgb24(0xff, 0xab, 0x61);
4210
4211     twilight *= 0.35f + 0.65f * night;
4212     palette.zenith = mix_rgb24(palette.zenith, rgb24(0x52, 0x3d, 0x5d), twilight * 0.22f);
4213     palette.high = mix_rgb24(palette.high, rgb24(0xa0, 0x5f, 0x62), twilight * 0.28f);
4214     palette.mid = mix_rgb24(palette.mid, rgb24(0xe7, 0x88, 0x56), twilight * 0.45f);
4215     palette.horizon = mix_rgb24(palette.horizon, sunset, twilight * 0.78f);
4216     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);

```

```

4275 update_world_light_from_sun(ctx, palette_sun_dir);
4276 daylight = ctx->world_daylight;
4277 night = 1.0f - daylight;
4278 palette = make_sky_palette(palette_sun_dir);
4279 upload_sky_palette(ctx, &palette);
4280 draw_sky_gradient(ctx, &palette);
4281
4282 /* Sprite positions use continuous time so movement is smooth. */
4283 sun_dir = sun_direction_for_time(time_seconds);
4284 moon_dir = (Vec3){ -sun_dir.x, -sun_dir.y, -sun_dir.z };
4285
4286 if (night > 0.2f)
4287     draw_randomized_stars(ctx);
4288
4289 if (sun_dir.y > 0.0f)
4290     draw_sky_sprite(ctx, sun_dir, 34.0f, TEX_TILE_SUN);
4291 if (moon_dir.y > 0.0f)
4292     draw_sky_sprite(ctx, moon_dir, 30.0f, TEX_TILE_MOON);
4293
4294 for (size_t i = 0; i < sizeof(cloud_defs) / sizeof(cloud_defs[0]); i++) {
4295     float azimuth = cloud_defs[i].azimuth + cloud_defs[i].drift * time_seconds;
4296     float elevation = cloud_defs[i].elevation +
4297         0.04f * sinf(time_seconds * cloud_defs[i].wobble + (float)i * 1.7f);
4298
4299     draw_sky_sprite(ctx,
4300         direction_from_azimuth_elevation(azimuth, elevation),
4301         cloud_defs[i].size_px,
4302         TEX_TILE_CLOUD);
4303 }
4304
4305 return ctx->n_quads - before;
4306 }
4307
4308 /* Project a world-space vertical line segment to screen and draw it as a
4309 * 1-pixel-wide line using a screen-space quad. Returns true if visible. */
4310 static bool draw_projected_line(RenderContext *ctx,
4311     float wx0, float wy0, float wz0,
4312     float wx1, float wy1, float wz1,
4313     uint8_t palette_index)
4314 {
4315     CameraVertex cam0, cam1;
4316     Vertex2D scr0, scr1;
4317
4318     world_to_camera(ctx, (Vec3){wx0, wy0, wz0}, &cam0);
4319     world_to_camera(ctx, (Vec3){wx1, wy1, wz1}, &cam1);
4320     if (!project_camera_vertex(ctx, &cam0, &scr0) ||
4321         !project_camera_vertex(ctx, &cam1, &scr1))
4322         return false;
4323
4324     float dx = scr1.x - scr0.x;
4325     float dy = scr1.y - scr0.y;
4326     float len = sqrtf(dx * dx + dy * dy);
4327     if (len < 0.1f) return false;
4328
4329     /* Normal vector to the line in screen space */
4330     float nx = -dy / len;
4331     float ny = dx / len;
4332
4333     /* 0.5px half-width for a 1-pixel wide line */

```

```

4334     float hw = 0.5f;
4335     nx *= hw;
4336     ny *= hw;
4337
4338     /* Build a quad connecting the two points, 1px thick.
4339     * Force z=0.0f (nearest possible depth) so the debug lines
4340     * draw cleanly over all scene blocks. */
4341     RenderQuad quad = {0};
4342     quad.color_tint = palette_index;
4343     quad.flags = 0;
4344
4345     float inv_w0 = scr0.one_over_w;
4346     float inv_w1 = scr1.one_over_w;
4347
4348     quad.vertices[0] = (Vertex2D){ scr0.x - nx, scr0.y - ny, 0.0f, 0.0f, inv_w0, 0.0f };
4349     quad.vertices[1] = (Vertex2D){ scr0.x + nx, scr0.y + ny, 0.0f, 0.0f, inv_w0, 0.0f };
4350     quad.vertices[2] = (Vertex2D){ scr1.x + nx, scr1.y + ny, 0.0f, 0.0f, inv_w1, 0.0f };
4351     quad.vertices[3] = (Vertex2D){ scr1.x - nx, scr1.y - ny, 0.0f, 0.0f, inv_w1, 0.0f };
4352
4353     if (projected_quad_fully_inside_viewport(quad.vertices))
4354         return stage_projected_quad_no_clip(ctx, &quad);
4355     return renderer_push_quad(ctx, &quad);
4356 }
4357
4358 int renderer_draw_chunk_borders(RenderContext *ctx,
4359                                float player_x, float player_z,
4360                                int render_distance)
4361 {
4362     if (!ctx || render_distance <= 0)
4363         return 0;
4364
4365     int drawn = 0;
4366     int player_cx = (int)floorf(player_x / (float)WORLD_CHUNK_SIZE);
4367     int player_cz = (int)floorf(player_z / (float)WORLD_CHUNK_SIZE);
4368
4369     /* Draw vertical lines at chunk boundaries within render distance.
4370     * Lines run from y=0 to y=WORLD_CHUNK_HEIGHT at each chunk edge. */
4371     float y_bot = 0.0f;
4372     float y_top = (float)WORLD_CHUNK_HEIGHT;
4373
4374     for (int dx = -render_distance; dx <= render_distance + 1; dx++) {
4375         float wx = (float)((player_cx + dx) * WORLD_CHUNK_SIZE);
4376         /* Highlight the player's chunk borders in yellow (palette 8),
4377         * other chunk borders in white (palette 5). */
4378         bool is_player_edge = (dx == 0 || dx == 1);
4379         uint8_t color = is_player_edge ? 8 : 5;
4380
4381         for (int dz = -render_distance; dz <= render_distance + 1; dz++) {
4382             float wz = (float)((player_cz + dz) * WORLD_CHUNK_SIZE);
4383             bool is_player_z = (dz == 0 || dz == 1);
4384             uint8_t line_color = (is_player_edge && is_player_z) ? 8 : color;
4385
4386             /* Vertical line at this grid intersection. */
4387             if (draw_projected_line(ctx, wx, y_bot, wz, wx, y_top, wz,
4388                                   line_color))
4389                 drawn++;
4390         }
4391     }
4392 }

```

```

4393     return drawn;
4394 }
4395
4396 bool renderer_draw_crosshair(RenderContext *ctx)
4397 {
4398     const float cx = SCREEN_WIDTH * 0.5f;
4399     const float cy = SCREEN_HEIGHT * 0.5f;
4400     const float half = 4.5f * HUD_SCALE;
4401     const float thickness = 0.75f * HUD_SCALE;
4402     const uint8_t white = 5;
4403     bool ok = true;
4404
4405     ok &= renderer_fill_rect(ctx,
4406                             cx - half, cy - thickness * 0.5f,
4407                             cx + half, cy + thickness * 0.5f,
4408                             white, QUAD_ALPHA_75);
4409     ok &= renderer_fill_rect(ctx,
4410                             cx - thickness * 0.5f, cy - half,
4411                             cx + thickness * 0.5f, cy + half,
4412                             white, QUAD_ALPHA_75);
4413     return ok;
4414 }

```

Complete voxel_gpu.sv RTL listing

Listing 3: Complete voxel_gpu.sv RTL source.

```

1 // voxel_gpu.sv - SDRAM-backed display path with RGB565 external frames.
2 //
3 // The rasterizer renders at true 640x480 through an explicit 640x60 BRAM band
4 // cache. Userspace bins descriptors into eight vertical passes and brackets each
5 // pass with BEGIN_BAND / END_BAND CSRs. Palette entries are still the
6 // source-color ABI, but resolved pixels are stored as RGB565 so translucent
7 // quads can alpha blend against the existing destination pixel.
8 // FLIP no longer copies a full BRAM framebuffer. Instead:
9 // * BEGIN_BAND clears the resident Z band; untouched color is lazy and
10 //   regenerated as sky/clear where Z still equals 0xFFFF,
11 // * the rasterizer writes only pixels that fall inside that resident band,
12 // * END_BAND flushes the color band to the inactive SDRAM color frame, and
13 // * scanout reads the active SDRAM frame through small line buffers.
14 //
15 // This keeps BRAM usage close to the indexed design while making the external
16 // framebuffer itself true color.
17 //
18 // Reading map for teammates:
19 // 1. Register/FIFO front door: HPS writes CSRs and descriptor words.
20 // 2. Main engine FSM: fetches one descriptor, prepares one resident band,
21 //   rasterizes pixels, and kicks background band flushes.
22 // 3. Pixel pipeline: edge test -> reciprocal/texture -> palette -> fog ->
23 //   color/Z commit. Valid bits move alongside data through every stage.
24 // 4. SDRAM background flush and VGA scanout run beside the main FSM. Most
25 //   "weird" gates below protect ownership between those concurrent users.
26 // 5. Leaf helper modules/'svh' files contain pure math/color helpers only;
27 //   stateful timing stays in this module so the control flow remains visible.
28
29 module voxel_gpu (
30     input logic      clk,

```

```

31     input logic      reset,
32
33     // Avalon-MM slave (address units = WORDS; writedata is 32 bits).
34     input logic [12:0] address,
35     input logic      chipselect,
36     input logic      write,
37     input logic [31:0] writedata,
38     input logic [3:0] byteenable,
39     output logic [31:0] readdata,
40
41     // VGA conduit
42     output logic [7:0] VGA_R,
43     output logic [7:0] VGA_G,
44     output logic [7:0] VGA_B,
45     output logic      VGA_CLK,
46     output logic      VGA_HS,
47     output logic      VGA_VS,
48     output logic      VGA_BLANK_n,
49     output logic      VGA_SYNC_n,
50
51     // Board SDR SDRAM conduit
52     output logic [12:0] DRAM_ADDR,
53     output logic [1:0] DRAM_BA,
54     output logic      DRAM_CAS_N,
55     output logic      DRAM_CKE,
56     output logic      DRAM_CLK,
57     output logic      DRAM_CS_N,
58     inout wire [15:0] DRAM_DQ,
59     output logic      DRAM_LDQM,
60     output logic      DRAM_RAS_N,
61     output logic      DRAM_UDQM,
62     output logic      DRAM_WE_N
63 );
64
65     // -----
66     // Register Map, Frame Geometry, and Packed Descriptor Constants
67     // -----
68
69     localparam logic [12:0] ADDR_CONTROL = 13'h000;
70     localparam logic [12:0] ADDR_STATUS = 13'h001;
71     localparam logic [12:0] ADDR_FRAMECNT = 13'h002;
72     localparam logic [12:0] ADDR_PAL_ADDR = 13'h003;
73     localparam logic [12:0] ADDR_PAL_DATA = 13'h004;
74     localparam logic [12:0] ADDR_FOG_RANGE = 13'h005;
75     localparam logic [12:0] ADDR_FOG_CTRL = 13'h006;
76     localparam logic [12:0] ADDR_EXTMEM_CTRL = 13'h007;
77     localparam logic [12:0] ADDR_EXTMEM_FRONT = 13'h008;
78     localparam logic [12:0] ADDR_EXTMEM_BACK = 13'h009;
79     localparam logic [12:0] ADDR_EXTMEM_STRIDE = 13'h00A;
80     localparam logic [12:0] ADDR_EXTMEM_TILE = 13'h00B;
81     localparam logic [12:0] ADDR_EXTMEM_STAT = 13'h00C;
82     localparam logic [12:0] ADDR_BAND_INDEX = 13'h00D;
83     localparam logic [12:0] ADDR_BAND_CTRL = 13'h00E;
84     localparam logic [12:0] ADDR_BAND_WINDOW = 13'h00F;
85     localparam logic [12:0] ADDR_PERF_DRAW_ACT = 13'h010;
86     localparam logic [12:0] ADDR_PERF_DRAW_IDLE = 13'h011;
87     localparam logic [12:0] ADDR_PERF_FLUSH_ACT = 13'h012;
88     localparam logic [12:0] ADDR_PERF_FLUSH_STL = 13'h013;
89     localparam logic [12:0] ADDR_PERF_INIT = 13'h014;

```

```

90 localparam logic [12:0] ADDR_PERF_LOAD      = 13'h015;
91 localparam logic [12:0] ADDR_PERF_FLUSH_LOAD = 13'h016;
92 localparam logic [12:0] ADDR_PERF_FLUSH_FIFO = 13'h017;
93 localparam logic [12:0] ADDR_PERF_FLUSH_DATA = 13'h018;
94 localparam logic [12:0] ADDR_PERF_FLUSH_DRAIN = 13'h019;
95 localparam logic [12:0] ADDR_SKY_PAL_ADDR = 13'h01A;
96 localparam logic [12:0] ADDR_SKY_PAL_DATA = 13'h01B;
97 localparam logic [12:0] ADDR_FIFO_LO = 13'h400; // 0x1000
98 localparam logic [12:0] ADDR_FIFO_HI = 13'h800; // 0x2000 (exclusive)
99
100 localparam int FB_WIDTH      = 640;
101 localparam int BAND_HEIGHT   = 60;
102 localparam int BAND_PIXELS   = FB_WIDTH * BAND_HEIGHT;
103 localparam int FIFO_DEPTH    = 1024;
104 localparam int BASE_QUAD_WORDS = 16;
105 localparam int UV_QUAD_WORDS  = 9;
106 localparam int MAX_DESC_WORDS = BASE_QUAD_WORDS + UV_QUAD_WORDS;
107 localparam logic [5:0] BASE_QUAD_WORDS_6 = 6'd16;
108 localparam logic [5:0] MAX_DESC_WORDS_6 = 6'd25;
109 localparam int TEXTURE_BYTES  = 128 * 16 * 16;
110 localparam int LINE_WORDS     = FB_WIDTH;
111 /* Stage background-flush words into the 512-word SDRAM WR FIFO while
112  * scanout owns the bus; keep headroom for in-flight pushes. */
113 localparam logic [8:0] COPY_WR_FIFO_HIGH_WATER = 9'd224;
114 localparam logic [7:0] COPY_DRAIN_CYCLES = 8'd128;
115 /* Drain every valid pipeline stage between fetch and color commit. */
116 localparam logic [3:0] DRAW_FLUSH_CYCLES = 4'd14;
117 localparam logic [24:0] READ_BURST_WORDS_25 = 25'd64;
118 localparam logic [9:0] LINE_WORDS_10 = 10'd640;
119 localparam logic [8:0] COPY_BURST_WORDS_9 = 9'd128;
120 localparam logic [8:0] READ_BURST_WORDS_9 = 9'd64;
121 localparam logic [10:0] ACTIVE_WRITE_END_HCOUNT = 11'd1120;
122 localparam int EXTMEM_SKY_GRADIENT_CLEAR_BIT = 5;
123 localparam logic [7:0] PAL_SKY_GRADIENT_BASE = 8'd40;
124 localparam logic [7:0] PAL_SKY_GRADIENT_LAST = 8'd63;
125 localparam int SKY_GRADIENT_COLORS = 24;
126 localparam logic [4:0] SKY_GRADIENT_LAST_INDEX = 5'd23;
127 localparam logic [5:0] SKY_GRADIENT_COLORS_6 = 6'd24;
128 localparam logic [15:0] Z_CLEAR_SENTINEL = 16'hFFFF;
129 localparam logic [15:0] Z_VALID_FAR      = 16'hFFFE;
130 localparam logic [31:0] DEFAULT_EXTMEM_CTRL = 32'h0000_002B;
131 localparam logic [31:0] DEFAULT_EXTMEM_FRONT_BASE = 32'd0;
132 localparam logic [31:0] DEFAULT_EXTMEM_BACK_BASE = 32'd1048576; // 1MB
133 localparam logic [31:0] DEFAULT_EXTMEM_STRIDE = 32'd1280;
134 localparam logic [17:0] SDRAM_POWERUP_HOLD_LAST = 18'd199999;
135 localparam logic [15:0] SDRAM_INIT_WAIT_LAST = 16'd31999;
136 localparam int FLAG_TEX_BIT      = 0;
137 localparam int FLAG_ZTEST_BIT    = 1;
138 localparam int FLAG_ALPHA_KEY_BIT = 2;
139 localparam int FLAG_FOG_BIT      = 3;
140 localparam int FLAG_LIGHT_LSB    = 4;
141 localparam int FLAG_LIGHT_MSB    = 5;
142 localparam int FLAG_ALPHA_LSB    = 6;
143 localparam int FLAG_ALPHA_MSB    = 7;
144
145 // -----
146 // Main Engine State Map
147 // -----
148 //

```

```

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;

```

```

267 logic signed [63:0] vw_row_val, vw_cur_val;
268 logic signed [63:0] iw_row_val, iw_cur_val;
269 // -----
270 // Pixel Pipeline Registers
271 // -----
272 //
273 // Unsuffixed lane = even pixel of the 2-pixel pair.
274 // '_o' lane       = odd pixel of the same pair.
275 //
276 // Stage notes name the field that changes; unmentioned metadata is carried
277 // forward to keep the two lanes aligned.
278
279 // pipe0: raster pair accepted; cache read addresses, z, and UV/w are sampled.
280 logic      pipe0_valid;
281 logic      pipe0_inside;
282 logic      pipe0_ztest;
283 logic      pipe0_textured;
284 logic      pipe0_alpha_key;
285 logic [1:0] pipe0_alpha;
286 logic      pipe0_fog;
287 logic [1:0] pipe0_light_bank;
288 logic [7:0] pipe0_tex_or_color;
289 logic [15:0] pipe0_addr;
290 logic [15:0] pipe0_z;
291 logic [9:0] pipe0_x;
292 logic [8:0] pipe0_y;
293 logic signed [31:0] pipe0_uw_q;
294 logic signed [31:0] pipe0_vw_q;
295 logic [31:0] pipe0_iw_q;
296 // recip0: 1/w is normalized and the LUT index/fraction is derived.
297 logic      recip0_valid;
298 logic      recip0_inside;
299 logic      recip0_ztest;
300 logic      recip0_textured;
301 logic      recip0_alpha_key;
302 logic [1:0] recip0_alpha;
303 logic      recip0_fog;
304 logic [1:0] recip0_light_bank;
305 logic [7:0] recip0_tex_or_color;
306 logic [15:0] recip0_addr;
307 logic [15:0] recip0_z;
308 logic [9:0] recip0_x;
309 logic [8:0] recip0_y;
310 logic signed [31:0] recip0_uw_q;
311 logic signed [31:0] recip0_vw_q;
312 logic      recip0_iw_zero;
313 logic [5:0] recip0_iw_msb;
314 logic [31:0] recip0_iw_norm_q;
315 // recip1: destination color/Z return from the band cache; reciprocal LUT read.
316 logic      recip1_valid;
317 logic      recip1_inside;
318 logic      recip1_ztest;
319 logic      recip1_textured;
320 logic      recip1_alpha_key;
321 logic [1:0] recip1_alpha;
322 logic      recip1_fog;
323 logic [1:0] recip1_light_bank;
324 logic [7:0] recip1_tex_or_color;
325 logic [15:0] recip1_addr;

```

```

326 logic [15:0] recip1_z;
327 logic [15:0] recip1_z_ref;
328 logic [15:0] recip1_dst_rgb565;
329 logic [9:0] recip1_x;
330 logic [8:0] recip1_y;
331 logic signed [31:0] recip1_uw_q;
332 logic signed [31:0] recip1_vw_q;
333 logic recip1_iw_zero;
334 logic [5:0] recip1_iw_msb;
335 logic [5:0] recip1_iw_lut_frac;
336 logic [31:0] recip1_w_norm_lo;
337 logic [31:0] recip1_w_norm_hi;
338 // recip2: reciprocal interpolation finishes; normalized w is registered.
339 logic recip2_valid;
340 logic recip2_inside;
341 logic recip2_ztest;
342 logic recip2_textured;
343 logic recip2_alpha_key;
344 logic [1:0] recip2_alpha;
345 logic recip2_fog;
346 logic [1:0] recip2_light_bank;
347 logic [7:0] recip2_tex_or_color;
348 logic [15:0] recip2_addr;
349 logic [15:0] recip2_z;
350 logic [15:0] recip2_z_ref;
351 logic [15:0] recip2_dst_rgb565;
352 logic [9:0] recip2_x;
353 logic [8:0] recip2_y;
354 logic signed [31:0] recip2_uw_q;
355 logic signed [31:0] recip2_vw_q;
356 logic recip2_iw_zero;
357 logic [5:0] recip2_iw_msb;
358 logic [31:0] recip2_w_norm_q;
359 // pipe1: w is denormalized and aligned with UV and destination metadata.
360 logic pipe1_valid;
361 logic pipe1_inside;
362 logic pipe1_ztest;
363 logic pipe1_textured;
364 logic pipe1_alpha_key;
365 logic [1:0] pipe1_alpha;
366 logic pipe1_fog;
367 logic [1:0] pipe1_light_bank;
368 logic [7:0] pipe1_tex_or_color;
369 logic [15:0] pipe1_addr;
370 logic [15:0] pipe1_z;
371 logic [15:0] pipe1_z_ref;
372 logic [15:0] pipe1_dst_rgb565;
373 logic [9:0] pipe1_x;
374 logic [8:0] pipe1_y;
375 logic signed [31:0] pipe1_uw_q;
376 logic signed [31:0] pipe1_vw_q;
377 logic [31:0] pipe1_w_q;
378 // tex0: perspective UV products are registered.
379 logic tex0_valid;
380 logic tex0_inside;
381 logic tex0_ztest;
382 logic tex0_textured;
383 logic tex0_alpha_key;
384 logic [1:0] tex0_alpha;

```

```

385     logic         tex0_fog;
386     logic  [1:0] tex0_light_bank;
387     logic  [7:0] tex0_tex_or_color;
388     logic [15:0] tex0_addr;
389     logic [15:0] tex0_z;
390     logic [15:0] tex0_z_ref;
391     logic [15:0] tex0_dst_rgb565;
392     logic  [9:0] tex0_x;
393     logic  [8:0] tex0_y;
394     logic [31:0] tex0_w_q;
395     logic signed [63:0] tex0_u_prod;
396     logic signed [63:0] tex0_v_prod;
397     // pipe2: texture coordinates become atlas addresses.
398     logic         pipe2_valid;
399     logic         pipe2_inside;
400     logic         pipe2_ztest;
401     logic         pipe2_textured;
402     logic         pipe2_alpha_key;
403     logic  [1:0] pipe2_alpha;
404     logic         pipe2_fog;
405     logic  [1:0] pipe2_light_bank;
406     logic  [7:0] pipe2_tex_or_color;
407     logic [15:0] pipe2_addr;
408     logic [15:0] pipe2_z;
409     logic [15:0] pipe2_z_ref;
410     logic [15:0] pipe2_dst_rgb565;
411     logic  [9:0] pipe2_x;
412     logic  [8:0] pipe2_y;
413     logic [31:0] pipe2_w_q;
414     logic [14:0] pipe2_tex_addr;
415     // draw_pipe: ROM texel data is aligned with color/Z metadata.
416     logic         draw_pipe_valid;
417     logic         draw_pipe_inside;
418     logic         draw_pipe_ztest;
419     logic         draw_pipe_textured;
420     logic         draw_pipe_alpha_key;
421     logic  [1:0] draw_pipe_alpha;
422     logic         draw_pipe_fog;
423     logic  [1:0] draw_pipe_light_bank;
424     logic  [7:0] draw_pipe_tex_or_color;
425     logic [15:0] draw_pipe_addr;
426     logic [15:0] draw_pipe_z;
427     logic [15:0] draw_pipe_z_ref;
428     logic [15:0] draw_pipe_dst_rgb565;
429     logic  [9:0] draw_pipe_x;
430     logic  [8:0] draw_pipe_y;
431     logic [31:0] draw_pipe_w_q;
432     logic         draw_is_band_primer;
433
434     // pal_rd: palette/fog addresses are flopped before the array read.
435     logic         pal_rd_valid;
436     logic         pal_rd_pass;
437     logic         pal_rd_ztest;
438     logic  [1:0] pal_rd_alpha;
439     logic         pal_rd_fog;
440     logic [15:0] pal_rd_addr;
441     logic [15:0] pal_rd_z;
442     logic  [7:0] pal_rd_src_addr;
443     logic  [7:0] pal_rd_fog_addr;

```

```

444 logic pal_rd_src_sky;
445 logic [4:0] pal_rd_sky_index;
446 logic [15:0] pal_rd_dst_rgb565;
447 logic [31:0] pal_rd_w_q;
448 logic [33:0] pal_rd_ray_scale_q16;
449
450 // plr: palette/fog RGB values are captured and kept with fog metadata.
451 logic plr_valid;
452 logic plr_pass;
453 logic plr_ztest;
454 logic [1:0] plr_alpha;
455 logic plr_fog;
456 logic [15:0] plr_addr;
457 logic [15:0] plr_z;
458 logic [23:0] plr_src_rgb;
459 logic [15:0] plr_dst_rgb565;
460 logic [23:0] plr_fog_rgb;
461 logic [31:0] plr_w_q;
462 logic [33:0] plr_ray_scale_q16;
463
464 // fog0: RGB888 converts to RGB565; radial distance multiply starts.
465 logic fog0_valid;
466 logic fog0_pass;
467 logic fog0_ztest;
468 logic [1:0] fog0_alpha;
469 logic fog0_fog;
470 logic [15:0] fog0_addr;
471 logic [15:0] fog0_z;
472 logic [15:0] fog0_src_rgb565;
473 logic [15:0] fog0_dst_rgb565;
474 logic [15:0] fog0_fog_rgb565;
475 logic [31:0] fog0_w_q;
476 logic [33:0] fog0_ray_scale_q16;
477 // fog1: radial distance is in Q8.8 for fog blend.
478 logic fog1_valid;
479 logic fog1_pass;
480 logic fog1_ztest;
481 logic [1:0] fog1_alpha;
482 logic fog1_fog;
483 logic [15:0] fog1_addr;
484 logic [15:0] fog1_z;
485 logic [15:0] fog1_src_rgb565;
486 logic [15:0] fog1_dst_rgb565;
487 logic [15:0] fog1_fog_rgb565;
488 logic [15:0] fog1_radial_q8_8;
489 // commit: final RGB565 and Z write-back controls.
490 logic commit_valid;
491 logic commit_pass;
492 logic commit_ztest;
493 logic [15:0] commit_addr;
494 logic [15:0] commit_z;
495 logic [15:0] commit_color;
496
497 /* Odd lane for the 2 px/cycle raster pipe. Existing unsuffixed
498 * registers are lane0/even; '_o' registers carry lane1/odd. */
499 logic pipe0_valid_o;
500 logic pipe0_inside_o;
501 logic pipe0_ztest_o;
502 logic pipe0_textured_o;

```

```

503     logic        pipe0_alpha_key_o;
504     logic  [1:0] pipe0_alpha_o;
505     logic        pipe0_fog_o;
506     logic  [1:0] pipe0_light_bank_o;
507     logic  [7:0] pipe0_tex_or_color_o;
508     logic [15:0] pipe0_addr_o;
509     logic [15:0] pipe0_z_o;
510     logic  [9:0] pipe0_x_o;
511     logic  [8:0] pipe0_y_o;
512     logic signed [31:0] pipe0_uw_q_o;
513     logic signed [31:0] pipe0_vw_q_o;
514     logic [31:0] pipe0_iw_q_o;
515     logic        recip0_valid_o;
516     logic        recip0_inside_o;
517     logic        recip0_ztest_o;
518     logic        recip0_textured_o;
519     logic        recip0_alpha_key_o;
520     logic  [1:0] recip0_alpha_o;
521     logic        recip0_fog_o;
522     logic  [1:0] recip0_light_bank_o;
523     logic  [7:0] recip0_tex_or_color_o;
524     logic [15:0] recip0_addr_o;
525     logic [15:0] recip0_z_o;
526     logic  [9:0] recip0_x_o;
527     logic  [8:0] recip0_y_o;
528     logic signed [31:0] recip0_uw_q_o;
529     logic signed [31:0] recip0_vw_q_o;
530     logic        recip0_iw_zero_o;
531     logic  [5:0] recip0_iw_msb_o;
532     logic [31:0] recip0_iw_norm_q_o;
533     logic        recip1_valid_o;
534     logic        recip1_inside_o;
535     logic        recip1_ztest_o;
536     logic        recip1_textured_o;
537     logic        recip1_alpha_key_o;
538     logic  [1:0] recip1_alpha_o;
539     logic        recip1_fog_o;
540     logic  [1:0] recip1_light_bank_o;
541     logic  [7:0] recip1_tex_or_color_o;
542     logic [15:0] recip1_addr_o;
543     logic [15:0] recip1_z_o;
544     logic [15:0] recip1_z_ref_o;
545     logic [15:0] recip1_dst_rgb565_o;
546     logic  [9:0] recip1_x_o;
547     logic  [8:0] recip1_y_o;
548     logic signed [31:0] recip1_uw_q_o;
549     logic signed [31:0] recip1_vw_q_o;
550     logic        recip1_iw_zero_o;
551     logic  [5:0] recip1_iw_msb_o;
552     logic  [5:0] recip1_iw_lut_frac_o;
553     logic [31:0] recip1_w_norm_lo_o;
554     logic [31:0] recip1_w_norm_hi_o;
555     logic        recip2_valid_o;
556     logic        recip2_inside_o;
557     logic        recip2_ztest_o;
558     logic        recip2_textured_o;
559     logic        recip2_alpha_key_o;
560     logic  [1:0] recip2_alpha_o;
561     logic        recip2_fog_o;

```

```

562 logic [1:0] recip2_light_bank_o;
563 logic [7:0] recip2_tex_or_color_o;
564 logic [15:0] recip2_addr_o;
565 logic [15:0] recip2_z_o;
566 logic [15:0] recip2_z_ref_o;
567 logic [15:0] recip2_dst_rgb565_o;
568 logic [9:0] recip2_x_o;
569 logic [8:0] recip2_y_o;
570 logic signed [31:0] recip2_uw_q_o;
571 logic signed [31:0] recip2_vw_q_o;
572 logic recip2_iw_zero_o;
573 logic [5:0] recip2_iw_msb_o;
574 logic [31:0] recip2_w_norm_q_o;
575 logic pipe1_valid_o;
576 logic pipe1_inside_o;
577 logic pipe1_ztest_o;
578 logic pipe1_textured_o;
579 logic pipe1_alpha_key_o;
580 logic [1:0] pipe1_alpha_o;
581 logic pipe1_fog_o;
582 logic [1:0] pipe1_light_bank_o;
583 logic [7:0] pipe1_tex_or_color_o;
584 logic [15:0] pipe1_addr_o;
585 logic [15:0] pipe1_z_o;
586 logic [15:0] pipe1_z_ref_o;
587 logic [15:0] pipe1_dst_rgb565_o;
588 logic [9:0] pipe1_x_o;
589 logic [8:0] pipe1_y_o;
590 logic signed [31:0] pipe1_uw_q_o;
591 logic signed [31:0] pipe1_vw_q_o;
592 logic [31:0] pipe1_w_q_o;
593 logic tex0_valid_o;
594 logic tex0_inside_o;
595 logic tex0_ztest_o;
596 logic tex0_textured_o;
597 logic tex0_alpha_key_o;
598 logic [1:0] tex0_alpha_o;
599 logic tex0_fog_o;
600 logic [1:0] tex0_light_bank_o;
601 logic [7:0] tex0_tex_or_color_o;
602 logic [15:0] tex0_addr_o;
603 logic [15:0] tex0_z_o;
604 logic [15:0] tex0_z_ref_o;
605 logic [15:0] tex0_dst_rgb565_o;
606 logic [9:0] tex0_x_o;
607 logic [8:0] tex0_y_o;
608 logic [31:0] tex0_w_q_o;
609 logic signed [63:0] tex0_u_prod_o;
610 logic signed [63:0] tex0_v_prod_o;
611 logic pipe2_valid_o;
612 logic pipe2_inside_o;
613 logic pipe2_ztest_o;
614 logic pipe2_textured_o;
615 logic pipe2_alpha_key_o;
616 logic [1:0] pipe2_alpha_o;
617 logic pipe2_fog_o;
618 logic [1:0] pipe2_light_bank_o;
619 logic [7:0] pipe2_tex_or_color_o;
620 logic [15:0] pipe2_addr_o;

```

```

621 logic [15:0] pipe2_z_o;
622 logic [15:0] pipe2_z_ref_o;
623 logic [15:0] pipe2_dst_rgb565_o;
624 logic [9:0] pipe2_x_o;
625 logic [8:0] pipe2_y_o;
626 logic [31:0] pipe2_w_q_o;
627 logic [14:0] pipe2_tex_addr_o;
628 logic draw_pipe_valid_o;
629 logic draw_pipe_inside_o;
630 logic draw_pipe_ztest_o;
631 logic draw_pipe_textured_o;
632 logic draw_pipe_alpha_key_o;
633 logic [1:0] draw_pipe_alpha_o;
634 logic draw_pipe_fog_o;
635 logic [1:0] draw_pipe_light_bank_o;
636 logic [7:0] draw_pipe_tex_or_color_o;
637 logic [15:0] draw_pipe_addr_o;
638 logic [15:0] draw_pipe_z_o;
639 logic [15:0] draw_pipe_z_ref_o;
640 logic [15:0] draw_pipe_dst_rgb565_o;
641 logic [9:0] draw_pipe_x_o;
642 logic [8:0] draw_pipe_y_o;
643 logic [31:0] draw_pipe_w_q_o;
644 logic pal_rd_valid_o;
645 logic pal_rd_pass_o;
646 logic pal_rd_ztest_o;
647 logic [1:0] pal_rd_alpha_o;
648 logic pal_rd_fog_o;
649 logic [15:0] pal_rd_addr_o;
650 logic [15:0] pal_rd_z_o;
651 logic [7:0] pal_rd_src_addr_o;
652 logic [7:0] pal_rd_fog_addr_o;
653 logic pal_rd_src_sky_o;
654 logic [4:0] pal_rd_sky_index_o;
655 logic [15:0] pal_rd_dst_rgb565_o;
656 logic [31:0] pal_rd_w_q_o;
657 logic [33:0] pal_rd_ray_scale_q16_o;
658 logic plr_valid_o;
659 logic plr_pass_o;
660 logic plr_ztest_o;
661 logic [1:0] plr_alpha_o;
662 logic plr_fog_o;
663 logic [15:0] plr_addr_o;
664 logic [15:0] plr_z_o;
665 logic [23:0] plr_src_rgb_o;
666 logic [15:0] plr_dst_rgb565_o;
667 logic [23:0] plr_fog_rgb_o;
668 logic [31:0] plr_w_q_o;
669 logic [33:0] plr_ray_scale_q16_o;
670 logic fog0_valid_o;
671 logic fog0_pass_o;
672 logic fog0_ztest_o;
673 logic [1:0] fog0_alpha_o;
674 logic fog0_fog_o;
675 logic [15:0] fog0_addr_o;
676 logic [15:0] fog0_z_o;
677 logic [15:0] fog0_src_rgb565_o;
678 logic [15:0] fog0_dst_rgb565_o;
679 logic [15:0] fog0_fog_rgb565_o;

```

```

680 logic [31:0] fog0_w_q_o;
681 logic [33:0] fog0_ray_scale_q16_o;
682 logic fog1_valid_o;
683 logic fog1_pass_o;
684 logic fog1_ztest_o;
685 logic [1:0] fog1_alpha_o;
686 logic fog1_fog_o;
687 logic [15:0] fog1_addr_o;
688 logic [15:0] fog1_z_o;
689 logic [15:0] fog1_src_rgb565_o;
690 logic [15:0] fog1_dst_rgb565_o;
691 logic [15:0] fog1_fog_rgb565_o;
692 logic [15:0] fog1_radial_q8_8_o;
693 logic commit_valid_o;
694 logic commit_pass_o;
695 logic commit_ztest_o;
696 logic [15:0] commit_addr_o;
697 logic [15:0] commit_z_o;
698 logic [15:0] commit_color_o;
699 // tex_rd_data is driven combinationally by voxel_texture_rom's
700 // registered output. The ROM takes pipe2_tex_addr on cycle T and
701 // presents mem[pipe2_tex_addr[T]] on cycle T+1, which is the same
702 // 1-cycle latency the draw_pipe stage expects (see the instance
703 // below and the voxel_texture_rom module header for rationale).
704 wire [7:0] tex_rd_data;
705 wire [7:0] tex_rd_data_o;
706
707 logic [10:0] hcount;
708 logic [9:0] vcount;
709
710 logic scan_visible_now;
711 logic [15:0] scan_rgb565_r;
712 logic scan_visible_r;
713 logic [15:0] draw_addr;
714 logic [15:0] draw_addr_o;
715 logic [15:0] fb_back_rd_addr;
716 logic [15:0] fb_back_rd_addr_o;
717 // fb_back_rd_data is now a wire driven by the ping-pong cache mux
718 logic [15:0] fb_wr_addr;
719 logic [15:0] fb_wr_data;
720 logic fb_back_wr_en;
721 logic [15:0] fb_wr_addr_e;
722 logic [15:0] fb_wr_data_e;
723 logic fb_back_wr_en_e;
724 logic [15:0] fb_wr_addr_o;
725 logic [15:0] fb_wr_data_o;
726 logic fb_back_wr_en_o;
727 logic [15:0] z_rd_addr;
728 logic [15:0] z_rd_addr_o;
729 // z_rd_data is now a wire driven by the ping-pong cache mux
730 logic [15:0] z_wr_addr;
731 logic [15:0] z_wr_data;
732 logic z_wr_en;
733 logic [15:0] z_wr_addr_e;
734 logic [15:0] z_wr_data_e;
735 logic z_wr_en_e;
736 logic [15:0] z_wr_addr_o;
737 logic [15:0] z_wr_data_o;
738 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_sdrwr_addr; // SDRAM write base for flush
807 logic [24:0] flush_sdrwr_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 :
903     extmem_front_base_words;
904 wire [24:0] copy_target_base_words = copy_target_sel ? extmem_back_base_words :
905     extmem_front_base_words;
906 /*
907  * vcount counts 0..524 (VTOTAL-1) but scan_current_row is only 9 bits, so
908  * a naive vcount[8:0] wraps at vcount=512 -- during the back porch the
909  * scanout consumer would see scan_current_row=0..12 and race the advance
910  * logic ahead, popping ~13 prefetched rows before the visible frame even
911  * starts. That shows up as black at the top (source rows 0..12 gone) and
912  * black at the bottom (prefetcher caps at row 479, so active_row stalls
913  * there for the last 13 visible vcounts). Pin scan_current_row to 0 outside
914  * the active region so neither the comparator nor the +1 target lookahead
915  * 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) &&

```

```

1032         !scan_bank2_protected;
1033 wire [1:0] scan_prefetch_bank = scan_bank0_free ? 2'd0 :
1034         scan_bank1_free ? 2'd1 :
1035         scan_bank2_free ? 2'd2 : 2'd3;
1036 wire scan_read_idle = !scan_fill_active && !scan_fill_armed &&
1037         !scan_fill_load_pending &&
1038         sdram_rd_empty;
1039 wire cache_init_state = (state == ST_CACHE_INIT);
1040 /* True when the main FSM still owns the active cache port. */
1041 wire cache_used_by_main = cache_sky_patch_state ||
1042         (state == ST_DRAW) ||
1043         (state == ST_DRAW_FLUSH) ||
1044         cache_init_state;
1045 /* Hold BEGIN until any queued END_BAND has captured its flush indices. */
1046 wire band_begin_cache_available =
1047         !band_flush_pending &&
1048         (!flush_active || flush_generated_sky || ((~draw_cache_sel) != flush_cache_sel));
1049 wire scan_vsync_read_req = vsync_pulse && sdram_ready &&
1050         (copy_complete_pending || display_valid);
1051 /*
1052  * Defense-in-depth: never spawn a scan-fill burst while the SDRAM RD
1053  * FIFO still has residuals. This keeps one scanline burst from poisoning
1054  * the next linebuf fill with stale tail words.
1055  */
1056 wire scan_prefetch_critical =
1057         scan_prefetch_recover_current ||
1058         scan_prefetch_need_target ||
1059         scan_prefetch_need_next;
1060 wire scan_prefetch_best_effort =
1061         scan_prefetch_need_far && !scan_prefetch_critical;
1062 wire scan_write_pressure =
1063         flush_active ||
1064         band_flush_pending ||
1065         flush_word_pending_valid ||
1066         (sdram_wr_use[8:0] != 9'd0);
1067 wire scan_prefetch_req = !scan_fill_active && display_valid &&
1068         sdram_ready && scan_active_valid &&
1069         scan_prefetch_valid && !scan_prefetch_ready &&
1070         (scan_prefetch_bank != 2'd3) &&
1071         (!scan_prefetch_best_effort ||
1072         !scan_write_pressure) &&
1073         sdram_rd_empty;
1074 /* Let writes use active-video time once scanout has enough line headroom. */
1075 wire scan_prefetch_margin_ready =
1076         scan_target_or_active_ready &&
1077         scan_immediate_next_line_ready &&
1078         (!scan_prefetch_critical || scan_prefetch_ready);
1079 wire scan_active_write_window = vcount_visible &&
1080         (hcount < ACTIVE_WRITE_END_HCOUNT);
1081 wire scanout_write_slack = !display_valid ||
1082         (scan_read_idle &&
1083         !scan_prefetch_req &&
1084         (!VGA_BLANK_n ||
1085         (scan_prefetch_margin_ready &&
1086         scan_active_write_window)));
1087 wire scanout_read_load_req = scan_vsync_read_req ||
1088         scan_fill_load_pending ||
1089         scan_prefetch_req;
1090 wire bg_flush_wr_load_req = flush_active && flush_load_pending &&

```

```

1091         scanout_write_slack &&
1092         !scanout_read_load_req;
1093 wire      bg_flush_stream_active = flush_active && !flush_load_pending;
1094 wire      scan_visible_data_ready = display_valid && sdram_ready &&
1095         scan_active_bank_ready &&
1096         (scan_active_bank_row == scan_active_row);
1097 wire [9:0] scan_fill_words_complete =
1098     scan_fill_store_idx + (scan_rd_capture ? 10'd1 : 10'd0);
1099 wire      scan_fill_line_done =
1100     scan_rd_capture && (scan_fill_words_complete == LINE_WORDS_10);
1101 wire      scan_fill_chunk_done =
1102     scan_rd_capture && (scan_fill_words_complete[5:0] == 6'd0);
1103 /* Background flush may stage into the internal WR FIFO while scanout reads. */
1104 wire      bg_flush_wr_push      = bg_flush_stream_active && flush_word_pending_valid &&
1105     !sdram_wr_full;
1106 wire      sdram_wr_push         = bg_flush_wr_push;
1107
1108 /* While armed, ignore residual RD FIFO data from the previous burst. */
1109 wire      scan_rd_pop = scan_fill_active && !scan_fill_armed && !sdram_rd_empty &&
1110     (scan_fill_store_idx + (scan_rd_capture ? 10'd1 : 10'd0) <
1111     LINE_WORDS_10) &&
1112     !scan_fill_chunk_done;
1113 wire      sdram_rd_pop = scan_rd_pop;
1114
1115 /* Keep the background flush-to-FIFO pipeline moving until the WR FIFO high-water mark. */
1116 wire      flush_can_issue_read =
1117     bg_flush_stream_active &&
1118     !flush_generated_sky &&
1119     (flush_words_issued < flush_pixels_total) &&
1120     !flush_fetch_inflight &&
1121     (sdram_wr_use[8:0] < COPY_WR_FIFO_HIGH_WATER) &&
1122     (!flush_word_pending_valid || bg_flush_wr_push);
1123 wire      flush_can_issue_sky =
1124     bg_flush_stream_active &&
1125     flush_generated_sky &&
1126     (flush_words_issued < flush_pixels_total) &&
1127     (sdram_wr_use[8:0] < COPY_WR_FIFO_HIGH_WATER) &&
1128     (!flush_word_pending_valid || bg_flush_wr_push);
1129
1130 wire [8:0] sdram_wr_length_cfg = (bg_flush_stream_active && scanout_write_slack) ?
1131     COPY_BURST_WORDS_9 : 9'd0;
1132 /*
1133  * Keep SDRAM reads in 64-word chunks. A full scanline burst can cross the
1134  * SDRAM row/column boundary; 64-word chunks stay aligned because both the
1135  * frame line width and SDRAM column size are multiples of 64.
1136  */
1137 /*
1138  * While scan_fill_load_pending is high, the next burst address is being
1139  * staged but RD_LOAD has not reached Sdram_Control yet. Holding LENGTH at
1140  * zero for that one cycle prevents the controller from launching a burst
1141  * at the previous 64-word chunk address.
1142  */
1143 wire [8:0] sdram_rd_length_cfg = (scan_fill_armed && !scan_fill_load_pending) ?
1144     READ_BURST_WORDS_9 : 9'd0;
1145
1146 /* Ping-pong cache port muxing */
1147 /* Rasterizer read data from active cache */
1148 wire [15:0] fb_back_rd_data = draw_cache_sel ? fb_B_rd_data : fb_A_rd_data;
1149 wire [15:0] z_rd_data      = draw_cache_sel ? z_B_rd_data : z_A_rd_data;

```

```

1150 wire [15:0] fb_draw_rd_data_e = draw_cache_sel ? fb_B_e_rd_data : fb_A_e_rd_data;
1151 wire [15:0] fb_draw_rd_data_o = draw_cache_sel ? fb_B_o_rd_data : fb_A_o_rd_data;
1152 wire [15:0] z_draw_rd_data_e = draw_cache_sel ? z_B_e_rd_data : z_A_e_rd_data;
1153 wire [15:0] z_draw_rd_data_o = draw_cache_sel ? z_B_o_rd_data : z_A_o_rd_data;
1154 /* Flush read data from flush_cache_sel's cache */
1155 wire [15:0] flush_fb_rd_data = flush_cache_sel ? fb_B_rd_data : fb_A_rd_data;
1156 wire [15:0] flush_z_rd_data = flush_cache_sel ? z_B_rd_data : z_A_rd_data;
1157
1158 wire signed [15:0] desc_x_min_raw = $signed(desc_words[0][15:0]);
1159 wire signed [15:0] desc_y_min_raw = $signed(desc_words[0][31:16]);
1160 wire signed [15:0] desc_x_max_raw = $signed(desc_words[1][15:0]);
1161 wire signed [15:0] desc_y_max_raw = $signed(desc_words[1][31:16]);
1162 wire [9:0] desc_x_min = clamp_x(desc_x_min_raw);
1163 wire [9:0] desc_x_max = clamp_x(desc_x_max_raw);
1164 wire [8:0] desc_y_min = clamp_y(desc_y_min_raw);
1165 wire [8:0] desc_y_max = clamp_y(desc_y_max_raw);
1166 wire [15:0] desc_z0 = desc_words[14][15:0];
1167 wire signed [15:0] desc_dz_dx = $signed(desc_words[14][31:16]);
1168 wire signed [15:0] desc_dz_dy = $signed(desc_words[15][15:0]);
1169 wire [7:0] desc_tex_or_color = desc_words[15][23:16];
1170 wire [7:0] desc_flags = desc_words[15][31:24];
1171 wire [8:0] desc_band_base_y = band_base_row(cache_band_index);
1172 wire [4:0] desc_sky_index =
1173     desc_tex_or_color[4:0] - PAL_SKY_GRADIENT_BASE[4:0];
1174 wire desc_redundant_sky_clear =
1175     sky_gradient_clear_enabled &&
1176     (desc_flags == 8'd0) &&
1177     (desc_tex_or_color >= PAL_SKY_GRADIENT_BASE) &&
1178     (desc_tex_or_color <= PAL_SKY_GRADIENT_LAST) &&
1179     (desc_x_min == 10'd0) &&
1180     (desc_x_max == 10'd639);
1181 wire desc_band_primer =
1182     (desc_flags == 8'd0) &&
1183     (desc_tex_or_color == 8'd0) &&
1184     (desc_x_min == 10'd0) &&
1185     (desc_x_max == 10'd0) &&
1186     (desc_y_min == desc_band_base_y) &&
1187     (desc_y_max == desc_band_base_y);
1188 wire cache_sky_patch_state =
1189     (state == ST_FETCH) &&
1190     (fetch_count == fetch_target_words) &&
1191     desc_redundant_sky_clear;
1192 // Perspective-correct UV: 9 Q16.16 plane coefficients packed directly
1193 // after the base descriptor (words 16..24). One_over_w is guaranteed positive
1194 // at any pixel in front of the near plane -- software does not emit quads
1195 // that touch w <= 0.
1196 wire signed [31:0] desc_uw_0 = $signed(desc_words[16]);
1197 wire signed [31:0] desc_uw_dx = $signed(desc_words[17]);
1198 wire signed [31:0] desc_uw_dy = $signed(desc_words[18]);
1199 wire signed [31:0] desc_vw_0 = $signed(desc_words[19]);
1200 wire signed [31:0] desc_vw_dx = $signed(desc_words[20]);
1201 wire signed [31:0] desc_vw_dy = $signed(desc_words[21]);
1202 wire signed [31:0] desc_iw_0 = $signed(desc_words[22]);
1203 wire signed [31:0] desc_iw_dx = $signed(desc_words[23]);
1204 wire signed [31:0] desc_iw_dy = $signed(desc_words[24]);
1205
1206 wire [9:0] desc_x_start_even = {desc_x_min[9:1], 1'b0};
1207 wire [9:0] draw_x_start_even = {draw_x_min[9:1], 1'b0};
1208 wire [9:0] draw_x_next = draw_x_cur + 10'd1;

```

```

1209 wire signed [63:0] edge_eval0;
1210 wire signed [63:0] edge_eval1;
1211 wire signed [63:0] edge_eval2;
1212 wire signed [63:0] edge_eval3;
1213 wire signed [47:0] draw_z_start_val;
1214 wire signed [63:0] draw_uw_start_val;
1215 wire signed [63:0] draw_vw_start_val;
1216 wire signed [63:0] draw_iw_start_val;
1217
1218 voxel_raster_setup raster_setup (
1219     .draw_x_start_even (draw_x_start_even),
1220     .draw_y_min        (draw_y_min),
1221     .draw_x_min        (draw_x_min),
1222     .draw_z0           (draw_z0),
1223     .draw_dz_dx        (draw_dz_dx),
1224     .draw_uw_0         (draw_uw_0),
1225     .draw_uw_dx        (draw_uw_dx),
1226     .draw_vw_0         (draw_vw_0),
1227     .draw_vw_dx        (draw_vw_dx),
1228     .draw_iw_0         (draw_iw_0),
1229     .draw_iw_dx        (draw_iw_dx),
1230     .edge_a0           (edge_A[0]),
1231     .edge_a1           (edge_A[1]),
1232     .edge_a2           (edge_A[2]),
1233     .edge_a3           (edge_A[3]),
1234     .edge_b0           (edge_B[0]),
1235     .edge_b1           (edge_B[1]),
1236     .edge_b2           (edge_B[2]),
1237     .edge_b3           (edge_B[3]),
1238     .edge_c0           (edge_C[0]),
1239     .edge_c1           (edge_C[1]),
1240     .edge_c2           (edge_C[2]),
1241     .edge_c3           (edge_C[3]),
1242     .edge_eval0        (edge_eval0),
1243     .edge_eval1        (edge_eval1),
1244     .edge_eval2        (edge_eval2),
1245     .edge_eval3        (edge_eval3),
1246     .z_start_val       (draw_z_start_val),
1247     .uw_start_val      (draw_uw_start_val),
1248     .vw_start_val      (draw_vw_start_val),
1249     .iw_start_val      (draw_iw_start_val)
1250 );
1251
1252 wire draw_inside = (edge_cur_val[0] >= 0) && (edge_cur_val[1] >= 0) &&
1253                   (edge_cur_val[2] >= 0) && (edge_cur_val[3] >= 0);
1254 wire signed [63:0] edge_cur_val_o0;
1255 wire signed [63:0] edge_cur_val_o1;
1256 wire signed [63:0] edge_cur_val_o2;
1257 wire signed [63:0] edge_cur_val_o3;
1258 wire draw_inside_o_edge = (edge_cur_val_o0 >= 0) &&
1259                           (edge_cur_val_o1 >= 0) &&
1260                           (edge_cur_val_o2 >= 0) &&
1261                           (edge_cur_val_o3 >= 0);
1262 wire draw_lane0_in_bounds = (draw_x_cur >= draw_x_min) &&
1263                             (draw_x_cur <= draw_x_max);
1264 wire draw_lane1_in_bounds = (draw_x_next >= draw_x_min) &&
1265                             (draw_x_next <= draw_x_max);
1266 wire draw_inside_lane0 = draw_inside && draw_lane0_in_bounds;
1267 wire draw_inside_lane1 = draw_inside_o_edge && draw_lane1_in_bounds;

```

```

1268 wire draw_pair_edge_inside = draw_inside || draw_inside_o_edge;
1269 wire draw_pair_exited = draw_row_inside && !draw_inside && !draw_inside_o_edge;
1270 wire draw_pair_last = (draw_x_next >= draw_x_max);
1271 wire [15:0] draw_z_value = clamp_z(z_cur_val);
1272 wire signed [63:0] edge_next_pair0;
1273 wire signed [63:0] edge_next_pair1;
1274 wire signed [63:0] edge_next_pair2;
1275 wire signed [63:0] edge_next_pair3;
1276 wire signed [63:0] edge_next_row0;
1277 wire signed [63:0] edge_next_row1;
1278 wire signed [63:0] edge_next_row2;
1279 wire signed [63:0] edge_next_row3;
1280 wire signed [47:0] draw_z_lane1_val;
1281 wire signed [47:0] z_next_pair;
1282 wire signed [47:0] z_next_row;
1283 wire signed [63:0] draw_uw_lane1_val;
1284 wire signed [63:0] draw_vw_lane1_val;
1285 wire signed [63:0] draw_iw_lane1_val;
1286 wire signed [63:0] uw_next_pair;
1287 wire signed [63:0] uw_next_row;
1288 wire signed [63:0] vw_next_pair;
1289 wire signed [63:0] vw_next_row;
1290 wire signed [63:0] iw_next_pair;
1291 wire signed [63:0] iw_next_row;
1292
1293 voxel_draw_step draw_step (
1294     .edge_cur0      (edge_cur_val[0]),
1295     .edge_cur1      (edge_cur_val[1]),
1296     .edge_cur2      (edge_cur_val[2]),
1297     .edge_cur3      (edge_cur_val[3]),
1298     .edge_row0      (edge_row_val[0]),
1299     .edge_row1      (edge_row_val[1]),
1300     .edge_row2      (edge_row_val[2]),
1301     .edge_row3      (edge_row_val[3]),
1302     .edge_a0        (edge_A[0]),
1303     .edge_a1        (edge_A[1]),
1304     .edge_a2        (edge_A[2]),
1305     .edge_a3        (edge_A[3]),
1306     .edge_b0        (edge_B[0]),
1307     .edge_b1        (edge_B[1]),
1308     .edge_b2        (edge_B[2]),
1309     .edge_b3        (edge_B[3]),
1310     .z_cur          (z_cur_val),
1311     .z_row          (z_row_val),
1312     .dz_dx          (draw_dz_dx),
1313     .dz_dy          (draw_dz_dy),
1314     .uw_cur         (uw_cur_val),
1315     .uw_row         (uw_row_val),
1316     .uw_dx          (draw_uw_dx),
1317     .uw_dy          (draw_uw_dy),
1318     .vw_cur         (vw_cur_val),
1319     .vw_row         (vw_row_val),
1320     .vw_dx          (draw_vw_dx),
1321     .vw_dy          (draw_vw_dy),
1322     .iw_cur         (iw_cur_val),
1323     .iw_row         (iw_row_val),
1324     .iw_dx          (draw_iw_dx),
1325     .iw_dy          (draw_iw_dy),
1326     .edge_lane1_0   (edge_cur_val_o0),

```

```

1327     .edge_lane1_1      (edge_cur_val_o1),
1328     .edge_lane1_2      (edge_cur_val_o2),
1329     .edge_lane1_3      (edge_cur_val_o3),
1330     .edge_next_pair0   (edge_next_pair0),
1331     .edge_next_pair1   (edge_next_pair1),
1332     .edge_next_pair2   (edge_next_pair2),
1333     .edge_next_pair3   (edge_next_pair3),
1334     .edge_next_row0    (edge_next_row0),
1335     .edge_next_row1    (edge_next_row1),
1336     .edge_next_row2    (edge_next_row2),
1337     .edge_next_row3    (edge_next_row3),
1338     .z_lane1           (draw_z_lane1_val),
1339     .z_next_pair       (z_next_pair),
1340     .z_next_row        (z_next_row),
1341     .uw_lane1         (draw_uw_lane1_val),
1342     .uw_next_pair     (uw_next_pair),
1343     .uw_next_row      (uw_next_row),
1344     .vw_lane1         (draw_vw_lane1_val),
1345     .vw_next_pair     (vw_next_pair),
1346     .vw_next_row      (vw_next_row),
1347     .iw_lane1         (draw_iw_lane1_val),
1348     .iw_next_pair     (iw_next_pair),
1349     .iw_next_row      (iw_next_row)
1350 );
1351
1352 wire [15:0] draw_z_value_o = clamp_z(draw_z_lane1_val);
1353 wire [2:0]  draw_band_index = y_to_band(draw_y_cur);
1354 wire       draw_cache_hit = cache_valid && (cache_band_index == draw_band_index);
1355 wire [15:0] draw_x_offset = {6'd0, draw_x_cur} - {6'd0, draw_x_start_even};
1356 wire [15:0] draw_cache_addr = draw_row_base + draw_x_offset;
1357 wire [15:0] draw_cache_addr_o = draw_cache_addr + 16'd1;
1358 wire signed [31:0] draw_uw_q = clamp_s32(uw_cur_val);
1359 wire signed [31:0] draw_vw_q = clamp_s32(vw_cur_val);
1360 wire [31:0] draw_iw_q = clamp_pos_u32(iw_cur_val);
1361 wire signed [31:0] draw_uw_q_o = clamp_s32(draw_uw_lane1_val);
1362 wire signed [31:0] draw_vw_q_o = clamp_s32(draw_vw_lane1_val);
1363 wire [31:0] draw_iw_q_o = clamp_pos_u32(draw_iw_lane1_val);
1364 wire [5:0] pipe0_iw_msb;
1365 wire [31:0] pipe0_iw_norm_q;
1366 wire [5:0] pipe0_iw_msb_o;
1367 wire [31:0] pipe0_iw_norm_q_o;
1368
1369 voxel_iw_normalize iw_norm_lane0 (
1370     .iw_q      (pipe0_iw_q),
1371     .iw_msb    (pipe0_iw_msb),
1372     .iw_norm_q (pipe0_iw_norm_q)
1373 );
1374
1375 voxel_iw_normalize iw_norm_lane1 (
1376     .iw_q      (pipe0_iw_q_o),
1377     .iw_msb    (pipe0_iw_msb_o),
1378     .iw_norm_q (pipe0_iw_norm_q_o)
1379 );
1380
1381 wire [15:0] recip0_iw_phase = recip0_iw_norm_q[15:0];
1382 wire [10:0] recip0_iw_lut_idx = {1'b0, recip0_iw_phase[15:6]};
1383 wire [5:0] recip0_iw_lut_frac = recip0_iw_phase[5:0];
1384 wire [15:0] recip0_iw_phase_o = recip0_iw_norm_q_o[15:0];
1385 wire [10:0] recip0_iw_lut_idx_o = {1'b0, recip0_iw_phase_o[15:6]};

```

```

1386 wire [5:0] recip0_iw_lut_frac_o = recip0_iw_phase_o[5:0];
1387 wire [31:0] recip1_w_norm_q;
1388 wire [31:0] recip1_w_norm_q_o;
1389 wire [31:0] recip2_w_q;
1390 wire [31:0] recip2_w_q_o;
1391
1392 voxel_recip_interpolate recip_interp_lane0 (
1393     .w_norm_lo    (recip1_w_norm_lo),
1394     .w_norm_hi    (recip1_w_norm_hi),
1395     .iw_lut_frac  (recip1_iw_lut_frac),
1396     .w_norm_q     (recip1_w_norm_q)
1397 );
1398
1399 voxel_recip_interpolate recip_interp_lane1 (
1400     .w_norm_lo    (recip1_w_norm_lo_o),
1401     .w_norm_hi    (recip1_w_norm_hi_o),
1402     .iw_lut_frac  (recip1_iw_lut_frac_o),
1403     .w_norm_q     (recip1_w_norm_q_o)
1404 );
1405
1406 voxel_w_denormalize w_denorm_lane0 (
1407     .iw_zero      (recip2_iw_zero),
1408     .iw_msb       (recip2_iw_msb),
1409     .w_norm_q     (recip2_w_norm_q),
1410     .w_q          (recip2_w_q)
1411 );
1412
1413 voxel_w_denormalize w_denorm_lane1 (
1414     .iw_zero      (recip2_iw_zero_o),
1415     .iw_msb       (recip2_iw_msb_o),
1416     .w_norm_q     (recip2_w_norm_q_o),
1417     .w_q          (recip2_w_q_o)
1418 );
1419 wire signed [63:0] pipe1_u_prod = $signed(pipe1_uw_q) * $signed(pipe1_w_q);
1420 wire signed [63:0] pipe1_v_prod = $signed(pipe1_vw_q) * $signed(pipe1_w_q);
1421 wire signed [63:0] pipe1_u_prod_o = $signed(pipe1_uw_q_o) * $signed(pipe1_w_q_o);
1422 wire signed [63:0] pipe1_v_prod_o = $signed(pipe1_vw_q_o) * $signed(pipe1_w_q_o);
1423 wire tex0_repeat_uv = tex0_tex_or_color[7];
1424 wire [3:0] tex0_tex_u = texture_coord(tex0_u_prod, tex0_repeat_uv);
1425 wire [3:0] tex0_tex_v = texture_coord(tex0_v_prod, tex0_repeat_uv);
1426 wire [14:0] tex0_tex_addr = tex0_textured ?
1427     {tex0_tex_or_color[6:0], tex0_tex_v, tex0_tex_u} :
1428     15'd0;
1429 wire tex0_repeat_uv_o = tex0_tex_or_color_o[7];
1430 wire [3:0] tex0_tex_u_o = texture_coord(tex0_u_prod_o, tex0_repeat_uv_o);
1431 wire [3:0] tex0_tex_v_o = texture_coord(tex0_v_prod_o, tex0_repeat_uv_o);
1432 wire [14:0] tex0_tex_addr_o = tex0_textured_o ?
1433     {tex0_tex_or_color_o[6:0], tex0_tex_v_o, tex0_tex_u_o} :
1434     15'd0;
1435 wire [7:0] draw_pipe_raw_color = draw_pipe_textured ? tex_rd_data : draw_pipe_tex_or_color;
1436 wire [7:0] draw_pipe_color = apply_light_bank(draw_pipe_raw_color, draw_pipe_light_bank);
1437 wire [7:0] draw_pipe_raw_color_o =
1438     draw_pipe_textured_o ? tex_rd_data_o : draw_pipe_tex_or_color_o;
1439 wire [7:0] draw_pipe_color_o =
1440     apply_light_bank(draw_pipe_raw_color_o, draw_pipe_light_bank_o);
1441 wire
1442     !draw_pipe_textured &&
1443     !draw_pipe_ztest &&
1444     !draw_pipe_fog &&

```

```

1445     (draw_pipe_alpha == 2'd0) &&
1446     (draw_pipe_light_bank == 2'd0) &&
1447     (draw_pipe_raw_color >= PAL_SKY_GRADIENT_BASE) &&
1448     (draw_pipe_raw_color <= PAL_SKY_GRADIENT_LAST);
1449 wire     draw_pipe_generated_sky_o =
1450 !draw_pipe_textured_o &&
1451 !draw_pipe_ztest_o &&
1452 !draw_pipe_fog_o &&
1453 (draw_pipe_alpha_o == 2'd0) &&
1454 (draw_pipe_light_bank_o == 2'd0) &&
1455 (draw_pipe_raw_color_o >= PAL_SKY_GRADIENT_BASE) &&
1456 (draw_pipe_raw_color_o <= PAL_SKY_GRADIENT_LAST);
1457 wire [4:0] draw_pipe_sky_index =
1458 draw_pipe_raw_color[4:0] - PAL_SKY_GRADIENT_BASE[4:0];
1459 wire [4:0] draw_pipe_sky_index_o =
1460 draw_pipe_raw_color_o[4:0] - PAL_SKY_GRADIENT_BASE[4:0];
1461 wire [7:0] palette_src_addr = (state == ST_CLEAR) ? 8'd0 : draw_pipe_color;
1462 wire [7:0] palette_src_addr_o = (state == ST_CLEAR) ? 8'd0 : draw_pipe_color_o;
1463 /* Palette reads for the draw pipeline are sampled into plr_* one
1464 * cycle before fog0 (see the plr_* register block). The
1465 * rgb888_to_rgb565 conversion is cheap bit slicing, so we leave it
1466 * as a combinational derivation on the registered palette output
1467 * and let fog0_src_rgb565 / fog0_fog_rgb565 pick it up on the next
1468 * cycle. */
1469 wire [15:0] plr_src_rgb565 = rgb888_to_rgb565(plr_src_rgb);
1470 wire [15:0] plr_fog_rgb565 = rgb888_to_rgb565(plr_fog_rgb);
1471 wire [15:0] plr_src_rgb565_o = rgb888_to_rgb565(plr_src_rgb_o);
1472 wire [15:0] plr_fog_rgb565_o = rgb888_to_rgb565(plr_fog_rgb_o);
1473
1474 /* Separate combinational reads used by clear/cache-init paths, which write
1475 * background colors straight to the local band cache and do not go through
1476 * the draw pipeline. */
1477 wire [15:0] clear_rgb565 = rgb888_to_rgb565(palette[8'd0]);
1478 wire     sky_gradient_clear_enabled =
1479 extmem_ctrl[EXTMEM_SKY_GRADIENT_CLEAR_BIT];
1480 wire [15:0] flush_clear_rgb565 =
1481 sky_gradient_clear_enabled ?
1482 rgb888_to_rgb565(sky_palette[flush_sky_palette]) :
1483 clear_rgb565;
1484 wire [15:0] draw_clear_rgb565 =
1485 sky_gradient_clear_enabled ?
1486 rgb888_to_rgb565(sky_palette[sky_palette_for_y( recip0_y)]) :
1487 clear_rgb565;
1488 wire draw_pipe_transparent = draw_pipe_textured &&
1489 draw_pipe_alpha_key &&
1490 (draw_pipe_raw_color == 8'd0);
1491 wire draw_pipe_transparent_o = draw_pipe_textured_o &&
1492 draw_pipe_alpha_key_o &&
1493 (draw_pipe_raw_color_o == 8'd0);
1494 /*
1495 * Radial distance estimate: starting from the per-pixel w (linear depth
1496 * along the camera forward axis) we scale by sqrt(1 + r^2/f^2) where r is
1497 * the pixel offset from screen center and f is the projection depth in
1498 * pixels. The square-root is approximated with 1 + 3/8 * r^2/f^2, which
1499 * is within a couple of percent for our field of view. The final distance
1500 * is produced in Q8.8 so it can be compared directly against the
1501 * fog_start_dist / fog_end_dist registers.
1502 */
1503 wire signed [11:0] draw_pipe_dx_center =

```

```

1504     $signed({1'b0, draw_pipe_x}) - 12'sd320;
1505 wire signed [10:0] draw_pipe_dy_center =
1506     11'sd240 - $signed({1'b0, draw_pipe_y});
1507 wire [23:0] draw_pipe_dx_sq = draw_pipe_dx_center * draw_pipe_dx_center;
1508 wire [23:0] draw_pipe_dy_sq = draw_pipe_dy_center * draw_pipe_dy_center;
1509 wire [24:0] draw_pipe_radius_sq = draw_pipe_dx_sq + draw_pipe_dy_sq;
1510 wire [40:0] draw_pipe_r2_prod = draw_pipe_radius_sq * fog_inv_proj_sq;
1511 wire [31:0] draw_pipe_r2_q16 = draw_pipe_r2_prod[31:0];
1512 wire [33:0] draw_pipe_ray_scale_q16 =
1513     34'd65536 + (({2'b00, draw_pipe_r2_q16} * 3'd3) >> 3);
1514 wire signed [11:0] draw_pipe_dx_center_o =
1515     $signed({1'b0, draw_pipe_x_o}) - 12'sd320;
1516 wire signed [10:0] draw_pipe_dy_center_o =
1517     11'sd240 - $signed({1'b0, draw_pipe_y_o});
1518 wire [23:0] draw_pipe_dx_sq_o = draw_pipe_dx_center_o * draw_pipe_dx_center_o;
1519 wire [23:0] draw_pipe_dy_sq_o = draw_pipe_dy_center_o * draw_pipe_dy_center_o;
1520 wire [24:0] draw_pipe_radius_sq_o = draw_pipe_dx_sq_o + draw_pipe_dy_sq_o;
1521 wire [40:0] draw_pipe_r2_prod_o = draw_pipe_radius_sq_o * fog_inv_proj_sq;
1522 wire [31:0] draw_pipe_r2_q16_o = draw_pipe_r2_prod_o[31:0];
1523 wire [33:0] draw_pipe_ray_scale_q16_o =
1524     34'd65536 + (({2'b00, draw_pipe_r2_q16_o} * 3'd3) >> 3);
1525 wire [65:0] fog0_radial_prod = fog0_w_q * fog0_ray_scale_q16;
1526 wire [31:0] fog0_radial_q16 = fog0_radial_prod[47:16];
1527 wire [15:0] fog0_radial_q8_8 = fog0_radial_q16[23:8];
1528 wire [65:0] fog0_radial_prod_o = fog0_w_q_o * fog0_ray_scale_q16_o;
1529 wire [31:0] fog0_radial_q16_o = fog0_radial_prod_o[47:16];
1530 wire [15:0] fog0_radial_q8_8_o = fog0_radial_q16_o[23:8];
1531
1532 wire [15:0] fog1_out_rgb565;
1533 wire [15:0] fog1_out_rgb565_o;
1534
1535 voxel_fog_blend fog_blend_lane0 (
1536     .fog_enable      (fog_enable),
1537     .pixel_fog      (fog1_fog),
1538     .radial_q8_8    (fog1_radial_q8_8),
1539     .fog_start_dist (fog_start_dist),
1540     .fog_end_dist   (fog_end_dist),
1541     .src_rgb565     (fog1_src_rgb565),
1542     .dst_rgb565     (fog1_dst_rgb565),
1543     .fog_rgb565     (fog1_fog_rgb565),
1544     .alpha          (fog1_alpha),
1545     .out_rgb565     (fog1_out_rgb565)
1546 );
1547
1548 voxel_fog_blend fog_blend_lane1 (
1549     .fog_enable      (fog_enable),
1550     .pixel_fog      (fog1_fog_o),
1551     .radial_q8_8    (fog1_radial_q8_8_o),
1552     .fog_start_dist (fog_start_dist),
1553     .fog_end_dist   (fog_end_dist),
1554     .src_rgb565     (fog1_src_rgb565_o),
1555     .dst_rgb565     (fog1_dst_rgb565_o),
1556     .fog_rgb565     (fog1_fog_rgb565_o),
1557     .alpha          (fog1_alpha_o),
1558     .out_rgb565     (fog1_out_rgb565_o)
1559 );
1560 wire draw_commit_pass = draw_pipe_inside &&
1561     !draw_pipe_transparent &&
1562     (!draw_pipe_ztest || (draw_pipe_z < draw_pipe_z_ref));

```

```

1563 wire draw_commit_pass_o = draw_pipe_inside_o &&
1564                             !draw_pipe_transparent_o &&
1565                             (!draw_pipe_ztest_o || (draw_pipe_z_o < draw_pipe_z_ref_o));
1566 wire [31:0] status_word = {
1567     12'h0,
1568     {5'h0, fifo_count},
1569     vsy_latch,
1570     fifo_empty,
1571     fifo_full,
1572     engine_busy
1573 };
1574
1575 wire [31:0] control_word = {
1576     28'h0,
1577     (clear_pending || (state == ST_CLEAR)),
1578     ctrl_ien,
1579     ctrl_flp_pending,
1580     ctrl_en
1581 };
1582
1583 wire [23:0] pixel_rgb = {
1584     expand5_to_8(scan_rgb565_r[15:11]),
1585     expand6_to_8(scan_rgb565_r[10:5]),
1586     expand5_to_8(scan_rgb565_r[4:0])
1587 };
1588
1589 `include "voxel_raster_helpers.svh"
1590 `include "voxel_color_helpers.svh"
1591
1592 voxel_vga_counters counters (
1593     .clk50      (clk),
1594     .reset      (reset),
1595     .hcount     (hcount),
1596     .vcount     (vcount),
1597     .VGA_CLK    (VGA_CLK),
1598     .VGA_HS     (VGA_HS),
1599     .VGA_VS     (VGA_VS),
1600     .VGA_BLANK_n (VGA_BLANK_n),
1601     .VGA_SYNC_n (VGA_SYNC_n)
1602 );
1603
1604 Sdram_Control sdram_ctrl (
1605     .REF_CLK      (clk),
1606     .RESET_N      (sdram_ctrl_reset_n),
1607     .CLK          (sdram_ctrl_clk),
1608     .WR_DATA      (flush_word_pending),
1609     .WR           (sdram_wr_push),
1610     .WR_ADDR      (flush_sdram_wr_addr),
1611     .WR_MAX_ADDR  (flush_sdram_wr_max_addr),
1612     .WR_LENGTH    (sdram_wr_length_cfg),
1613     .WR_LOAD      (bg_flush_wr_load_req),
1614     .WR_CLK       (clk),
1615     .WR_FULL      (sdram_wr_full),
1616     .WR_USE       (sdram_wr_use),
1617     .RD_DATA      (sdram_rd_data),
1618     .RD           (sdram_rd_pop),
1619     .RD_ADDR      (sdram_rd_addr_cfg),
1620     .RD_MAX_ADDR  (sdram_rd_max_addr_cfg),
1621     .RD_LENGTH    (sdram_rd_length_cfg),

```

```

1622     .RD_LOAD      (sdram_rd_load_out),
1623     .RD_CLK      (clk),
1624     .RD_EMPTY    (sdram_rd_empty),
1625     .RD_USE      (sdram_rd_use),
1626     .SA          (DRAM_ADDR),
1627     .BA          (DRAM_BA),
1628     .CS_N        (dram_cs_n_bus),
1629     .CKE         (DRAM_CKE),
1630     .RAS_N       (DRAM_RAS_N),
1631     .CAS_N       (DRAM_CAS_N),
1632     .WE_N        (DRAM_WE_N),
1633     .DQ          (DRAM_DQ),
1634     .DQM         ({DRAM_UDQM, DRAM_LDQM}),
1635     .SDR_CLK     (DRAM_CLK)
1636 );
1637
1638 assign DRAM_CS_N = dram_cs_n_bus[0];
1639
1640 /* Ping-pong band caches A and B. Even and odd linear addresses land in
1641  * separate banks so the two raster lanes can read/write independently. */
1642 voxel_banked_sdp_ram #(
1643     .DATA_W(16),
1644     .ADDR_W(16),
1645     .DEPTH(BAND_PIXELS)
1646 ) fb_back_ram_A (
1647     .clk      (clk),
1648     .rd_addr_e (fb_A_e_rd_addr),
1649     .rd_data_e (fb_A_e_rd_data),
1650     .rd_addr_o (fb_A_o_rd_addr),
1651     .rd_data_o (fb_A_o_rd_data),
1652     .wr_addr_e (fb_A_e_wr_addr),
1653     .wr_data_e (fb_A_e_wr_data),
1654     .wr_en_e   (fb_A_e_wr_en),
1655     .wr_addr_o (fb_A_o_wr_addr),
1656     .wr_data_o (fb_A_o_wr_data),
1657     .wr_en_o   (fb_A_o_wr_en)
1658 );
1659
1660 voxel_banked_sdp_ram #(
1661     .DATA_W(16),
1662     .ADDR_W(16),
1663     .DEPTH(BAND_PIXELS)
1664 ) fb_back_ram_B (
1665     .clk      (clk),
1666     .rd_addr_e (fb_B_e_rd_addr),
1667     .rd_data_e (fb_B_e_rd_data),
1668     .rd_addr_o (fb_B_o_rd_addr),
1669     .rd_data_o (fb_B_o_rd_data),
1670     .wr_addr_e (fb_B_e_wr_addr),
1671     .wr_data_e (fb_B_e_wr_data),
1672     .wr_en_e   (fb_B_e_wr_en),
1673     .wr_addr_o (fb_B_o_wr_addr),
1674     .wr_data_o (fb_B_o_wr_data),
1675     .wr_en_o   (fb_B_o_wr_en)
1676 );
1677
1678 voxel_banked_sdp_ram #(
1679     .DATA_W(16),
1680     .ADDR_W(16),

```

```

1681     .DEPTH(BAND_PIXELS)
1682 ) z_ram_A (
1683     .clk      (clk),
1684     .rd_addr_e (z_A_e_rd_addr),
1685     .rd_data_e (z_A_e_rd_data),
1686     .rd_addr_o (z_A_o_rd_addr),
1687     .rd_data_o (z_A_o_rd_data),
1688     .wr_addr_e (z_A_e_wr_addr),
1689     .wr_data_e (z_A_e_wr_data),
1690     .wr_en_e   (z_A_e_wr_en),
1691     .wr_addr_o (z_A_o_wr_addr),
1692     .wr_data_o (z_A_o_wr_data),
1693     .wr_en_o   (z_A_o_wr_en)
1694 );
1695
1696 voxel_banked_sdp_ram #(
1697     .DATA_W(16),
1698     .ADDR_W(16),
1699     .DEPTH(BAND_PIXELS)
1700 ) z_ram_B (
1701     .clk      (clk),
1702     .rd_addr_e (z_B_e_rd_addr),
1703     .rd_data_e (z_B_e_rd_data),
1704     .rd_addr_o (z_B_o_rd_addr),
1705     .rd_data_o (z_B_o_rd_data),
1706     .wr_addr_e (z_B_e_wr_addr),
1707     .wr_data_e (z_B_e_wr_data),
1708     .wr_en_e   (z_B_e_wr_en),
1709     .wr_addr_o (z_B_o_wr_addr),
1710     .wr_data_o (z_B_o_wr_data),
1711     .wr_en_o   (z_B_o_wr_en)
1712 );
1713
1714 /* Rebuild the single-cache read view from the bank selected last cycle. */
1715 assign fb_A_rd_data = fb_A_rd_sel_q ? fb_A_o_rd_data : fb_A_e_rd_data;
1716 assign fb_B_rd_data = fb_B_rd_sel_q ? fb_B_o_rd_data : fb_B_e_rd_data;
1717 assign z_A_rd_data  = z_A_rd_sel_q  ? z_A_o_rd_data  : z_A_e_rd_data;
1718 assign z_B_rd_data  = z_B_rd_sel_q  ? z_B_o_rd_data  : z_B_e_rd_data;
1719
1720 /*
1721  * Texture-atlas ROM. Two explicit 1-cycle read ports keep texels aligned
1722  * with draw_pipe metadata; tex_or_color[6:0] selects the tile and bit 7
1723  * controls UV wrapping.
1724  */
1725 voxel_texture_rom #(
1726     .DATA_W(8),
1727     .ADDR_W(15),
1728     .DEPTH(TEXTURE_BYTES),
1729     .INIT_FILE("voxel_gpu/assets/textures.mif")
1730 ) texture_rom (
1731     .clk      (clk),
1732     .rd_addr  (pipe2_tex_addr),
1733     .rd_data  (tex_rd_data),
1734     .rd_addr_b (pipe2_tex_addr_o),
1735     .rd_data_b (tex_rd_data_o)
1736 );
1737
1738 integer i;
1739 integer ei;

```

```

1740  initial begin
1741      ctrl_en          = 1'b0;
1742      ctrl_ien         = 1'b0;
1743      ctrl_flp_pending = 1'b0;
1744      clear_pending    = 1'b0;
1745      frame_count      = 32'h0;
1746      pal_addr         = 8'h0;
1747      sky_pal_addr     = 5'd0;
1748      fog_enable       = 1'b0;
1749      fog_color        = 8'd0;
1750      fog_start_dist   = 16'd0;
1751      fog_end_dist     = 16'd0;
1752      fog_inv_proj_sq  = 16'd0;
1753      extmem_ctrl      = DEFAULT_EXTMEM_CTRL;
1754      extmem_front_base = DEFAULT_EXTMEM_FRONT_BASE;
1755      extmem_back_base  = DEFAULT_EXTMEM_BACK_BASE;
1756      extmem_stride_bytes = DEFAULT_EXTMEM_STRIDE;
1757      extmem_tile_cfg   = 32'd0;
1758      extmem_dma_status = 32'd0;
1759      vsy_latch        = 1'b0;
1760      display_sel      = 1'b0;
1761      display_valid    = 1'b0;
1762      copy_target_sel  = 1'b1;
1763      copy_complete_pending = 1'b0;
1764      state            = ST_IDLE;
1765      fifo_wr_ptr      = 10'd0;
1766      fifo_rd_ptr      = 10'd0;
1767      fifo_count       = 11'd0;
1768      fetch_count      = 6'd0;
1769      prefetch_count   = 6'd0;
1770      prefetch_target_words = BASE_QUAD_WORDS_6;
1771      prefetch_active  = 1'b0;
1772      prefetch_valid   = 1'b0;
1773      draw_flush_count = 4'd0;
1774      clear_addr       = 16'd0;
1775      draw_row_base    = 16'd0;
1776      draw_x_min       = 10'd0;
1777      draw_x_max       = 10'd0;
1778      draw_x_cur       = 10'd0;
1779      draw_y_min       = 9'd0;
1780      draw_y_max       = 9'd0;
1781      draw_y_cur       = 9'd0;
1782      draw_tex_or_color = 8'd0;
1783      draw_flags       = 8'd0;
1784      draw_z0          = 16'd0;
1785      draw_dz_dx       = 16'sd0;
1786      draw_dz_dy       = 16'sd0;
1787      draw_uw_0        = 32'sd0;
1788      draw_uw_dx       = 32'sd0;
1789      draw_uw_dy       = 32'sd0;
1790      draw_vw_0        = 32'sd0;
1791      draw_vw_dx       = 32'sd0;
1792      draw_vw_dy       = 32'sd0;
1793      draw_iw_0        = 32'sd0;
1794      draw_iw_dx       = 32'sd0;
1795      draw_iw_dy       = 32'sd0;
1796      z_row_val        = 48'sd0;
1797      z_cur_val        = 48'sd0;
1798      uw_row_val       = 64'sd0;

```

```

1799     uw_cur_val      = 64'sd0;
1800     vw_row_val     = 64'sd0;
1801     vw_cur_val     = 64'sd0;
1802     iw_row_val     = 64'sd0;
1803     iw_cur_val     = 64'sd0;
1804     pipe0_valid    = 1'b0;
1805     pipe0_inside   = 1'b0;
1806     pipe0_ztest    = 1'b0;
1807     pipe0_textured = 1'b0;
1808     pipe0_alpha_key = 1'b0;
1809     pipe0_alpha    = 2'd0;
1810     pipe0_fog      = 1'b0;
1811     pipe0_light_bank = 2'd0;
1812     pipe0_tex_or_color = 8'd0;
1813     pipe0_addr     = 16'd0;
1814     pipe0_z        = 16'd0;
1815     pipe0_x        = 10'd0;
1816     pipe0_y        = 9'd0;
1817     pipe0_uw_q     = 32'sd0;
1818     pipe0_vw_q     = 32'sd0;
1819     pipe0_iw_q     = 32'd0;
1820     recip0_valid   = 1'b0;
1821     recip0_inside  = 1'b0;
1822     recip0_ztest   = 1'b0;
1823     recip0_textured = 1'b0;
1824     recip0_alpha_key = 1'b0;
1825     recip0_alpha   = 2'd0;
1826     recip0_fog     = 1'b0;
1827     recip0_light_bank = 2'd0;
1828     recip0_tex_or_color = 8'd0;
1829     recip0_addr    = 16'd0;
1830     recip0_z       = 16'd0;
1831     recip0_x       = 10'd0;
1832     recip0_y       = 9'd0;
1833     recip0_uw_q    = 32'sd0;
1834     recip0_vw_q    = 32'sd0;
1835     recip0_iw_zero = 1'b1;
1836     recip0_iw_msb  = 6'd0;
1837     recip0_iw_norm_q = 32'd0;
1838     recip1_valid   = 1'b0;
1839     recip1_inside  = 1'b0;
1840     recip1_ztest   = 1'b0;
1841     recip1_textured = 1'b0;
1842     recip1_alpha_key = 1'b0;
1843     recip1_alpha   = 2'd0;
1844     recip1_fog     = 1'b0;
1845     recip1_light_bank = 2'd0;
1846     recip1_tex_or_color = 8'd0;
1847     recip1_addr    = 16'd0;
1848     recip1_z       = 16'd0;
1849     recip1_z_ref   = 16'd0;
1850     recip1_dst_rgb565 = 16'h0000;
1851     recip1_x       = 10'd0;
1852     recip1_y       = 9'd0;
1853     recip1_uw_q    = 32'sd0;
1854     recip1_vw_q    = 32'sd0;
1855     recip1_iw_zero = 1'b1;
1856     recip1_iw_msb  = 6'd0;
1857     recip1_iw_lut_frac = 6'd0;

```

```

1858 recip1_w_norm_lo = 32'd0;
1859 recip1_w_norm_hi = 32'd0;
1860 recip2_valid     = 1'b0;
1861 recip2_inside   = 1'b0;
1862 recip2_ztest    = 1'b0;
1863 recip2_textured = 1'b0;
1864 recip2_alpha_key = 1'b0;
1865 recip2_alpha    = 2'd0;
1866 recip2_fog      = 1'b0;
1867 recip2_light_bank = 2'd0;
1868 recip2_tex_or_color = 8'd0;
1869 recip2_addr     = 16'd0;
1870 recip2_z        = 16'd0;
1871 recip2_z_ref    = 16'd0;
1872 recip2_dst_rgb565 = 16'h0000;
1873 recip2_x        = 10'd0;
1874 recip2_y        = 9'd0;
1875 recip2_uw_q     = 32'sd0;
1876 recip2_vw_q     = 32'sd0;
1877 recip2_iw_zero  = 1'b1;
1878 recip2_iw_msb   = 6'd0;
1879 recip2_w_norm_q = 32'd0;
1880 pipe1_valid     = 1'b0;
1881 pipe1_inside    = 1'b0;
1882 pipe1_ztest     = 1'b0;
1883 pipe1_textured  = 1'b0;
1884 pipe1_alpha_key = 1'b0;
1885 pipe1_alpha     = 2'd0;
1886 pipe1_fog       = 1'b0;
1887 pipe1_light_bank = 2'd0;
1888 pipe1_tex_or_color = 8'd0;
1889 pipe1_addr      = 16'd0;
1890 pipe1_z         = 16'd0;
1891 pipe1_z_ref     = 16'd0;
1892 pipe1_dst_rgb565 = 16'h0000;
1893 pipe1_x         = 10'd0;
1894 pipe1_y         = 9'd0;
1895 pipe1_uw_q      = 32'sd0;
1896 pipe1_vw_q      = 32'sd0;
1897 pipe1_w_q       = 32'd0;
1898 tex0_valid      = 1'b0;
1899 tex0_inside     = 1'b0;
1900 tex0_ztest      = 1'b0;
1901 tex0_textured   = 1'b0;
1902 tex0_alpha_key  = 1'b0;
1903 tex0_alpha      = 2'd0;
1904 tex0_fog        = 1'b0;
1905 tex0_light_bank = 2'd0;
1906 tex0_tex_or_color = 8'd0;
1907 tex0_addr       = 16'd0;
1908 tex0_z          = 16'd0;
1909 tex0_z_ref      = 16'd0;
1910 tex0_dst_rgb565 = 16'h0000;
1911 tex0_x          = 10'd0;
1912 tex0_y          = 9'd0;
1913 tex0_w_q        = 32'd0;
1914 tex0_u_prod     = 64'sd0;
1915 tex0_v_prod     = 64'sd0;
1916 pipe2_valid     = 1'b0;

```

```

1917     pipe2_inside      = 1'b0;
1918     pipe2_ztest      = 1'b0;
1919     pipe2_textured   = 1'b0;
1920     pipe2_alpha_key  = 1'b0;
1921     pipe2_alpha      = 2'd0;
1922     pipe2_fog        = 1'b0;
1923     pipe2_light_bank = 2'd0;
1924     pipe2_tex_or_color = 8'd0;
1925     pipe2_addr       = 16'd0;
1926     pipe2_z          = 16'd0;
1927     pipe2_z_ref      = 16'd0;
1928     pipe2_dst_rgb565 = 16'h0000;
1929     pipe2_x          = 10'd0;
1930     pipe2_y          = 9'd0;
1931     pipe2_w_q        = 32'd0;
1932     pipe2_tex_addr   = 15'd0;
1933     draw_pipe_valid  = 1'b0;
1934     draw_pipe_inside = 1'b0;
1935     draw_pipe_ztest  = 1'b0;
1936     draw_pipe_textured = 1'b0;
1937     draw_pipe_alpha_key = 1'b0;
1938     draw_pipe_alpha  = 2'd0;
1939     draw_pipe_fog    = 1'b0;
1940     draw_pipe_light_bank = 2'd0;
1941     draw_pipe_tex_or_color = 8'd0;
1942     draw_pipe_addr   = 16'd0;
1943     draw_pipe_z      = 16'd0;
1944     draw_pipe_z_ref  = 16'd0;
1945     draw_pipe_dst_rgb565 = 16'h0000;
1946     draw_pipe_x      = 10'd0;
1947     draw_pipe_y      = 9'd0;
1948     draw_pipe_w_q    = 32'd0;
1949     draw_is_band_primer = 1'b0;
1950     pal_rd_valid     = 1'b0;
1951     pal_rd_pass      = 1'b0;
1952     pal_rd_ztest     = 1'b0;
1953     pal_rd_alpha     = 2'd0;
1954     pal_rd_fog       = 1'b0;
1955     pal_rd_addr      = 16'd0;
1956     pal_rd_z         = 16'd0;
1957     pal_rd_src_addr  = 8'd0;
1958     pal_rd_fog_addr  = 8'd0;
1959     pal_rd_src_sky   = 1'b0;
1960     pal_rd_sky_index = 5'd0;
1961     pal_rd_dst_rgb565 = 16'h0000;
1962     pal_rd_w_q       = 32'd0;
1963     pal_rd_ray_scale_q16 = 34'd0;
1964     plr_valid        = 1'b0;
1965     plr_pass         = 1'b0;
1966     plr_ztest        = 1'b0;
1967     plr_alpha        = 2'd0;
1968     plr_fog          = 1'b0;
1969     plr_addr         = 16'd0;
1970     plr_z           = 16'd0;
1971     plr_src_rgb      = 24'h000000;
1972     plr_dst_rgb565   = 16'h0000;
1973     plr_fog_rgb      = 24'h000000;
1974     plr_w_q          = 32'd0;
1975     plr_ray_scale_q16 = 34'd0;

```

```

1976     fog0_valid      = 1'b0;
1977     fog0_pass      = 1'b0;
1978     fog0_ztest    = 1'b0;
1979     fog0_alpha    = 2'd0;
1980     fog0_fog      = 1'b0;
1981     fog0_addr     = 16'd0;
1982     fog0_z        = 16'd0;
1983     fog0_src_rgb565 = 16'h0000;
1984     fog0_dst_rgb565 = 16'h0000;
1985     fog0_fog_rgb565 = 16'h0000;
1986     fog0_w_q      = 32'd0;
1987     fog0_ray_scale_q16 = 34'd0;
1988     fog1_valid    = 1'b0;
1989     fog1_pass     = 1'b0;
1990     fog1_ztest    = 1'b0;
1991     fog1_alpha    = 2'd0;
1992     fog1_fog     = 1'b0;
1993     fog1_addr    = 16'd0;
1994     fog1_z       = 16'd0;
1995     fog1_src_rgb565 = 16'h0000;
1996     fog1_dst_rgb565 = 16'h0000;
1997     fog1_fog_rgb565 = 16'h0000;
1998     fog1_radial_q8_8 = 16'd0;
1999     commit_valid  = 1'b0;
2000     commit_pass   = 1'b0;
2001     commit_ztest  = 1'b0;
2002     commit_addr  = 16'd0;
2003     commit_z     = 16'd0;
2004     commit_color  = 16'h0000;
2005     pipe0_valid_o = 1'b0;
2006     recip0_valid_o = 1'b0;
2007     recip1_valid_o = 1'b0;
2008     recip2_valid_o = 1'b0;
2009     pipe1_valid_o = 1'b0;
2010     tex0_valid_o  = 1'b0;
2011     pipe2_valid_o = 1'b0;
2012     draw_pipe_valid_o = 1'b0;
2013     pal_rd_valid_o = 1'b0;
2014     pal_rd_src_sky_o = 1'b0;
2015     pal_rd_sky_index_o = 5'd0;
2016     plr_valid_o   = 1'b0;
2017     fog0_valid_o  = 1'b0;
2018     fog1_valid_o  = 1'b0;
2019     commit_valid_o = 1'b0;
2020     scan_rgb565_r = 16'h0000;
2021     scan_visible_r = 1'b0;
2022     scan_line0_ready = 1'b0;
2023     scan_line1_ready = 1'b0;
2024     scan_line2_ready = 1'b0;
2025     scan_line0_row  = 9'd0;
2026     scan_line1_row  = 9'd0;
2027     scan_line2_row  = 9'd0;
2028     scan_active_bank = 2'd0;
2029     scan_active_valid = 1'b0;
2030     scan_active_row  = 9'd0;
2031     scan_fill_active = 1'b0;
2032     scan_fill_armed  = 1'b0;
2033     scan_fill_load_pending = 1'b0;
2034     scan_fill_bank   = 2'd0;

```

```

2035     scan_fill_row      = 9'd0;
2036     scan_fill_base_words = 25'd0;
2037     scan_fill_store_idx = 10'd0;
2038     scan_rd_capture    = 1'b0;
2039     sdram_rd_addr_cfg  = 25'd0;
2040     sdram_rd_max_addr_cfg = 25'd0;
2041     sdram_rd_load_pulse = 1'b0;
2042     sdram_rd_load_stretch_req = 1'b0;
2043     sdram_rd_load_hold  = 4'd0;
2044     cache_band_index   = 3'd0;
2045     cache_target_band  = 3'd0;
2046     band_flush_y_min_cfg = 6'd0;
2047     band_flush_y_max_cfg = 6'd59;
2048     cache_flush_y_min  = 6'd0;
2049     cache_window_start = 16'd0;
2050     cache_window_pixels = 16'd38400;
2051     cache_window_end   = 16'd38400;
2052     cache_valid        = 1'b0;
2053     cache_dirty        = 1'b0;
2054     cache_draw_dirty   = 1'b0;
2055     cache_maint_addr   = 16'd0;
2056     flush_active       = 1'b0;
2057     flush_maint_addr   = 16'd0;
2058     flush_window_start = 16'd0;
2059     flush_pixels_total = 16'd0;
2060     flush_words_issued = 16'd0;
2061     flush_words_done   = 16'd0;
2062     flush_fetch_inflight = 1'b0;
2063     flush_word_pending_valid = 1'b0;
2064     flush_word_pending = 16'd0;
2065     flush_load_pending = 1'b0;
2066     flush_drain_count  = 8'd0;
2067     flush_band_index   = 3'd0;
2068     flush_sdram_wr_addr = 25'd0;
2069     flush_sdram_wr_max_addr = 25'd0;
2070     flush_cache_sel    = 1'b0;
2071     flush_generated_sky = 1'b0;
2072     flush_sky_x        = 10'd0;
2073     flush_sky_row_count = 5'd0;
2074     flush_sky_palette  = 5'd0;
2075     flush_fetch_clear_rgb565 = 16'h0000;
2076     sdram_powerup_counter = 18'd0;
2077     sdram_init_wait_counter = 16'd0;
2078     sdram_ctrl_reset_n = 1'b0;
2079     sdram_ready        = 1'b0;
2080     vga_vs_d           = 1'b1;
2081
2082     palette[0] = 24'h101018;
2083     palette[1] = 24'h6BA43A;
2084     palette[2] = 24'h8B6341;
2085     palette[3] = 24'h6F5737;
2086     palette[4] = 24'h7C7C7C;
2087     palette[5] = 24'hFFFFFF;
2088     palette[6] = 24'hFF4040;
2089     palette[7] = 24'h40A0FF;
2090     palette[8] = 24'hFFD040;
2091     palette[9] = 24'h5C8634;
2092     palette[10] = 24'h6A4A2C;
2093     palette[11] = 24'h9D7B4D;

```

```

2094 palette[12] = 24'h533823;
2095 palette[13] = 24'h989898;
2096 palette[14] = 24'h5C5C5C;
2097 palette[15] = 24'hA77952;
2098 palette[16] = 24'h59412A;
2099 palette[17] = 24'h4F782D;
2100 palette[18] = 24'h84BA57;
2101 palette[19] = 24'h6F4F32;
2102 palette[20] = 24'hAA815A;
2103 palette[21] = 24'h886A44;
2104 palette[22] = 24'h503B24;
2105 palette[23] = 24'h636363;
2106 palette[24] = 24'h9A9A9A;
2107 for (i = 25; i < 256; i = i + 1)
2108     palette[i] = {i[7:0], i[7:0], i[7:0]};
2109 for (i = 0; i < SKY_GRADIENT_COLORS; i = i + 1)
2110     sky_palette[i] = 24'h78B4F0;
2111
2112 for (i = 0; i < FIFO_DEPTH; i = i + 1)
2113     fifo_mem[i] = 32'h0;
2114
2115 /*
2116  * The texture atlas is loaded by voxel_texture_rom via
2117  * altsyncram init_file (voxel_gpu/assets/textures.mif); no $readmemh needed
2118  * for it here anymore.
2119  */
2120 $readmemh("voxel_gpu/assets/recv_lut.hex", recip_lut);
2121
2122 for (i = 0; i < MAX_DESC_WORDS; i = i + 1)
2123     desc_words[i] = 32'h0;
2124 for (i = 0; i < MAX_DESC_WORDS; i = i + 1)
2125     prefetch_words[i] = 32'h0;
2126
2127 for (i = 0; i < 4; i = i + 1) begin
2128     edge_A[i] = 32'sd0;
2129     edge_B[i] = 32'sd0;
2130     edge_C[i] = 32'sd0;
2131     edge_row_val[i] = 64'sd0;
2132     edge_cur_val[i] = 64'sd0;
2133 end
2134
2135 for (i = 0; i < LINE_WORDS; i = i + 1) begin
2136     scan_linebuf0[i] = 16'h0000;
2137     scan_linebuf1[i] = 16'h0000;
2138     scan_linebuf2[i] = 16'h0000;
2139 end
2140 end
2141
2142 always_comb begin
2143     if (!scan_current_x_valid)
2144         scan_rgb565_now = 16'h0000;
2145     else if (scan_active_bank == 2'd1)
2146         scan_rgb565_now = scan_linebuf1[scan_current_x];
2147     else if (scan_active_bank == 2'd2)
2148         scan_rgb565_now = scan_linebuf2[scan_current_x];
2149     else
2150         scan_rgb565_now = scan_linebuf0[scan_current_x];
2151
2152 // 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;

```

```

2330         z_A_e_wr_en    = 1'b0;
2331         z_A_o_wr_en    = 1'b0;
2332     end
2333
2334     /* Cache B ports */
2335     if (draw_cache_sel == 1'b1 && cache_used_by_main) begin
2336         /* B is active: rasterizer read+write. */
2337         fb_B_e_rd_addr = fb_back_rd_addr;
2338         fb_B_o_rd_addr = fb_back_rd_addr_o;
2339         fb_B_e_wr_addr = fb_wr_addr_e;
2340         fb_B_o_wr_addr = fb_wr_addr_o;
2341         fb_B_e_wr_data = fb_wr_data_e;
2342         fb_B_o_wr_data = fb_wr_data_o;
2343         fb_B_e_wr_en   = fb_back_wr_en_e;
2344         fb_B_o_wr_en   = fb_back_wr_en_o;
2345         z_B_e_rd_addr  = z_rd_addr;
2346         z_B_o_rd_addr  = z_rd_addr_o;
2347         z_B_e_wr_addr  = z_wr_addr_e;
2348         z_B_o_wr_addr  = z_wr_addr_o;
2349         z_B_e_wr_data  = z_wr_data_e;
2350         z_B_o_wr_data  = z_wr_data_o;
2351         z_B_e_wr_en    = z_wr_en_e;
2352         z_B_o_wr_en    = z_wr_en_o;
2353     end else if (flush_cache_sel == 1'b1 && flush_active) begin
2354         /* B is being flushed: flush reads. */
2355         fb_B_e_rd_addr = flush_maint_addr;
2356         fb_B_o_rd_addr = flush_maint_addr;
2357         fb_B_e_wr_addr = 16'd0;
2358         fb_B_o_wr_addr = 16'd0;
2359         fb_B_e_wr_data = 16'd0;
2360         fb_B_o_wr_data = 16'd0;
2361         fb_B_e_wr_en   = 1'b0;
2362         fb_B_o_wr_en   = 1'b0;
2363         z_B_e_rd_addr  = flush_maint_addr;
2364         z_B_o_rd_addr  = flush_maint_addr;
2365         z_B_e_wr_addr  = 16'd0;
2366         z_B_o_wr_addr  = 16'd0;
2367         z_B_e_wr_data  = 16'd0;
2368         z_B_o_wr_data  = 16'd0;
2369         z_B_e_wr_en    = 1'b0;
2370         z_B_o_wr_en    = 1'b0;
2371     end else begin
2372         /* B is idle */
2373         fb_B_e_rd_addr = 16'd0;
2374         fb_B_o_rd_addr = 16'd0;
2375         fb_B_e_wr_addr = 16'd0;
2376         fb_B_o_wr_addr = 16'd0;
2377         fb_B_e_wr_data = 16'd0;
2378         fb_B_o_wr_data = 16'd0;
2379         fb_B_e_wr_en   = 1'b0;
2380         fb_B_o_wr_en   = 1'b0;
2381         z_B_e_rd_addr  = 16'd0;
2382         z_B_o_rd_addr  = 16'd0;
2383         z_B_e_wr_addr  = 16'd0;
2384         z_B_o_wr_addr  = 16'd0;
2385         z_B_e_wr_data  = 16'd0;
2386         z_B_o_wr_data  = 16'd0;
2387         z_B_e_wr_en    = 1'b0;
2388         z_B_o_wr_en    = 1'b0;

```

```

2389     end
2390 end
2391
2392 always_ff @(posedge clk) begin
2393     if (reset) begin
2394         ctrl_en           <= 1'b0;
2395         ctrl_ien         <= 1'b0;
2396         ctrl_flp_pending <= 1'b0;
2397         clear_pending    <= 1'b0;
2398         frame_count      <= 32'h0;
2399         pal_addr         <= 8'h0;
2400         sky_pal_addr     <= 5'd0;
2401         fog_enable       <= 1'b0;
2402         fog_color        <= 8'd0;
2403         fog_start_dist   <= 16'd0;
2404         fog_end_dist     <= 16'd0;
2405         fog_inv_proj_sq  <= 16'd0;
2406         extmem_ctrl      <= DEFAULT_EXTMEM_CTRL;
2407         extmem_front_base <= DEFAULT_EXTMEM_FRONT_BASE;
2408         extmem_back_base  <= DEFAULT_EXTMEM_BACK_BASE;
2409         extmem_stride_bytes <= DEFAULT_EXTMEM_STRIDE;
2410         extmem_tile_cfg   <= 32'd0;
2411         extmem_dma_status <= 32'd0;
2412         vsy_latch        <= 1'b0;
2413         display_sel      <= 1'b0;
2414         display_valid    <= 1'b0;
2415         copy_target_sel  <= 1'b1;
2416         copy_complete_pending <= 1'b0;
2417         state            <= ST_IDLE;
2418         fifo_wr_ptr      <= 10'd0;
2419         fifo_rd_ptr      <= 10'd0;
2420         fifo_count       <= 11'd0;
2421         fetch_count      <= 6'd0;
2422         prefetch_count   <= 6'd0;
2423         prefetch_target_words <= BASE_QUAD_WORDS_6;
2424         prefetch_active  <= 1'b0;
2425         prefetch_valid   <= 1'b0;
2426         draw_flush_count <= 4'd0;
2427         clear_addr       <= 16'd0;
2428         draw_row_base    <= 16'd0;
2429         draw_x_min       <= 10'd0;
2430         draw_x_max       <= 10'd0;
2431         draw_x_cur       <= 10'd0;
2432         draw_y_min       <= 9'd0;
2433         draw_y_max       <= 9'd0;
2434         draw_y_cur       <= 9'd0;
2435         draw_tex_or_color <= 8'd0;
2436         draw_flags       <= 8'd0;
2437         draw_z0          <= 16'd0;
2438         draw_dz_dx       <= 16'sd0;
2439         draw_dz_dy       <= 16'sd0;
2440         draw_uw_0        <= 32'sd0;
2441         draw_uw_dx       <= 32'sd0;
2442         draw_uw_dy       <= 32'sd0;
2443         draw_vw_0        <= 32'sd0;
2444         draw_vw_dx       <= 32'sd0;
2445         draw_vw_dy       <= 32'sd0;
2446         draw_iw_0        <= 32'sd0;
2447         draw_iw_dx       <= 32'sd0;

```

```

2448     draw_iw_dy         <= 32'sd0;
2449     z_row_val         <= 48'sd0;
2450     z_cur_val         <= 48'sd0;
2451     uw_row_val        <= 64'sd0;
2452     uw_cur_val        <= 64'sd0;
2453     vw_row_val        <= 64'sd0;
2454     vw_cur_val        <= 64'sd0;
2455     iw_row_val        <= 64'sd0;
2456     iw_cur_val        <= 64'sd0;
2457     pipe0_valid       <= 1'b0;
2458     pipe0_inside     <= 1'b0;
2459     pipe0_ztest      <= 1'b0;
2460     pipe0_textured   <= 1'b0;
2461     pipe0_alpha_key  <= 1'b0;
2462     pipe0_alpha      <= 2'd0;
2463     pipe0_fog        <= 1'b0;
2464     pipe0_light_bank <= 2'd0;
2465     pipe0_tex_or_color <= 8'd0;
2466     pipe0_addr       <= 16'd0;
2467     pipe0_z          <= 16'd0;
2468     pipe0_x          <= 10'd0;
2469     pipe0_y          <= 9'd0;
2470     pipe0_uw_q       <= 32'sd0;
2471     pipe0_vw_q       <= 32'sd0;
2472     pipe0_iw_q       <= 32'd0;
2473     recip0_valid    <= 1'b0;
2474     recip0_inside   <= 1'b0;
2475     recip0_ztest    <= 1'b0;
2476     recip0_textured <= 1'b0;
2477     recip0_alpha_key <= 1'b0;
2478     recip0_alpha    <= 2'd0;
2479     recip0_fog      <= 1'b0;
2480     recip0_light_bank <= 2'd0;
2481     recip0_tex_or_color <= 8'd0;
2482     recip0_addr     <= 16'd0;
2483     recip0_z        <= 16'd0;
2484     recip0_x        <= 10'd0;
2485     recip0_y        <= 9'd0;
2486     recip0_uw_q     <= 32'sd0;
2487     recip0_vw_q     <= 32'sd0;
2488     recip0_iw_zero  <= 1'b1;
2489     recip0_iw_msb   <= 6'd0;
2490     recip0_iw_norm_q <= 32'd0;
2491     recip1_valid    <= 1'b0;
2492     recip1_inside   <= 1'b0;
2493     recip1_ztest    <= 1'b0;
2494     recip1_textured <= 1'b0;
2495     recip1_alpha_key <= 1'b0;
2496     recip1_alpha    <= 2'd0;
2497     recip1_fog      <= 1'b0;
2498     recip1_light_bank <= 2'd0;
2499     recip1_tex_or_color <= 8'd0;
2500     recip1_addr     <= 16'd0;
2501     recip1_z        <= 16'd0;
2502     recip1_z_ref    <= 16'd0;
2503     recip1_dst_rgb565 <= 16'h0000;
2504     recip1_x        <= 10'd0;
2505     recip1_y        <= 9'd0;
2506     recip1_uw_q     <= 32'sd0;

```

```

2507     recip1_vw_q      <= 32'sd0;
2508     recip1_iw_zero  <= 1'b1;
2509     recip1_iw_msb   <= 6'd0;
2510     recip1_iw_lut_frac <= 6'd0;
2511     recip1_w_norm_lo <= 32'd0;
2512     recip1_w_norm_hi <= 32'd0;
2513     recip2_valid    <= 1'b0;
2514     recip2_inside   <= 1'b0;
2515     recip2_ztest    <= 1'b0;
2516     recip2_textured <= 1'b0;
2517     recip2_alpha_key <= 1'b0;
2518     recip2_alpha    <= 2'd0;
2519     recip2_fog      <= 1'b0;
2520     recip2_light_bank <= 2'd0;
2521     recip2_tex_or_color <= 8'd0;
2522     recip2_addr     <= 16'd0;
2523     recip2_z        <= 16'd0;
2524     recip2_z_ref    <= 16'd0;
2525     recip2_dst_rgb565 <= 16'h0000;
2526     recip2_x        <= 10'd0;
2527     recip2_y        <= 9'd0;
2528     recip2_uw_q     <= 32'sd0;
2529     recip2_vw_q     <= 32'sd0;
2530     recip2_iw_zero  <= 1'b1;
2531     recip2_iw_msb   <= 6'd0;
2532     recip2_w_norm_q <= 32'd0;
2533     pipe1_valid     <= 1'b0;
2534     pipe1_inside   <= 1'b0;
2535     pipe1_ztest    <= 1'b0;
2536     pipe1_textured <= 1'b0;
2537     pipe1_alpha_key <= 1'b0;
2538     pipe1_alpha    <= 2'd0;
2539     pipe1_fog      <= 1'b0;
2540     pipe1_light_bank <= 2'd0;
2541     pipe1_tex_or_color <= 8'd0;
2542     pipe1_addr     <= 16'd0;
2543     pipe1_z        <= 16'd0;
2544     pipe1_z_ref    <= 16'd0;
2545     pipe1_dst_rgb565 <= 16'h0000;
2546     pipe1_x        <= 10'd0;
2547     pipe1_y        <= 9'd0;
2548     pipe1_uw_q     <= 32'sd0;
2549     pipe1_vw_q     <= 32'sd0;
2550     pipe1_w_q      <= 32'd0;
2551     tex0_valid     <= 1'b0;
2552     tex0_inside    <= 1'b0;
2553     tex0_ztest    <= 1'b0;
2554     tex0_textured <= 1'b0;
2555     tex0_alpha_key <= 1'b0;
2556     tex0_alpha    <= 2'd0;
2557     tex0_fog      <= 1'b0;
2558     tex0_light_bank <= 2'd0;
2559     tex0_tex_or_color <= 8'd0;
2560     tex0_addr     <= 16'd0;
2561     tex0_z        <= 16'd0;
2562     tex0_z_ref    <= 16'd0;
2563     tex0_dst_rgb565 <= 16'h0000;
2564     tex0_x        <= 10'd0;
2565     tex0_y        <= 9'd0;

```

```

2566     tex0_w_q           <= 32'd0;
2567     tex0_u_prod       <= 64'sd0;
2568     tex0_v_prod       <= 64'sd0;
2569     pipe2_valid        <= 1'b0;
2570     pipe2_inside      <= 1'b0;
2571     pipe2_ztest       <= 1'b0;
2572     pipe2_textured    <= 1'b0;
2573     pipe2_alpha_key   <= 1'b0;
2574     pipe2_alpha       <= 2'd0;
2575     pipe2_fog         <= 1'b0;
2576     pipe2_light_bank  <= 2'd0;
2577     pipe2_tex_or_color <= 8'd0;
2578     pipe2_addr        <= 16'd0;
2579     pipe2_z           <= 16'd0;
2580     pipe2_z_ref       <= 16'd0;
2581     pipe2_dst_rgb565  <= 16'h0000;
2582     pipe2_x           <= 10'd0;
2583     pipe2_y           <= 9'd0;
2584     pipe2_w_q         <= 32'd0;
2585     pipe2_tex_addr    <= 15'd0;
2586     draw_pipe_valid   <= 1'b0;
2587     draw_pipe_inside  <= 1'b0;
2588     draw_pipe_ztest   <= 1'b0;
2589     draw_pipe_textured <= 1'b0;
2590     draw_pipe_alpha_key <= 1'b0;
2591     draw_pipe_alpha   <= 2'd0;
2592     draw_pipe_fog     <= 1'b0;
2593     draw_pipe_light_bank <= 2'd0;
2594     draw_pipe_tex_or_color <= 8'd0;
2595     draw_pipe_addr    <= 16'd0;
2596     draw_pipe_z       <= 16'd0;
2597     draw_pipe_z_ref   <= 16'd0;
2598     draw_pipe_dst_rgb565 <= 16'h0000;
2599     draw_pipe_x       <= 10'd0;
2600     draw_pipe_y       <= 9'd0;
2601     draw_pipe_w_q     <= 32'd0;
2602     draw_is_band_primer <= 1'b0;
2603     pal_rd_valid      <= 1'b0;
2604     pal_rd_pass       <= 1'b0;
2605     pal_rd_ztest      <= 1'b0;
2606     pal_rd_alpha      <= 2'd0;
2607     pal_rd_fog        <= 1'b0;
2608     pal_rd_addr       <= 16'd0;
2609     pal_rd_z          <= 16'd0;
2610     pal_rd_src_addr   <= 8'd0;
2611     pal_rd_fog_addr   <= 8'd0;
2612     pal_rd_src_sky    <= 1'b0;
2613     pal_rd_sky_index  <= 5'd0;
2614     pal_rd_dst_rgb565 <= 16'h0000;
2615     pal_rd_w_q        <= 32'd0;
2616     pal_rd_ray_scale_q16 <= 34'd0;
2617     plr_valid         <= 1'b0;
2618     plr_pass          <= 1'b0;
2619     plr_ztest         <= 1'b0;
2620     plr_alpha         <= 2'd0;
2621     plr_fog           <= 1'b0;
2622     plr_addr          <= 16'd0;
2623     plr_z             <= 16'd0;
2624     plr_src_rgb       <= 24'h000000;

```

```

2625     plr_dst_rgb565    <= 16'h0000;
2626     plr_fog_rgb      <= 24'h000000;
2627     plr_w_q          <= 32'd0;
2628     plr_ray_scale_q16 <= 34'd0;
2629     fog0_valid       <= 1'b0;
2630     fog0_pass        <= 1'b0;
2631     fog0_ztest       <= 1'b0;
2632     fog0_alpha       <= 2'd0;
2633     fog0_fog         <= 1'b0;
2634     fog0_addr        <= 16'd0;
2635     fog0_z           <= 16'd0;
2636     fog0_src_rgb565  <= 16'h0000;
2637     fog0_dst_rgb565  <= 16'h0000;
2638     fog0_fog_rgb565  <= 16'h0000;
2639     fog0_w_q         <= 32'd0;
2640     fog0_ray_scale_q16 <= 34'd0;
2641     fog1_valid       <= 1'b0;
2642     fog1_pass        <= 1'b0;
2643     fog1_ztest       <= 1'b0;
2644     fog1_alpha       <= 2'd0;
2645     fog1_fog         <= 1'b0;
2646     fog1_addr        <= 16'd0;
2647     fog1_z           <= 16'd0;
2648     fog1_src_rgb565  <= 16'h0000;
2649     fog1_dst_rgb565  <= 16'h0000;
2650     fog1_fog_rgb565  <= 16'h0000;
2651     fog1_radial_q8_8 <= 16'd0;
2652     commit_valid     <= 1'b0;
2653     commit_pass      <= 1'b0;
2654     commit_ztest     <= 1'b0;
2655     commit_addr      <= 16'd0;
2656     commit_z         <= 16'd0;
2657     commit_color     <= 16'h0000;
2658     pipe0_valid_o    <= 1'b0;
2659     recip0_valid_o   <= 1'b0;
2660     recip1_valid_o   <= 1'b0;
2661     recip2_valid_o   <= 1'b0;
2662     pipe1_valid_o    <= 1'b0;
2663     tex0_valid_o     <= 1'b0;
2664     pipe2_valid_o    <= 1'b0;
2665     draw_pipe_valid_o <= 1'b0;
2666     pal_rd_valid_o   <= 1'b0;
2667     pal_rd_src_sky_o <= 1'b0;
2668     pal_rd_sky_index_o <= 5'd0;
2669     plr_valid_o      <= 1'b0;
2670     fog0_valid_o     <= 1'b0;
2671     fog1_valid_o     <= 1'b0;
2672     commit_valid_o   <= 1'b0;
2673     scan_rgb565_r    <= 16'h0000;
2674     scan_line0_ready <= 1'b0;
2675     scan_line1_ready <= 1'b0;
2676     scan_line2_ready <= 1'b0;
2677     scan_line0_row   <= 9'd0;
2678     scan_line1_row   <= 9'd0;
2679     scan_line2_row   <= 9'd0;
2680     scan_active_bank <= 2'd0;
2681     scan_active_valid <= 1'b0;
2682     scan_active_row  <= 9'd0;
2683     scan_fill_active <= 1'b0;

```

```

2684     scan_fill_armed    <= 1'b0;
2685     scan_fill_load_pending <= 1'b0;
2686     scan_fill_bank    <= 2'd0;
2687     scan_fill_row     <= 9'd0;
2688     scan_fill_base_words <= 25'd0;
2689     scan_fill_store_idx <= 10'd0;
2690     scan_rd_capture   <= 1'b0;
2691     scan_late_count   <= 16'd0;
2692     sdram_rd_addr_cfg <= 25'd0;
2693     sdram_rd_max_addr_cfg <= 25'd0;
2694     sdram_rd_load_pulse <= 1'b0;
2695     sdram_rd_load_stretch_req <= 1'b0;
2696     sdram_rd_load_hold <= 4'd0;
2697     cache_band_index <= 3'd0;
2698     cache_target_band <= 3'd0;
2699     band_index_cfg <= 3'd0;
2700     band_flush_y_min_cfg <= 6'd0;
2701     band_flush_y_max_cfg <= 6'd59;
2702     cache_flush_y_min <= 6'd0;
2703     cache_window_start <= 16'd0;
2704     cache_window_pixels <= 16'd38400;
2705     cache_window_end <= 16'd38400;
2706     band_begin_pending <= 1'b0;
2707     band_flush_pending <= 1'b0;
2708     cache_valid <= 1'b0;
2709     cache_dirty <= 1'b0;
2710     cache_draw_dirty <= 1'b0;
2711     cache_maint_addr <= 16'd0;
2712     /* Ping-pong cache reset */
2713     draw_cache_sel <= 1'b0;
2714     fb_A_rd_sel_q <= 1'b0;
2715     fb_B_rd_sel_q <= 1'b0;
2716     z_A_rd_sel_q <= 1'b0;
2717     z_B_rd_sel_q <= 1'b0;
2718     flush_active <= 1'b0;
2719     flush_maint_addr <= 16'd0;
2720     flush_window_start <= 16'd0;
2721     flush_pixels_total <= 16'd0;
2722     flush_words_issued <= 16'd0;
2723     flush_words_done <= 16'd0;
2724     flush_fetch_inflight <= 1'b0;
2725     flush_word_pending_valid <= 1'b0;
2726     flush_word_pending <= 16'd0;
2727     flush_load_pending <= 1'b0;
2728     flush_drain_count <= 8'd0;
2729     flush_band_index <= 3'd0;
2730     flush_sdram_wr_addr <= 25'd0;
2731     flush_sdram_wr_max_addr <= 25'd0;
2732     flush_cache_sel <= 1'b0;
2733     flush_generated_sky <= 1'b0;
2734     flush_sky_x <= 10'd0;
2735     flush_sky_row_count <= 5'd0;
2736     flush_sky_palette <= 5'd0;
2737     flush_fetch_clear_rgb565 <= 16'h0000;
2738     draw_row_inside <= 1'b0;
2739     sdram_powerup_counter <= 18'd0;
2740     sdram_init_wait_counter <= 16'd0;
2741     sdram_ctrl_reset_n <= 1'b0;
2742     sdram_ready <= 1'b0;

```

```

2743     vga_vs_d <= 1'b1;
2744     scan_visible_r <= 1'b0;
2745     for (ei = 0; ei < 4; ei = ei + 1) begin
2746         edge_A[ei] <= 32'sd0;
2747         edge_B[ei] <= 32'sd0;
2748         edge_C[ei] <= 32'sd0;
2749         edge_row_val[ei] <= 64'sd0;
2750         edge_cur_val[ei] <= 64'sd0;
2751     end
2752 end else begin
2753     vga_vs_d <= VGA_VS;
2754     scan_rgb565_r <= scan_rgb565_now;
2755     scan_visible_r <= scan_visible_now;
2756     sdram_rd_load_pulse <= 1'b0;
2757     sdram_rd_load_stretch_req <= 1'b0;
2758     scan_rd_capture <= scan_rd_pop;
2759     fb_A_rd_sel_q <= fb_A_e_rd_addr[0];
2760     fb_B_rd_sel_q <= fb_B_e_rd_addr[0];
2761     z_A_rd_sel_q <= z_A_e_rd_addr[0];
2762     z_B_rd_sel_q <= z_B_e_rd_addr[0];
2763
2764     /*
2765     * Whenever a stretched pulse is requested (sampled here as the
2766     * previous-cycle scheduler result), prime the hold counter so
2767     * RD_LOAD stays high for 3 additional REF_CLK cycles (4 total).
2768     */
2769     if (sdram_rd_load_pulse && sdram_rd_load_stretch_req) begin
2770         sdram_rd_load_hold <= 4'd3;
2771     end else if (sdram_rd_load_hold != 4'd0) begin
2772         sdram_rd_load_hold <= sdram_rd_load_hold - 4'd1;
2773     end
2774
2775     if (!sdram_ctrl_reset_n) begin
2776         if (sdram_powerup_counter == SDRAM_POWERUP_HOLD_LAST) begin
2777             sdram_ctrl_reset_n <= 1'b1;
2778         end else begin
2779             sdram_powerup_counter <= sdram_powerup_counter + 18'd1;
2780         end
2781     end else if (!sdram_ready) begin
2782         if (sdram_init_wait_counter == SDRAM_INIT_WAIT_LAST) begin
2783             sdram_ready <= 1'b1;
2784         end else begin
2785             sdram_init_wait_counter <= sdram_init_wait_counter + 16'd1;
2786         end
2787     end
2788
2789     if (vsync_pulse) begin
2790         frame_count <= frame_count + 32'd1;
2791         scan_line0_ready <= 1'b0;
2792         scan_line1_ready <= 1'b0;
2793         scan_line2_ready <= 1'b0;
2794         scan_line0_row <= 9'd0;
2795         scan_line1_row <= 9'd0;
2796         scan_line2_row <= 9'd0;
2797         scan_active_bank <= 2'd0;
2798         scan_active_valid <= 1'b0;
2799         scan_active_row <= 9'd0;
2800         scan_fill_active <= 1'b0;
2801         scan_fill_armed <= 1'b0;

```

```

2802     scan_fill_load_pending <= 1'b0;
2803     scan_fill_bank <= 2'd0;
2804     scan_fill_row <= 9'd0;
2805     scan_fill_base_words <= 25'd0;
2806     scan_fill_store_idx <= 10'd0;
2807     scan_rd_capture <= 1'b0;
2808
2809     if (copy_complete_pending) begin
2810         display_sel <= copy_target_sel;
2811         display_valid <= 1'b1;
2812         ctrl_flp_pending <= 1'b0;
2813         copy_complete_pending <= 1'b0;
2814         vsy_latch <= 1'b1;
2815         if (sdram_ready) begin
2816             scan_fill_active <= 1'b1;
2817             scan_fill_armed <= 1'b1;
2818             scan_fill_load_pending <= 1'b0;
2819             scan_fill_bank <= 2'd0;
2820             scan_fill_row <= 9'd0;
2821             scan_fill_base_words <= copy_target_base_words;
2822             scan_fill_store_idx <= 10'd0;
2823             sdram_rd_addr_cfg <= copy_target_base_words;
2824             sdram_rd_max_addr_cfg <= copy_target_base_words + READ_BURST_WORDS_25;
2825             sdram_rd_load_pulse <= 1'b1;
2826             sdram_rd_load_stretch_req <= 1'b1;
2827         end
2828     end else begin
2829         vsy_latch <= !ctrl_flp_pending;
2830         if (display_valid && sdram_ready) begin
2831             scan_fill_active <= 1'b1;
2832             scan_fill_armed <= 1'b1;
2833             scan_fill_load_pending <= 1'b0;
2834             scan_fill_bank <= 2'd0;
2835             scan_fill_row <= 9'd0;
2836             scan_fill_base_words <= display_base_words;
2837             scan_fill_store_idx <= 10'd0;
2838             sdram_rd_addr_cfg <= display_base_words;
2839             sdram_rd_max_addr_cfg <= display_base_words + READ_BURST_WORDS_25;
2840             sdram_rd_load_pulse <= 1'b1;
2841             sdram_rd_load_stretch_req <= 1'b1;
2842         end
2843     end
2844 end
2845
2846 if (wr) begin
2847     case (address)
2848     ADDR_CONTROL: begin
2849         ctrl_en <= writedata[0];
2850         ctrl_ien <= writedata[2];
2851         if (writedata[1]) begin
2852             ctrl_flp_pending <= 1'b1;
2853             vsy_latch <= 1'b0;
2854         end
2855         if (writedata[3]) begin
2856             clear_pending <= 1'b1;
2857         end
2858     end
2859     ADDR_PAL_ADDR: pal_addr <= writedata[7:0];
2860     ADDR_PAL_DATA: begin

```

```

2861         palette[pal_addr] <= writedata[23:0];
2862         pal_addr <= pal_addr + 8'd1;
2863     end
2864     ADDR_SKY_PAL_ADDR: sky_pal_addr <= writedata[4:0];
2865     ADDR_SKY_PAL_DATA: begin
2866         if (sky_pal_addr <= SKY_GRADIENT_LAST_INDEX)
2867             sky_palette[sky_pal_addr] <= writedata[23:0];
2868             sky_pal_addr <= (sky_pal_addr == SKY_GRADIENT_LAST_INDEX) ?
2869                 5'd0 : sky_pal_addr + 5'd1;
2870         end
2871     ADDR_FOG_RANGE: begin
2872         fog_start_dist <= writedata[15:0];
2873         fog_end_dist <= writedata[31:16];
2874     end
2875     ADDR_FOG_CTRL: begin
2876         fog_color <= writedata[7:0];
2877         fog_enable <= writedata[8];
2878         fog_inv_proj_sq <= writedata[31:16];
2879     end
2880     ADDR_EXTMEM_CTRL: extmem_ctrl <= writedata;
2881     ADDR_EXTMEM_FRONT: extmem_front_base <= writedata;
2882     ADDR_EXTMEM_BACK: extmem_back_base <= writedata;
2883     ADDR_EXTMEM_STRIDE: extmem_stride_bytes <= writedata;
2884     ADDR_EXTMEM_TILE: extmem_tile_cfg <= writedata;
2885     ADDR_BAND_INDEX: begin
2886         band_index_cfg <= writedata[2:0];
2887     end
2888     ADDR_BAND_WINDOW: begin
2889         band_flush_y_min_cfg <= writedata[5:0];
2890         band_flush_y_max_cfg <= writedata[13:8];
2891     end
2892     ADDR_BAND_CTRL: begin
2893         if (writedata[0])
2894             band_begin_pending <= 1'b1;
2895         if (writedata[1])
2896             band_flush_pending <= 1'b1;
2897         end
2898     default: ;
2899     endcase
2900 end
2901
2902 if (fifo_push_req)
2903     fifo_mem[fifo_wr_ptr] <= writedata;
2904 if (fetch_pop_req)
2905     desc_words[fetch_count[4:0]] <= fifo_head;
2906 else if (prefetch_pop_req)
2907     prefetch_words[prefetch_count[4:0]] <= fifo_head;
2908
2909 if (prefetch_can_start) begin
2910     prefetch_active <= 1'b1;
2911     prefetch_valid <= 1'b0;
2912     prefetch_count <= 6'd0;
2913     prefetch_target_words <= BASE_QUAD_WORDS_6;
2914 end else if (prefetch_pop_req) begin
2915     prefetch_count <= prefetch_count + 6'd1;
2916     prefetch_target_words <= prefetch_capture_target_words;
2917 if (prefetch_finishes_on_pop) begin
2918     prefetch_active <= 1'b0;
2919     prefetch_valid <= 1'b1;

```

```

2920         end
2921     end
2922
2923     case ({fifo_push_req, fifo_pop_req})
2924     2'b10: begin
2925         fifo_wr_ptr <= fifo_wr_ptr + 10'd1;
2926         fifo_count <= fifo_count + 11'd1;
2927     end
2928     2'b01: begin
2929         fifo_rd_ptr <= fifo_rd_ptr + 10'd1;
2930         fifo_count <= fifo_count - 11'd1;
2931     end
2932     2'b11: begin
2933         fifo_wr_ptr <= fifo_wr_ptr + 10'd1;
2934         fifo_rd_ptr <= fifo_rd_ptr + 10'd1;
2935     end
2936     default: ;
2937 endcase
2938
2939 if (!vsync_pulse) begin
2940     if (scan_hblank_start && display_valid && scan_active_valid &&
2941         scan_target_valid && !scan_target_line_ready &&
2942         (scan_late_count != 16'hffff))
2943         scan_late_count <= scan_late_count + 16'd1;
2944
2945     if (scan_fill_load_pending) begin
2946         scan_fill_load_pending <= 1'b0;
2947         sdram_rd_addr_cfg <= scan_fill_base_words +
2948             {15'd0, scan_fill_store_idx};
2949         sdram_rd_max_addr_cfg <= scan_fill_base_words +
2950             {15'd0, scan_fill_store_idx} +
2951             READ_BURST_WORDS_25;
2952         sdram_rd_load_pulse <= 1'b1;
2953     end else if (scan_fill_armed && !sdram_rd_load_out && !sdram_rd_empty) begin
2954         scan_fill_armed <= 1'b0;
2955     end
2956
2957     if (scan_rd_capture) begin
2958         case (scan_fill_bank)
2959             2'd0: scan_linebuf0[scan_fill_store_idx] <= sdram_rd_data;
2960             2'd1: scan_linebuf1[scan_fill_store_idx] <= sdram_rd_data;
2961             default: scan_linebuf2[scan_fill_store_idx] <= sdram_rd_data;
2962         endcase
2963
2964         if (scan_fill_line_done) begin
2965             /*
2966              * If the SDRAM RD FIFO output register coughs up a
2967              * stale word on the first pop after RD_LOAD, it
2968              * lands in linebuf[0] and looks like the rightmost
2969              * column wrapping to the left edge. Once the full
2970              * line is resident, copy column 1 over column 0 as
2971              * a one-pixel edge guard. This does not shift the
2972              * scanout timing or touch the actual framebuffer.
2973              */
2974             case (scan_fill_bank)
2975                 2'd0: scan_linebuf0[10'd0] <= scan_linebuf0[10'd1];
2976                 2'd1: scan_linebuf1[10'd0] <= scan_linebuf1[10'd1];
2977                 default: scan_linebuf2[10'd0] <= scan_linebuf2[10'd1];
2978             endcase

```

```

2979         scan_fill_active <= 1'b0;
2980         scan_fill_armed <= 1'b0;
2981         case (scan_fill_bank)
2982             2'd0: begin
2983                 scan_line0_ready <= 1'b1;
2984                 scan_line0_row <= scan_fill_row;
2985             end
2986             2'd1: begin
2987                 scan_line1_ready <= 1'b1;
2988                 scan_line1_row <= scan_fill_row;
2989             end
2990             default: begin
2991                 scan_line2_ready <= 1'b1;
2992                 scan_line2_row <= scan_fill_row;
2993             end
2994         endcase
2995
2996         if (!scan_active_valid) begin
2997             scan_active_valid <= 1'b1;
2998             scan_active_bank <= scan_fill_bank;
2999             scan_active_row <= scan_fill_row;
3000         end
3001     end else begin
3002         scan_fill_store_idx <= scan_fill_store_idx + 10'd1;
3003         if (scan_fill_chunk_done) begin
3004             scan_fill_armed <= 1'b1;
3005             scan_fill_load_pending <= 1'b1;
3006         end
3007     end
3008 end
3009
3010 /* Only change the displayed linebuffer in horizontal blank.
3011 * If a line is late, keep showing the last complete line for
3012 * this scanline instead of switching partway across the row. */
3013 if (scan_hblank_window && display_valid && scan_active_valid &&
3014     (scan_active_row != scan_target_row)) begin
3015     if (scan_line0_ready && (scan_line0_row == scan_target_row)) begin
3016         scan_active_bank <= 2'd0;
3017         scan_active_row <= scan_target_row;
3018     end else if (scan_line1_ready && (scan_line1_row == scan_target_row)) begin
3019         scan_active_bank <= 2'd1;
3020         scan_active_row <= scan_target_row;
3021     end else if (scan_line2_ready && (scan_line2_row == scan_target_row)) begin
3022         scan_active_bank <= 2'd2;
3023         scan_active_row <= scan_target_row;
3024     end else if ((scan_active_row != scan_current_row) &&
3025         scan_line0_ready && (scan_line0_row == scan_current_row)) begin
3026         scan_active_bank <= 2'd0;
3027         scan_active_row <= scan_current_row;
3028     end else if ((scan_active_row != scan_current_row) &&
3029         scan_line1_ready && (scan_line1_row == scan_current_row)) begin
3030         scan_active_bank <= 2'd1;
3031         scan_active_row <= scan_current_row;
3032     end else if ((scan_active_row != scan_current_row) &&
3033         scan_line2_ready && (scan_line2_row == scan_current_row)) begin
3034         scan_active_bank <= 2'd2;
3035         scan_active_row <= scan_current_row;
3036     end else if (scan_line0_ready && (scan_line0_row == scan_active_next_row))
begin

```

```

3037         scan_active_bank <= 2'd0;
3038         scan_active_row <= scan_active_next_row;
3039     end else if (scan_line1_ready && (scan_line1_row == scan_active_next_row))
begin
3040         scan_active_bank <= 2'd1;
3041         scan_active_row <= scan_active_next_row;
3042     end else if (scan_line2_ready && (scan_line2_row == scan_active_next_row))
begin
3043         scan_active_bank <= 2'd2;
3044         scan_active_row <= scan_active_next_row;
3045     end
3046 end
3047
3048 if (scan_prefetch_req) begin
3049     case (scan_prefetch_bank)
3050     2'd0: begin
3051         scan_fill_active <= 1'b1;
3052         scan_fill_armed <= 1'b1;
3053         scan_fill_load_pending <= 1'b0;
3054         scan_fill_bank <= 2'd0;
3055         scan_fill_row <= scan_prefetch_row;
3056         scan_fill_base_words <= scan_prefetch_base_words;
3057         scan_fill_store_idx <= 10'd0;
3058         scan_line0_ready <= 1'b0;
3059         sdram_rd_addr_cfg <= scan_prefetch_base_words;
3060         sdram_rd_max_addr_cfg <= scan_prefetch_base_words + READ_BURST_WORDS_25;
3061         sdram_rd_load_pulse <= 1'b1;
3062         sdram_rd_load_stretch_req <= 1'b1;
3063     end
3064     2'd1: begin
3065         scan_fill_active <= 1'b1;
3066         scan_fill_armed <= 1'b1;
3067         scan_fill_load_pending <= 1'b0;
3068         scan_fill_bank <= 2'd1;
3069         scan_fill_row <= scan_prefetch_row;
3070         scan_fill_base_words <= scan_prefetch_base_words;
3071         scan_fill_store_idx <= 10'd0;
3072         scan_line1_ready <= 1'b0;
3073         sdram_rd_addr_cfg <= scan_prefetch_base_words;
3074         sdram_rd_max_addr_cfg <= scan_prefetch_base_words + READ_BURST_WORDS_25;
3075         sdram_rd_load_pulse <= 1'b1;
3076         sdram_rd_load_stretch_req <= 1'b1;
3077     end
3078     default: begin
3079         scan_fill_active <= 1'b1;
3080         scan_fill_armed <= 1'b1;
3081         scan_fill_load_pending <= 1'b0;
3082         scan_fill_bank <= 2'd2;
3083         scan_fill_row <= scan_prefetch_row;
3084         scan_fill_base_words <= scan_prefetch_base_words;
3085         scan_fill_store_idx <= 10'd0;
3086         scan_line2_ready <= 1'b0;
3087         sdram_rd_addr_cfg <= scan_prefetch_base_words;
3088         sdram_rd_max_addr_cfg <= scan_prefetch_base_words + READ_BURST_WORDS_25;
3089         sdram_rd_load_pulse <= 1'b1;
3090         sdram_rd_load_stretch_req <= 1'b1;
3091     end
3092     endcase
3093 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

```

```

3153     end
3154
3155     /* Flush complete: all words pushed to SDRAM, FIFO drained,
3156     * and the SDRAM controller has had time to finish the final
3157     * burst it already pulled out of the FIFO. */
3158     if ((flush_words_done == flush_pixels_total) &&
3159         !flush_word_pending_valid &&
3160         !flush_fetch_inflight &&
3161         (sdram_wr_use[8:0] == 9'd0)) begin
3162         if (flush_drain_count == COPY_DRAIN_CYCLES) begin
3163             flush_active <= 1'b0;
3164             flush_generated_sky <= 1'b0;
3165             flush_drain_count <= 8'd0;
3166         end else begin
3167             flush_drain_count <= flush_drain_count + 8'd1;
3168         end
3169     end else begin
3170         flush_drain_count <= 8'd0;
3171     end
3172 end
3173
3174 // draw_pipe -> pal_rd: register palette/fog addresses.
3175 pal_rd_valid      <= draw_pipe_valid;
3176 pal_rd_pass      <= draw_commit_pass;
3177 pal_rd_ztest     <= draw_pipe_ztest;
3178 pal_rd_alpha     <= draw_pipe_alpha;
3179 pal_rd_fog       <= draw_pipe_fog;
3180 pal_rd_addr      <= draw_pipe_addr;
3181 pal_rd_z         <= draw_pipe_z;
3182 pal_rd_src_addr  <= palette_src_addr;
3183 pal_rd_fog_addr  <= fog_color;
3184 pal_rd_src_sky   <= draw_pipe_generated_sky;
3185 pal_rd_sky_index <= draw_pipe_sky_index;
3186 pal_rd_dst_rgb565 <= draw_pipe_dst_rgb565;
3187 pal_rd_w_q       <= draw_pipe_w_q;
3188 pal_rd_ray_scale_q16 <= draw_pipe_ray_scale_q16;
3189 pal_rd_valid_o   <= draw_pipe_valid_o;
3190 pal_rd_pass_o    <= draw_commit_pass_o;
3191 pal_rd_ztest_o   <= draw_pipe_ztest_o;
3192 pal_rd_alpha_o   <= draw_pipe_alpha_o;
3193 pal_rd_fog_o     <= draw_pipe_fog_o;
3194 pal_rd_addr_o    <= draw_pipe_addr_o;
3195 pal_rd_z_o       <= draw_pipe_z_o;
3196 pal_rd_src_addr_o <= palette_src_addr_o;
3197 pal_rd_fog_addr_o <= fog_color;
3198 pal_rd_src_sky_o <= draw_pipe_generated_sky_o;
3199 pal_rd_sky_index_o <= draw_pipe_sky_index_o;
3200 pal_rd_dst_rgb565_o <= draw_pipe_dst_rgb565_o;
3201 pal_rd_w_q_o     <= draw_pipe_w_q_o;
3202 pal_rd_ray_scale_q16_o <= draw_pipe_ray_scale_q16_o;
3203
3204 // pal_rd -> plr: capture palette RGB values.
3205 plr_valid        <= pal_rd_valid;
3206 plr_pass         <= pal_rd_pass;
3207 plr_ztest        <= pal_rd_ztest;
3208 plr_alpha        <= pal_rd_alpha;
3209 plr_fog          <= pal_rd_fog;
3210 plr_addr         <= pal_rd_addr;
3211 plr_z            <= pal_rd_z;

```

```

3212     plr_src_rgb         <= pal_rd_src_sky ?
3213                     sky_palette[pal_rd_sky_index] :
3214                     palette[pal_rd_src_addr];
3215     plr_dst_rgb565     <= pal_rd_dst_rgb565;
3216     plr_fog_rgb       <= palette[pal_rd_fog_addr];
3217     plr_w_q           <= pal_rd_w_q;
3218     plr_ray_scale_q16 <= pal_rd_ray_scale_q16;
3219     plr_valid_o       <= pal_rd_valid_o;
3220     plr_pass_o        <= pal_rd_pass_o;
3221     plr_ztest_o       <= pal_rd_ztest_o;
3222     plr_alpha_o       <= pal_rd_alpha_o;
3223     plr_fog_o         <= pal_rd_fog_o;
3224     plr_addr_o        <= pal_rd_addr_o;
3225     plr_z_o           <= pal_rd_z_o;
3226     plr_src_rgb_o     <= pal_rd_src_sky_o ?
3227                     sky_palette[pal_rd_sky_index_o] :
3228                     palette[pal_rd_src_addr_o];
3229     plr_dst_rgb565_o  <= pal_rd_dst_rgb565_o;
3230     plr_fog_rgb_o     <= palette[pal_rd_fog_addr_o];
3231     plr_w_q_o         <= pal_rd_w_q_o;
3232     plr_ray_scale_q16_o <= pal_rd_ray_scale_q16_o;
3233
3234     // plr -> fog0: convert source/fog colors to RGB565.
3235     fog0_valid <= plr_valid;
3236     fog0_pass <= plr_pass;
3237     fog0_ztest <= plr_ztest;
3238     fog0_alpha <= plr_alpha;
3239     fog0_fog <= plr_fog;
3240     fog0_addr <= plr_addr;
3241     fog0_z <= plr_z;
3242     fog0_src_rgb565 <= plr_src_rgb565;
3243     fog0_dst_rgb565 <= plr_dst_rgb565;
3244     fog0_fog_rgb565 <= plr_fog_rgb565;
3245     fog0_w_q <= plr_w_q;
3246     fog0_ray_scale_q16 <= plr_ray_scale_q16;
3247     fog0_valid_o <= plr_valid_o;
3248     fog0_pass_o <= plr_pass_o;
3249     fog0_ztest_o <= plr_ztest_o;
3250     fog0_alpha_o <= plr_alpha_o;
3251     fog0_fog_o <= plr_fog_o;
3252     fog0_addr_o <= plr_addr_o;
3253     fog0_z_o <= plr_z_o;
3254     fog0_src_rgb565_o <= plr_src_rgb565_o;
3255     fog0_dst_rgb565_o <= plr_dst_rgb565_o;
3256     fog0_fog_rgb565_o <= plr_fog_rgb565_o;
3257     fog0_w_q_o <= plr_w_q_o;
3258     fog0_ray_scale_q16_o <= plr_ray_scale_q16_o;
3259
3260     // fog0 -> fog1: finish radial distance in Q8.8.
3261     fog1_valid <= fog0_valid;
3262     fog1_pass <= fog0_pass;
3263     fog1_ztest <= fog0_ztest;
3264     fog1_alpha <= fog0_alpha;
3265     fog1_fog <= fog0_fog;
3266     fog1_addr <= fog0_addr;
3267     fog1_z <= fog0_z;
3268     fog1_src_rgb565 <= fog0_src_rgb565;
3269     fog1_dst_rgb565 <= fog0_dst_rgb565;
3270     fog1_fog_rgb565 <= fog0_fog_rgb565;

```

```

3271     fog1_radial_q8_8 <= fog0_radial_q8_8;
3272     fog1_valid_o <= fog0_valid_o;
3273     fog1_pass_o <= fog0_pass_o;
3274     fog1_ztest_o <= fog0_ztest_o;
3275     fog1_alpha_o <= fog0_alpha_o;
3276     fog1_fog_o <= fog0_fog_o;
3277     fog1_addr_o <= fog0_addr_o;
3278     fog1_z_o <= fog0_z_o;
3279     fog1_src_rgb565_o <= fog0_src_rgb565_o;
3280     fog1_dst_rgb565_o <= fog0_dst_rgb565_o;
3281     fog1_fog_rgb565_o <= fog0_fog_rgb565_o;
3282     fog1_radial_q8_8_o <= fog0_radial_q8_8_o;
3283
3284     // fog1 -> commit: final fog/alpha-blended color.
3285     commit_valid <= fog1_valid;
3286     commit_pass <= fog1_pass;
3287     commit_ztest <= fog1_ztest;
3288     commit_addr <= fog1_addr;
3289     commit_z <= fog1_z;
3290     commit_color <= fog1_out_rgb565;
3291     commit_valid_o <= fog1_valid_o;
3292     commit_pass_o <= fog1_pass_o;
3293     commit_ztest_o <= fog1_ztest_o;
3294     commit_addr_o <= fog1_addr_o;
3295     commit_z_o <= fog1_z_o;
3296     commit_color_o <= fog1_out_rgb565_o;
3297     if ((state == ST_DRAW || state == ST_DRAW_FLUSH) &&
3298         ((commit_valid && commit_pass) ||
3299          (commit_valid_o && commit_pass_o))) begin
3300         cache_dirty <= 1'b1;
3301         if (!draw_is_band_primer)
3302             cache_draw_dirty <= 1'b1;
3303     end
3304
3305     case (state)
3306     ST_IDLE: begin
3307         fetch_count <= 6'd0;
3308         draw_flush_count <= 4'd0;
3309         pipe0_valid <= 1'b0;
3310         recip0_valid <= 1'b0;
3311         recip1_valid <= 1'b0;
3312         recip2_valid <= 1'b0;
3313         pipe1_valid <= 1'b0;
3314         tex0_valid <= 1'b0;
3315         pipe2_valid <= 1'b0;
3316         draw_pipe_valid <= 1'b0;
3317         pal_rd_valid <= 1'b0;
3318         plr_valid <= 1'b0;
3319         fog0_valid <= 1'b0;
3320         fog1_valid <= 1'b0;
3321         commit_valid <= 1'b0;
3322         pipe0_valid_o <= 1'b0;
3323         recip0_valid_o <= 1'b0;
3324         recip1_valid_o <= 1'b0;
3325         recip2_valid_o <= 1'b0;
3326         pipe1_valid_o <= 1'b0;
3327         tex0_valid_o <= 1'b0;
3328         pipe2_valid_o <= 1'b0;
3329         draw_pipe_valid_o <= 1'b0;

```

```

3330     pal_rd_valid_o <= 1'b0;
3331     plr_valid_o <= 1'b0;
3332     fog0_valid_o <= 1'b0;
3333     fog1_valid_o <= 1'b0;
3334     commit_valid_o <= 1'b0;
3335     if (clear_pending) begin
3336         state         <= ST_CLEAR;
3337         clear_pending <= 1'b0;
3338         clear_addr    <= 16'd0;
3339     end else if (band_begin_pending && band_begin_cache_available) begin
3340         /* Only two local caches exist; wait here rather than
3341          * toggling back into a cache still being flushed. */
3342         band_begin_pending <= 1'b0;
3343         cache_target_band <= band_index_cfg;
3344         cache_band_index <= band_index_cfg;
3345         cache_flush_y_min <= band_flush_y_min_cfg;
3346         cache_window_start <= band_local_row_offset(band_flush_y_min_cfg);
3347         cache_window_pixels <= band_row_window_count(
3348             band_flush_y_min_cfg, band_flush_y_max_cfg);
3349         cache_window_end <= band_local_row_offset(band_flush_y_min_cfg) +
3350             band_row_window_count(
3351                 band_flush_y_min_cfg,
3352                 band_flush_y_max_cfg);
3353         cache_maint_addr <= band_local_row_offset(band_flush_y_min_cfg);
3354         cache_valid <= 1'b0;
3355         cache_dirty <= 1'b0;
3356         cache_draw_dirty <= 1'b0;
3357         /* Toggle cache selector: draw into the other cache */
3358         draw_cache_sel <= ~draw_cache_sel;
3359         state <= ST_CACHE_INIT;
3360     end else if (band_flush_pending && !flush_active) begin
3361         /* Do not require fifo_count==0: pipelined next-band
3362          * descriptors may already be queued behind this flush. */
3363         band_flush_pending <= 1'b0;
3364         if (cache_valid && cache_dirty) begin
3365             /* Flush this cache while the next band uses the other. */
3366             flush_active <= 1'b1;
3367             flush_band_index <= cache_band_index;
3368             flush_pixels_total <= cache_window_pixels;
3369             flush_window_start <= cache_window_start;
3370             flush_maint_addr <= cache_window_start;
3371             flush_words_issued <= 16'd0;
3372             flush_words_done <= 16'd0;
3373             flush_fetch_inflight <= 1'b0;
3374             flush_word_pending_valid <= 1'b0;
3375             flush_load_pending <= 1'b1;
3376             flush_drain_count <= 8'd0;
3377             flush_cache_sel <= draw_cache_sel;
3378             flush_generated_sky <= sky_gradient_clear_enabled &&
3379                 !cache_draw_dirty;
3380             flush_sky_x <= 10'd0;
3381             flush_sky_row_count <=
3382                 sky_row_count_for_local_y(cache_flush_y_min);
3383             flush_sky_palette <=
3384                 sky_clear_index_for_local_y(cache_band_index,
3385                     cache_flush_y_min);
3386             flush_sdram_wr_addr <= copy_target_base_words +
3387                 band_word_offset(cache_band_index) +
3388                 {9'd0, cache_window_start};

```

```

3389         flush_sdram_wr_max_addr <= copy_target_base_words +
3390             band_word_offset(cache_band_index) +
3391             {9'd0, cache_window_start} +
3392             {9'd0, cache_window_pixels};
3393
3394         cache_valid <= 1'b0;
3395         cache_dirty <= 1'b0;
3396         cache_draw_dirty <= 1'b0;
3397     end
3398     /* Background flush owns SDRAM writeback; stay in
3399      * ST_IDLE so the next BEGIN_BAND can proceed. */
3400 end else if (ctrl_flp_pending && sdram_ready &&
3401     !copy_complete_pending && !flush_active) begin
3402     copy_complete_pending <= 1'b1;
3403 end else if (prefetch_valid && !band_flush_pending) begin
3404     for (ei = 0; ei < MAX_DESC_WORDS; ei = ei + 1)
3405         desc_words[ei] <= prefetch_words[ei];
3406     fetch_count <= prefetch_target_words;
3407     prefetch_valid <= 1'b0;
3408     state <= ST_FETCH;
3409 end else if (!prefetch_active && ctrl_en && (fifo_count >= 11'd16) &&
3410     !band_flush_pending) begin
3411     /* Wait until the queued flush/begin pair advances the
3412      * resident cache before fetching next-band descriptors. */
3413     state <= ST_FETCH;
3414     fetch_count <= 6'd0;
3415 end
3416
3417 ST_CLEAR: begin
3418     copy_target_sel <= ~display_sel;
3419     cache_valid <= 1'b0;
3420     cache_dirty <= 1'b0;
3421     cache_draw_dirty <= 1'b0;
3422     cache_flush_y_min <= 6'd0;
3423     cache_window_start <= 16'd0;
3424     cache_window_pixels <= 16'd38400;
3425     cache_window_end <= 16'd38400;
3426     band_begin_pending <= 1'b0;
3427     band_flush_pending <= 1'b0;
3428     prefetch_count <= 6'd0;
3429     prefetch_target_words <= BASE_QUAD_WORDS_6;
3430     prefetch_active <= 1'b0;
3431     prefetch_valid <= 1'b0;
3432     copy_complete_pending <= 1'b0;
3433     draw_cache_sel <= 1'b0;
3434     /* Do not kill flush_active here -- see the comment in
3435      * ctrl_clear_write. Let the background flush drain
3436      * naturally so the SDRAM framebuffer is fully written
3437      * before scanout displays it. */
3438     state <= ST_IDLE;
3439 end
3440
3441 ST_FETCH: begin
3442     if (fetch_count == fetch_target_words) begin
3443         draw_x_min <= desc_x_min;
3444         draw_x_max <= desc_x_max;
3445         draw_y_min <= desc_y_min;
3446         draw_y_max <= desc_y_max;
3447         draw_row_base <= band_local_addr(desc_x_start_even, desc_y_min,

```

```

3448                                     cache_band_index);
3449 draw_x_cur <= desc_x_start_even;
3450 draw_y_cur <= desc_y_min;
3451 draw_row_inside <= 1'b0;
3452 draw_tex_or_color <= desc_tex_or_color;
3453 draw_flags <= desc_flags;
3454 draw_is_band_primer <= desc_band_primer;
3455 draw_z0 <= desc_z0;
3456 draw_dz_dx <= desc_dz_dx;
3457 draw_dz_dy <= desc_dz_dy;
3458 if (desc_has_uv) begin
3459     draw_uw_0 <= desc_uw_0;
3460     draw_uw_dx <= desc_uw_dx;
3461     draw_uw_dy <= desc_uw_dy;
3462     draw_vw_0 <= desc_vw_0;
3463     draw_vw_dx <= desc_vw_dx;
3464     draw_vw_dy <= desc_vw_dy;
3465     draw_iw_0 <= desc_iw_0;
3466     draw_iw_dx <= desc_iw_dx;
3467     draw_iw_dy <= desc_iw_dy;
3468 end else begin
3469     draw_uw_0 <= 32'sd0;
3470     draw_uw_dx <= 32'sd0;
3471     draw_uw_dy <= 32'sd0;
3472     draw_vw_0 <= 32'sd0;
3473     draw_vw_dx <= 32'sd0;
3474     draw_vw_dy <= 32'sd0;
3475     draw_iw_0 <= 32'sd0;
3476     draw_iw_dx <= 32'sd0;
3477     draw_iw_dy <= 32'sd0;
3478 end
3479 pipe0_valid <= 1'b0;
3480 recip0_valid <= 1'b0;
3481 recip1_valid <= 1'b0;
3482 recip2_valid <= 1'b0;
3483 pipe1_valid <= 1'b0;
3484 tex0_valid <= 1'b0;
3485 pipe2_valid <= 1'b0;
3486 draw_pipe_valid <= 1'b0;
3487 pal_rd_valid <= 1'b0;
3488 plr_valid <= 1'b0;
3489 fog0_valid <= 1'b0;
3490 fog1_valid <= 1'b0;
3491 commit_valid <= 1'b0;
3492 pipe0_valid_o <= 1'b0;
3493 recip0_valid_o <= 1'b0;
3494 recip1_valid_o <= 1'b0;
3495 recip2_valid_o <= 1'b0;
3496 pipe1_valid_o <= 1'b0;
3497 tex0_valid_o <= 1'b0;
3498 pipe2_valid_o <= 1'b0;
3499 draw_pipe_valid_o <= 1'b0;
3500 pal_rd_valid_o <= 1'b0;
3501 plr_valid_o <= 1'b0;
3502 fog0_valid_o <= 1'b0;
3503 fog1_valid_o <= 1'b0;
3504 commit_valid_o <= 1'b0;
3505 for (ei = 0; ei < 4; ei = ei + 1) begin
3506     edge_A[ei] <= $signed(desc_words[2 + ei * 3]);

```

```

3507         edge_B[ei] <= $signed(desc_words[3 + ei * 3]);
3508         edge_C[ei] <= $signed(desc_words[4 + ei * 3]);
3509     end
3510
3511     if ((desc_x_min > desc_x_max) || (desc_y_min > desc_y_max) ||
3512         desc_redundant_sky_clear)
3513         state <= ST_IDLE;
3514     else
3515         state <= ST_SETUP;
3516     end else if (fifo_pop_req) begin
3517         fetch_count <= fetch_count + 6'd1;
3518     end
3519 end
3520
3521 ST_SETUP: begin
3522     edge_row_val[0] <= edge_eval0;
3523     edge_cur_val[0] <= edge_eval0;
3524     edge_row_val[1] <= edge_eval1;
3525     edge_cur_val[1] <= edge_eval1;
3526     edge_row_val[2] <= edge_eval2;
3527     edge_cur_val[2] <= edge_eval2;
3528     edge_row_val[3] <= edge_eval3;
3529     edge_cur_val[3] <= edge_eval3;
3530
3531     z_row_val <= draw_z_start_val;
3532     z_cur_val <= draw_z_start_val;
3533     uw_row_val <= draw_uw_start_val;
3534     uw_cur_val <= draw_uw_start_val;
3535     vw_row_val <= draw_vw_start_val;
3536     vw_cur_val <= draw_vw_start_val;
3537     iw_row_val <= draw_iw_start_val;
3538     iw_cur_val <= draw_iw_start_val;
3539
3540     state <= ST_DRAW;
3541 end
3542
3543 ST_DRAW,
3544 ST_DRAW_FLUSH: begin
3545     // pipe0 -> recip0: normalize 1/w; all raster metadata carries.
3546     recip0_valid <= pipe0_valid;
3547     recip0_inside <= pipe0_inside;
3548     recip0_ztest <= pipe0_ztest;
3549     recip0_textured <= pipe0_textured;
3550     recip0_alpha_key <= pipe0_alpha_key;
3551     recip0_alpha <= pipe0_alpha;
3552     recip0_fog <= pipe0_fog;
3553     recip0_light_bank <= pipe0_light_bank;
3554     recip0_tex_or_color <= pipe0_tex_or_color;
3555     recip0_addr <= pipe0_addr;
3556     recip0_z <= pipe0_z;
3557     recip0_x <= pipe0_x;
3558     recip0_y <= pipe0_y;
3559     recip0_uw_q <= pipe0_uw_q;
3560     recip0_vw_q <= pipe0_vw_q;
3561     recip0_iw_zero <= (pipe0_iw_q == 32'd0);
3562     recip0_iw_msb <= pipe0_iw_msb;
3563     recip0_iw_norm_q <= pipe0_iw_norm_q;
3564     recip0_valid_o <= pipe0_valid_o;
3565     recip0_inside_o <= pipe0_inside_o;

```

```

3566     recip0_ztest_o <= pipe0_ztest_o;
3567     recip0_textured_o <= pipe0_textured_o;
3568     recip0_alpha_key_o <= pipe0_alpha_key_o;
3569     recip0_alpha_o <= pipe0_alpha_o;
3570     recip0_fog_o <= pipe0_fog_o;
3571     recip0_light_bank_o <= pipe0_light_bank_o;
3572     recip0_tex_or_color_o <= pipe0_tex_or_color_o;
3573     recip0_addr_o <= pipe0_addr_o;
3574     recip0_z_o <= pipe0_z_o;
3575     recip0_x_o <= pipe0_x_o;
3576     recip0_y_o <= pipe0_y_o;
3577     recip0_uw_q_o <= pipe0_uw_q_o;
3578     recip0_vw_q_o <= pipe0_vw_q_o;
3579     recip0_iw_zero_o <= (pipe0_iw_q_o == 32'd0);
3580     recip0_iw_msb_o <= pipe0_iw_msb_o;
3581     recip0_iw_norm_q_o <= pipe0_iw_norm_q_o;
3582
3583     // recip0 -> recip1: read reciprocal LUT and destination cache.
3584     recip1_valid <= recip0_valid;
3585     recip1_inside <= recip0_inside;
3586     recip1_ztest <= recip0_ztest;
3587     recip1_textured <= recip0_textured;
3588     recip1_alpha_key <= recip0_alpha_key;
3589     recip1_alpha <= recip0_alpha;
3590     recip1_fog <= recip0_fog;
3591     recip1_light_bank <= recip0_light_bank;
3592     recip1_tex_or_color <= recip0_tex_or_color;
3593     recip1_addr <= recip0_addr;
3594     recip1_z <= recip0_z;
3595     recip1_z_ref <= z_draw_rd_data_e;
3596     recip1_dst_rgb565 <=
3597         (z_draw_rd_data_e == Z_CLEAR_SENTINEL) ?
3598         draw_clear_rgb565 : fb_draw_rd_data_e;
3599     recip1_x <= recip0_x;
3600     recip1_y <= recip0_y;
3601     recip1_uw_q <= recip0_uw_q;
3602     recip1_vw_q <= recip0_vw_q;
3603     recip1_iw_zero <= recip0_iw_zero;
3604     recip1_iw_msb <= recip0_iw_msb;
3605     recip1_iw_lut_frac <= recip0_iw_lut_frac;
3606     recip1_w_norm_lo <= recip_lut[recip0_iw_lut_idx];
3607     recip1_w_norm_hi <= recip_lut[recip0_iw_lut_idx + 11'd1];
3608     recip1_valid_o <= recip0_valid_o;
3609     recip1_inside_o <= recip0_inside_o;
3610     recip1_ztest_o <= recip0_ztest_o;
3611     recip1_textured_o <= recip0_textured_o;
3612     recip1_alpha_key_o <= recip0_alpha_key_o;
3613     recip1_alpha_o <= recip0_alpha_o;
3614     recip1_fog_o <= recip0_fog_o;
3615     recip1_light_bank_o <= recip0_light_bank_o;
3616     recip1_tex_or_color_o <= recip0_tex_or_color_o;
3617     recip1_addr_o <= recip0_addr_o;
3618     recip1_z_o <= recip0_z_o;
3619     recip1_z_ref_o <= z_draw_rd_data_o;
3620     recip1_dst_rgb565_o <=
3621         (z_draw_rd_data_o == Z_CLEAR_SENTINEL) ?
3622         draw_clear_rgb565 : fb_draw_rd_data_o;
3623     recip1_x_o <= recip0_x_o;
3624     recip1_y_o <= recip0_y_o;

```

```

3625 recip1_uw_q_o <= recip0_uw_q_o;
3626 recip1_vw_q_o <= recip0_vw_q_o;
3627 recip1_iw_zero_o <= recip0_iw_zero_o;
3628 recip1_iw_msb_o <= recip0_iw_msb_o;
3629 recip1_iw_lut_frac_o <= recip0_iw_lut_frac_o;
3630 recip1_w_norm_lo_o <= recip_lut[recip0_iw_lut_idx_o];
3631 recip1_w_norm_hi_o <= recip_lut[recip0_iw_lut_idx_o + 11'd1];
3632
3633 // recip1 -> recip2: interpolate reciprocal.
3634 recip2_valid <= recip1_valid;
3635 recip2_inside <= recip1_inside;
3636 recip2_ztest <= recip1_ztest;
3637 recip2_textured <= recip1_textured;
3638 recip2_alpha_key <= recip1_alpha_key;
3639 recip2_alpha <= recip1_alpha;
3640 recip2_fog <= recip1_fog;
3641 recip2_light_bank <= recip1_light_bank;
3642 recip2_tex_or_color <= recip1_tex_or_color;
3643 recip2_addr <= recip1_addr;
3644 recip2_z <= recip1_z;
3645 recip2_z_ref <= recip1_z_ref;
3646 recip2_dst_rgb565 <= recip1_dst_rgb565;
3647 recip2_x <= recip1_x;
3648 recip2_y <= recip1_y;
3649 recip2_uw_q <= recip1_uw_q;
3650 recip2_vw_q <= recip1_vw_q;
3651 recip2_iw_zero <= recip1_iw_zero;
3652 recip2_iw_msb <= recip1_iw_msb;
3653 recip2_w_norm_q <= recip1_w_norm_q;
3654 recip2_valid_o <= recip1_valid_o;
3655 recip2_inside_o <= recip1_inside_o;
3656 recip2_ztest_o <= recip1_ztest_o;
3657 recip2_textured_o <= recip1_textured_o;
3658 recip2_alpha_key_o <= recip1_alpha_key_o;
3659 recip2_alpha_o <= recip1_alpha_o;
3660 recip2_fog_o <= recip1_fog_o;
3661 recip2_light_bank_o <= recip1_light_bank_o;
3662 recip2_tex_or_color_o <= recip1_tex_or_color_o;
3663 recip2_addr_o <= recip1_addr_o;
3664 recip2_z_o <= recip1_z_o;
3665 recip2_z_ref_o <= recip1_z_ref_o;
3666 recip2_dst_rgb565_o <= recip1_dst_rgb565_o;
3667 recip2_x_o <= recip1_x_o;
3668 recip2_y_o <= recip1_y_o;
3669 recip2_uw_q_o <= recip1_uw_q_o;
3670 recip2_vw_q_o <= recip1_vw_q_o;
3671 recip2_iw_zero_o <= recip1_iw_zero_o;
3672 recip2_iw_msb_o <= recip1_iw_msb_o;
3673 recip2_w_norm_q_o <= recip1_w_norm_q_o;
3674
3675 // recip2 -> pipe1: denormalize w.
3676 pipe1_valid <= recip2_valid;
3677 pipe1_inside <= recip2_inside;
3678 pipe1_ztest <= recip2_ztest;
3679 pipe1_textured <= recip2_textured;
3680 pipe1_alpha_key <= recip2_alpha_key;
3681 pipe1_alpha <= recip2_alpha;
3682 pipe1_fog <= recip2_fog;
3683 pipe1_light_bank <= recip2_light_bank;

```

```

3684     pipe1_tex_or_color <= recip2_tex_or_color;
3685     pipe1_addr <= recip2_addr;
3686     pipe1_z <= recip2_z;
3687     pipe1_z_ref <= recip2_z_ref;
3688     pipe1_dst_rgb565 <= recip2_dst_rgb565;
3689     pipe1_x <= recip2_x;
3690     pipe1_y <= recip2_y;
3691     pipe1_uw_q <= recip2_uw_q;
3692     pipe1_vw_q <= recip2_vw_q;
3693     pipe1_w_q <= recip2_w_q;
3694     pipe1_valid_o <= recip2_valid_o;
3695     pipe1_inside_o <= recip2_inside_o;
3696     pipe1_ztest_o <= recip2_ztest_o;
3697     pipe1_textured_o <= recip2_textured_o;
3698     pipe1_alpha_key_o <= recip2_alpha_key_o;
3699     pipe1_alpha_o <= recip2_alpha_o;
3700     pipe1_fog_o <= recip2_fog_o;
3701     pipe1_light_bank_o <= recip2_light_bank_o;
3702     pipe1_tex_or_color_o <= recip2_tex_or_color_o;
3703     pipe1_addr_o <= recip2_addr_o;
3704     pipe1_z_o <= recip2_z_o;
3705     pipe1_z_ref_o <= recip2_z_ref_o;
3706     pipe1_dst_rgb565_o <= recip2_dst_rgb565_o;
3707     pipe1_x_o <= recip2_x_o;
3708     pipe1_y_o <= recip2_y_o;
3709     pipe1_uw_q_o <= recip2_uw_q_o;
3710     pipe1_vw_q_o <= recip2_vw_q_o;
3711     pipe1_w_q_o <= recip2_w_q_o;
3712
3713     // pipe1 -> tex0: multiply (u/w, v/w) by w.
3714     tex0_valid <= pipe1_valid;
3715     tex0_inside <= pipe1_inside;
3716     tex0_ztest <= pipe1_ztest;
3717     tex0_textured <= pipe1_textured;
3718     tex0_alpha_key <= pipe1_alpha_key;
3719     tex0_alpha <= pipe1_alpha;
3720     tex0_fog <= pipe1_fog;
3721     tex0_light_bank <= pipe1_light_bank;
3722     tex0_tex_or_color <= pipe1_tex_or_color;
3723     tex0_addr <= pipe1_addr;
3724     tex0_z <= pipe1_z;
3725     tex0_z_ref <= pipe1_z_ref;
3726     tex0_dst_rgb565 <= pipe1_dst_rgb565;
3727     tex0_x <= pipe1_x;
3728     tex0_y <= pipe1_y;
3729     tex0_w_q <= pipe1_w_q;
3730     tex0_u_prod <= pipe1_u_prod;
3731     tex0_v_prod <= pipe1_v_prod;
3732     tex0_valid_o <= pipe1_valid_o;
3733     tex0_inside_o <= pipe1_inside_o;
3734     tex0_ztest_o <= pipe1_ztest_o;
3735     tex0_textured_o <= pipe1_textured_o;
3736     tex0_alpha_key_o <= pipe1_alpha_key_o;
3737     tex0_alpha_o <= pipe1_alpha_o;
3738     tex0_fog_o <= pipe1_fog_o;
3739     tex0_light_bank_o <= pipe1_light_bank_o;
3740     tex0_tex_or_color_o <= pipe1_tex_or_color_o;
3741     tex0_addr_o <= pipe1_addr_o;
3742     tex0_z_o <= pipe1_z_o;

```

```

3743 tex0_z_ref_o <= pipe1_z_ref_o;
3744 tex0_dst_rgb565_o <= pipe1_dst_rgb565_o;
3745 tex0_x_o <= pipe1_x_o;
3746 tex0_y_o <= pipe1_y_o;
3747 tex0_w_q_o <= pipe1_w_q_o;
3748 tex0_u_prod_o <= pipe1_u_prod_o;
3749 tex0_v_prod_o <= pipe1_v_prod_o;
3750
3751 // tex0 -> pipe2: build texture atlas address.
3752 pipe2_valid <= tex0_valid;
3753 pipe2_inside <= tex0_inside;
3754 pipe2_ztest <= tex0_ztest;
3755 pipe2_textured <= tex0_textured;
3756 pipe2_alpha_key <= tex0_alpha_key;
3757 pipe2_alpha <= tex0_alpha;
3758 pipe2_fog <= tex0_fog;
3759 pipe2_light_bank <= tex0_light_bank;
3760 pipe2_tex_or_color <= tex0_tex_or_color;
3761 pipe2_addr <= tex0_addr;
3762 pipe2_z <= tex0_z;
3763 pipe2_z_ref <= tex0_z_ref;
3764 pipe2_dst_rgb565 <= tex0_dst_rgb565;
3765 pipe2_x <= tex0_x;
3766 pipe2_y <= tex0_y;
3767 pipe2_w_q <= tex0_w_q;
3768 pipe2_tex_addr <= tex0_tex_addr;
3769 pipe2_valid_o <= tex0_valid_o;
3770 pipe2_inside_o <= tex0_inside_o;
3771 pipe2_ztest_o <= tex0_ztest_o;
3772 pipe2_textured_o <= tex0_textured_o;
3773 pipe2_alpha_key_o <= tex0_alpha_key_o;
3774 pipe2_alpha_o <= tex0_alpha_o;
3775 pipe2_fog_o <= tex0_fog_o;
3776 pipe2_light_bank_o <= tex0_light_bank_o;
3777 pipe2_tex_or_color_o <= tex0_tex_or_color_o;
3778 pipe2_addr_o <= tex0_addr_o;
3779 pipe2_z_o <= tex0_z_o;
3780 pipe2_z_ref_o <= tex0_z_ref_o;
3781 pipe2_dst_rgb565_o <= tex0_dst_rgb565_o;
3782 pipe2_x_o <= tex0_x_o;
3783 pipe2_y_o <= tex0_y_o;
3784 pipe2_w_q_o <= tex0_w_q_o;
3785 pipe2_tex_addr_o <= tex0_tex_addr_o;
3786
3787 // pipe2 -> draw_pipe: texture ROM output is now aligned.
3788 draw_pipe_valid <= pipe2_valid;
3789 draw_pipe_inside <= pipe2_inside;
3790 draw_pipe_ztest <= pipe2_ztest;
3791 draw_pipe_textured <= pipe2_textured;
3792 draw_pipe_alpha_key <= pipe2_alpha_key;
3793 draw_pipe_alpha <= pipe2_alpha;
3794 draw_pipe_fog <= pipe2_fog;
3795 draw_pipe_light_bank <= pipe2_light_bank;
3796 draw_pipe_tex_or_color <= pipe2_tex_or_color;
3797 draw_pipe_addr <= pipe2_addr;
3798 draw_pipe_z <= pipe2_z;
3799 draw_pipe_z_ref <= pipe2_z_ref;
3800 draw_pipe_dst_rgb565 <= pipe2_dst_rgb565;
3801 draw_pipe_x <= pipe2_x;

```

```

3802     draw_pipe_y <= pipe2_y;
3803     draw_pipe_w_q <= pipe2_w_q;
3804     draw_pipe_valid_o <= pipe2_valid_o;
3805     draw_pipe_inside_o <= pipe2_inside_o;
3806     draw_pipe_ztest_o <= pipe2_ztest_o;
3807     draw_pipe_textured_o <= pipe2_textured_o;
3808     draw_pipe_alpha_key_o <= pipe2_alpha_key_o;
3809     draw_pipe_alpha_o <= pipe2_alpha_o;
3810     draw_pipe_fog_o <= pipe2_fog_o;
3811     draw_pipe_light_bank_o <= pipe2_light_bank_o;
3812     draw_pipe_tex_or_color_o <= pipe2_tex_or_color_o;
3813     draw_pipe_addr_o <= pipe2_addr_o;
3814     draw_pipe_z_o <= pipe2_z_o;
3815     draw_pipe_z_ref_o <= pipe2_z_ref_o;
3816     draw_pipe_dst_rgb565_o <= pipe2_dst_rgb565_o;
3817     draw_pipe_x_o <= pipe2_x_o;
3818     draw_pipe_y_o <= pipe2_y_o;
3819     draw_pipe_w_q_o <= pipe2_w_q_o;
3820
3821     if (state == ST_DRAW && draw_cache_hit) begin
3822         pipe0_valid <= 1'b1;
3823         pipe0_inside <= draw_inside_lane0;
3824         pipe0_ztest <= draw_flags[FLAG_ZTEST_BIT];
3825         pipe0_textured <= draw_flags[FLAG_TEX_BIT];
3826         pipe0_alpha_key <= draw_flags[FLAG_ALPHA_KEY_BIT];
3827         pipe0_alpha <= draw_flags[FLAG_ALPHA_MSB:FLAG_ALPHA_LSB];
3828         pipe0_fog <= draw_flags[FLAG_FOG_BIT];
3829         pipe0_light_bank <= draw_flags[FLAG_LIGHT_MSB:FLAG_LIGHT_LSB];
3830         pipe0_tex_or_color <= draw_tex_or_color;
3831         pipe0_addr <= draw_addr;
3832         pipe0_z <= draw_z_value;
3833         pipe0_x <= draw_x_cur;
3834         pipe0_y <= draw_y_cur;
3835         pipe0_uw_q <= draw_uw_q;
3836         pipe0_vw_q <= draw_vw_q;
3837         pipe0_iw_q <= draw_iw_q;
3838         pipe0_valid_o <= 1'b1;
3839         pipe0_inside_o <= draw_inside_lane1;
3840         pipe0_ztest_o <= draw_flags[FLAG_ZTEST_BIT];
3841         pipe0_textured_o <= draw_flags[FLAG_TEX_BIT];
3842         pipe0_alpha_key_o <= draw_flags[FLAG_ALPHA_KEY_BIT];
3843         pipe0_alpha_o <= draw_flags[FLAG_ALPHA_MSB:FLAG_ALPHA_LSB];
3844         pipe0_fog_o <= draw_flags[FLAG_FOG_BIT];
3845         pipe0_light_bank_o <= draw_flags[FLAG_LIGHT_MSB:FLAG_LIGHT_LSB];
3846         pipe0_tex_or_color_o <= draw_tex_or_color;
3847         pipe0_addr_o <= draw_addr_o;
3848         pipe0_z_o <= draw_z_value_o;
3849         pipe0_x_o <= draw_x_next;
3850         pipe0_y_o <= draw_y_cur;
3851         pipe0_uw_q_o <= draw_uw_q_o;
3852         pipe0_vw_q_o <= draw_vw_q_o;
3853         pipe0_iw_q_o <= draw_iw_q_o;
3854
3855         /* Track inside-to-outside transition for early row exit */
3856         if (draw_pair_edge_inside)
3857             draw_row_inside <= 1'b1;
3858
3859         /* Early scanline exit: if we were inside the quad on
3860          * this row and now we're outside, all remaining pixels

```

```

3861     * on this row are also outside (convex quad property).
3862     * Skip directly to the next row. */
3863     if (draw_pair_last || draw_pair_exited) begin
3864         if (draw_y_cur == draw_y_max) begin
3865             state <= ST_DRAW_FLUSH;
3866             draw_flush_count <= DRAW_FLUSH_CYCLES;
3867         end else begin
3868             draw_row_base <= draw_row_base + 16'd640;
3869             draw_x_cur <= draw_x_start_even;
3870             draw_y_cur <= draw_y_cur + 9'd1;
3871             draw_row_inside <= 1'b0;
3872             edge_row_val[0] <= edge_next_row0;
3873             edge_cur_val[0] <= edge_next_row0;
3874             edge_row_val[1] <= edge_next_row1;
3875             edge_cur_val[1] <= edge_next_row1;
3876             edge_row_val[2] <= edge_next_row2;
3877             edge_cur_val[2] <= edge_next_row2;
3878             edge_row_val[3] <= edge_next_row3;
3879             edge_cur_val[3] <= edge_next_row3;
3880             z_row_val <= z_next_row;
3881             z_cur_val <= z_next_row;
3882             uw_row_val <= uw_next_row;
3883             uw_cur_val <= uw_next_row;
3884             vw_row_val <= vw_next_row;
3885             vw_cur_val <= vw_next_row;
3886             iw_row_val <= iw_next_row;
3887             iw_cur_val <= iw_next_row;
3888         end
3889     end else begin
3890         draw_x_cur <= draw_x_cur + 10'd2;
3891         edge_cur_val[0] <= edge_next_pair0;
3892         edge_cur_val[1] <= edge_next_pair1;
3893         edge_cur_val[2] <= edge_next_pair2;
3894         edge_cur_val[3] <= edge_next_pair3;
3895         z_cur_val <= z_next_pair;
3896         uw_cur_val <= uw_next_pair;
3897         vw_cur_val <= vw_next_pair;
3898         iw_cur_val <= iw_next_pair;
3899     end
3900 end else if (state == ST_DRAW) begin
3901     pipe0_valid <= 1'b0;
3902     pipe0_valid_o <= 1'b0;
3903     /*
3904     * Software has already binned this descriptor into each
3905     * overlapping band pass. Pixels outside the resident
3906     * band are ignored here and will be drawn during their
3907     * own BEGIN_BAND/END_BAND pass.
3908     */
3909     /* Early scanline exit for cache-miss path too */
3910     if (draw_pair_edge_inside)
3911         draw_row_inside <= 1'b1;
3912
3913     if (draw_pair_last || draw_pair_exited) begin
3914         if (draw_y_cur == draw_y_max) begin
3915             state <= ST_DRAW_FLUSH;
3916             draw_flush_count <= DRAW_FLUSH_CYCLES;
3917         end else begin
3918             draw_row_base <= draw_row_base + 16'd640;
3919             draw_x_cur <= draw_x_start_even;

```

```

3920         draw_y_cur <= draw_y_cur + 9'd1;
3921         draw_row_inside <= 1'b0;
3922         edge_row_val[0] <= edge_next_row0;
3923         edge_cur_val[0] <= edge_next_row0;
3924         edge_row_val[1] <= edge_next_row1;
3925         edge_cur_val[1] <= edge_next_row1;
3926         edge_row_val[2] <= edge_next_row2;
3927         edge_cur_val[2] <= edge_next_row2;
3928         edge_row_val[3] <= edge_next_row3;
3929         edge_cur_val[3] <= edge_next_row3;
3930         z_row_val <= z_next_row;
3931         z_cur_val <= z_next_row;
3932         uw_row_val <= uw_next_row;
3933         uw_cur_val <= uw_next_row;
3934         vw_row_val <= vw_next_row;
3935         vw_cur_val <= vw_next_row;
3936         iw_row_val <= iw_next_row;
3937         iw_cur_val <= iw_next_row;
3938     end
3939 end else begin
3940     draw_x_cur <= draw_x_cur + 10'd2;
3941     edge_cur_val[0] <= edge_next_pair0;
3942     edge_cur_val[1] <= edge_next_pair1;
3943     edge_cur_val[2] <= edge_next_pair2;
3944     edge_cur_val[3] <= edge_next_pair3;
3945     z_cur_val <= z_next_pair;
3946     uw_cur_val <= uw_next_pair;
3947     vw_cur_val <= vw_next_pair;
3948     iw_cur_val <= iw_next_pair;
3949 end
3950 end else begin
3951     pipe0_valid <= 1'b0;
3952     pipe0_inside <= 1'b0;
3953     pipe0_ztest <= 1'b0;
3954     pipe0_textured <= 1'b0;
3955     pipe0_alpha_key <= 1'b0;
3956     pipe0_alpha <= 2'd0;
3957     pipe0_fog <= 1'b0;
3958     pipe0_light_bank <= 2'd0;
3959     pipe0_tex_or_color <= 8'd0;
3960     pipe0_addr <= 16'd0;
3961     pipe0_z <= 16'd0;
3962     pipe0_x <= 10'd0;
3963     pipe0_y <= 9'd0;
3964     pipe0_uw_q <= 32'sd0;
3965     pipe0_vw_q <= 32'sd0;
3966     pipe0_iw_q <= 32'd0;
3967     pipe0_valid_o <= 1'b0;
3968     pipe0_inside_o <= 1'b0;
3969     pipe0_ztest_o <= 1'b0;
3970     pipe0_textured_o <= 1'b0;
3971     pipe0_alpha_key_o <= 1'b0;
3972     pipe0_alpha_o <= 2'd0;
3973     pipe0_fog_o <= 1'b0;
3974     pipe0_light_bank_o <= 2'd0;
3975     pipe0_tex_or_color_o <= 8'd0;
3976     pipe0_addr_o <= 16'd0;
3977     pipe0_z_o <= 16'd0;
3978     pipe0_x_o <= 10'd0;

```

```

3979         pipe0_y_o <= 9'd0;
3980         pipe0_uw_q_o <= 32'sd0;
3981         pipe0_vw_q_o <= 32'sd0;
3982         pipe0_iw_q_o <= 32'd0;
3983
3984         if (draw_flush_count == 4'd1) begin
3985             draw_flush_count <= 4'd0;
3986             if (prefetch_valid) begin
3987                 for (ei = 0; ei < MAX_DESC_WORDS; ei = ei + 1)
3988                     desc_words[ei] <= prefetch_words[ei];
3989                 fetch_count <= prefetch_target_words;
3990                 prefetch_valid <= 1'b0;
3991                 state <= ST_FETCH;
3992             end else if (prefetch_active) begin
3993                 draw_flush_count <= 4'd1;
3994                 state <= ST_DRAW_FLUSH;
3995             end else begin
3996                 state <= ST_IDLE;
3997             end
3998         end else begin
3999             draw_flush_count <= draw_flush_count - 4'd1;
4000         end
4001     end
4002 end
4003
4004 ST_CACHE_INIT: begin
4005     if (cache_maint_addr == cache_window_end - 16'd2) begin
4006         cache_valid <= 1'b1;
4007         cache_dirty <= 1'b1;
4008         cache_draw_dirty <= 1'b0;
4009         cache_band_index <= cache_target_band;
4010         cache_maint_addr <= cache_window_start;
4011         state <= ST_IDLE;
4012     end else begin
4013         cache_maint_addr <= cache_maint_addr + 16'd2;
4014     end
4015 end
4016
4017     default: state <= ST_IDLE;
4018 endcase
4019
4020 /*
4021  * CLR is a frame-level abort/restart request from the driver. Do
4022  * not wait for ST_IDLE: cache maintenance can legitimately take a
4023  * long time at 640x480, and if a previous frame wedges we need the
4024  * next CLEAR_FRAME ioctl to recover the engine instead of timing
4025  * out behind stale BUSY.
4026  */
4027 if (ctrl_clear_write) begin
4028     state <= ST_IDLE;
4029     clear_pending <= 1'b1;
4030     ctrl_flp_pending <= 1'b0;
4031     copy_complete_pending <= 1'b0;
4032     copy_target_sel <= ~display_sel;
4033     cache_valid <= 1'b0;
4034     cache_dirty <= 1'b0;
4035     cache_draw_dirty <= 1'b0;
4036     cache_flush_y_min <= 6'd0;
4037     cache_window_start <= 16'd0;

```

```

4038     cache_window_pixels <= 16'd38400;
4039     cache_window_end <= 16'd38400;
4040     band_begin_pending <= 1'b0;
4041     band_flush_pending <= 1'b0;
4042     draw_cache_sel <= 1'b0;
4043     /* CLEAR_FRAME must not kill an in-flight background flush. */
4044     cache_maint_addr <= 16'd0;
4045     scan_late_count <= 16'd0;
4046     /* Leave scanout RD state alone; CLEAR_FRAME can arrive mid-line. */
4047     fifo_wr_ptr <= 10'd0;
4048     fifo_rd_ptr <= 10'd0;
4049     fifo_count <= 11'd0;
4050     fetch_count <= 6'd0;
4051     prefetch_count <= 6'd0;
4052     prefetch_target_words <= BASE_QUAD_WORDS_6;
4053     prefetch_active <= 1'b0;
4054     prefetch_valid <= 1'b0;
4055     draw_flush_count <= 4'd0;
4056     pipe0_valid <= 1'b0;
4057     recip0_valid <= 1'b0;
4058     recip1_valid <= 1'b0;
4059     recip2_valid <= 1'b0;
4060     pipe1_valid <= 1'b0;
4061     tex0_valid <= 1'b0;
4062     pipe2_valid <= 1'b0;
4063     draw_pipe_valid <= 1'b0;
4064     pal_rd_valid <= 1'b0;
4065     plr_valid <= 1'b0;
4066     fog0_valid <= 1'b0;
4067     fog1_valid <= 1'b0;
4068     commit_valid <= 1'b0;
4069     pipe0_valid_o <= 1'b0;
4070     recip0_valid_o <= 1'b0;
4071     recip1_valid_o <= 1'b0;
4072     recip2_valid_o <= 1'b0;
4073     pipe1_valid_o <= 1'b0;
4074     tex0_valid_o <= 1'b0;
4075     pipe2_valid_o <= 1'b0;
4076     draw_pipe_valid_o <= 1'b0;
4077     pal_rd_valid_o <= 1'b0;
4078     plr_valid_o <= 1'b0;
4079     fog0_valid_o <= 1'b0;
4080     fog1_valid_o <= 1'b0;
4081     commit_valid_o <= 1'b0;
4082 end
4083
4084     extmem_dma_status <= {
4085         scan_late_count,
4086         cache_band_index,
4087         display_sel,
4088         copy_target_sel,
4089         copy_complete_pending,
4090         cache_init_state,
4091         flush_active,
4092         scan_fill_active,
4093         display_valid,
4094         sdram_ready,
4095         scan_active_bank[0],
4096         scan_fill_bank[0],

```

```

4097         scan_line2_ready,
4098         scan_line1_ready,
4099         scan_line0_ready
4100     };
4101     end
4102 end
4103
4104 wire perf_flip_write = wr && (address == ADDR_CONTROL) && writedata[1];
4105 wire perf_in_draw    = (state == ST_DRAW) || (state == ST_DRAW_FLUSH);
4106 wire perf_draw_commit = commit_valid || commit_valid_o;
4107 wire perf_flush_push = bg_flush_wr_push;
4108
4109 /*
4110  * Per-frame perf counters accumulate while a frame is in flight and reset
4111  * on the FLIP write that ends the frame. Counter categories can overlap
4112  * because background flush, cache init, and draw can run concurrently.
4113  */
4114 voxel_perf_counters #(
4115     .COPY_WR_FIFO_HIGH_WATER(COPY_WR_FIFO_HIGH_WATER)
4116 ) perf_counters (
4117     .clk             (clk),
4118     .reset           (reset),
4119     .flip_write      (perf_flip_write),
4120     .in_draw         (perf_in_draw),
4121     .draw_commit     (perf_draw_commit),
4122     .in_cache_init   (cache_init_state),
4123     .flush_active    (flush_active),
4124     .flush_push      (perf_flush_push),
4125     .flush_load_pending (flush_load_pending),
4126     .sdram_wr_full   (sdram_wr_full),
4127     .sdram_wr_use    (sdram_wr_use),
4128     .flush_words_done (flush_words_done),
4129     .flush_pixels_total (flush_pixels_total),
4130     .flush_word_pending_valid (flush_word_pending_valid),
4131     .flush_fetch_inflight (flush_fetch_inflight),
4132     .perf_draw_active  (perf_draw_active),
4133     .perf_draw_idle    (perf_draw_idle),
4134     .perf_flush_active (perf_flush_active),
4135     .perf_flush_stall  (perf_flush_stall),
4136     .perf_init         (perf_init),
4137     .perf_load         (perf_load),
4138     .perf_flush_wait_load (perf_flush_wait_load),
4139     .perf_flush_wait_fifo (perf_flush_wait_fifo),
4140     .perf_flush_wait_data (perf_flush_wait_data),
4141     .perf_flush_wait_drain (perf_flush_wait_drain)
4142 );
4143
4144 always_comb begin
4145     case (address)
4146         ADDR_CONTROL : readdata = control_word;
4147         ADDR_STATUS  : readdata = status_word;
4148         ADDR_FRAMECNT: readdata = frame_count;
4149         ADDR_PAL_ADDR: readdata = {24'h0, pal_addr};
4150         ADDR_SKY_PAL_ADDR: readdata = {27'h0, sky_pal_addr};
4151         ADDR_FOG_RANGE: readdata = {fog_end_dist, fog_start_dist};
4152         ADDR_FOG_CTRL: readdata = {fog_inv_proj_sq, 7'h0, fog_enable, fog_color};
4153         ADDR_EXTMEM_CTRL: readdata = extmem_ctrl;
4154         ADDR_EXTMEM_FRONT: readdata = extmem_front_base;
4155         ADDR_EXTMEM_BACK: readdata = extmem_back_base;

```

```

4156     ADDR_EXTMEM_STRIDE: readdata = extmem_stride_bytes;
4157     ADDR_EXTMEM_TILE: readdata = extmem_tile_cfg;
4158     ADDR_EXTMEM_STAT: readdata = extmem_dma_status;
4159     ADDR_BAND_INDEX: readdata = {29'h0, band_index_cfg};
4160     ADDR_BAND_CTRL: readdata = {30'h0, band_flush_pending, band_begin_pending};
4161     ADDR_BAND_WINDOW: readdata = {18'd0, band_flush_y_max_cfg,
4162                                   2'd0, band_flush_y_min_cfg};
4163     ADDR_PERF_DRAW_ACT : readdata = perf_draw_active;
4164     ADDR_PERF_DRAW_IDLE: readdata = perf_draw_idle;
4165     ADDR_PERF_FLUSH_ACT: readdata = perf_flush_active;
4166     ADDR_PERF_FLUSH_STL: readdata = perf_flush_stall;
4167     ADDR_PERF_INIT      : readdata = perf_init;
4168     ADDR_PERF_LOAD      : readdata = perf_load;
4169     ADDR_PERF_FLUSH_LOAD : readdata = perf_flush_wait_load;
4170     ADDR_PERF_FLUSH_FIFO : readdata = perf_flush_wait_fifo;
4171     ADDR_PERF_FLUSH_DATA : readdata = perf_flush_wait_data;
4172     ADDR_PERF_FLUSH_DRAIN: readdata = perf_flush_wait_drain;
4173     default              : readdata = 32'h0; /* palette readback not needed by driver */
4174   endcase
4175 end
4176
4177 always_comb begin
4178   if (scan_visible_r) begin
4179     VGA_R = pixel_rgb[23:16];
4180     VGA_G = pixel_rgb[15:8];
4181     VGA_B = pixel_rgb[7:0];
4182   end else begin
4183     VGA_R = 8'h00;
4184     VGA_G = 8'h00;
4185     VGA_B = 8'h00;
4186   end
4187 end
4188
4189 endmodule

```

Complete voxel_math_utils.svRTLlisting

Listing 4: Complete voxel_math_utils.svRTLsource.

```

1 // Pure combinational math helpers used by voxel_gpu.sv.
2 //
3 // Keep tiny stateless utility modules together here so the RTL directory
4 // reflects real ownership boundaries without scattering one-screen files.
5
6 module voxel_raster_setup (
7   input logic      [9:0] draw_x_start_even,
8   input logic      [8:0] draw_y_min,
9   input logic      [9:0] draw_x_min,
10  input logic       [15:0] draw_z0,
11  input logic signed [15:0] draw_dz_dx,
12  input logic signed [31:0] draw_uw_0,
13  input logic signed [31:0] draw_uw_dx,
14  input logic signed [31:0] draw_vw_0,
15  input logic signed [31:0] draw_vw_dx,
16  input logic signed [31:0] draw_iw_0,
17  input logic signed [31:0] draw_iw_dx,
18  input logic signed [31:0] edge_a0,

```

```

19   input logic signed [31:0] edge_a1,
20   input logic signed [31:0] edge_a2,
21   input logic signed [31:0] edge_a3,
22   input logic signed [31:0] edge_b0,
23   input logic signed [31:0] edge_b1,
24   input logic signed [31:0] edge_b2,
25   input logic signed [31:0] edge_b3,
26   input logic signed [31:0] edge_c0,
27   input logic signed [31:0] edge_c1,
28   input logic signed [31:0] edge_c2,
29   input logic signed [31:0] edge_c3,
30   output logic signed [63:0] edge_eval0,
31   output logic signed [63:0] edge_eval1,
32   output logic signed [63:0] edge_eval2,
33   output logic signed [63:0] edge_eval3,
34   output logic signed [47:0] z_start_val,
35   output logic signed [63:0] uw_start_val,
36   output logic signed [63:0] vw_start_val,
37   output logic signed [63:0] iw_start_val
38 );
39   function automatic signed [63:0] sext32_to_s64(input logic signed [31:0] value);
40       begin
41           sext32_to_s64 = {{32{value[31]}}, value};
42       end
43   endfunction
44
45   function automatic signed [47:0] sext16_to_s48(input logic signed [15:0] value);
46       begin
47           sext16_to_s48 = {{32{value[15]}}, value};
48       end
49   endfunction
50
51   wire signed [10:0] setup_start_x = $signed({1'b0, draw_x_start_even});
52   wire signed [9:0] setup_start_y = $signed({1'b0, draw_y_min});
53
54   wire signed [63:0] edge_ax0 = $signed(edge_a0) * setup_start_x;
55   wire signed [63:0] edge_ax1 = $signed(edge_a1) * setup_start_x;
56   wire signed [63:0] edge_ax2 = $signed(edge_a2) * setup_start_x;
57   wire signed [63:0] edge_ax3 = $signed(edge_a3) * setup_start_x;
58   wire signed [63:0] edge_by0 = $signed(edge_b0) * setup_start_y;
59   wire signed [63:0] edge_by1 = $signed(edge_b1) * setup_start_y;
60   wire signed [63:0] edge_by2 = $signed(edge_b2) * setup_start_y;
61   wire signed [63:0] edge_by3 = $signed(edge_b3) * setup_start_y;
62
63   wire signed [47:0] draw_dz_dx_ext = sext16_to_s48(draw_dz_dx);
64   wire signed [63:0] draw_uw_dx_ext = sext32_to_s64(draw_uw_dx);
65   wire signed [63:0] draw_vw_dx_ext = sext32_to_s64(draw_vw_dx);
66   wire signed [63:0] draw_iw_dx_ext = sext32_to_s64(draw_iw_dx);
67
68   always_comb begin
69       edge_eval0 = edge_ax0 + edge_by0 + sext32_to_s64(edge_c0);
70       edge_eval1 = edge_ax1 + edge_by1 + sext32_to_s64(edge_c1);
71       edge_eval2 = edge_ax2 + edge_by2 + sext32_to_s64(edge_c2);
72       edge_eval3 = edge_ax3 + edge_by3 + sext32_to_s64(edge_c3);
73
74       z_start_val =
75           $signed({32'd0, draw_z0}) -
76           (draw_x_min[0] ? draw_dz_dx_ext : 48'sd0);
77       uw_start_val =

```

```

78         sext32_to_s64(draw_uw_0) -
79         (draw_x_min[0] ? draw_uw_dx_ext : 64'sd0);
80     vw_start_val =
81         sext32_to_s64(draw_vw_0) -
82         (draw_x_min[0] ? draw_vw_dx_ext : 64'sd0);
83     iw_start_val =
84         sext32_to_s64(draw_iw_0) -
85         (draw_x_min[0] ? draw_iw_dx_ext : 64'sd0);
86     end
87 endmodule
88
89 module voxel_draw_step (
90     input  logic signed [63:0] edge_cur0,
91     input  logic signed [63:0] edge_cur1,
92     input  logic signed [63:0] edge_cur2,
93     input  logic signed [63:0] edge_cur3,
94     input  logic signed [63:0] edge_row0,
95     input  logic signed [63:0] edge_row1,
96     input  logic signed [63:0] edge_row2,
97     input  logic signed [63:0] edge_row3,
98     input  logic signed [31:0] edge_a0,
99     input  logic signed [31:0] edge_a1,
100    input  logic signed [31:0] edge_a2,
101    input  logic signed [31:0] edge_a3,
102    input  logic signed [31:0] edge_b0,
103    input  logic signed [31:0] edge_b1,
104    input  logic signed [31:0] edge_b2,
105    input  logic signed [31:0] edge_b3,
106    input  logic signed [47:0] z_cur,
107    input  logic signed [47:0] z_row,
108    input  logic signed [15:0] dz_dx,
109    input  logic signed [15:0] dz_dy,
110    input  logic signed [63:0] uw_cur,
111    input  logic signed [63:0] uw_row,
112    input  logic signed [31:0] uw_dx,
113    input  logic signed [31:0] uw_dy,
114    input  logic signed [63:0] vw_cur,
115    input  logic signed [63:0] vw_row,
116    input  logic signed [31:0] vw_dx,
117    input  logic signed [31:0] vw_dy,
118    input  logic signed [63:0] iw_cur,
119    input  logic signed [63:0] iw_row,
120    input  logic signed [31:0] iw_dx,
121    input  logic signed [31:0] iw_dy,
122    output logic signed [63:0] edge_lane1_0,
123    output logic signed [63:0] edge_lane1_1,
124    output logic signed [63:0] edge_lane1_2,
125    output logic signed [63:0] edge_lane1_3,
126    output logic signed [63:0] edge_next_pair0,
127    output logic signed [63:0] edge_next_pair1,
128    output logic signed [63:0] edge_next_pair2,
129    output logic signed [63:0] edge_next_pair3,
130    output logic signed [63:0] edge_next_row0,
131    output logic signed [63:0] edge_next_row1,
132    output logic signed [63:0] edge_next_row2,
133    output logic signed [63:0] edge_next_row3,
134    output logic signed [47:0] z_lane1,
135    output logic signed [47:0] z_next_pair,
136    output logic signed [47:0] z_next_row,

```

```

137 output logic signed [63:0] uw_lane1,
138 output logic signed [63:0] uw_next_pair,
139 output logic signed [63:0] uw_next_row,
140 output logic signed [63:0] vw_lane1,
141 output logic signed [63:0] vw_next_pair,
142 output logic signed [63:0] vw_next_row,
143 output logic signed [63:0] iw_lane1,
144 output logic signed [63:0] iw_next_pair,
145 output logic signed [63:0] iw_next_row
146 );
147 function automatic signed [63:0] sext32_to_s64(input logic signed [31:0] value);
148     begin
149         sext32_to_s64 = {{32{value[31]}}, value};
150     end
151 endfunction
152
153 function automatic signed [47:0] sext16_to_s48(input logic signed [15:0] value);
154     begin
155         sext16_to_s48 = {{32{value[15]}}, value};
156     end
157 endfunction
158
159 wire signed [63:0] edge_a0_ext = sext32_to_s64(edge_a0);
160 wire signed [63:0] edge_a1_ext = sext32_to_s64(edge_a1);
161 wire signed [63:0] edge_a2_ext = sext32_to_s64(edge_a2);
162 wire signed [63:0] edge_a3_ext = sext32_to_s64(edge_a3);
163 wire signed [63:0] edge_b0_ext = sext32_to_s64(edge_b0);
164 wire signed [63:0] edge_b1_ext = sext32_to_s64(edge_b1);
165 wire signed [63:0] edge_b2_ext = sext32_to_s64(edge_b2);
166 wire signed [63:0] edge_b3_ext = sext32_to_s64(edge_b3);
167 wire signed [47:0] dz_dx_ext = sext16_to_s48(dz_dx);
168 wire signed [47:0] dz_dy_ext = sext16_to_s48(dz_dy);
169 wire signed [63:0] uw_dx_ext = sext32_to_s64(uw_dx);
170 wire signed [63:0] uw_dy_ext = sext32_to_s64(uw_dy);
171 wire signed [63:0] vw_dx_ext = sext32_to_s64(vw_dx);
172 wire signed [63:0] vw_dy_ext = sext32_to_s64(vw_dy);
173 wire signed [63:0] iw_dx_ext = sext32_to_s64(iw_dx);
174 wire signed [63:0] iw_dy_ext = sext32_to_s64(iw_dy);
175
176 always_comb begin
177     edge_lane1_0 = edge_cur0 + edge_a0_ext;
178     edge_lane1_1 = edge_cur1 + edge_a1_ext;
179     edge_lane1_2 = edge_cur2 + edge_a2_ext;
180     edge_lane1_3 = edge_cur3 + edge_a3_ext;
181
182     edge_next_pair0 = edge_cur0 + edge_a0_ext + edge_a0_ext;
183     edge_next_pair1 = edge_cur1 + edge_a1_ext + edge_a1_ext;
184     edge_next_pair2 = edge_cur2 + edge_a2_ext + edge_a2_ext;
185     edge_next_pair3 = edge_cur3 + edge_a3_ext + edge_a3_ext;
186
187     edge_next_row0 = edge_row0 + edge_b0_ext;
188     edge_next_row1 = edge_row1 + edge_b1_ext;
189     edge_next_row2 = edge_row2 + edge_b2_ext;
190     edge_next_row3 = edge_row3 + edge_b3_ext;
191
192     z_lane1 = z_cur + dz_dx_ext;
193     z_next_pair = z_cur + dz_dx_ext + dz_dx_ext;
194     z_next_row = z_row + dz_dy_ext;
195

```

```

196     uw_lane1 = uw_cur + uw_dx_ext;
197     uw_next_pair = uw_cur + uw_dx_ext + uw_dx_ext;
198     uw_next_row = uw_row + uw_dy_ext;
199
200     vw_lane1 = vw_cur + vw_dx_ext;
201     vw_next_pair = vw_cur + vw_dx_ext + vw_dx_ext;
202     vw_next_row = vw_row + vw_dy_ext;
203
204     iw_lane1 = iw_cur + iw_dx_ext;
205     iw_next_pair = iw_cur + iw_dx_ext + iw_dx_ext;
206     iw_next_row = iw_row + iw_dy_ext;
207     end
208 endmodule
209
210 module voxel_iw_normalize (
211     input  logic [31:0] iw_q,
212     output logic [5:0]  iw_msb,
213     output logic [31:0] iw_norm_q
214 );
215     function automatic [5:0] msb_index32(input logic [31:0] value);
216         integer bit_idx;
217         logic found;
218         begin
219             msb_index32 = 6'd0;
220             found = 1'b0;
221             for (bit_idx = 31; bit_idx >= 0; bit_idx = bit_idx - 1) begin
222                 if (!found && value[bit_idx]) begin
223                     msb_index32 = bit_idx[5:0];
224                     found = 1'b1;
225                 end
226             end
227         end
228     endfunction
229
230     wire [5:0] iw_msb_calc = msb_index32(iw_q);
231
232     always_comb begin
233         iw_msb = iw_msb_calc;
234         iw_norm_q =
235             (iw_q == 32'd0) ? 32'd0 :
236             (iw_msb_calc >= 6'd16) ?
237             (iw_q >> (iw_msb_calc - 6'd16)) :
238             (iw_q << (6'd16 - iw_msb_calc));
239     end
240 endmodule
241
242 module voxel_recip_interpolate (
243     input  logic [31:0] w_norm_lo,
244     input  logic [31:0] w_norm_hi,
245     input  logic [5:0]  iw_lut_frac,
246     output logic [31:0] w_norm_q
247 );
248     wire [31:0] w_norm_delta = w_norm_lo - w_norm_hi;
249     wire [37:0] w_interp_prod = w_norm_delta * iw_lut_frac;
250     wire [37:0] w_interp_step_ext = (w_interp_prod + 38'd32) >> 6;
251     wire [31:0] w_interp_step = w_interp_step_ext[31:0];
252
253     assign w_norm_q = w_norm_lo - w_interp_step;
254 endmodule

```

```

255
256 module voxel_w_denormalize (
257     input logic      iw_zero,
258     input logic [5:0] iw_msb,
259     input logic [31:0] w_norm_q,
260     output logic [31:0] w_q
261 );
262     always_comb begin
263         w_q =
264             iw_zero ? 32'd0 :
265             (iw_msb >= 6'd16) ?
266             (w_norm_q >> (iw_msb - 6'd16)) :
267             (w_norm_q << (6'd16 - iw_msb));
268     end
269 endmodule
270
271 module voxel_fog_blend (
272     input logic      fog_enable,
273     input logic      pixel_fog,
274     input logic [15:0] radial_q8_8,
275     input logic [15:0] fog_start_dist,
276     input logic [15:0] fog_end_dist,
277     input logic [15:0] src_rgb565,
278     input logic [15:0] dst_rgb565,
279     input logic [15:0] fog_rgb565,
280     input logic [1:0] alpha,
281     output logic [15:0] out_rgb565
282 );
283 'include "voxel_color_helpers.svh"
284
285     wire fog_active =
286         fog_enable && pixel_fog && (fog_end_dist > fog_start_dist) &&
287         (radial_q8_8 > fog_start_dist);
288     wire fog_full = fog_active && (radial_q8_8 >= fog_end_dist);
289     wire [15:0] fog_range = fog_end_dist - fog_start_dist;
290     wire [15:0] fog_q1 = fog_start_dist + {2'b00, fog_range[15:2]};
291     wire [15:0] fog_q2 = fog_start_dist + {1'b0, fog_range[15:1]};
292     wire [1:0] fog_alpha =
293         !fog_active ? 2'd0 :
294         fog_full    ? 2'd0 :
295         (radial_q8_8 < fog_q1) ? 2'd1 :
296         (radial_q8_8 < fog_q2) ? 2'd2 : 2'd3;
297     wire [15:0] fog_blended = blend_rgb565(src_rgb565, fog_rgb565, fog_alpha);
298     wire [15:0] fogged_rgb565 = fog_full ? fog_rgb565 : fog_blended;
299
300     assign out_rgb565 = blend_rgb565(fogged_rgb565, dst_rgb565, alpha);
301 endmodule

```

Complete voxel_raster_helpers.svRTLlisting

Listing 5: Complete voxel_raster_helpers.svRTLsource.

```

1     function automatic [9:0] clamp_x(input logic signed [15:0] value);
2         begin
3             if (value < 0)
4                 clamp_x = 10'd0;
5             else if (value > 16'sd639)

```

```

6         clamp_x = 10'd639;
7     else
8         clamp_x = value[9:0];
9     end
10    endfunction
11
12    function automatic [8:0] clamp_y(input logic signed [15:0] value);
13    begin
14        if (value < 0)
15            clamp_y = 9'd0;
16        else if (value > 16'sd479)
17            clamp_y = 9'd479;
18        else
19            clamp_y = value[8:0];
20        end
21    endfunction
22
23    function automatic [15:0] band_local_addr(input logic [9:0] x,
24                                              input logic [8:0] y,
25                                              input logic [2:0] band);
26    logic [8:0] band_base_y;
27    logic [8:0] local_y;
28    begin
29        band_base_y = band_base_row(band);
30        local_y = (y >= band_base_y) ? (y - band_base_y) : 9'd0;
31        band_local_addr = local_y * 16'd640 + {6'd0, x};
32    end
33    endfunction
34
35    function automatic [15:0] band_pixel_count(input logic [2:0] band);
36    logic [9:0] base_row;
37    logic [9:0] band_end;
38    logic [15:0] visible_rows;
39    begin
40        base_row = {1'b0, band_base_row(band)};
41        band_end = base_row + 10'd60;
42        if (base_row >= 10'd480)
43            band_pixel_count = 16'd0;
44        else if (band_end > 10'd480) begin
45            visible_rows = {6'd0, (10'd480 - base_row)};
46            band_pixel_count = visible_rows * 16'd640;
47        end else begin
48            band_pixel_count = 16'd38400;
49        end
50    end
51    endfunction
52
53    function automatic [15:0] band_local_row_offset(input logic [5:0] local_y);
54    begin
55        band_local_row_offset = {local_y, 9'd0} + {3'd0, local_y, 7'd0};
56    end
57    endfunction
58
59    function automatic [15:0] band_row_window_count(input logic [5:0] y_min,
60                                                  input logic [5:0] y_max);
61    logic [6:0] rows;
62    begin
63        rows = {1'b0, y_max} - {1'b0, y_min} + 7'd1;
64        band_row_window_count = {rows, 9'd0} + {2'd0, rows, 7'd0};

```

```

65     end
66 endfunction
67
68 function automatic [24:0] band_word_count(input logic [2:0] band);
69     begin
70         band_word_count = {9'd0, band_pixel_count(band)};
71     end
72 endfunction
73
74 function automatic [24:0] band_word_offset(input logic [2:0] band);
75     begin
76         band_word_offset = {7'd0, band, 15'd0} +
77                             {10'd0, band, 12'd0} +
78                             {12'd0, band, 10'd0} +
79                             {13'd0, band, 9'd0};
80     end
81 endfunction
82
83 function automatic [8:0] band_base_row(input logic [2:0] band);
84     begin
85         band_base_row = {band, 6'd0} - {4'd0, band, 2'd0};
86     end
87 endfunction
88
89 function automatic [4:0] sky_clear_start_index(input logic [2:0] band);
90     begin
91         case (band)
92             3'd0: sky_clear_start_index = 5'd0;
93             3'd1: sky_clear_start_index = 5'd3;
94             3'd2: sky_clear_start_index = 5'd6;
95             3'd3: sky_clear_start_index = 5'd9;
96             3'd4: sky_clear_start_index = 5'd12;
97             3'd5: sky_clear_start_index = 5'd15;
98             3'd6: sky_clear_start_index = 5'd18;
99             default: sky_clear_start_index = 5'd21;
100        endcase
101    end
102 endfunction
103
104 function automatic [4:0] sky_row_count_for_local_y(input logic [5:0] local_y);
105     logic [4:0] local_mod;
106     begin
107         local_mod = local_y[4:0];
108         if (local_y >= 6'd40)
109             sky_row_count_for_local_y = local_mod - 5'd8;
110         else if (local_y >= 6'd32)
111             sky_row_count_for_local_y = local_mod + 5'd12;
112         else if (local_y >= 6'd20)
113             sky_row_count_for_local_y = local_mod - 5'd20;
114         else
115             sky_row_count_for_local_y = local_mod;
116     end
117 endfunction
118
119 function automatic [4:0] sky_clear_index_for_local_y(input logic [2:0] band,
120                                                     input logic [5:0] local_y);
121     logic [5:0] pal;
122     begin
123         pal = {1'b0, sky_clear_start_index(band)};

```

```

124         if (local_y >= 6'd40)
125             pal = pal + 6'd2;
126         else if (local_y >= 6'd20)
127             pal = pal + 6'd1;
128         sky_clear_index_for_local_y =
129             (pal >= SKY_GRADIENT_COLORS_6) ?
130             SKY_GRADIENT_LAST_INDEX : pal[4:0];
131     end
132 endfunction
133
134 function automatic [4:0] sky_palette_for_y(input logic [8:0] y);
135     logic [2:0] band;
136     logic [8:0] local_y_wide;
137     logic [5:0] local_y;
138     logic [6:0] row_sum;
139     logic [1:0] pal_offset;
140     logic [5:0] pal;
141     begin
142         band = y_to_band(y);
143         local_y_wide = y - band_base_row(band);
144         local_y = local_y_wide[5:0];
145         row_sum = {1'b0, local_y};
146         if (row_sum >= 7'd60)
147             pal_offset = 2'd3;
148         else if (row_sum >= 7'd40)
149             pal_offset = 2'd2;
150         else if (row_sum >= 7'd20)
151             pal_offset = 2'd1;
152         else
153             pal_offset = 2'd0;
154
155         pal = {1'b0, sky_clear_start_index(band)} +
156             {4'b0, pal_offset};
157         sky_palette_for_y = (pal >= SKY_GRADIENT_COLORS_6) ?
158             SKY_GRADIENT_LAST_INDEX : pal[4:0];
159     end
160 endfunction
161
162 function automatic [2:0] y_to_band(input logic [8:0] y);
163     logic [9:0] y_adjusted;
164     begin
165         y_adjusted = {1'b0, y} + {5'd0, y[8:5], 1'b0} + 10'd2;
166         y_to_band = y_adjusted[8:6];
167     end
168 endfunction
169
170 function automatic [15:0] clamp_z(input logic signed [47:0] value);
171     begin
172         if (value < 0)
173             clamp_z = 16'h0000;
174         else if (value > 48'sd65534)
175             clamp_z = Z_VALID_FAR;
176         else
177             clamp_z = value[15:0];
178     end
179 endfunction
180
181 function automatic signed [31:0] clamp_s32(input logic signed [63:0] value);
182     begin

```

```

183         if (value > 64'sh7FFF_FFFF)
184             clamp_s32 = 32'sh7FFF_FFFF;
185         else if (value < -64'sh8000_0000)
186             clamp_s32 = -32'sh8000_0000;
187         else
188             clamp_s32 = value[31:0];
189         end
190     endfunction
191
192     function automatic [31:0] clamp_pos_u32(input logic signed [63:0] value);
193     begin
194         if (value <= 0)
195             clamp_pos_u32 = 32'd0;
196         else if (value > 64'sh7FFF_FFFF)
197             clamp_pos_u32 = 32'h7FFF_FFFF;
198         else
199             clamp_pos_u32 = value[31:0];
200         end
201     endfunction
202
203     function automatic [3:0] texture_coord(input logic signed [63:0] value,
204                                           input logic repeat_uv);
205     begin
206         if (repeat_uv)
207             texture_coord = value[35:32];
208         else if (value <= 64'sd0)
209             texture_coord = 4'd0;
210         else if (value >= 64'sh0000_0010_0000_0000)
211             texture_coord = 4'd15;
212         else
213             texture_coord = value[35:32];
214         end
215     endfunction

```