3D Hardware Rasterizer Design Document

Sathvik Rajampalli (vr2618), Shlok Desai (sbd2150), Aarush Agarwal (aa5763),

Gianna Belmont (gb2973), Srika Chagarlamudi (sc5800)

Spring 2026

1. Introduction **2**
2. System Block Diagram **4**
3. Algorithms **7**
3.1 Software 7
3.1.1 Overview 7
3.1.2 Pipeline Flow 7
3.1.3 Data Structures 8
3.1.4 Fixed-Point Format 10
3.1.5 Matrix Transforms 10
3.1.10 Vertex Projection 11
3.1.7 Per-Face Lighting 11
3.1.8 Triangle Setup - Pineda Edge Equations 12
3.1.9 Complete Function List 13
3.1.10 Test Vector File Format 13
3.2 Hardware Rasterization 14
3.2.1 Edge-function Rasterization 14
3.2.2 Triangle Setup 14
3.2.3 Pixel Unit Inner Loop 15
3.2.4 Y-partition Clipping and skip 17
3.2.5 Depth Test and Z-buffer 17
4. Resource Budgets **18**
5. Hardware/Software Interface **21**
5.1 Communication Pattern: 22
5.2 Data flow 22
5.3 Kernel Driver Design 22
5.4 Handshake Protocol
A. Appendix A 24

1. Introduction

Real-time 3D graphics rendering is one of the most computation-intensive workloads in modern computing, requiring the rapid transformation, projection, and rasterization of thousands of triangles per frame. This project addresses that challenge by designing and implementing a hardware-accelerated 3D triangle rasterizer on the DE1-SoC FPGA platform, capable of rendering rotating 3D geometry on a VGA monitor at interactive frame rates. The goal is to demonstrate that a carefully designed hardware pipeline, built around a well-chosen algorithm, can achieve real-time rendering performance within the tight resource constraints of an embedded FPGA system.

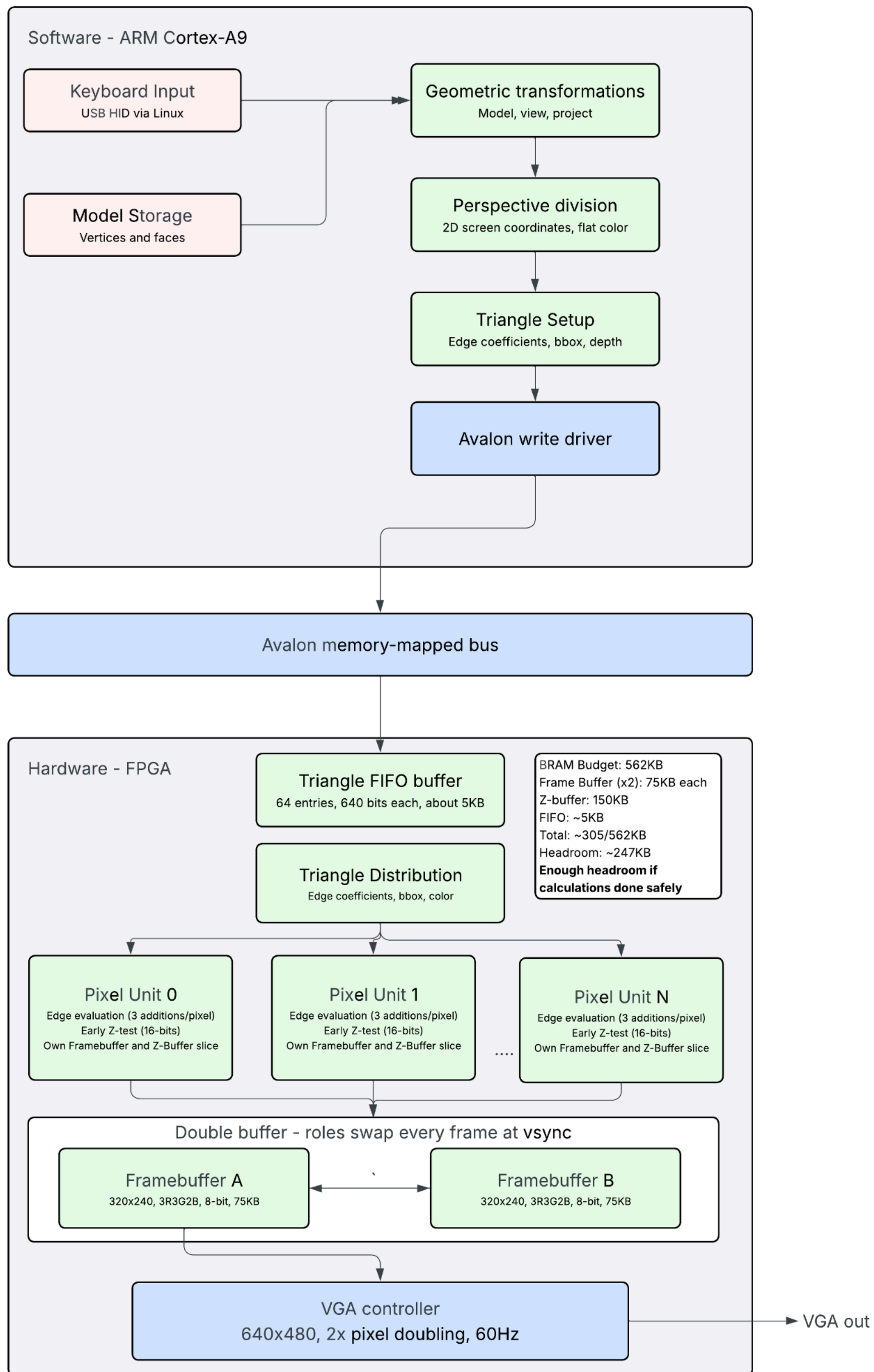
The system is partitioned between two tasks. Computation that occurs once per triangle belongs in software, while computation that occurs once per pixel belongs in dedicated hardware. The ARM processor handles user input and performs all 3D mathematics such as applying rotation and projection matrices to transform vertex coordinates from 3D world space into 2D screen coordinates. These projected triangle vertices are then passed to the FPGA over the Avalon memory-mapped bus. The hardware rasterizer takes over from there, determining which pixels on screen fall inside each triangle, computing their depth and color values, and writing the results into a framebuffer that drives the VGA display.

The core rasterization algorithm is the Pineda edge equation method, introduced in the SIGGRAPH paper "A Parallel Algorithm for Polygon Rasterization." For each triangle, a linear edge function is defined for each of its three edges. A pixel is interior to the triangle if and only if it lies on the positive side of all three edge functions simultaneously. Since these functions are linear, their values change by a constant amount with each unit step in the x or y direction. This means that after an initial setup, every pixel in the triangle's bounding box can be tested using only three additions and three sign comparisons. This property makes the algorithm exceptionally well-suited to hardware implementation, where adders are cheap and dividers are expensive.

This approach represents a significant architectural improvement over prior scanline-based implementations. A previous implementation of a similar project relied on a scanline rasterizer requiring four dedicated hardware divider modules and five chained pipeline stages, with the pipeline stalling at each division operation. The Pineda method eliminates per-pixel division entirely, replacing that entire rasterization pipeline with a single state machine whose inner loop consists of adders and comparators. The rendering performance of the resulting design is

memory-bandwidth limited rather than computation limited, which is the ideal operating regime for a hardware rasterizer.

2. System Block Diagram



Top Level System Diagram

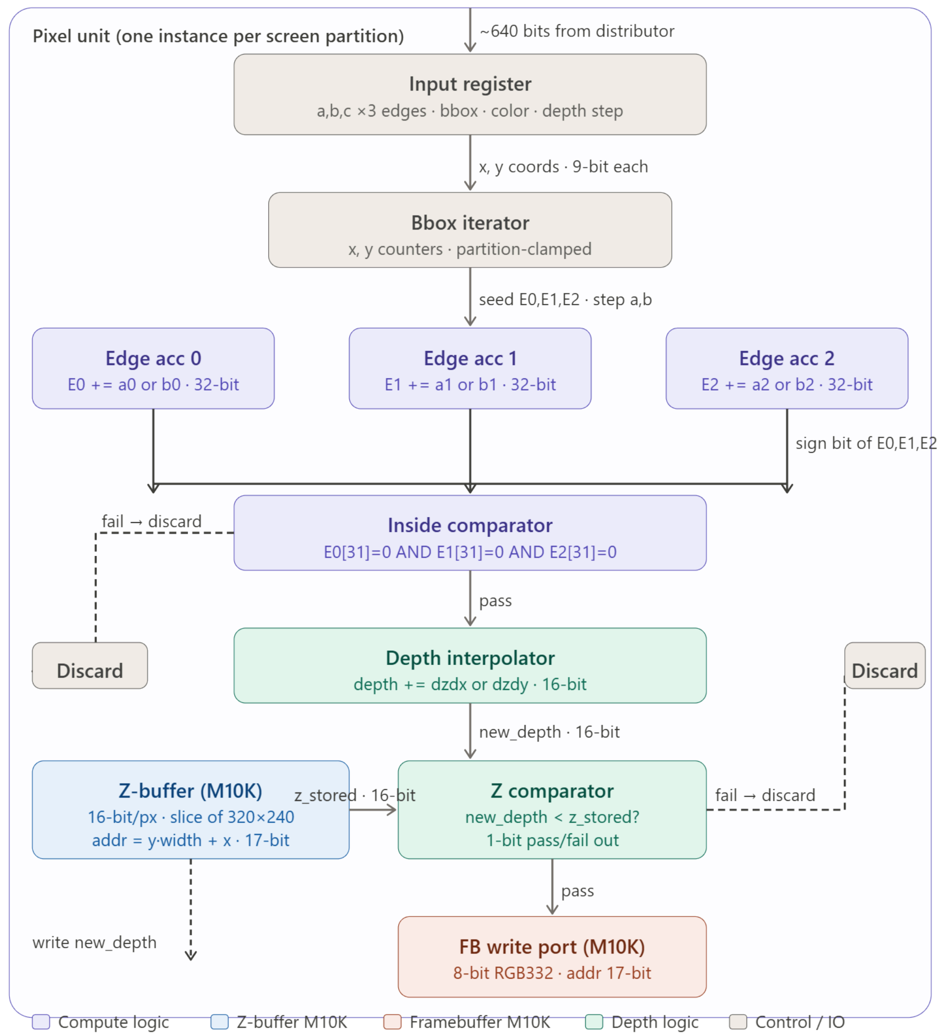
The top level system diagram shows the system divided into a software component running on the ARM Cortex-A9 and a hardware component implemented on the FPGA, communicating over the Avalon memory-mapped bus.

On the software side, keyboard input is read via the Linux USB HID driver and used to update rotation angles each frame. The 3D model is stored in memory as a list of vertices and faces. Each frame, the ARM processor applies model, view, and projection transforms to every vertex, producing 2D screen coordinates and a depth value per vertex. Perspective division is performed in software to complete the projection. For each triangle, software computes a flat shade color and runs triangle setting, computing the 3 edge function coefficients per edge, the bounding box, initial edge function values at the bounding box corner, and depth step values. This ensures hardware will only perform per-pixel work.

The Avalon write driver submits triangle packets by writing 20 consecutive 32-bit words to the FIFO write port, then strobing the commit register. It operates interrupt-driven.

The triangle FIFO buffers up to 64 entries in M10K BRAM. The triangle distributor pops one entry at a time and sends it to all pixel units simultaneously. Each pixel unit owns a fixed horizontal partition of the screen with its own private M10K slices for its framebuffer and Z-buffer, eliminating memory contention between units.

The two frame buffers are 320x240, 3R3G2B, and 75KB each. One is always read by the VGA and the other is written by the pixel units. On frame completion, their roles swap. The VGA controller reads the front buffer and generates a 640x480 signal by doubling each pixel in both dimensions at 60Hz.



Pixel Unit Internal Diagram

Each pixel unit processes its assigned screen partition independently. All units receive the same triangle information simultaneously and operate in parallel with no shared memory.

After receiving the triangle, the unit seeds 3 32-bit signed edge accumulators with the precomputed initial values. The bbox iterator then steps through pixels in the partition, adding the coefficient per x-step and reloading from a row register that increments by b per row. This gives exactly 3 additions per pixel with no multiplications.

The inside comparator checks the sign bit of each accumulator. If all three are non-negative the pixel is inside the triangle; otherwise it is discarded immediately.

Pixels that pass proceed to depth testing. A 16-bit depth accumulator tracks interpolated depth. The Z comparator reads the stored depth from the local Z-buffer and discards the pixel if it is not close. Pixels that pass both of these tests write their depth back to the Z buffer and write the color to the framebuffer at the same address. Both memories are private to the unit.

3. Algorithms

3.1 Software

3.1.1 Overview

The ARM Cortex-A9 processor on the DE1-SoC runs a C program that performs all per-frame and per-triangle computation. Its job is to take a 3D model stored as vertices and triangle faces, transform them into 2D screen coordinates, compute lighting and color for each face, set up the Pineda edge equation parameters for each triangle, and submit the resulting triangle packets to the FPGA hardware over the Avalon memory-mapped bus.

All floating-point computation happens in software. The hardware receives only fixed-point integer values and performs only addition and comparison in the pixel-processing loop. This division of labor puts expensive math (matrix multiplies, trigonometric functions, division) on the ARM, and puts the high-volume per-pixel work (edge function evaluation, depth testing) on the FPGA where it can be parallelized across multiple pixel units.

3.1.2 Pipeline Flow

The software pipeline executes the following steps in order, once per frame:

1. **Read user input.** Read keyboard state via the Linux input subsystem. Arrow keys adjust the object's rotation angles by a fixed increment per frame (e.g., 2 degrees). Up/down arrows rotate around the X axis, left/right around the Y axis.
2. **Build transformation matrix.** Construct the model-view-projection (MVP) matrix by composing rotation, view (camera translation), and perspective projection matrices. This single 4×4 matrix encapsulates all geometric transformations.
3. **Transform and project vertices.** Multiply each model vertex by the MVP matrix to produce clip coordinates. Perform perspective division (divide by w) to get normalized device coordinates. Apply the viewport transform to convert from NDC [-1, 1] to screen

coordinates [0, 319] × [0, 239]. The result is a 2D screen position and a depth value for each vertex.

4. **Per-face lighting.** For each triangle, compute the face normal from the cross product of two edge vectors. Transform the normal to world space. Take the dot product with a fixed light direction vector to determine brightness. Encode the shaded color as an 8-bit RGB332 value.
5. **Triangle setup.** For each triangle, compute the Pineda edge equation coefficients (a, b for each of the three edges), the bounding box clamped to screen boundaries, the initial edge function values at the bounding box origin, and the depth interpolation step values. Convert all values from floating-point to Q12.12 fixed-point. Pack everything into a triangle packet struct.
6. **Submit to hardware FIFO.** Write the triangle packet fields to the Avalon register interface. The final write to the COMMIT register pushes the assembled packet into the hardware FIFO. Check the FIFO status register before each submission to avoid overflow.

3.1.3 Data Structures

The following C structs define the data types used throughout the software pipeline.

3D vertex (vec3f_t):

```c
typedef struct {
    float x, y, z;
} vec3f_t;
```

4D homogeneous vector (vec4f_t):

```c
typedef struct {
    float x, y, z, w;
} vec4f_t;
```

4×4 transformation matrix (mat4f_t):

```c
typedef struct {
    float m[4][4];
} mat4f_t;
```

Triangle face (face_t):

```c
typedef struct {
    int v[3];  // indices into vertex array
} face_t;
```

Screen-space vertex (screen_vertex_t):

```c
typedef struct {
    float sx, sy;  // screen coords [0,319] x [0,239]
    float sz;      // normalized depth [0, 1]
} screen_vertex_t;
```

Triangle packet (to FIFO) (triangle_packet_t):

```c
typedef struct {
    int32_t  a0, b0, a1, b1, a2, b2;        // edge steps Q12.12
    int32_t  e0_init, e1_init, e2_init;       // initial edge vals
    int32_t  z_at_origin, z_step_x, z_step_y; // depth Q12.12
    uint16_t bbox_xmin, bbox_xmax;             // 0-319
    uint16_t bbox_ymin, bbox_ymax;              // 0-239
    uint8_t  color;                             // RGB332
    uint8_t  front_facing;                      // 0=cull, 1=draw
} triangle_packet_t;
```

3.1.4 Fixed-Point Format

All values sent to hardware use Q12.12 signed fixed-point representation stored in 32-bit integers.

Property	Value
Format	Q12.12 (1 sign + 11 integer + 12 fractional)
Storage	int32_t (32 bits)
Range	-2048.0 to +2047.999755859375
Precision	1/4096 ≈ 0.000244
Conversion	fixed = (int32_t)(float_val * 4096.0 + 0.5)

Conversion function:

```
C/C++

int32_t to_fixed(float v) {

    return (int32_t)(v * 4096.0f + (v >= 0 ? 0.5f : -0.5f));

}
```

3.1.5 Matrix Transforms

The MVP matrix is constructed by multiplying: MVP = Projection × View × Rotation_Z × Rotation_Y × Rotation_X.

Rotation matrices are standard 4×4 rotations. For example, rotation around Y by angle θ uses cosθ and sinθ in the [0,0], [0,2], [2,0], [2,2] positions. The view matrix is a translation along negative Z by the camera distance (default 4.0). The projection matrix uses a 60° field of view, aspect ratio 320/240, near plane 0.1, and far plane 100.0.

3.1.10 Vertex Projection

For each vertex: (1) multiply by MVP to get clip coordinates, (2) perspective divide: ndc_x = clip.x/clip.w, ndc_y = clip.y/clip.w, ndc_z = clip.z/clip.w, (3) viewport transform:

```
C/C++

screen_x = (ndc_x + 1.0) * 0.5 * 320;

screen_y = (1.0 - ndc_y) * 0.5 * 240;   // Y flipped

screen_z = clamp((ndc_z + 1.0) * 0.5, 0.0, 1.0);
```

3.1.7 Per-Face Lighting

For each triangle face: (1) compute face normal from cross product of two edge vectors, normalized, (2) transform normal to world space using model matrix (w=0), (3) diffuse intensity: ndotl = max(0, dot(world_normal, light_dir)), (4) final intensity with ambient 0.25: intensity = clamp(0.25 + 0.75 * ndotl, 0, 1), (5) encode as RGB332:

```
C/C++

R = clamp(base_r * intensity * 7 + 0.5, 0, 7);   // 3 bits

G = clamp(base_g * intensity * 7 + 0.5, 0, 7);   // 3 bits

B = clamp(base_b * intensity * 3 + 0.5, 0, 3);   // 2 bits

color = (R << 5) | (G << 2) | B;
```

Light direction: normalize(0.4, 0.7, 0.5). Base color: R=0.2, G=0.7, B=1.0 (teal/cyan).

3.1.8 Triangle Setup - Pineda Edge Equations

This function converts three screen-space vertices into the precomputed coefficients that the FPGA uses. Edge equation coefficients for each edge E(x,y) = a*x + b*y + c:

```
Edge 0 (v1->v2):  a0 = v1.sy - v2.sy    b0 = v2.sx - v1.sx

Edge 1 (v2->v0):  a1 = v2.sy - v0.sy    b1 = v0.sx - v2.sx

Edge 2 (v0->v1):  a2 = v0.sy - v1.sy    b2 = v1.sx - v0.sx
```

The a coefficient is the x-step (hardware adds a per pixel right). The b coefficient is the y-step (hardware adds b per pixel down). No multiplication or division in the inner loop.

Back-face culling: The signed area = a0*v0.sx + b0*v0.sy + c0. If area > 0: front-facing. If area < 0: back-facing (coefficients negated, front_facing set to 0 for hardware to cull).

Bounding box: Clamped to screen: bbox_xmin = max(0, floor(min(v0.sx, v1.sx, v2.sx))), bbox_xmax = min(319, ceil(max(...))), etc.

Initial edge values: Evaluated at pixel center (bbox_xmin + 0.5, bbox_ymin + 0.5): e0_init = a0*(bbox_xmin+0.5) + b0*(bbox_ymin+0.5) + c0.

Depth interpolation: Precomputed step values using barycentric coordinates:

```
z_step_x = (a0*z0 + a1*z1 + a2*z2) / area

z_step_y = (b0*z0 + b1*z1 + b2*z2) / area

z_at_origin = (e0_init*z0 + e1_init*z1 + e2_init*z2) / area
```

where z0, z1, z2 are the per-vertex depths in [0, 1] from screen_vertex_t.sz. All values converted to Q12.12 fixed-point.

3.1.9 Complete Function List

Function	Input	Output	Description
update_rotation	keyboard state	rot_x, rot_y, rot_z	Read arrow keys, update angles
rotation_x/y/z	angle (degrees)	mat4f_t	4×4 rotation matrix
translation	tx, ty, tz	mat4f_t	4×4 translation matrix
perspective	fov, aspect, near, far	mat4f_t	Perspective projection
mat4_multiply	mat4f_t a, b	mat4f_t	Matrix product a × b
mat4_transform	mat4f_t m, vec4f_t v	vec4f_t	Matrix-vector product
transform_vertices	model, MVP	screen_vertex_t[]	Project all vertices to screen
face_normal	v0, v1, v2	vec3f_t	Cross product of edges
compute_face_color	verts, model_mat, light	uint8_t	RGB332 flat color
to_fixed	float	int32_t	Float to Q12.12 fixed point
setup_triangle	3 screen verts, color	triangle_packet_t	Compute edge coefficients
submit_triangle	fd, packet	(writes to HW)	Write packet to Avalon regs
render_frame	fd, model, verts	(writes to HW)	Submit all triangles for frame

3.1.10 Test Vector File Format

The golden reference software produces test_vectors.txt for hardware verification. It contains two sections:

Triangle packets: TRI <index> <a0> <b0> <a1> <b1> <a2> <b2><xmin> <ymin> <xmax> <ymax> <z_origin> <z_step_x> <z_step_y> <color> <front_facing>. All values are decimal Q12.12 fixed-point integers.

Expected pixel outputs: PIX <x> <y> <depth> <color>. The testbench reads TRI lines, feeds them into the pixel unit, captures every output pixel, and compares against the PIX lines. Any mismatch indicates a hardware bug.

3.2 Hardware Rasterization

3.2.1 Edge-function Rasterization

The rasterizer uses the parallel algorithm from Pineda. Each directed edge of a triangle defines a linear function E(x,y) that is positive on one side of the edge and negative on the other. For an edge from vertex (x_0,y_0) to (x_1,y_1):

$$E(x, y) = a \cdot x + b \cdot y + c$$

Where $a = (y_1 - y_0)$, $b = (x_0 - x_1)$, and c is the constant that makes $E(x_0,y_0) = 0$. A pixel lies inside the triangle if and only if all three edge functions E_0, E_1, E_2 are all non-negative. Software guarantees this by ordering triangle vertices consistently before submission.

Because E is linear, incrementing x by one adds a to the edge value and incrementing y by one adds b. Every per-pixel update is therefore a signed add and a sign check. No multipliers are in the per-pixel datapath. Depth z is updated the same way. One add per pixel, using z_step_x when walking in x and z_step_y when stepping to the next row.

3.2.2 Triangle Setup

Software handles all the per-triangle setup on the ARM. For each triangle, the software prepares a triangle_packet_t.

```
None

        `ifndef TRIANGLE_PACKET_SVH

        `define TRIANGLE_PACKET_SVH

        typedef struct packed {

            // bounding box (integer screen coords)

            logic [9:0]  bbox_xmin;   // 0-319

            logic [9:0]  bbox_xmax;

            logic [8:0]  bbox_ymin;   // 0-239

            logic [8:0]  bbox_ymax;

            // edge coefficients Q12.12 signed
```

```
    // step x: E_i += a_i   step y: E_i += b_i
    logic signed [31:0] a0, b0;  // edge 0: v1->v2
    logic signed [31:0] a1, b1;  // edge 1: v2->v0
    logic signed [31:0] a2, b2;  // edge 2: v0->v1
    // initial edge values at bounding box origin
    // evaluated at pixel center (bbox_xmin+0.5, bbox_ymin+0.5)
    logic signed [31:0] e0_init;
    logic signed [31:0] e1_init;
    logic signed [31:0] e2_init;
    // depth interpolation Q12.12 signed
    logic signed [31:0] z_at_origin; // depth at bbox origin
    logic signed [31:0] z_step_x;    // dZ per x step
    logic signed [31:0] z_step_y;    // dZ per y step
    // color and culling
    logic [7:0]  color;        // RGB332
    logic        front_facing; // 0 = cull, 1 = draw
} triangle_packet_t;
`endif
```

The packet (~480 bits) is pushed across the avalon bus into the hardware triangle FIFO. The pixel unit pulls packets from the FIFO one at a time.

3.2.3 Pixel Unit Inner Loop

Each pixel unit is responsible for a vertical partition of the screen, parametrized by Y_MIN_CLIP and Y_MAX_CLIP. Control is a single flag, active. When low, the unit waits for a valid packet on the FIFO interface. When high, it walks the bounding box and emits pixels. The following pseudocode matches the RTL in pixel_unit.sv closely:

None

```
    if active:

        if e0 ≥ 0 && e1 ≥ 0 && e2 ≥ 0:

            emit (x, y, lat_color, z[15:0])

            pixel_valid = 1

        if x == lat_bbox_xmax:            // end of row

            x = lat_bbox_xmin, y = y + 1

            e_i_row += lat_b_i, z_row += lat_z_step_y

            e_i = e_i_row, z = z_row

        else:                             // walk x

            x = x + 1

            e_i += lat_a_i, z += lat_z_step_x


        if y == lat_clip_ymax && x == lat_bbox_xmax:

            active = 0, ready = 1

    else if valid_in && ready:

        if packet.front_facing:

            clip_ymin = max(packet.bbox_ymin, Y_MIN_CLIP)

            clip_ymax = min(packet.bbox_ymax, Y_MAX_CLIP)

            skip = clip_ymin - packet.bbox_ymin

            if clip_ymin ≤ clip_ymax:

                latch all packet fields into lat_* regs

                x = bbox_xmin, y = clip_ymin

                e_i_row = e_i = e_i_init + b_i · skip

                z_row  = z  = z_at_origin + z_step_y · skip

                active = 1, ready = 0

        // backfaces and empty partitions fall through with ready = 1
```

The pixel_valid = 0 default (not shown) guarantees no phantom pixels are emitted after the last inside pixel of a triangle when active drops. Packet fields are copied into local latched registers the cycle the triangle is accepted. This lets the upstream FIFO advance to the next triangle without corrupting the in-flight rasterization.

3.2.4 Y-partition Clipping and `skip`

When the pixel unit accepts a triangle whose bounding box extends above its partition, it advances the starting row to clip_ymin before the rasterization begins. The edge functions and depth at the new origin are:

$$E_i(x_{min}, clip_{ymin}) = E_i(x_{min}, y_{min}) + b_i \cdot skip$$

$$z(x_{min}, clip_{ymin}) = z_{origin} + z_{step-y} \cdot skip$$

Skip is bounded by the screen height (240) so it is declared as a 10-bit signed value. Without this width bound, the synthesizer would infer full 32x32 signed DSP cascades. If the clipped y-range is empty or the triangle is back-facing, the pixel unit drops the packet and leaves ready high so the FIFO advances on the next cycle.

3.2.5 Depth Test and Z-buffer

Depth is maintained in Q12.12 fixed-point and sliced to 16 bits for the z-buffer write. The slice range is a local param so it can be re-tuned once we profile real mesh depth ranges from the software golden model. After perspective division, post-projection z lives in roughly [0,1), so the integer bits are mostly zero and we keep the fractional bits.

The z-buffer lives in M10K with a one-cycle read latency. Rasterization is extended with a two stage pipeline: issue read at (x,y), compare the fetched depth against the incoming z on the following cycle, and conditionally write. This drops the theoretical fill rate from 1px/cycle to 0.5 px/cycle per pixel unit in the worst case. Section 4 shows this still fits 30fps with margin.

4. Resource Budgets

The DE1-SoC carries a Cyclone V 5CSEMA5F31C6N with ~32K ALMs, 3,970 Kbits (~497 KB) of M10K block RAM, and a 50 MHz fabric clock. VGA output is fixed at 640x480 @ 60Hz. We render at 320x240 internal resolution and a 2x pixel-double in the VGA controller, giving a 30fps render cadence (each internal frame is displayed for two VGA frames).

The three memory consumers are the double buffered frame buffers, the z-buffer, and the triangle FIFO. The budget assumes two parallel pixel units.

Component	**Size**	**Kbits**	**% of total BRAM**
Frame buffer A (320x240 x 8-bit RGB332)	75 KB	600	16.8%
Frame buffer B (double buffer)	75 KB	600	16.8%
Z-buffer (320x240 x 16-bit)	150 KB	1,200	33.6%
Triangle FIFO (64 entries x ~640 bits)	~5 KB	40	1.0%
Total used	**~305 KB**	**~2,440**	**61.5%**
Remaining headroom	~192 KB	~1,530	38.5%

Three design decisions drive this budget:

1) 8-bit RGB332 frame buffer. Flat per-face lighting only needs a small color space, and packing color into a byte halves frame buffer cost versus 16-bit. This is what buys us room for a full resolution z-buffer.
2) 320x240 internal resolution. At 640x480 the frame buffer alone would be 300 KB per buffer and a 16 bit z-buffer would be 600 KB, which exceeds the device. Pixel-doubling in the VGA controller gets us to full-screen output with no extra memory.
3) 16-bit z-buffer. Lower precision causes visible depth errors where faces overlap. 16 bits uses 33.6% of BRAM but was necessary for image quality.

The 38.5% BRAM headroom absorbs a third pixel unit (adds one framebuffer partition and one z-buffer partition), FIFO resizing, and any debug logic added during design.

Metric	Calculation	Result	Status
Cycles per frame @ 30 fps	50 MHz / 30	1,666,666 cycles	2x budget vs 60 fps
Pixels per frame	320 x 240	76,800 px	
Cycles per pixel (1 unit)	1,666,666 / 76,800	~21 cycles/px	very comfortable
Required write bandwidth	4.6 Mpx/s x 1 byte	4.6 MB/s	
M10K bandwidth (per port)	1 write/cycle x 50 MHz	50 MB/s	10x headroom
Theoretical fill rate (2 units)	2 x 1 px/cycle x 50 MHz	100 Mpx/s	21x over requirement
Triangle budget (320 tris, ~50x50 px)	320 x 2,500 cycles	800,000 cycles	fits in 1,666,666

The key ratio is 21 cycles per pixel even with a single pixel unit. The z-buffer read-modify-write pipeline doubles that cost in the worst case to roughly 2 cycles/pixel, which still leaves a ~10x margin against the 21 cycle budget. Two pixel units push the theoretical fill rate to 100 Mpx/s, 21x over requirement, which turns the throughput constraint into a triangle-setup constraint rather than a rasterization constraint.

The golden software model (non-accelerated, 320-triangle sphere) hits 53–54 fps on the ARM alone, so the triangle workload is tractable. Our 30 fps target leaves headroom to scale up.

Per pixel unit, the dominant logic is six 32-bit accumulators for the edge and edge-row values, two counters (10-bit and 9-bit), and ~400 bits of latched packet state. This is well under 1K ALMs per unit on the Cyclone V, leaving the 32K ALM device essentially unconstrained by the pixel units themselves.

DSP usage is confined to the skip adjustment. The skip value is bounded to 10 bits signed, so $b_i \cdot skip$ and $z_step_y \cdot skip$ each fit in a single DSP block. Four DSPs per pixel unit, eight total. This was the main reason skip was narrowed from 32 bits to 10 bits in the RTL. All other multiplies in the pipeline (triangle setup, lighting, perspective divide) happen in software on the ARM.

5. Hardware/Software Interface

The ARM processor and FPGA fabric communicate over the Avalon memory-mapped bus. Triangle generation happens entirely on the ARM core in software; only the essential rasterization parameters are sent to hardware:

- Bounding box (xmin, xmax, ymin, ymax)
- Edge steps (a, b per edge) and pre-evaluated edge values (e_init per edge)
- Depth interpolation constants (z at origin, Δz per pixel in x and y)
- Color and back-face culling flag

Below is the register info:

Note: All the hardware registers are 32 bit.

One triangle packet would be 15 regs and 2 extra regs for control and status transmission

Register	Access	Format / Description	
bbox_x	W	[xmin(10)|xmax(10) | pad(12)]	X bounds, pixel coords 0–319
bbox_y	W	[ymin(9) | ymax(9) | pad(14)]	Y bounds, pixel coords 0–239
a0	W	32-bit	Edge 0: coefficient a
b0	W	32-bit	Edge 0: coefficient b
a1	W	32-bit	Edge 1: coefficient a
b1	W	32-bit	Edge 1: coefficient b
a2	W	32-bit	Edge 2: coefficient a
b2	W	32-bit	Edge 2: coefficient b
e0_init	W	32-bit	Edge 0 value at bbox origin (c rolled in)
e1_init	W	32-bit	Edge 1 value at bbox origin (c rolled in)
e2_init	W	32-bit	Edge 2 value at bbox origin (c rolled in)
z_at_origin	W	32-bit	Depth at bbox origin (0.5 offset)
z_step_x	W	32-bit	Δz per pixel in x direction
z_step_y	W	32-bit	Δz per pixel in y direction
color_cull	W	[color(8)|front_facing(1)|pad(23)]	RGB332 color + back-face cull flag
commit	W	Any value triggers push	

status	R	[fifo_full(1)|fifo_level(6)|pad(25)]	Hardware FIFO state: occupancy + full flag

This approach minimizes the FPGA memory footprint by offloading expensive floating-point math (triangle setup) to the CPU.

5.1 HW->SW Datapath

The HW–SW interface mirrors Lab 3's design, using the ioctl mechanism to bridge userspace and the kernel driver. The kernel driver then accesses hardware registers via the Avalon bus using iowrite32() and ioread32().

Data flow:

1. Software generates triangle packets and buffers them in a FIFO in ARM RAM
2. A submission thread polls the hardware FIFO status register
3. If space is available, the thread submits the next triangle via ioctl
4. The kernel driver writes the 15-word packet to hardware staging registers, then asserts the commit signal
5. Hardware receives the packet, stages it into its internal FIFO, and feeds it to the rasterizer asynchronously

This design allows hardware and software to run asynchronously, with the hardware FIFO absorbing bursts from the CPU.

5.2 Kernel Driver Design

The driver is structured identically to Lab 3, with two primary ioctl macros/commands:

1. RASTERIZER_WRITE_TRIANGLE macro Submits a triangle packet (like VGA_BALL_WRITE_BACKGROUND)
2. RASTERIZER_READ_STATUS Reads the hardware FIFO level and full flag (like VGA_BALL_READ_BACKGROUND)

The kernel driver will:

- Use copy_from_user() to read the triangle struct from userspace to kernel space
- Write each of the 15 words to staging registers via iowrite32()

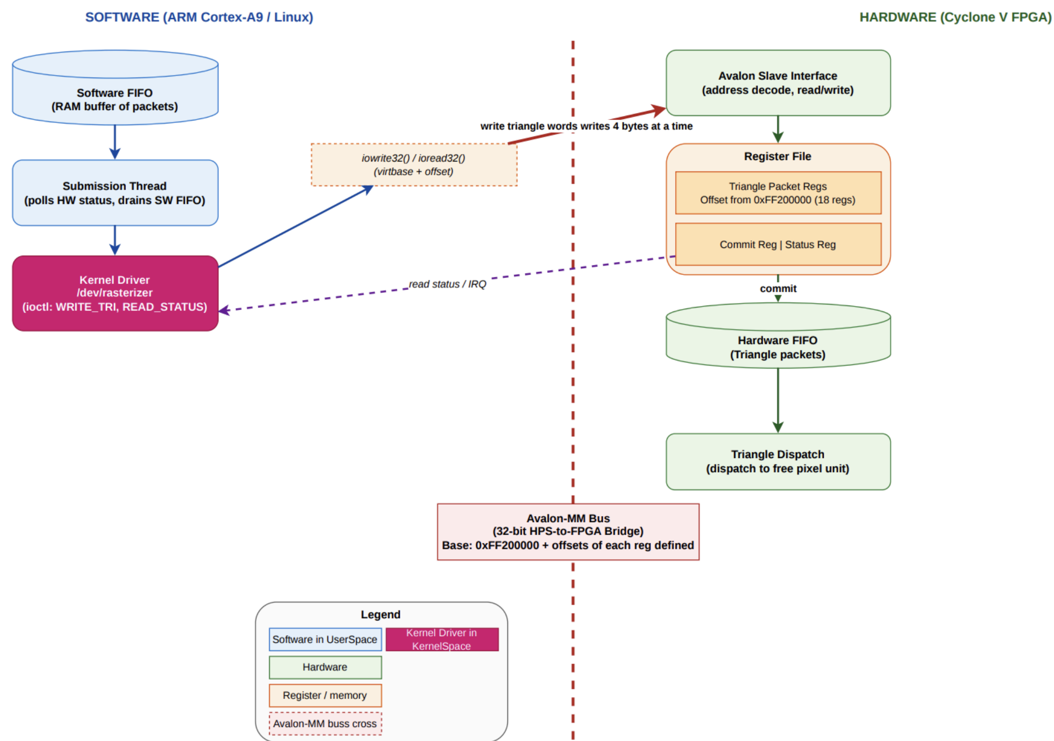
- Write the commit register to signal the hardware
- Maintain FIFO occupancy tracking in the status register

5.3 HW-SW Handshake

The interaction between software and hardware proceeds as follows:

1. Software generates a triangle packet and enqueues it into the application's software FIFO (a ring buffer in ARM RAM).
2. Submission thread polls hardware status by calling ioctl read. The kernel driver reads the hardware status register and returns the FIFO level.
3. If hardware FIFO has space, the submission thread calls ioctl send with the next triangle.
4. Kernel driver writes the packet: It copies the 15-word 60byte packet from userspace into staging registers then writes the commit register to signal the hardware that a complete packet is ready.
5. Hardware receives the commit and atomically pushes the staging register contents into its internal FIFO. The Triangle Dispatcher then pulls packets from this FIFO and dispatches them to idle pixel units.
6. Both sides continue independently: Software can keep pushing triangles to the staging registers while hardware is rasterizing previous packets from its FIFO. This asynchronous operation hides software latency.

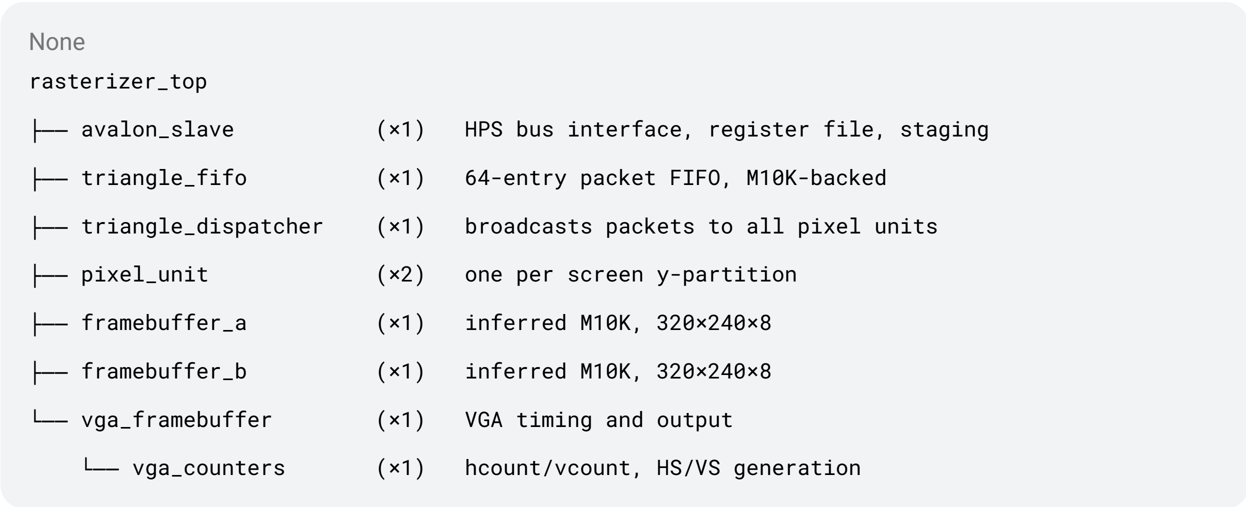
Below is a graphical representation of the overall HW SW interface



Appendix A. Module Interfaces and Hierarchy

This appendix documents the Verilog modules in the custom peripheral, their hierarchy, and the protocols connecting them. pixel_unit and vga_framebuffer are implemented; other modules are under active development. Module boundaries and port details may change during integration, but the protocols described here are the contracts each module must satisfy.

A.1 Module Hierarchy



```
None
rasterizer_top
├── avalon_slave          (×1)   HPS bus interface, register file, staging
├── triangle_fifo         (×1)   64-entry packet FIFO, M10K-backed
├── triangle_dispatcher   (×1)   broadcasts packets to all pixel units
├── pixel_unit            (×2)   one per screen y-partition
├── framebuffer_a         (×1)   inferred M10K, 320×240×8
├── framebuffer_b         (×1)   inferred M10K, 320×240×8
└── vga_framebuffer       (×1)   VGA timing and output
    └── vga_counters      (×1)   hcount/vcount, HS/VS generation
```

A.2 Inter-module Protocols

Connection	Protocol
HPS ↔ avalon_slave	Avalon-MM, 32-bit single-cycle reads/writes, no waitrequest
avalon_slave → triangle_fifo	One-cycle push pulse on commit register write; ignored when FIFO full
triangle_fifo → triangle_dispatcher	Pop handshake: dispatcher asserts pop when ready and FIFO non-empty
triangle_dispatcher → pixel_unit[i]	Broadcast; new packet issued only when all units assert ready (AND-reduction)
pixel_unit → framebuffer	M10K write port to unit's private slice, always ready, no backpressure
framebuffer → vga_framebuffer	M10K read port, 1-cycle read latency

vga_framebuffer → top-level	frame_done pulse toggles frame_parity, swapping front/back buffer

A.3 Module Interfaces

pixel_unit (implemented, file pixel_unit.sv)

```
None

module pixel_unit #(parameter int Y_MIN_CLIP = 0, Y_MAX_CLIP = 239) (

    input  logic clk, rst, valid_in,

    input  triangle_packet_t packet,

    output logic pixel_valid, ready,

    output logic [9:0]  pixel_x,

    output logic [8:0]  pixel_y,

    output logic [7:0]  pixel_color,

    output logic [15:0] pixel_depth

);
```

Behavior: Section 3.2.3 (pseudocode) and Section 3.2.4 (y-partition clipping).

vga_framebuffer (implemented, contains vga_counters)

```
None

module vga_framebuffer (

    input  logic clk, reset,                   // 50 MHz

    output logic [7:0] VGA_R, VGA_G, VGA_B,
```

```
    output logic VGA_CLK, VGA_HS, VGA_VS, VGA_BLANK_n, VGA_SYNC_n

);
```

Generates 640×480 @ 60 Hz with a 25 MHz pixel clock. Internally reads from a 320×240 × 8-bit framebuffer with 2× pixel doubling in both dimensions. In final integration the internal framebuffer will be replaced by the external double-buffered framebuffer_a/_b pair.

avalon_slave (design contract) Decodes Avalon-MM writes from the HPS and stages them into hardware registers. When software writes the commit register, the slave packs all staged data into a triangle packet and signals the FIFO to accept it. Exposes the standard Avalon-MM interface to the HPS.

triangle_fifo (design contract) 64-entry FIFO of triangle_packet_t, backed by M10K (~5 KB). Standard push/pop handshake with full, empty, and 6-bit level outputs.

triangle_dispatcher (design contract) Pops one packet from triangle_fifo and broadcasts to all pixel units. Issues the next packet only when all pixel_ready lines are high.

Framebuffers (inferred M10K, not standalone modules) Two 320×240×8-bit logical framebuffers (A and B), each physically partitioned into one M10K-backed slice per pixel unit. Each slice has one write port owned by its pixel unit and one read port routed to the VGA reader, which selects the correct slice by y-coordinate. A top-level frame_parity flip-flop toggled by frame_done selects front/back.

rasterizer_top (design contract) Top-level module. Instantiates all blocks above, exposes Avalon-MM slave to HPS on one side and VGA pins to the board on the other. Contains the frame-swap logic.