Real-Time Ray Tracer

CSEE 4840 Embedded System Design — Design Document

Innokentiy Kaurov (ik2492), Tony Giannini (aug2102), Matthew Lou (ml4855)

Spring 2026

Contents

1 Introduction

Ray tracing is a rendering algorithm that is parallel but arithmetically heavy: rendering each pixel requires several square roots, divisions, and dozens of multiplications. It is therefore a natural fit for an FPGA accelerator paired with a general-purpose CPU that handles scene setup, image composition, and I/O.

This project implements a simplified real-time ray tracer in SystemVerilog and maps it onto the DE1-SoC board. The scene is fixed: a unit sphere sitting on an infinite checkerboard plane, lit by a single point light that orbits the sphere. The user input is three 8-bit coordinate light positions. The output is a frame of the image in 24-bit RGB format. Our ultimate goal is to animate the images in real-time, i.e., at ≥ 24 FPS.

Three design choices drive everything else in this document:

1. **Fixed-point, not floating-point.** Every operand is a signed 24-bit word with 10 fractional bits. This removes the Cyclone V floating-point-DSP dependency.

2. **Parallel tracing pipelines.** The pixels are processed in parallel by six independent pipelines. A pipeline of four stages allows us to have four pixels processed simultaneously, increasing throughput. Pixels are consumed by the HPS after being stored in a line buffer in the on-chip RAM.

3. **Tiny register-map interface.** The HPS drives the hardware one row at a time. Each `ioctl` writes the light vector $\vec{\ell}$ and a row index y to a handful of 32-bit Avalon-MM registers, pulses `start`, and waits for `done`; the FPGA processes the pixels in the row in batches of 6, filling a 480-pixel row in an on-chip line buffer for the driver to drain. After 360 such calls the server has the whole frame.

2 Block Diagram

Figure 1 shows the top-down structure: a desktop client talks over TCP to a server on the HPS, sending the current light location. The server loops over the 360 rows of the output frame, issuing `ioctl`'s to the `raytracer` kernel driver, which reaches the FPGA through the HPS-to-FPGA lightweight bridge and writes to a small Avalon-MM slave; the slave feeds a row scheduler that processes the row in six-pixel batches over four-stage ray-tracing pipelines. The pipelines write the completed row to a line buffer, which the driver reads back out and sends to the server for TCP streaming to the client. After 360 rows the server has the whole frame and the loop repeats with the next light position for animation.

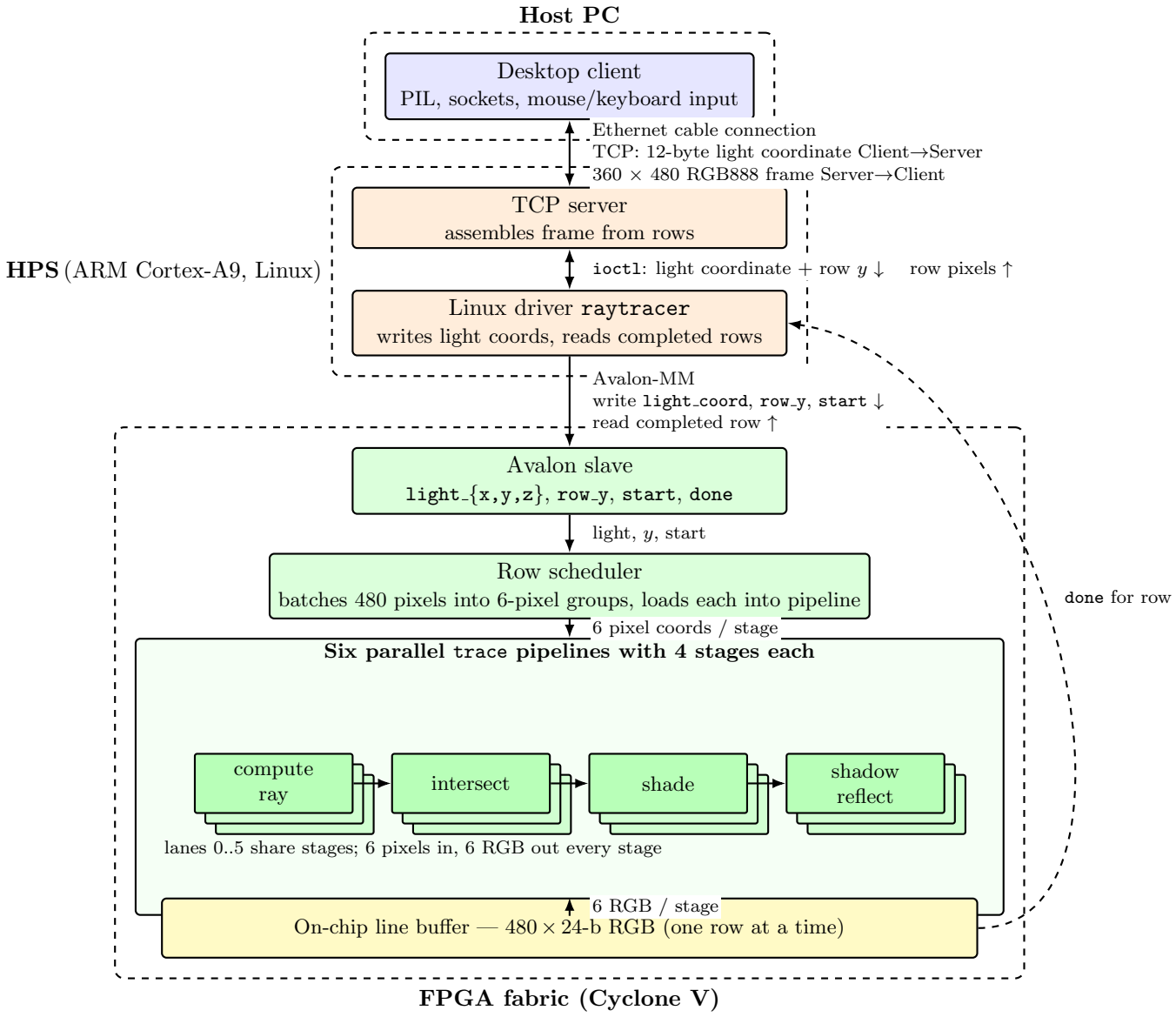

Figure 1: System block diagram. Blue = remote host, orange = HPS software, green = FPGA hardware, yellow = on-chip RAM. For each of the 360 rows the driver writes the light vector and row index y, pulses `start`, and waits for `done`. The row scheduler sweeps $x = 0, 6, \ldots, 474$ and feeds six parallel `trace` pipelines (compute ray, intersect, shade, shadow or reflect), which advance six pixels per stage. Finished pixels accumulate in the on-chip line buffer. The driver reads the completed row out, the server appends it to the frame, and the loop repeats with the next y; after 360 rows the server streams the assembled frame over TCP.

Block responsibilities and protocols.

1. **Desktop client.** A Python program that connects over TCP to the HPS, pushes a 12-byte light vector, and receives a 480 × 360 RGB888 frame. Decodes the frame with PIL for animation.

2. **TCP server.** Runs on the HPS. For each request it loops $y = 0 \ldots 359$, issuing one `ioctl` per row with the caller's light vector and row index. After all 360 rows return it streams the frame to the client.

3. **Linux kernel driver.** The `raytracer` kernel module. A misc-device platform driver whose `ioctl` takes one (`light`, `row_y`) pair, writes them to the slave, pulses `start`, waits on the hardware `done` flag (polling), memory-copies the 480-pixel row out of the line buffer, and returns it to userspace.

4. **Avalon slave.** Small Avalon-MM slave that registers `light_{x,y,z}` and `row_y`, captures the `start` pulse, and exposes a `done` status bit that software clears at the start of the next row.

5. **Row scheduler.** On `start`, emits successive six-pixel batches $(x, y), (x{+}1, y), \ldots, (x{+}5, y)$ at $x = 0, 6, \ldots, 474$ for the programmed row y, then raises `done` once the last batch has drained through the pipeline.

6. **Six parallel `trace` pipelines.** Each pipeline takes one (x, y) per stage and, after the four-stage fill latency, emits one shaded RGB pixel per stage — so the array delivers six pixels every stage. Intermediate primary-ray hits, normals, and light vectors flow between stages as pipeline registers.

7. **Per-stage logic.** The four pipeline stages (*compute ray*, *intersect*, *shade*, *shadow or reflect*) are described algorithmically in §3.1; stage-crossing signals and widths are in Table 1.

8. **On-chip line buffer.** A 480×24-bit pixel buffer in on-chip RAM (M10K blocks). The pipeline writes column by column for the current row; after `done` the Avalon-MM slave reads it back one 32-bit word at a time while the driver drains the row into DDR. The next `ioctl` re-uses the same buffer for the following row.

9. **Arithmetic primitives.** Fixed point multiplication, division, square root, vector dot product, and vector normalization modules. Each pipeline stage instantiates its own copies — not shared across stages — which is what makes the six-pixel-per-cycle throughput achievable.

3 Algorithms

3.1 The rendering pipeline

The four pipeline stages correspond directly to four algorithmic steps that a single pixel traverses, one stage per cycle.

Stage 1 — compute ray. Converts a pixel coordinate (x, y) on the screen to a ray in 3D space.

Stage 2 — intersect. Runs `intersect_sphere` and `intersect_plane` on the primary ray and picks the nearer hit (or a miss). Emits hit point, normal, and object id downstream.

Stage 3 — shade. Computes base color (solid blue for the sphere, checkerboard for the plane), surface normal $\vec{n}$, and light direction $\hat{\ell}$, and gets the luminosity from the dot product: $\max(0, \vec{n} \cdot \hat{\ell})$.

Stage 4 — shadow or reflect. A one-bounce secondary ray whose behaviour depends on the primary hit surface:

- **Plane hit ⇒ shadow.** Cast a ray from the hit point toward the light and test it against the sphere. If intersects, the pixel is in shadow and only ambient shading applies; otherwise, the pixel is lit and the diffuse term from S3 is added to ambient.
- **Sphere hit ⇒ reflection.** Reflect the primary direction about the surface normal and trace the reflected ray one bounce. The returned colour (plane checkerboard or sky gradient on miss) is blended with the sphere's own base shading.

The result is written to the line buffer at column x.

3.2 Pipeline stage interfaces

Table 1 lists the per-lane signals that cross each pipeline register boundary together with their widths. Every scalar is the project standard Q13.10 24-bit fixed-point word; three-component vectors (directions, colours, hit points) appear as $3 \times 24 = 72$ b, and multiplying each subtotal by $N = 6$ gives the physical wire count at that boundary. The payload signals are: `light` (world-space light position), `ray_dir` (normalized camera ray), `hit_point` (scene intersection $\vec{o} + t\vec{d}$), `normal` (unit surface normal at the hit), `sec_origin`/`sec_dir` (secondary-ray start and direction — shadow-to-light or mirror reflection), and `color_a`/`color_b` (primary/secondary shading terms blended by S4 per `obj_id`: `00` miss, `01` plane, `10` sphere). The output side carries `col_addr`, a 9-bit write index into the 480-entry line buffer, and `rgb888`, the packed 8-bit-per-channel output pixel.

Stage	Dir	Payload (b)	Signals (width in bits)
S1 (ray)	in	90	pixel_x(9), pixel_y(9), light(72), s1_vld(1), s2_rdy(1)
	out	153	s1_rdy(1), pixel_x(9), ray_dir(72), light(72), s2_vld(1)
S2 (intersect)	in	153	pixel_x(9), ray_dir(72), light(72), s2_vld(1), s3_rdy(1)
	out	299	s2_rdy(1), pixel_x(9), obj_id(2), hit_point(72), normal(72), ray_dir(72), light(72), s3_vld(1)
S3 (shade)	in	299	pixel_x(9), obj_id(2), hit_point(72), normal(72), ray_dir(72), light(72), s3_vld(1), s4_rdy(1)
	out	299	s3_rdy(1), pixel_x(9), obj_id(2), color_a(72), color_b(72), sec_origin(72), sec_dir(72), s4_vld(1)
S4 (combine)	in	299	pixel_x(9), obj_id(2), color_a(72), color_b(72), sec_origin(72), sec_dir(72), s4_vld(1), wr_rdy(1)
	out	33	s4_rdy(1), col_addr(9), rgb888(24), wr_vld(1)

Table 1: Per-lane port list of each pipeline stage; widths are in bits. Payload column excludes the 2-bit `vld`/`rdy` handshake; multiply by $N = 6$ lanes for the physical wire count. All 72-bit vectors are 3×24-bit Q12.10 xyz triples. `vld` travels with the data (producer drives), `rdy` travels against it (consumer drives); a transfer happens on any cycle where both are high, so pipeline bubbles are tolerated (e.g., S2's sphere-intersect takes 55 cycles, during which S1 holds its output and waits on `s2_rdy`). Pass-through signals (`pixel_x`, `light`, `ray_dir`) are re-registered into each stage's output flops, so they appear on both sides.

3.3 Fixed-point arithmetic

Every scalar is a 24-bit signed integer with 14 bits for the integer part and 10 bits for the fractional part. The representable range is $[-2^{13}, 2^{13})$ with resolution $2^{-10} = 1/1024$. We implement fixed point helper modules for multiplication, division, square root, vector dot product, and vector normalization.

3.4 Sphere and plane intersection

Sphere intersection solves the quadratic $at^2 + bt + c = 0$ and so needs one `fp_sqrt` (for the discriminant) and two parallel `fp_div` instances (for the two roots $t_{0,1}$). Plane intersection needs only one `fp_div` ($t = \vec{n} \cdot (\vec{p} - \vec{o})/(\vec{n} \cdot \vec{d})$); no square root.

4 Hardware/Software Interface

The accelerator exposes two memory regions on the HPS-to-FPGA lightweight bridge: a small *Avalon slave* for control, status, and per-row inputs, and a *line-buffer* — a window of on-chip RAM that the driver mmaps and reads as ordinary memory. Software submits one row at a time: write the light and row index into the Avalon slave, pulse `start`, poll for `done`, then read 480 packed RGB888 words out of the line buffer.

4.1 Avalon slave register map

Table 2: Avalon slave register map.

Word	Byte	Name	R/W	Meaning
0	`0x00`	`CONTROL`	W	Bit 0 `start`; bit 1 `clear_done`. Other bits ignored.
1	`0x04`	`STATUS`	R	Bit 0 `busy`; bit 1 `done`. Other bits read as zero.
2	`0x08`	`ROW_Y`	R/W	Row index for the next job. Only the low $\lceil \log_2(360) \rceil = 9$ bits are used.
3	`0x0C`	`LIGHT_X`	R/W	Signed 24-bit fixed-point x coordinate of the light source, stored in bits $[23:0]$.
4	`0x10`	`LIGHT_Y`	R/W	Signed 24-bit fixed-point y coordinate.
5	`0x14`	`LIGHT_Z`	R/W	Signed 24-bit fixed-point z coordinate.

4.2 Line-buffer RAM

The line buffer is a 480×32-bit on-chip RAM region that the driver mmaps into its address space. Word $x \in [0, 479]$ holds the RGB888 pixel at column x of the most recently completed row. The contents are valid only while `STATUS.done` is asserted; reads during a job return undefined data.

4.3 Bit-level meaning of control and status registers

`CONTROL` register (`0x00`, write-only).

Bit(s)	Name	Meaning
0	`start`	Writing a 1 issues a one-cycle start pulse. Begins rendering of the row given by the current `ROW_Y` and `LIGHT_{X,Y,Z}` registers. Starting a new job also clears any previously latched completion indication.
1	`clear_done`	Writing a 1 clears the sticky `done` flag in `STATUS`.
31:2	—	Reserved; ignored on write.

`STATUS` register (`0x04`, read-only).

Bit(s)	Name	Meaning
0	`busy`	Set to 1 while the accelerator is rendering a row. Cleared when hardware reports completion.
1	`done`	Sticky completion flag. Set when the 480-pixel row is ready in the line buffer; remains asserted until cleared by software or until a new job is started.
31:2	—	Reserved; always read as zero.

4.4 Software-visible job format

A single `ioctl` exchanges one row's worth of inputs and outputs: the caller supplies the row index and light position, and the driver returns the 480 RGB pixels of that row.

Listing 1: Userspace-visible data structure for one row request.

```
#define RT_WIDTH 480

struct rt_row {
    __u32 row_y;
    __s32 light_x;
    __s32 light_y;
    __s32 light_z;
    __u32 pixels[RT_WIDTH];   /* packed 0x00RRGGBB, filled by driver */
};

#define RT_IOCTL_RUN_ROW _IOWR('r', 0, struct rt_row)
```

Userspace fills `row_y` and `light_{x,y,z}` before the `ioctl`; on return, `pixels[x]` holds the RGB888 value for column x of row `row_y`, packed as `0x00RRGGBB`.

4.5 Complete software protocol

Per-frame the desktop client opens a TCP connection to the server on the HPS and sends one **12-byte request header** in network (big-endian) byte order:

Offset	Field	Format
`0x00`	`light_x`	signed 32-bit big-endian Q 13.10
`0x04`	`light_y`	signed 32-bit big-endian Q 13.10
`0x08`	`light_z`	signed 32-bit big-endian Q 13.10

The server byte-swaps the three light words, then loops $y = 0 \ldots 359$ driving the accelerator one row at a time. For each row:

1. Userspace fills an `rt_row` with `row_y` and the (already-decoded) light coordinates.
2. The driver writes `ROW_Y`, `LIGHT_X`, `LIGHT_Y`, `LIGHT_Z`.
3. The driver clears any stale completion flag and writes the `start` bit in `CONTROL`.
4. Software polls `STATUS.done` (or waits on an IRQ) until the row is ready.
5. The driver reads 480 consecutive 32-bit words from the mmapped line buffer into `pixels[]`.
6. The filled `rt_row` is returned to userspace.

After all 360 rows complete, the server streams a **518 400-byte RGB888 frame reply** (480×360 pixels, three bytes per pixel in row-major, top-to-bottom order) back to the client and closes the socket. The client reads exactly 518 400 bytes, hands them to PIL as an `RGB`-mode image of size 480×360, and displays/saves the result before sending the next light vector for the following frame.

4.6 Validity constraints

Valid software inputs satisfy

$$0 \le \texttt{row\_y} < 360, \qquad \texttt{light\_\{x,y,z\}} \in [-2^{13}, 2^{13}) \text{ as signed } Q\,13.10.$$

The hardware walks $x = 0, 6, \ldots, 474$ internally, so no `pixel_x` is exposed to software.

5 Resource Budget

The target device is the Terasic DE1-SoC's Cyclone V 5CSEMA5F31C6: 32 075 ALMs (128 300 equivalent logic elements), 87 variable-precision DSP blocks, and 397 kbit of M10K on-chip RAM.

5.1 Module-instance counts

Table 3: Arithmetic-primitive instances per `raytracer_pipe` and per batch after the pipeline rewrite. The per-pipe total flattens all sub-stages, so `rt_norm_stage` is counted once in the scope row and then three times (RAW / NORMAL / LIGHT) in the flattening. About half of the `fp_mul`s per pipe — the front-end basis projection (6),`rt_shadow_stage`'s offset (3), `rt_shade_stage`'s ambient (3), and `intersect_sphere`'s radius square (1 each) — have compile-time-constant multiplicands and fold to fabric shift-add rather than DSP blocks; the remaining $\sim$55 multiplies per pipe are variable$\times$variable and map onto Cyclone V DSP blocks.

Scope	`fp_mul`	`vec3_dot`	`fp_sqrt`	`fp_div`	Sub-FSMs
`intersect_sphere`	3	3	1	2	–
`intersect_plane`	0	2	0	1	–
`rt_norm_stage`	3	1	1	1	–
`rt_isect_stage` (excl. sub-FSMs)	0	0	0	0	1 `sphere`, 1 `plane`
`rt_shadow_stage` (excl. `sphere`)	3	0	0	0	1 `sphere`
`rt_shade_stage`	6	1	0	0	–
`trace_pipe` (excl. sub-stages)	0	0	0	0	1 `isect`, 1 `hp`, 2 `norm`, 1 `shadow`, 1 `shade`
`raytracer_pipe` (excl. `trace_pipe`)	6	0	0	0	1 RAW `norm`, 1 `trace_pipe`
Per-pipe total (flattened)	**30**	**12**	**5**	**8**	3 `norm`, 2 `sphere`, 1 `plane`, other stages

Among the four arithmetic primitives, only `fp_mul` contains a hardware multiplier — a combinational 24×24 signed multiply that Quartus maps to a Cyclone V DSP block when both operands are registered variables, or to fabric shift-add when one operand is a compile-time constant. `vec3_dot` contains no multiplier of its own; it is a thin wrapper around three `fp_mul`s plus an adder tree. `fp_sqrt` and `fp_div` are bit-serial iterative units with no multipliers at all — just registers, subtractors, and comparators.

6 Appendix

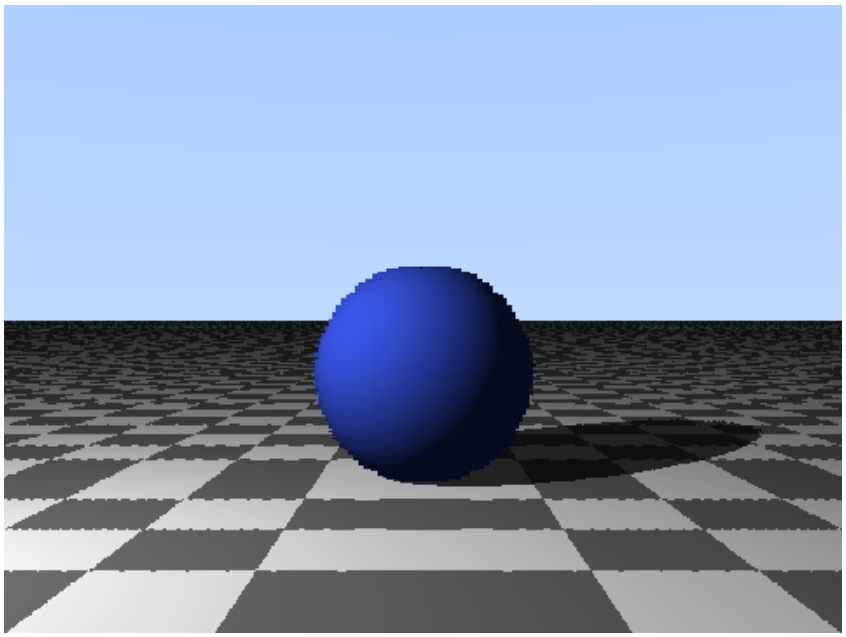

Figure 2: Example frame