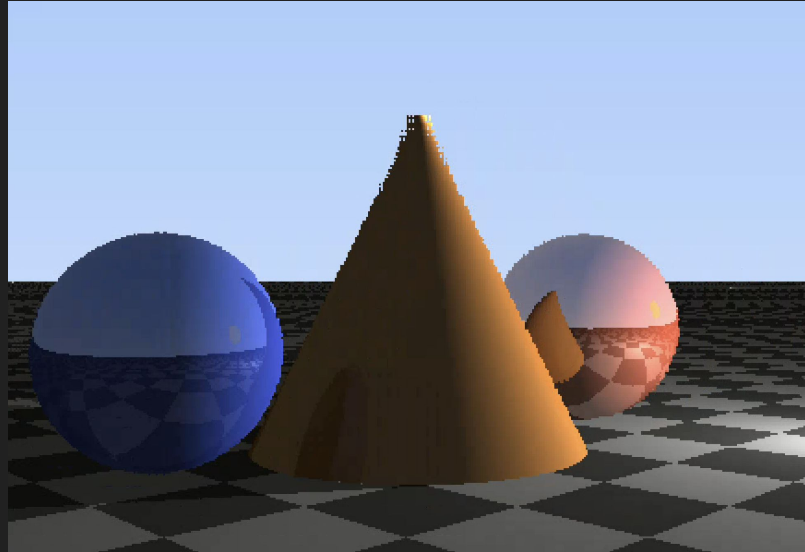
Real-Time Raytracer

Matthew Lou, Tony Giannini, Innokentiy Kaurov

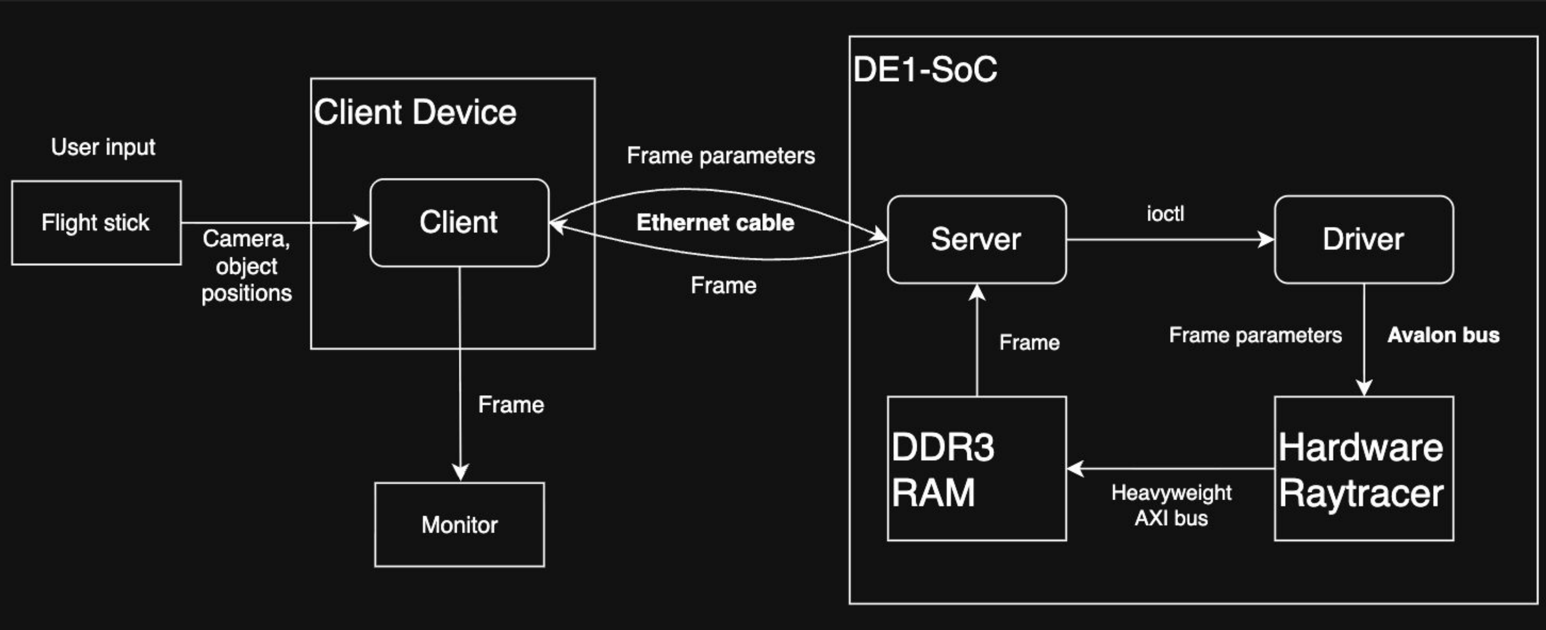
Goal

- Build a hardware accelerator for an interactive 3D scene with shadows and reflective objects: 2 spheres; 1 cone; all on 1 plane.
- Motivation: raytracing is highly parallelizable.
- Example result:



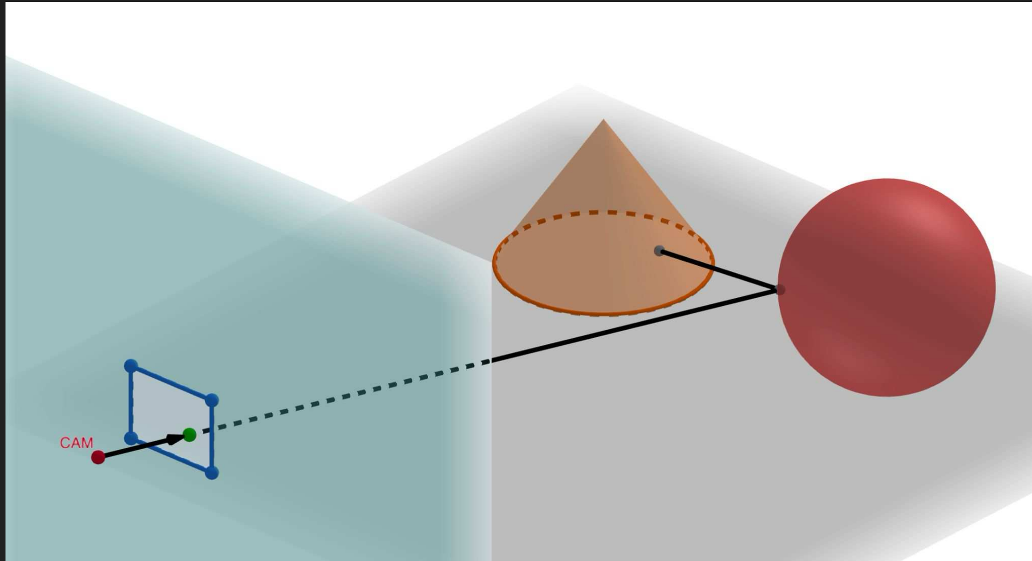
Resolution:
480x360
Goal FPS:
≥ 24 FPS

High-Level System Design



Ray Tracing Algorithm

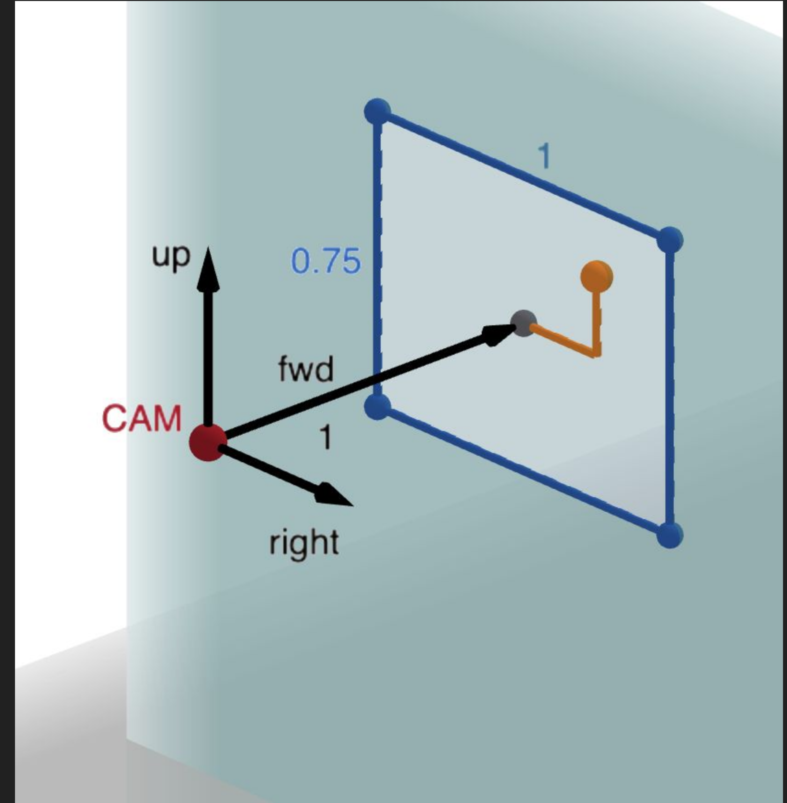
- High level idea:
  - Map each pixel on screen to a ray.
  - Launch this ray into the scene, see where it lands.
  - Make it bounce. Accumulate all colors along the path → this is the pixel color.



Step 1: 2D pixel to ray

- Camera can be rotated.
- Camera has 3 basis vectors:
  - fwd, right, up.
- Screen is 1x0.75 (matches 480x360)
  - Plane is 1 unit *fwd* from CAM, perpendicular to *fwd.*
- To translate (x, y):
  - Move 1 unit *fwd*.
  - Scale *up/right* appropriately.

- Computing basis vectors: some trigonometry on client SW.

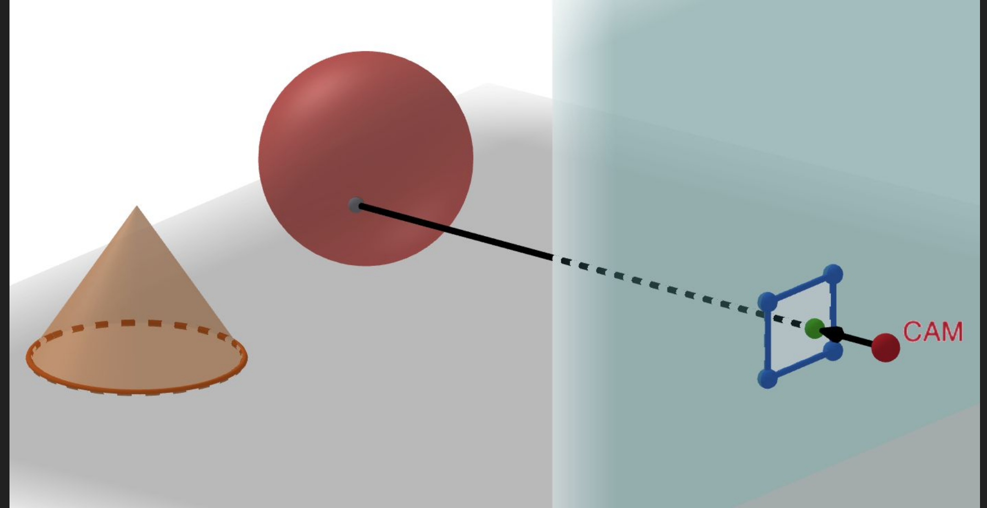


Step 2: Intersect with objects

- Try to intersect with both spheres and the cone.

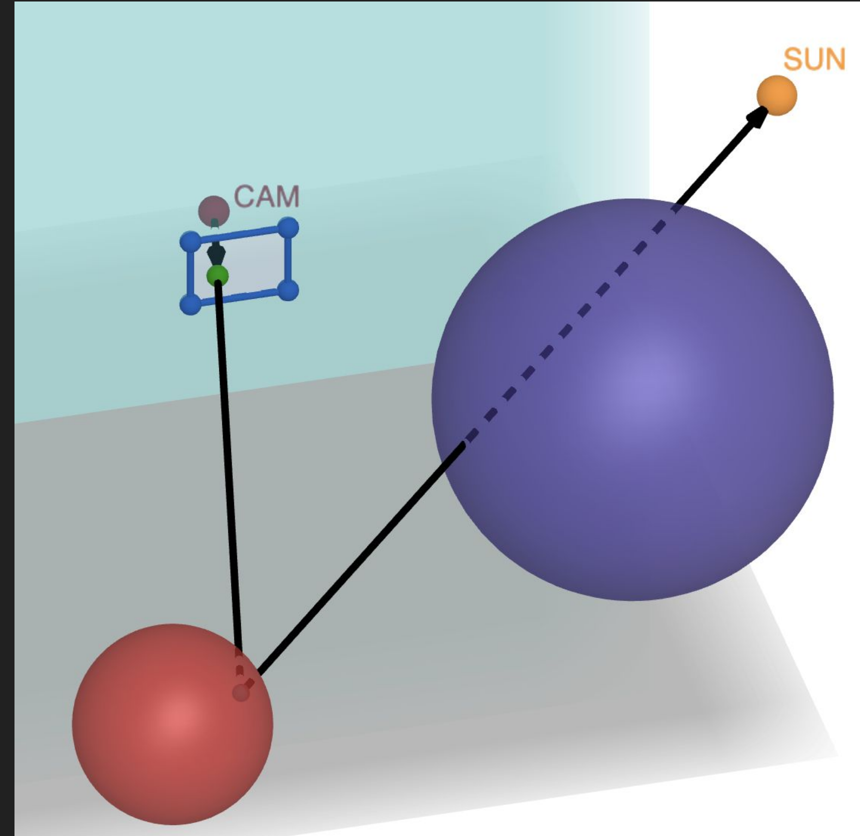
Sphere equation: $(x - x_{\text{center}})^2 + (y - y_{\text{center}})^2 + (z - z_{\text{center}})^2 = R^2$

Cone equation: $(x - x_{\text{apex}})^2 + (z - z_{\text{apex}})^2 = K^2(y - y_{\text{apex}})^2$



Step 3: Check for shadow

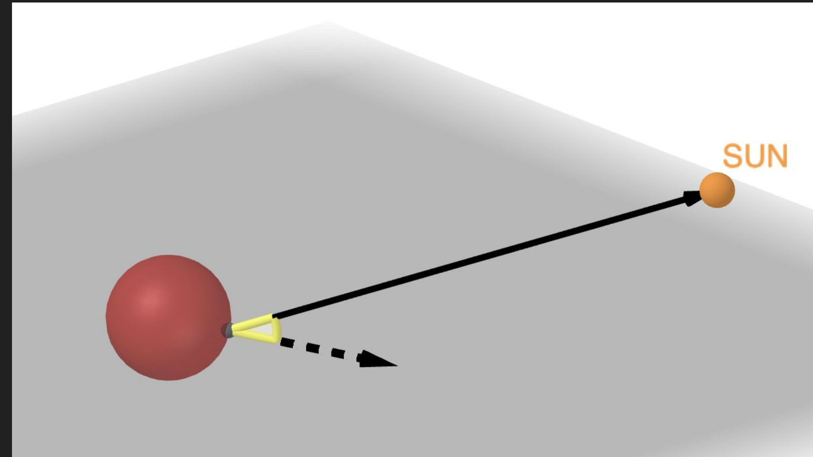
- Send a ray to the light source position from the intersection point.
- If it intersects any other object:
  - You are in shadow.
  - We only support sharp shadows.



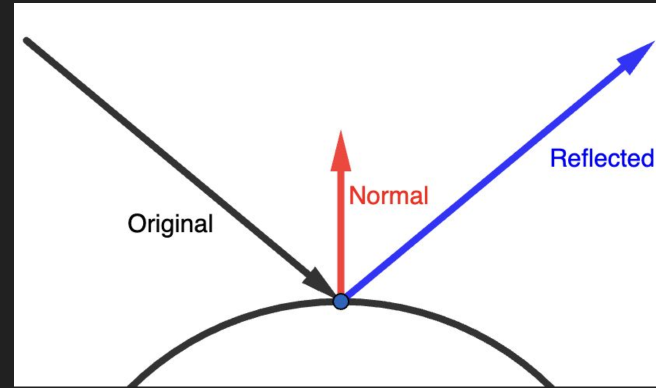
Step 4: Shading

- If we are in shadow:
  - ***Current color = ambient brightness * object color.***
  - ***Color = rgb $\in [0,1]^3$***
  - ***Ambient brightness = 5/32***
- Otherwise:
  - Consider (vector to light) • (surface normal).
  - Smaller angle = bigger product → more illuminated.
  - Scale dot product to [0, 1] range: *diffusion brightness*.
  - ***Current color =***

    (ambient + diffusion brightness) * object color.



Step 5: Reflect ray



- Reflecting across normal.

$$\mathbf{v}_{\text{reflected}} = \mathbf{v}_{\text{incident}} - 2(\mathbf{v}_{\text{incident}} \cdot \mathbf{n})\mathbf{n}$$

- Rerun from step 1 on new ray.
- Each surface has reflectivity parameter *k*. Lower *k* = more reflective:

$$\text{color} = (\text{object 1 color}) + (\text{object 2 color})/2^{k_1} + (\text{object 3 color})/2^{k_1+k_2} + \cdots$$

- Main parameters for the new ray:
  - **Depth (current reflection count):** start at 0, end at N.
  - **Ray intensity:** start at 1, divide by 2^k each bounce.

In summary

1. Convert 2D pixel to ray
2. Find intersection
3. Check for shadow
4. Shade
5. Reflect ray

Client → Server Frame Parameters

- Scene setup
  - Sphere/cone geometry
  - Colors, floor pattern,
- Camera setup
  - Camera X/Z position
  - Precomputed basis vectors
- Reflection setup
  - Max reflection depth
  - Reflectivity shift *k* for each object

Server ↔ Driver ↔ Hardware Flow

- Server accesses hardware through /dev/raytracer
  - ioctl gets interface offsets
  - mmap exposes control + frame buffers
- Driver manages frame execution
  - Server submits parameters
  - Hardware renders frame
  - Driver waits for DONE
- Completed frame returns to client
  - Server reads ready buffer
  - Sends image back over network

CSR groups — what SW writes to render a frame

one **CONTROL** write triggers the FPGA to walk all 360 batches autonomously

Frame control

lifecycle · DMA target

CONTROL	kick · clear
STATUS	done · slot
DMA_CTRL	enable
DMA_STATUS	busy · err
FB_BASE	DDR3 phys addr

Camera

origin + 3 basis vectors (right · up · fwd)

CAM.{X, Z}	cam.y locked at 1.5 m (no flying)
RIGHT.{X, Z}	right.y = 0 (no roll)
UP.{X, Y, Z}	unit vector, from yaw + pitch
FWD.{X, Y, Z}	unit vector, from yaw + pitch

→ feeds `basis_compute` (kicked once per primary ray issue)

Reflection

recursion · attenuation

MAX_DEPTH
0..7 (0 = no reflection)

SHIFT_0
sphere 0 (>>SHIFT per bounce)

SHIFT_1, SHIFT_CONE
sphere 1, cone

Light

direction vector

LIGHT.{X, Y, Z}

also derives K_const internally — sun billboard threshold (63/64 · |L|²)

Spheres (×2)

primaries

R0², R1²
SW pre-squares

SC0.{X, Y, Z}
SC1.{X, Y, Z}
centers

SCOL0.{R, G, B}
SCOL1.{R, G, B}
base colors

Floor / plane

infinite y = 0 plane

FL_MODE
check · sierp · stripe · fine

PC1.{R, G, B}
PC2.{R, G, B}
checker colors

Cone

finite, +y axis

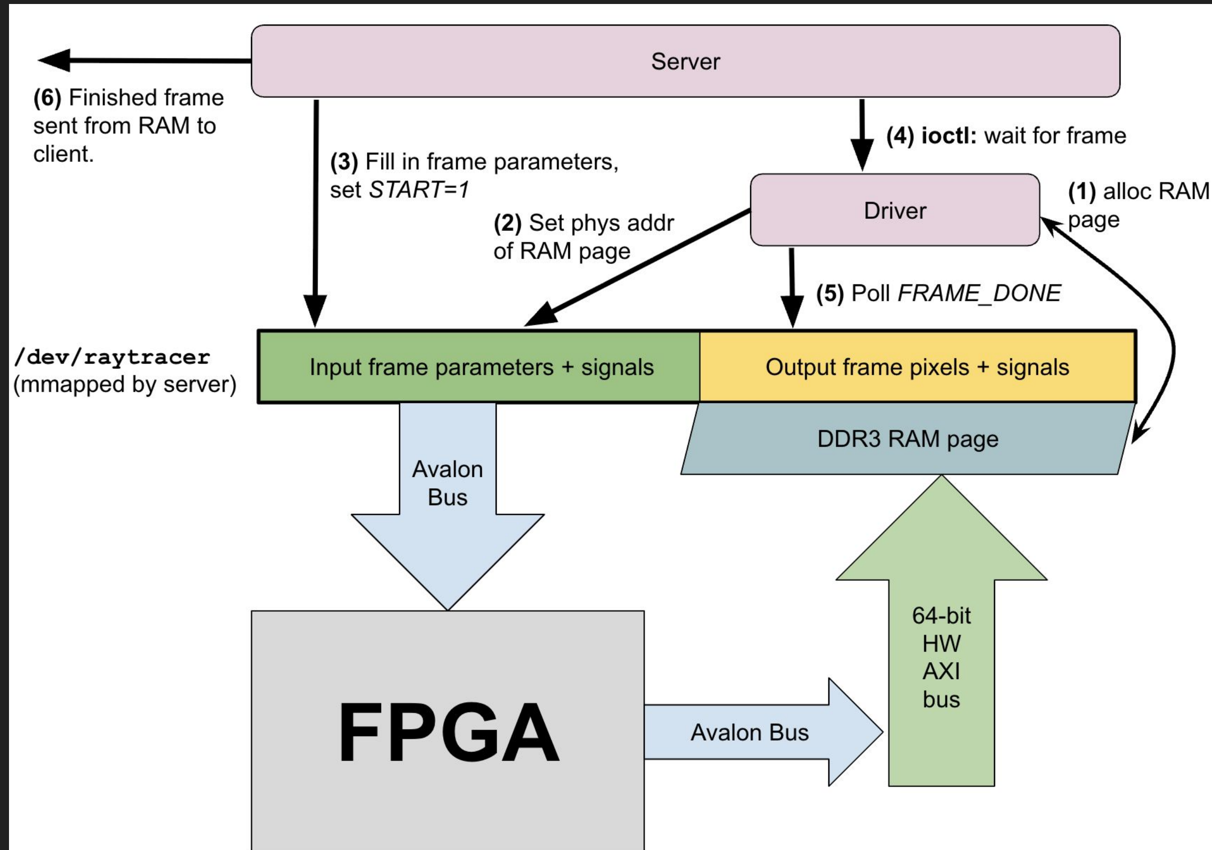
CONE.{X, Y, Z}
apex (top)

CONE.K², CONE.H
tan²(½-angle), height (0 = off)

CONE.NRM
SW pre-computed √(k²(1+k²))

CCOL.{R, G, B}
cone color

Server ↔ Driver ↔ Hardware Flow Diagram



Hardware implementation

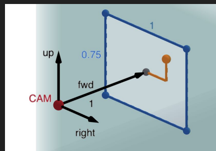
- Fixed point arithmetic
  - Q14.13 (matches DSP block width)
- Per-ray pipelines
  - 4 parallel pipelines, 8 Stages, 43 cycles each
  - Pipelines are offset by ~11 (43/4) cycles:
    - At most one pipeline finishes each cycle
  - **Line buffering:** after processing every **480** pixels, flush result to DDR3.
- Global Color Accumulator
  - M10K block holding accumulated color data

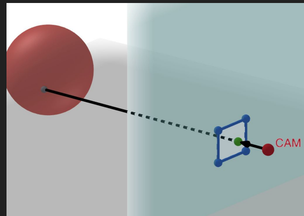
Fixed point arithmetic modules

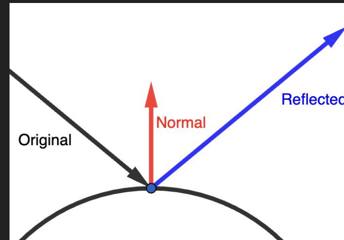
Module	Cycle count (≤ 43)	DSP usage (87 total)
Multiplier	1 (sequential)	1
Divider	41	0
Inverse	41	0
Square Root	20	0

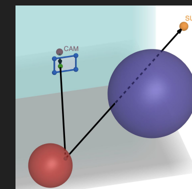
The Pipeline

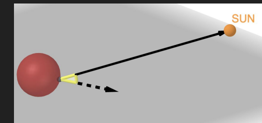
	pixel → ray	intersect		reflect		shadow check		shade
	A	**B**	**C**	**D**	**E**	**F**	**G**	**H**
	Convert 2D pixel to 3D ray	Precompute intersection coeffs	Determine the intersection point	Normalize surface normal vector	Compute reflected ray	Normalize light vector	Test shadow	Shade
× MUL	6	3	2	0	1	0	3	1
√ SQRT	0	3	1	0	1	0	3	0
÷ INV	0	3	0	1	0	1	0	0











Pipeline stage

Example stage interface:

Batch-wide input from pipe controller

Input from prev. stage (ready at tick=0)

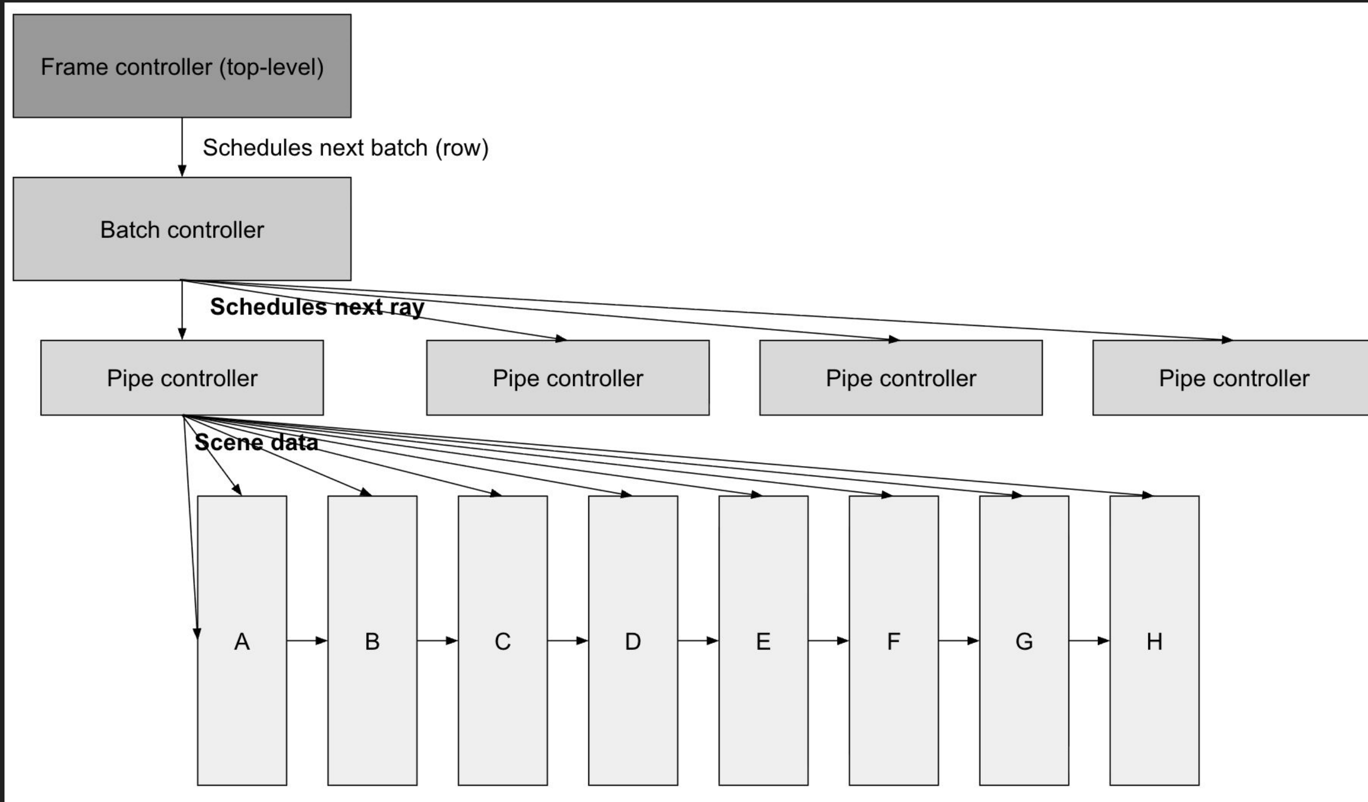
Output to next stage Ready by tick=43

```
// Stage G — SHADOW TEST: shadow-ray quadratic + visibility check
module rtl_shadow_test
    import raytracer_pkg::*;
#(
    parameter int FRAC_BITS = 13,
    parameter int WORD      = 27,
    parameter int TAG_W     = 7
)(
    input  logic                      clk, rst_n,                  // clock, async neg-edge reset
    input  logic [5:0]                tick,                        // pipe-local tick counter
    input  logic signed [WORD-1:0]    ldir_hx, ldir_hy, ldir_hz,   // surface hit point (used as shadow ray origin + bias)
    input  logic signed [WORD-1:0]    ldir_nx, ldir_ny, ldir_nz,   // unit normal (for shadow-origin bias hit + n/1024)
    input  logic signed [WORD-1:0]    ldir_tlx, ldir_tly, ldir_tlz, // to-light vector (multiplied by inv_ld → ldir)
    input  logic signed [WORD-1:0]    ldir_inv_ld,                 // 1/|to-light| (from Stage F)
    input  logic signed [WORD-1:0]    ldir_ld,                     // |to-light| — compared with shadow ray's t to test occlusion
    input  logic signed [WORD-1:0]    ldir_cr, ldir_cg, ldir_cb,   // base color (forwarded)
    input  logic [2:0]                ldir_hit_type,               // hit type (used for self-skip on shadow ray)
    input  sphere_t                   in_sc0,                      // sphere 0 (per-pipe relayed)
    input  sphere_t                   in_sc1,                      // sphere 1
    input  cone_t                     in_cone,                     // cone
    input  logic signed [WORD-1:0]    ldir_rx, ldir_ry, ldir_rz,   // reflection vector r (forwarded)
    input  pipe_baggage_t             ldir_bag,                    // baggage
    output logic signed [WORD-1:0]    shdw_hx, shdw_hy, shdw_hz,   // hit point (forwarded)
    output logic signed [WORD-1:0]    shdw_nx, shdw_ny, shdw_nz,   // unit normal (forwarded)
    output logic signed [WORD-1:0]    shdw_lx, shdw_ly, shdw_lz,   // ldir = tl·inv_ld — unit light direction (computed here)
    output logic signed [WORD-1:0]    shdw_cr, shdw_cg, shdw_cb,   // base color (forwarded)
    output logic [2:0]                shdw_hit_type,               // hit type (forwarded)
    output logic                      shdw_in_shadow,              // 1 = light blocked by any primitive (THIS STAGE'S WORK)
    output logic signed [WORD-1:0]    shdw_rx, shdw_ry, shdw_rz,   // reflection vector r (forwarded to Stage H)
    output pipe_baggage_t             shdw_bag                     // baggage forwarded to Stage H
);
```

Inside the stage: cycle-by-cycle FSM controlled by *tick*, e.g., “on cycle X / 43, compute Y”.

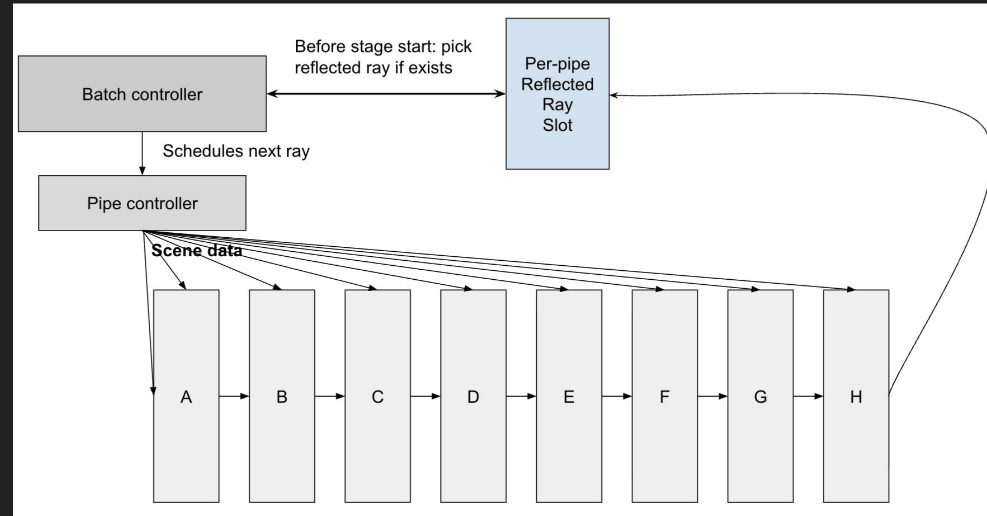
State stored on flip flops.

Managing the pipelines



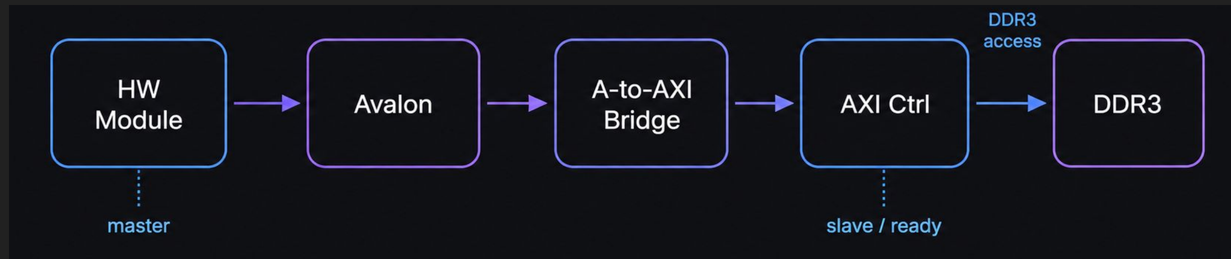
Scheduling rays

- Batch controller maintains the next unprocessed pixel in the row, and tells each pipe which ray to process.
- If a ray needs a reflection after stage H, put it back at the start of the current pipeline.
- Batch controller algorithm:
  - When pipe *i* needs to load:
    - If reflection slot of *i* nonempty:
      - schedule the reflection
    - Else:
      - schedule [next unprocessed pixel]++



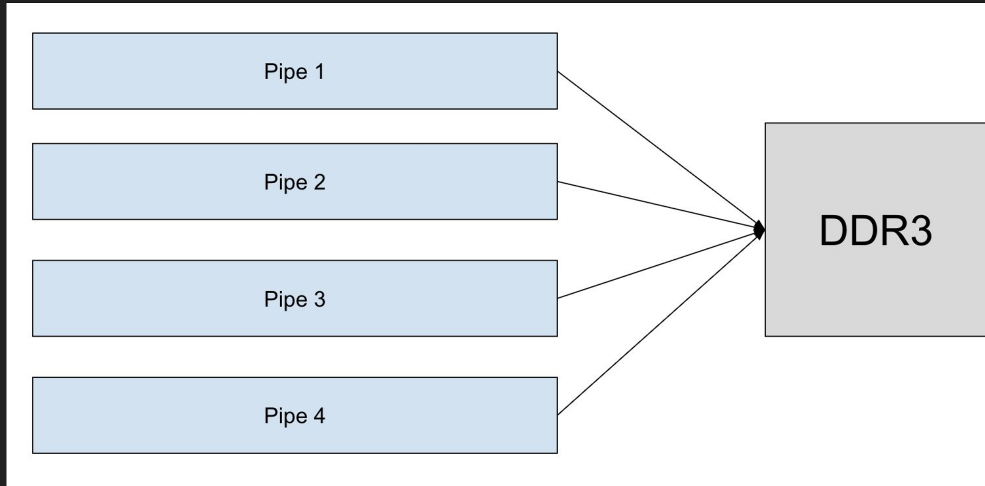
Storing pixel values

- Motivation for using DDR3
  - Bytes per frame: 3 (bytes per pixel) * 480 * 360 ~= 500 KB
  - 2 frames (double buffering) won't fit in M10K
  - DDR3 = 1Gb
- HW module sends Avalon signals through bridge to AXI bus
  - AXI memory controller talks to DDR3
  - HW module is master; AXI memory controller/slave accepts requests
  - When ready is asserted, the module provides valid address & data



Writing to DDR3: Naive approach

- **Approach 1:** When a pipe finishes a pixel, it writes it to DDR3 immediately.
- **Problems:**
    - DDR3 writes are long and take unpredictable no. of cycles. (may be >10)
    - We have only one Avalon-MM DDR3 interface.
    - Overall, not guaranteed to write 4 pixels in 43 cycles.



Writing to DDR3: Improved approach

- High level:
  - Store line buffer in M10K RAM (1 cycle access latency).
  - Flush line buffer to DDR3 at the end of batch.

How to combine colors across reflections?

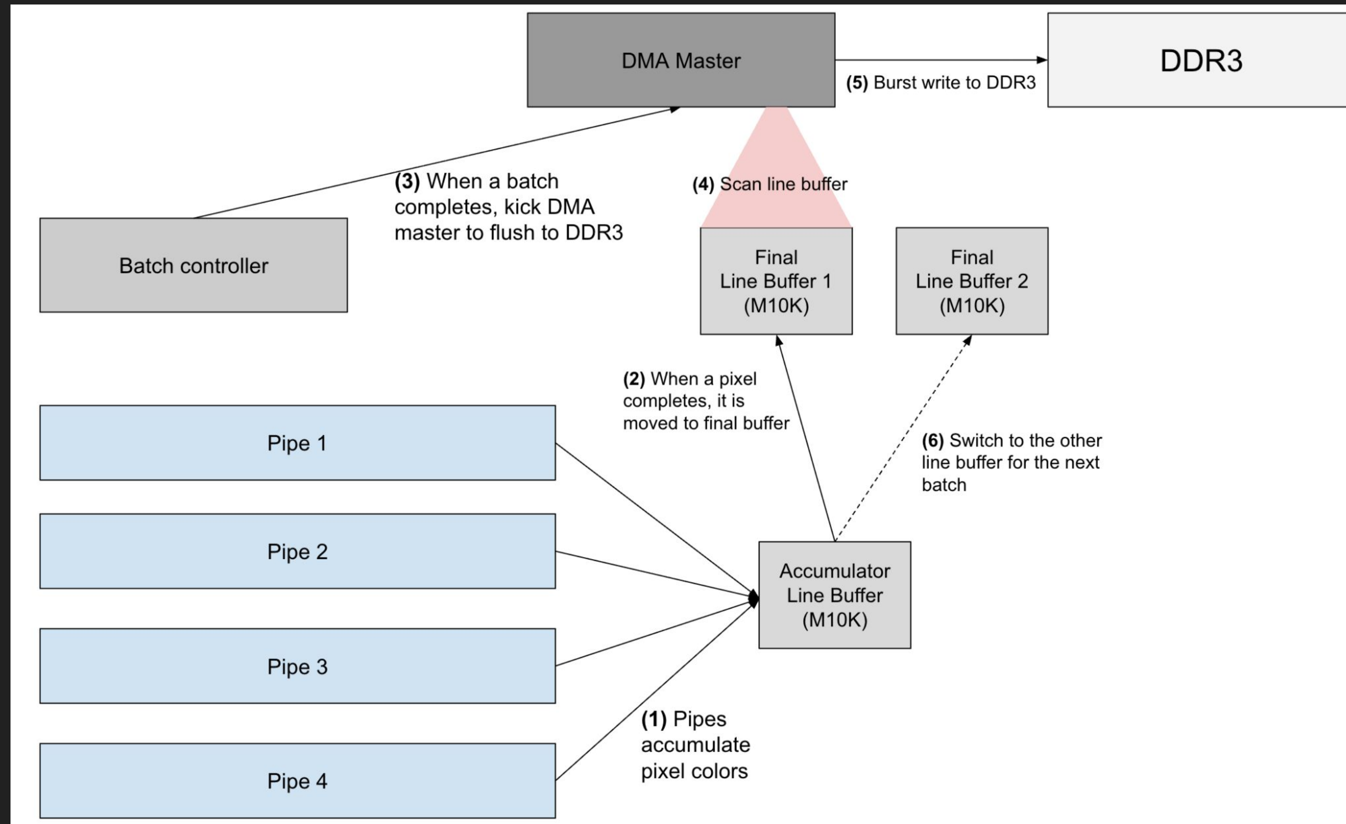
- Recall color formula

$$\text{color} = (\text{object 1 color}) + (\text{object 2 color})/2^{k_1} + (\text{object 3 color})/2^{k_1+k_2} + \cdots$$

- Tag each ray with its original pixel, and accumulate pixel color in the line buffer.

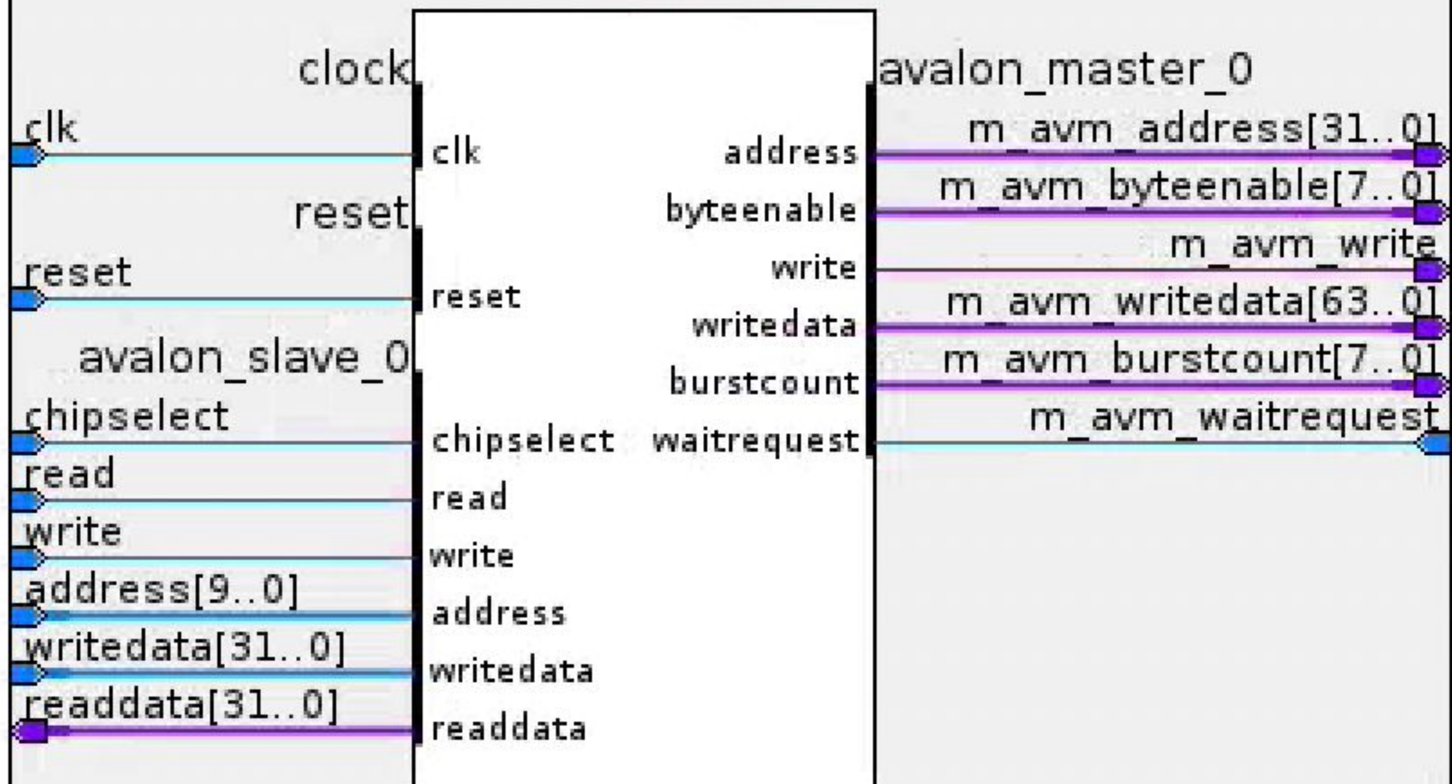
Writing to DDR3: Improved approach

- **Double buffering:** while writing to DDR3, can start the next row.



DDR3 Burst Writes

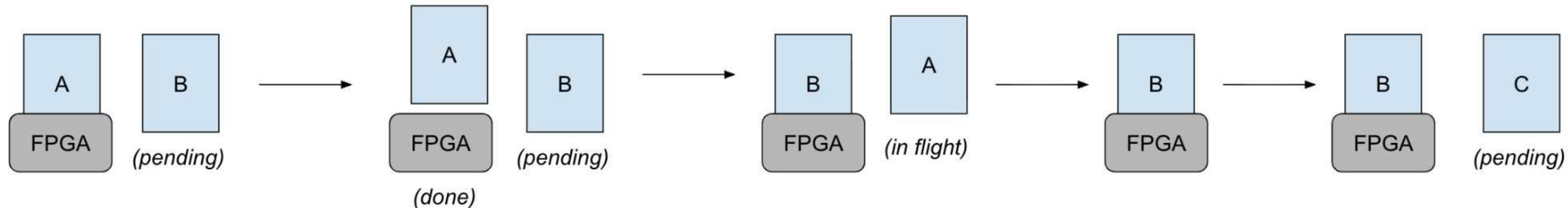
- DDR3 access has high latency; Burst writes reduce overhead
  - Send one start address + burst count
  - Then stream one 64-bit writedata word per cycle
  - Writes consecutive 64-bit words: pixels in one row are adjacent so it’s perfect.
  - Since each pixel is 24 bits (1 byte per color), we can fit 2 pixels in a word.
- Instead of paying 5+ cycles for each write, we pay 10-20 cycles for initial setup, then 1 cycle per word.


raytracer_inst
clock
clk
clk
reset
reset
reset
avalon_slave_0
chipselect
chipselect
read
read
write
write
address[9..0]
address
writedata[31..0]
writedata
readdata[31..0]
readdata
avalon_master_0
m_avm_address[31..0]
address
m_avm_byteenable[7..0]
byteenable
m_avm_write
write
m_avm_writedata[63..0]
writedata
m_avm_burstcount[7..0]
burstcount
m_avm_waitrequest
waitrequest
raytracer

Optimizations

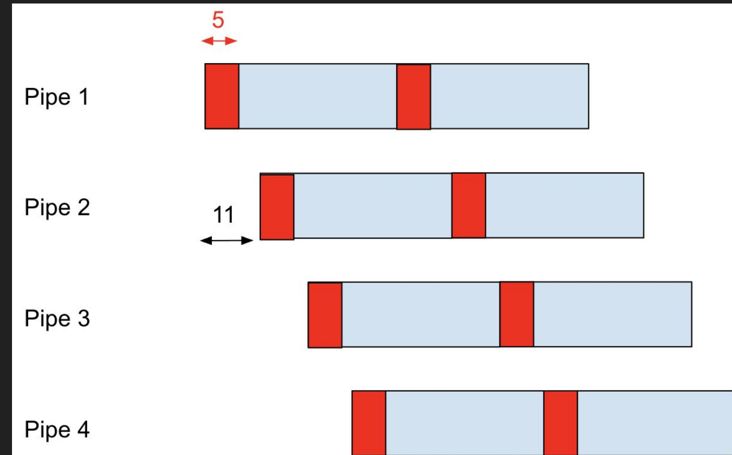
Frame Double Buffering

- Problem: transfer time causes idle hardware
  - Sending frame to client back takes time.
  - While frame is being sent, FPGA is idle.
- Solution: keep next frame ready
  - One frame processing
  - One frame pending
- Two frame buffers in DDR3 prevent overwrite
  - With the DONE signal, hardware also tells which buffer it wrote to.
  - Send completed frame while rendering next frame.
  - Hardware stays busy more consistently



Time-multiplexing ray computation module

- Recall stage A of the pipeline.
- Its first step is to call ray computation submodule:
  - The submodule converts 2D pixel (x, y) to ray (x, y, z).
  - It takes only 5 cycles to complete (heavily parallelized), but uses **5 DSP blocks**.
- **Optimization:** since pipelines are offset by 11 cycles and the module only takes 5, we reuse the same module for different pipelines.



Demo