

RAYTRACER: Real-time A+-quality TRiangle And Circle Renderer

William Feng
EECS, MIT
Cambridge, MA, USA
wyf@mit.edu

Calvin Rodrigue
EECS, MIT
Cambridge, MA, USA
calvinr@mit.edu

Abstract—We present the design for a ray tracing pipeline implemented entirely on an FPGA. Our system takes as input a three-dimensional scene consisting of objects (spheres, triangles, parallelograms, or planes) with associated spatial and material information. We discuss the implementation details of our system and evaluate its performance.

Index Terms—3D rendering, FPGAs, floating-point arithmetic, 3D graphics, ray tracing

I. INTRODUCTION

Rendering scenes with realistic graphics requires doing a large amount of repeated math to cast rays and is inefficient to do on a CPU due to low parallelization ability. Using an FPGA, ray tracing can be done both in parallel and in a pipelined fashion to speed up rendering by a tremendous amount. While GPUs are typically used in such cases, they can be expensive and power-demanding. By utilizing the parallelization and pipelining capabilities of an FPGA, we hope to render high-definition graphics of static scenes faster than a comparably-priced CPU.

II. SYSTEM OVERVIEW

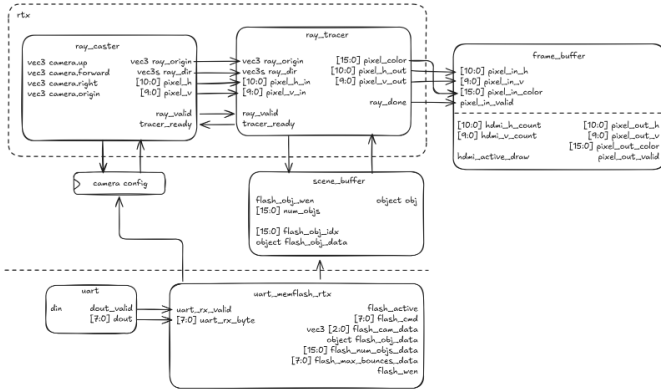


Fig. 1. Top-level schematic of the whole system

Our system takes as input a scene description, camera position, and some tracing-specific parameters such as the maximum number of bounces before a ray expires. All parameters are stored on-chip using BRAM or registers and are editable via a command script that communicates with the FPGA over UART. Our top level module consists of:

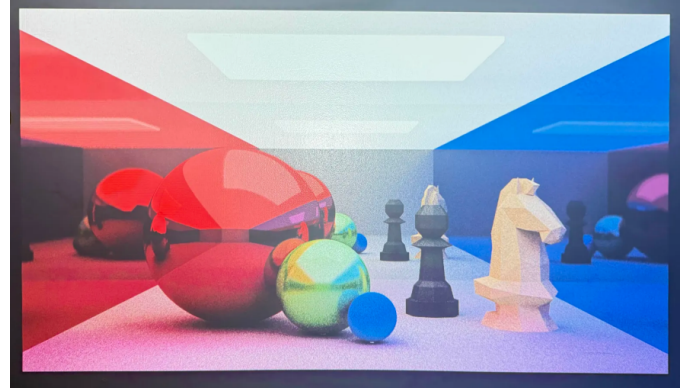


Fig. 2. Sample ray tracer output on projector. Scene and camera position were flashed over UART.

- 1) An **rtx** module, which takes as input a camera position and scene description, then serially outputs pixel values to construct a 1280×720 picture.
- 2) A **high_definition_frame_buffer** module, which accepts pixel values from the **rtx** module and communicates with DRAM to write them back to memory. This module also provides the correct signals for HDMI output.
- 3) A **uart_memflash_rtx** module, which listens for commands over UART to modify the camera position, rendering parameters, and scene buffer.

III. FLOATING-POINT ARITHMETIC (WILLIAM AND CALVIN)

We implemented floating-point arithmetic for all math operations in the ray tracer. This was a difficult decision to make, but we ultimately settled on it because floating-point numbers can express a larger range of numbers than fixed-point integers at the expense of area and additional cycles of delay. As shown in Figure 3, our floating-point format uses 24 bits, of which 1 bit is used for the sign, 7 for the exponent, and 16 for the mantissa. If $s \in \{-1, 1\}$ is the sign bit, $e \in [0, 2^7)$ is the exponent, and $m \in [0, 2^{16})$ is the mantissa, then the floating-point number represented by this 24-bit value is

$$f = 2^{e-63} \cdot (1 + m/2^{16}).$$

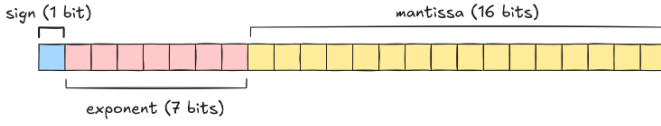


Fig. 3. Breakdown of bit usage for fp data type

This is in close correspondence to IEEE 754, which is the spec for 32-bit floating point numbers. We called our floating-point type **fp**. Core modules for handling fp operations include are described below.

Sidenote: Though we initially hardcoded the floating-point format, we reparameterized everything to support arbitrary-format floats later on to experiment with saving logic via smaller floats. We didn't really like how it looked though, and eventually got access to bigger boards (thanks Joe) so in the end decided to stick with fp.

A. Addition and Multiplication

The addition module **fp_add** is a two-stage pipeline that adds two fp values. It uses about 60-150 LUTs, the most resource-intensive operation being counting the number of leading zeros after shifting and adding in order to renormalize the final sum.

The multiplication module **fp_mul** is a one-stage pipeline that multiplies two fp values. It uses about 80-130 LUTs and one DSP48.

The module **fp_shift** multiplies fp values by constant powers of two by adding a shift amount to the exponent. It uses fewer resources than the full **fp_mul** module and is useful fairly often as we need to multiply by 2, 1/2, and 4 in a number of places. This module is fully combinational.

B. Fast Inverse Square Root

We designed a module **fp_inv_sqrt** to calculate $x \rightarrow 1/\sqrt{x}$ for use in the vector normalization module. Our design exploits the floating-point representation format to get a good initial guess for $1/\sqrt{x}$ by shifting x right one bit and subtracting from the magic constant 0x5e669c.¹ It then refines this guess through two Newton-Raphson stages with the formula

$$y_{n+1} = y_n \cdot \left(\frac{3 - xy_n^2}{2} \right).$$

This module achieves $\sim 0.0007\%$ relative error and is fully pipelined with 10 cycles of delay. Exact error margins for all math modules are reported in VII

C. Basic Vector Operations

The **fp_vec3** datatype is 72 bits wide and represents a vector with three fp components. The **fp_vec3_add** and **fp_vec3_mul** modules are the vector counterparts to the scalar addition and multiplication modules, and use around 3x the resources while maintaining the same cycle delay.

The **fp_vec3_scale** module multiplies a vector by a scalar.

D. Advanced Vector Operations

The **fp_vec3_dot** module computes the dot product between two vectors, and is useful in computing specular reflections as well as in finding the magnitude of vectors. It uses one **fp_mul** module and two **fp_add** modules to sum the component-wise products. This module uses about 400 LUTs and incurs 5 cycles of delay.

The **fp_vec3_cross** module computes the cross product between two vectors, and is used only for plane intersection in the triangle intersector. It uses six **fp_mul** modules and three **fp_add** modules to calculate the component-wise values. This module uses about 600 LUTs and incurs 3 cycles of delay.

The **fp_vec3_normalize** module is by far the most resource-intensive of our math modules. There are four of them total, and each requires about 960 LUTs. This module takes a vector as input and returns a scaled version of the vector that points in the same direction but has magnitude 1. Internally, it uses a dot product module, an inverse square root module, and a vector scaling module. **fp_vec3_normalize** is fully pipelined, like all other math modules, and incurs 16 cycles of delay.

The **fp_vec3_lerp** module performs linear interpolation between two vectors. It takes as input two **fp_vec3** values, a scalar amount t as the interpolation value, and $1 - t$. The reason we take $1 - t$ as input is to reduce the amount of extra logic and cycles needed in the reflection module. This module uses about 600 LUTs and incurs 3 cycles of delay.

IV. RTX

To render a scene using ray tracing, one casts rays from a point of origin and repeatedly calculates intersections between this ray and all objects in the scene. The ray is bounced off the nearest intersection and accumulates color/lighting information until it goes off into the void or reaches a set maximum number of bounces. Thus the primary problems that need to be solved are:

- 1) Initializing rays with the correct point of origin and direction given camera position and pixel coordinates
- 2) Ray-sphere and ray-triangle intersection
- 3) Reflecting a ray given the surface normal and material properties

They are encapsulated in the **ray_caster**, **ray_intersector**, and **ray_reflector** modules, respectively. Each is discussed in a separate subsection below.

We initially checked our understanding by writing a ray-tracer in C and verifying that our math produced reasonable images. The C implementation proved extremely helpful as a reference while we implemented the HDL.

A. Ray Caster (William)

The **ray_caster** module takes as input a camera position and pixel_h/pixel_v coordinates, and outputs a ray centered at the camera origin pointing in the direction of that pixel. This process is visualized in Figure 4.

The camera is parameterized with front, up, and right vectors (which need not be orthogonal). The process of casting then consists of:

¹found using a binary search-esque script. See `ctrl/find_constants.py`

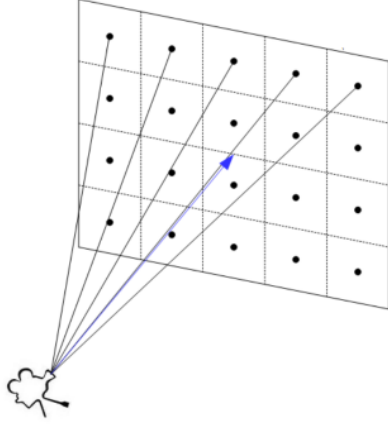


Fig. 4. Casting rays from a camera origin to a screen full of pixels



Fig. 5. Ray caster output with xyz coordinates visualized as rgb color values

- 1) converting pixel coordinates to normalized device coordinates (u, v) in the $[-1, 1]$ range,
- 2) scaling the right vector by u , scaling the up vector by v , summing both of these with the forward vector, and
- 3) normalizing the resulting vector to have magnitude 1.

This process is fully pipelined. The caster is told to generate a new ray whenever the tracer completes its current ray; and since the caster is much faster than the tracer, we can queue up the next ray to be cast before we actually need it, and release it immediately upon request to reduce latency.

Once a ray is cast, it is passed to the **ray_tracer** module, which contains the ray intersector and ray reflector modules. A ray goes through the intersector and reflector a number of times at most the `max_bounces` parameter, and accumulates **ray_color** and **incoming_light** information as it bounces. These values are tracked in registers that live for the lifetime of the ray.

`pixel_h` and `pixel_v` information accompany the ray along its

whole journey so that when it comes out the tracing pipeline, we know where on the screen to render it.

B. Ray Intersector (William and Calvin)

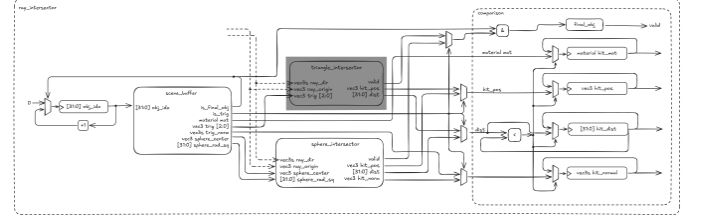


Fig. 6. Ray intersector module

The **ray_intersector** module takes as input a ray with given origin and direction and computes the nearest hit point with any object in the scene. Our design supports both sphere and triangle geometries.

The intersector works by calculating intersections with all objects in a scene and keeping the closest one. Information about the closest hit (position, surface normal, location) is latched in registers and updated whenever the distance to the current object's intersection point is less than the register's closest hit distance.

The **sphere_intersector** module² solves the equation

$$\|(O + tD) - C\|^2 = R^2,$$

where (O, D) are the origin and direction vectors of the ray, C is the center of the sphere, and R is the sphere's radius. Solving this equation requires the **quadratic_solver** module, which is capable of finding the lesser root of any monic quadratic $t^2 + bt + c = 0$. We care about the lesser root because we only want the closest intersection point. The sphere intersector module also computes the hit position $O + tD$ and the surface normal. In all, this module uses about 3400 LUTs.

The **trig_intersector** module³ solves the equation

$$O + tD = P_1 + uE_1 + vE_2$$

where (O, D) are the origin and direction vectors of the ray, P_n is the n^{th} corner of the triangle, $E_1 = P_2 - P_1$, $E_2 = P_3 - P_1$, and (u, v) are the barycentric coordinates of the intersection point. It does this using the Möller-Trumbore intersection algorithm, which returns the triangle-centric coordinates tuw . Since t is the same in both world space and triangle space, $O + tD$ gives the intersection point of the point with the triangle's plane. Actual intersection can be checked very easily, since in tuw space the triangle is always transformed to the unit triangle, so whether or not the point lying inside is as simple as checking that $u > 0$, $v > 0$, and $u + v < 1$.

This algorithm can also be expanded to represent intersections of parallelograms and planes, where a ray will always eventually intersect with a plane (that it is not parallel to), and

²authored by William

³authored by Calvin

for parallelograms it is as simple as changing the check from a unit triangle to a unit square ($u+v < 1$ to $u < 1$ and $v < 1$).

C. Ray Reflector (William and Calvin)

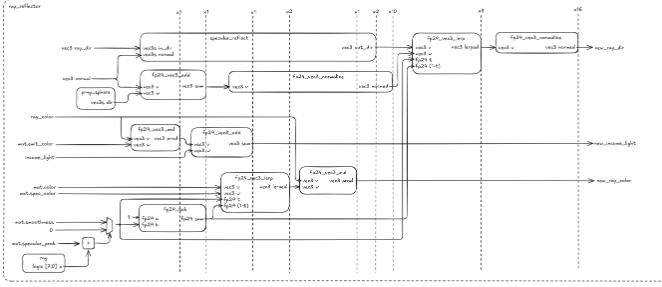


Fig. 7. Ray reflector module

The ray reflector takes as input a ray, the position of the closest hit as returned by ray_intersector, the surface normal at that position, and the material properties of the surface it hit. These properties consist of:

- 1) diffuse color (fp_vec3, 72 bits)
- 2) emissive color (fp_vec3, 72 bit)
- 3) specular color (fp_vec3, 72 bits)
- 4) specular probability (8 bits)
- 5) smoothness (fp, 24 bits)

In ray tracing, there are two kinds of reflections: diffuse and specular. A **diffuse** reflection bounces the ray to point in a random direction in the hemisphere centered on the surface normal. A **specular** reflection is deterministic and bounces a ray in accordance to the law of reflection: the angle between the incoming ray and the surface normal is the same as the angle between the surface normal and the outgoing ray. The two types of reflections are visualized in Figure 8.

Types of reflection

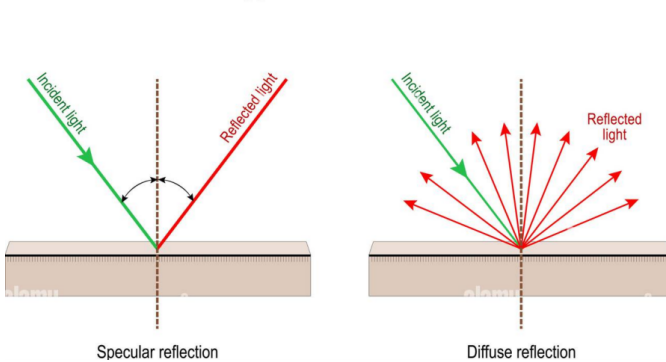


Fig. 8. Visualization of the two types of reflection

Diffuse reflections. In order to improve realism, some extra bias is given to directions closer to the surface normal using a standard technique called cosine weighted sampling. Thus the process of computing a diffuse reflection is:

- 1) Generate a random point on the unit sphere.
- 2) Add this to the surface normal.

- 3) Normalize the sum to have magnitude 1.

The most complicated part of this is generating a random point on the unit sphere. This is handled by the **prng_sphere** module ⁴, which is implemented as a **linear-feedback shift register** (LFSR) with random taps sampled from a **ring oscillator** ⁵. The LFSR has 48 bits and generates three sixteen-bit signed integers, which get converted to floating-point values through the **fp_convert** module. The ring oscillator consists of four unstable inverter loops of length three that each get sampled at a fixed point and then are XORed together. This gives three fairly random fp values which are then renormalized to give a point on a sphere. ⁶

Specular reflections. The formula for calculating specular reflections is

$$d - 2(\vec{d} \cdot \vec{n})\vec{n}.$$

We implement this formula in the **specular_reflect** module ⁷ using one fp_vec3_dot, one fp_vec3_scale, and one fp_vec3_add.

To render even more types of materials, we can blend the two types of reflections using the material's specular probability and smoothness properties. Essentially, we first randomly decide whether the bounce is a specular or diffuse bounce using the material's specular probability. If it is a diffuse reflection, we use the diffuse direction. If it is a specular reflection, we linearly interpolate between the diffuse and specular directions by the material's smoothness. This helps create materials that are look semi-smooth, like a tomato or partly glossy ball. See Figure 9 for an example.

Finally, the color is determined by multiplying the current ray_color value by the material's color (a combination of the specular and diffuse colors, determined in a similar fashion to the bounce direction) and adding the material's emissive color to the current incoming light value. The ray is then passed back to the intersector if it has not reached the maximum number of bounces, or released to the frame buffer if it has.

V. HIGH DEFINITION FRAME BUFFER (CALVIN)

To display the average of multiple high-definition frames, it was necessary to use a frame buffer with a very large capacity to store the sums of red, green, and blue color channels. The only viable option for this was to use the off-chip DRAM.

There are actually two internal frame buffers stored on the DRAM: one containing only the RGB values to be rendered to the screen in 5-6-5 8-packed words, and one containing the sum of all RGB values over every previously rendered frame.

For the **sum frame buffer**, color sums are represented by 64-bit 21-22-21 RGB integers. Each time a pixel is rendered from the **rtx** module, it is sent into the frame buffer **traffic_generator** as a 5-6-5 RGB value with screen position data. The **traffic_generator** then processes this data through either

⁴authored by Calvin

⁵authored by William

⁶While these points are technically not uniformly random due to the initial point being sampled from the unit cube, it looks good enough to us and most people couldn't tell when we asked them.

⁷authored by William

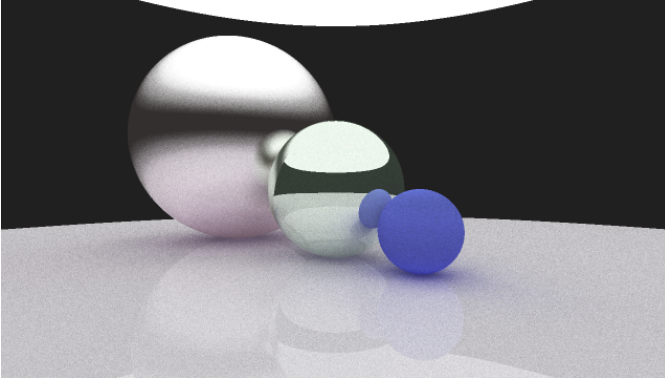


Fig. 9. Testbench rendering of specular reflections. The middle sphere is fully reflective while the leftmost one is only partially so. Observe the blending between specular color (white) and diffuse color (red) on the leftmost sphere.

2 or 4 stages, depending on the current frame count. The stages are as follows:

- 1) **FETCH_RTX**: Fetching the current sum in memory. This is necessary because the new data is being added to the old data, so the old data must first be retrieved.
- 2) **WB_RTX**: Writing back the new sum to memory. Because sums are 64 bits and memory words are 128 bits, this sum is packed with the sum of an adjacent pixel. The original value of this adjacent pixel is what is written back to memory along with the new sum of the target pixel. The **rtx** module generates pixels in an order such that no race conditions are created.

To avoid synthesizing division, averages are only computed when the frame count is 2^n so that bit shifting can be used instead. These averaged values are then stored in a separate section of memory and packed more densely so that they can be accessed at a higher throughput for HDMI display (8 pixels per memory request). These values are generated immediately after sum data is retrieved by a **FETCH_RTX** request. However, because of the pixel generation order chosen to avoid race conditions in the **WB_RTX** stage, it would require a significant amount of memory to pack all the averages and send them as a single memory request.

- 3) **FETCH_FB**: As a result, the **FETCH_FB** stage is necessary to retrieve the original 7 other values that are to be packed in with the new one.
- 4) **WB_FB**: A writeback request to the frame buffer that is to be displayed on the monitor. It contains 7 of the old values and 1 of the new averaged values.

In total, each pixel has a maximum of 4 memory requests, representing fetching the old sum, writing back the new sum, fetching the packed values from the frame buffer, and writing back the new packed value to the frame buffer.

VI. UART FLASHER (WLLIAM)

Our system supports reprogramming the scene buffer via UART. This is implemented in the **uart_memflash_rtx** module, which has the following interface:

- 1) Input byte **uart_rx_byte** from UART receiver
- 2) Single-cycle valid input bit **uart_rx_valid** denoting whether the bit is valid
- 3) 8-bit command type **flash_cmd**
- 4) 1024-bit generic output **flash_data**
- 5) Output write enable bit **flash_wen**

The data protocol is implemented as follows:

- 1) User sends a command byte according to the following lookup table:
 - a) 0x00 — cam.origin (24 bits)
 - b) 0x01 — cam.forward (24 bits)
 - c) 0x02 — cam.right (24 bits)
 - d) 0x03 — cam.up (24 bits)
 - e) 0x04 — object index (12 bits)
 - f) 0x05 — object data (298 bits)
 - g) 0x06 — scene buffer depth (12 bits)
 - h) 0x07 — max bounces (8 bits)

This value is latched into the **flash_cmd** output register.

- 2) The user sends a number of bytes corresponding to the command type, where we round up if the number of bytes in that type is not a whole number. Bytes are shifted in one at a time as the number used varies depending on what data is being flashed.

To write into the scene buffer, for instance, the user should send command byte 0x04, then send 2 bytes corresponding to the index of the object they want to overwrite, then send $\lceil 298/8 \rceil$ bytes corresponding to the object's binary representation.

VII. EVALUATION

On the test **knight.json** scene, our system runs in 11.3 seconds per frame while the reference C implementation runs for 1 minute 43 seconds to render one frame. This represents a speedup of $\sim 912\%$.

All our math modules are testbenched properly and tell us that fp operations achieve around 0.001% error.

Module	Mean relative error
fp_mul	0.000644%
fp_add	0.001344%
fp_sqrt	0.000710%
fp_inv	0.001128%
fp_inv_sqrt	0.000714%
fp_vec3_add	0.000549%
fp_vec3_dot	0.000907%
fp_cross	0.001026%
fp_vec3_mul	0.000656%
fp_vec3_scale	0.000634%
fp_vec3_normalize	0.000733%
fp_vec3_lerp	0.000871%

TABLE I
MEAN PERCENT ERROR FOR FLOATING-POINT MATH MODULES

The theoretical optimal throughput of our system, in units of pixels per cycle, is

$$\frac{1}{(\# \text{ objects}) \times (\text{max bounces})}$$

Currently, our system is not fully pipelined and has a throughput of

$$\frac{1}{(\# \text{ objects} + \text{intx delay} + \text{rflx delay}) \times \text{max bounces}}.$$

VIII. IMPLEMENTATION INSIGHTS

- Block diagrams were helpful for writing fully pipelined math modules.
- That said, if a module is meant to be pipelined, it should be testbenched very thoroughly. We ran into three instances where a module was improperly pipelined and caused major headaches down the road.
- Verilator is an alternative to Icarus that sacrifices some compile time for greatly reduced execution time.
- A great way to debug pipelining issues is to hold 100 cycles of **invalid** (X) input, have one cycle of **valid** input, and then 100 more cycles of **invalid**. This makes it very obvious in the waveform which signals are improperly pipelined, since correct pipelining will look like single-cycle valids but incorrect pipelining is invalid the whole time.
- For the Nexys board, xdc files are available online at the Digilent Github to help get the pin numbers for some components. The DDR3 pins still have to be manually numbered though.
- To stop Vivado from freaking out about combinational loops in the unstable ring oscillators, we used `set_property ALLOW_COMBINATORIAL_LOOPS TRUE [get_nets ...]` in the xdc file.

IX. CONTRIBUTIONS

Calvin implemented the high definition frame buffer, a bunch of floating point math modules, triangle intersection, and to this day is working on a multi-intersector design that will make the whole thing 10x faster.

William implemented sphere intersector, ray caster, a bunch of floating point math modules as well, sphere intersection, and the UART flashing infrastructure.

X. REFERENCES

- 1) Moller-Trumbore Triangle Intersection Algorithm
- 2) Sphere Intersection Algorithm
- 3) Sebastian Lague's GPU Ray Tracer