APRIL 2023

“RAYS THE BAR” HW Acceleration of Ray-Tracing on an FPGA System

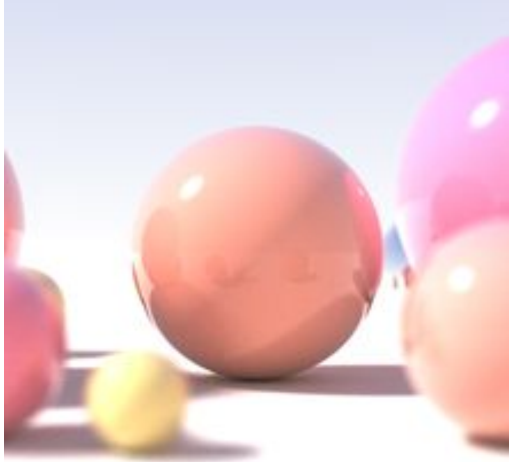
ECE 382N – Adv Embedded Microcontroller Sys

Course Project Presentation

Ganesh Ram Koushik, (gk7734)
Shreyas Ravishankar, (sr48925)
The University of Texas at Austin

What is Ray Tracing?

- Computer Graphics technique for Rendering “Realistic” images.
- Simulates the interaction between light rays and the objects.
- In simple terms, it is a Ray Optics physics model.



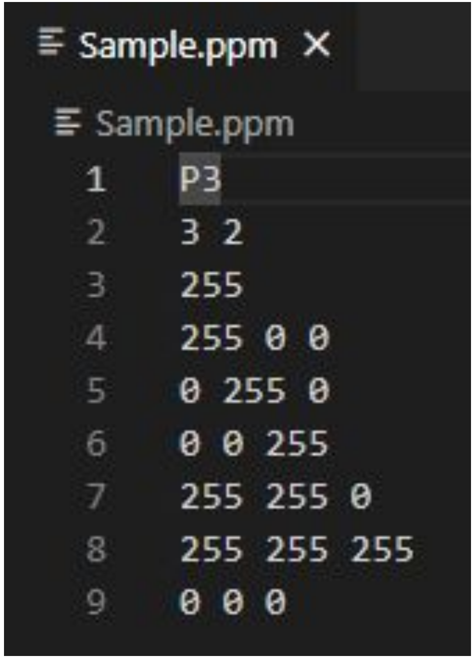
High Level Overview

- Implemented SW code for Ray Tracing.
- Used Verilog and HLS to offload sub-tasks to the HW.
- Achieved **5.07x** speedup over baseline software running on Ultra96.

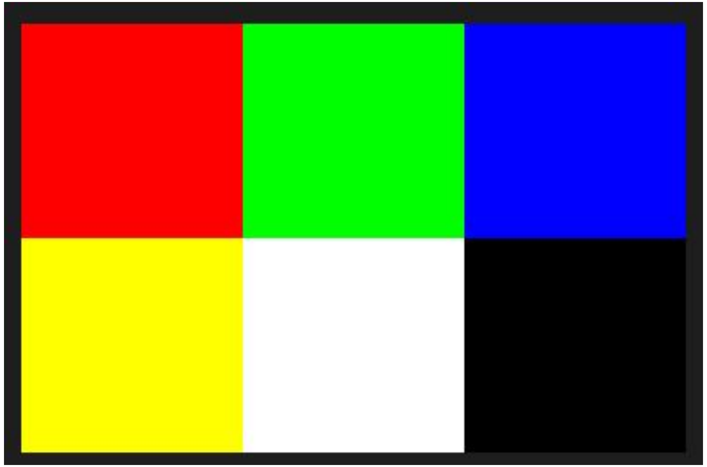
Algorithm Discussion

- The PPM Format
- The Ray Model
- The Camera & Viewport
- Ray-Object Interactions
- Anti-Aliasing
- Surfaces

The PPM Format



```
P3
3 2
255
255 0 0
0 255 0
0 0 255
255 255 0
255 255 255
0 0 0
```

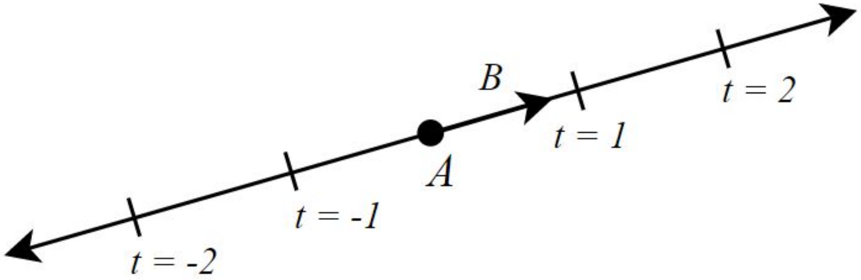


The Ray Model

$$\overrightarrow{v} = \overrightarrow{A} + \overrightarrow{B}*t$$

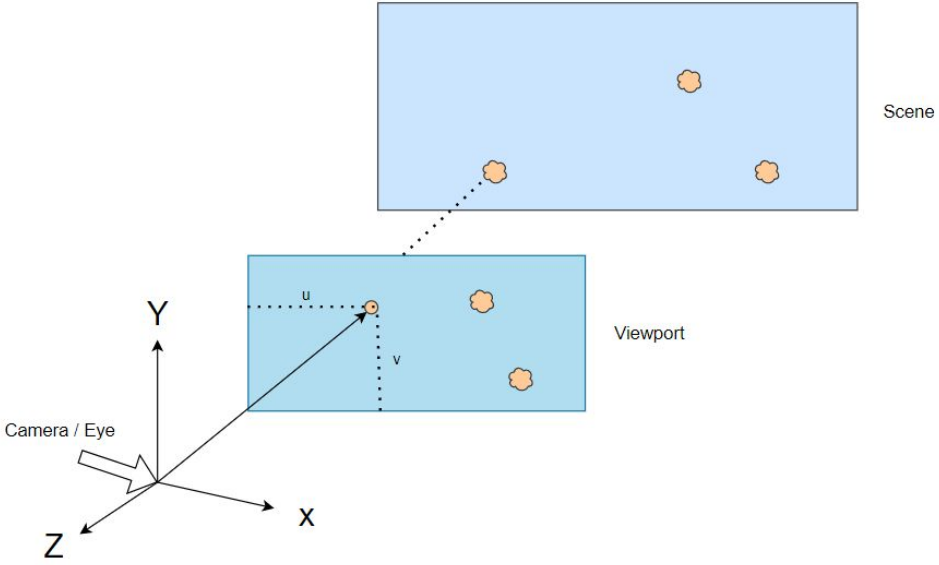
$\overrightarrow{A}$ *is the Ray Origin*

$\overrightarrow{B}$ *is the Ray Direction*



The Camera and the Viewport

- A scene is described using a global coordinate system.
- A ray is shot into every pixel of the viewport.
- Color of the pixel is determined by the object(s) that this ray "hits".

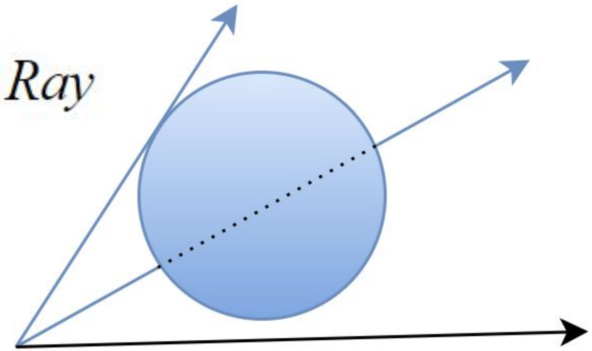


Ray Object Interactions

$$(x - C_x)^2 + (y - C_y)^2 + (z - C_z)^2 = r^2$$

Since the Intersection Point can be represented as a Ray

$$(P_x(t) - C_x)^2 + (P_y(t) - C_y)^2 + (P_z(t) - C_z)$$

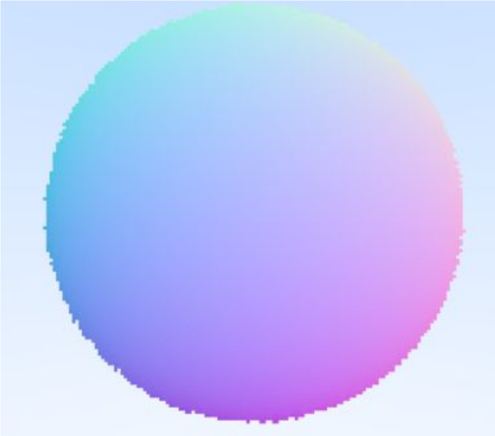


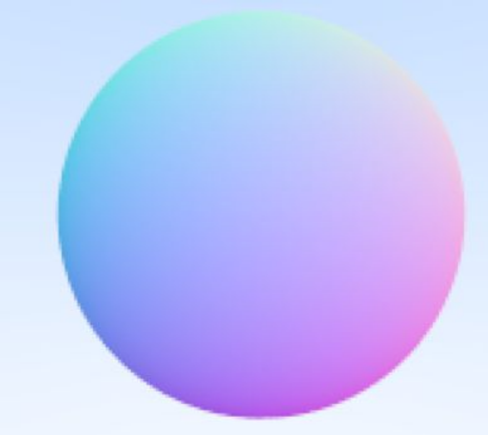
Simplifying

$$t^2*(B \bullet B) + 2t*((B) \bullet (A - C)) + (A - C) \bullet (A - C)$$

Anti-Aliasing

- Pixels at object borders contain multiple colors.
- Shooting one ray per pixel renders objects with jagged edges.
- Shoot multiple (~100) rays per pixel to smoothen the image.

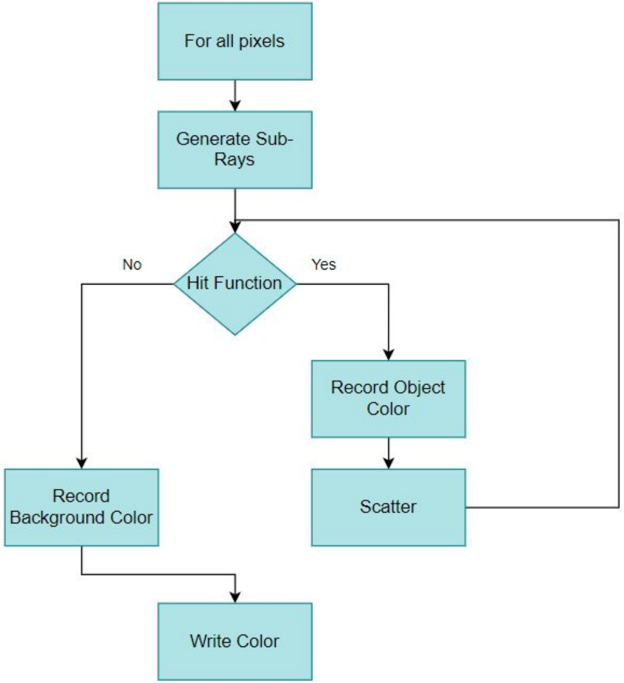




Surfaces

- Light interacts differently with various different objects in the natural world.
  - Lambertian (Random Scattering)
  - Metal (Laws of Reflection)
  - Glass (Snell’s Law- Refraction, Total Internal Reflection)
- Each time the ray “hits” an object, a scattered (child) ray is generated.
- Color of the pixel is a combination of the colors observed by all of the generated rays.

Software Implementation



Generate Sub-Rays

- Used for anti-aliasing.
- Generate 100 rays from a single base ray with small random offsets.
- Color of a pixel is cumulative sum of colors seen by all sub-rays.

Hit Function

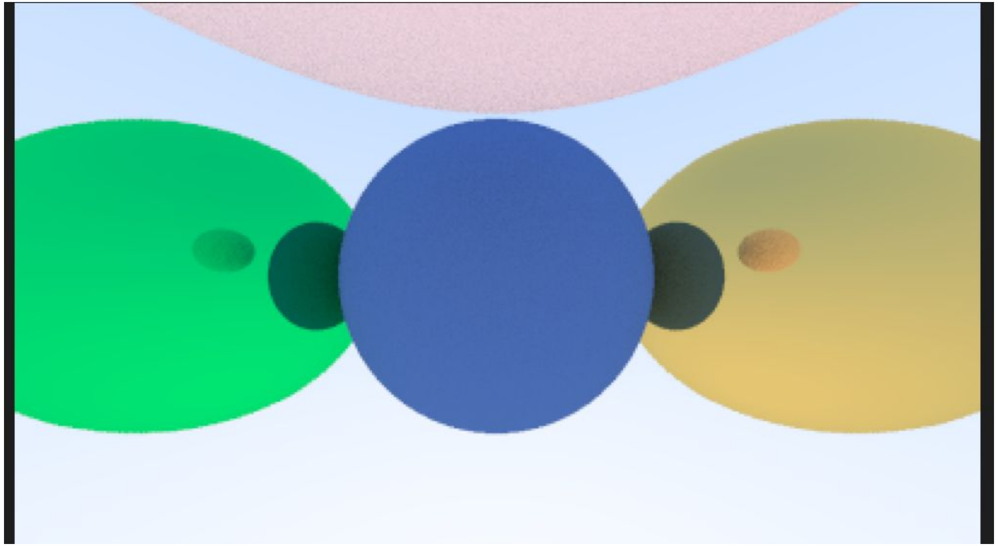
- Iterates through all objects in scene.
- Solve the quadratic equation to calculate points of intersection.
- Determines the closest valid intersection point.
- Calculate surface normal.

Scatter

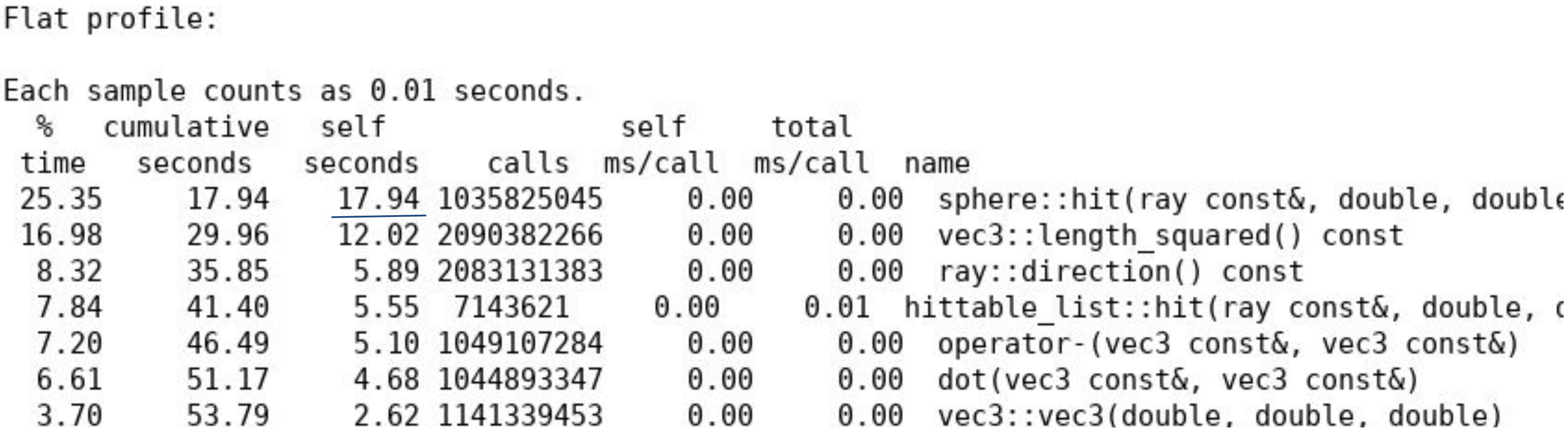
- Material dependent light interaction-
- Lambertian spheres- Scattered direction is a random offset from normal direction.
- Metallic Spheres- Scattered direction is determined by law of reflection.

SW Baseline

- Baseline SW code runs all functions on the arm processor.
- Time-
  - 49.36 s (Float)



Pre-Implementation: Profiling



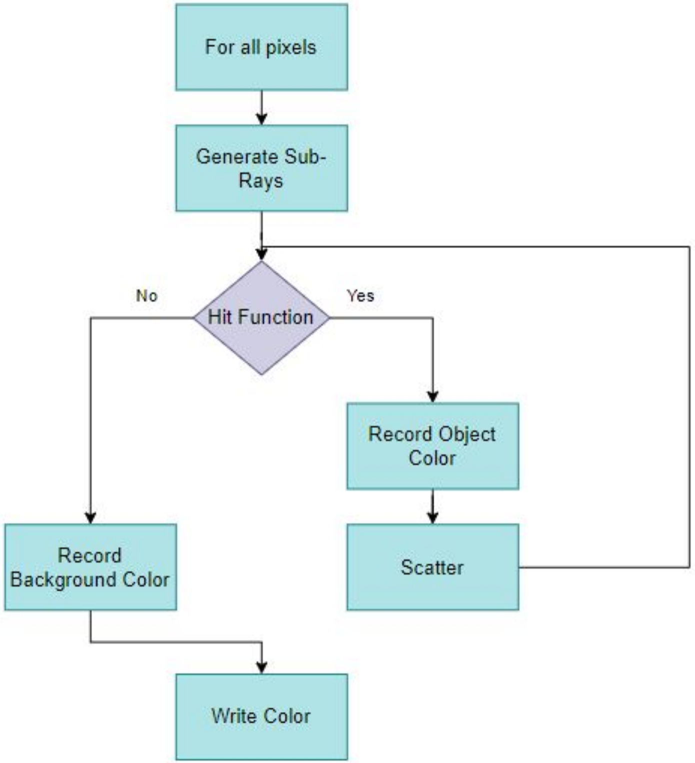
```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 25.35     17.94    17.94 1035825045     0.00     0.00  sphere::hit(ray const&, double, doubl
 16.98     29.96    12.02 2090382266     0.00     0.00  vec3::length_squared() const
  8.32     35.85     5.89 2083131383     0.00     0.00  ray::direction() const
  7.84     41.40     5.55  7143621     0.00     0.01  hittable_list::hit(ray const&, double,
  7.20     46.49     5.10 1049107284     0.00     0.00  operator-(vec3 const&, vec3 const&)
  6.61     51.17     4.68 1044893347     0.00     0.00  dot(vec3 const&, vec3 const&)
  3.70     53.79     2.62 1141339453     0.00     0.00  vec3::vec3(double, double, double)
```

Pre-Implementation: Quantization

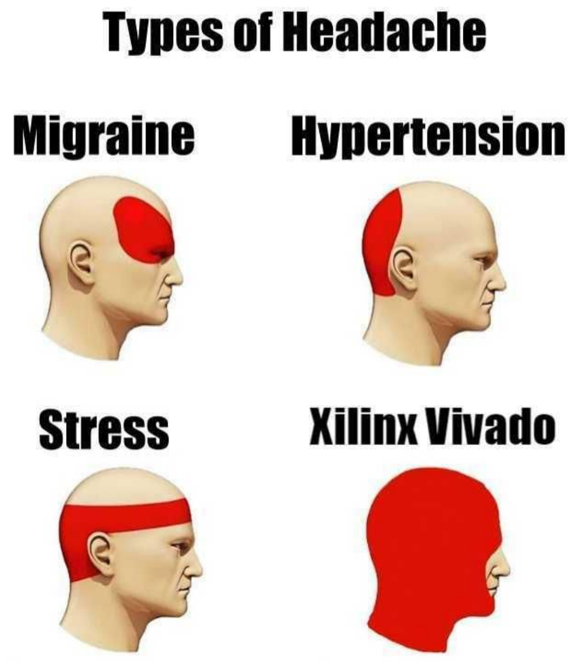
- Hit function in SW - Converted from floating point to fixed point.
- Scale Factor of the form 2^N.
- Downscaled the intermediate values dynamically to prevent overflow.
- 49.1s (Float) ⟶ 71.21 s(Quantized)

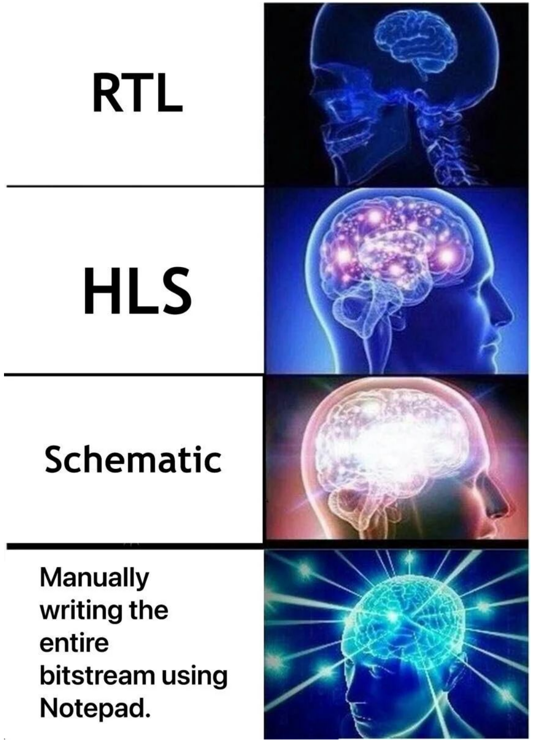
Implementation-1

- Implemented a Quadratic Equation Solver in Verilog (Single Object).
- Running at 250 MHz.
- Time- 145.14s

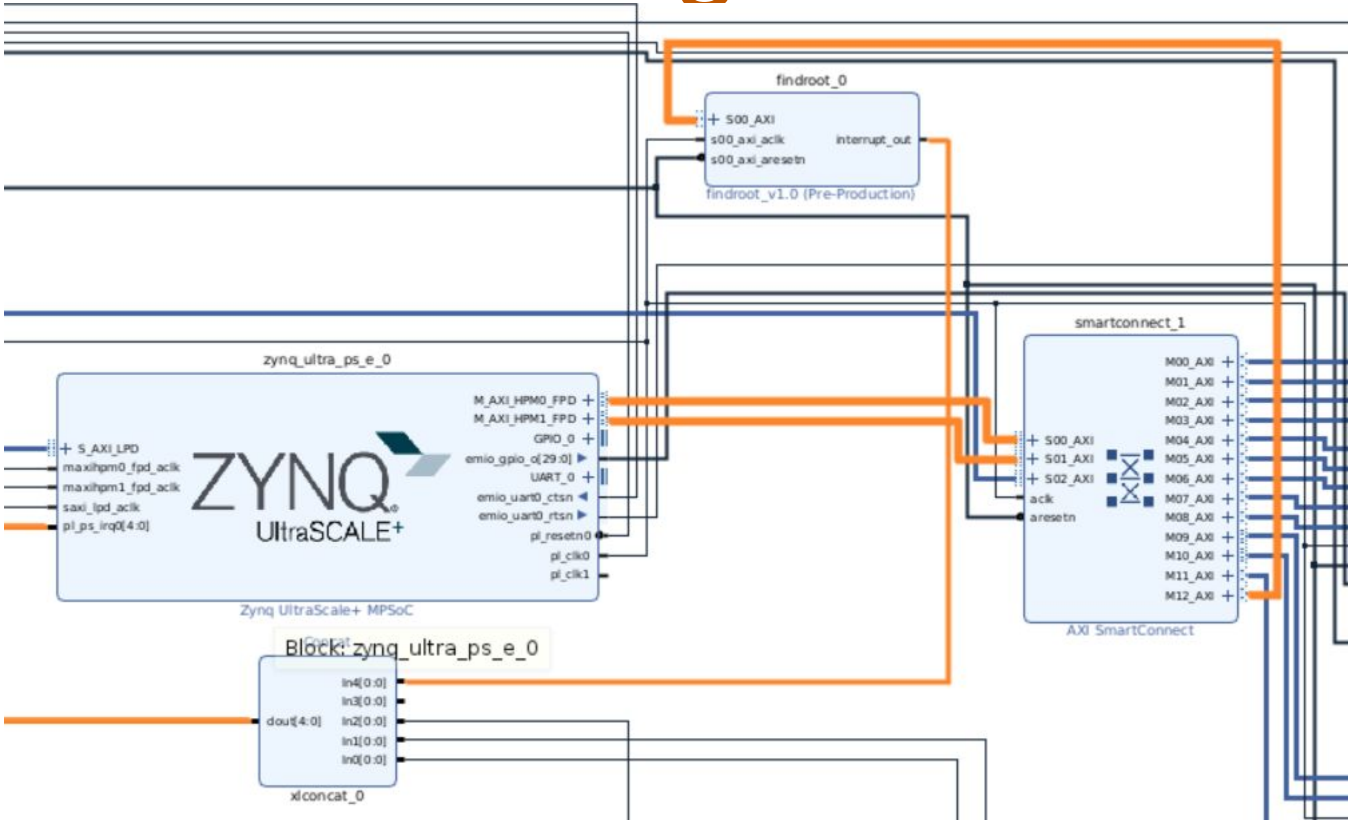


Vivado Block Diagram





Vivado Block Diagram



FSM HW design

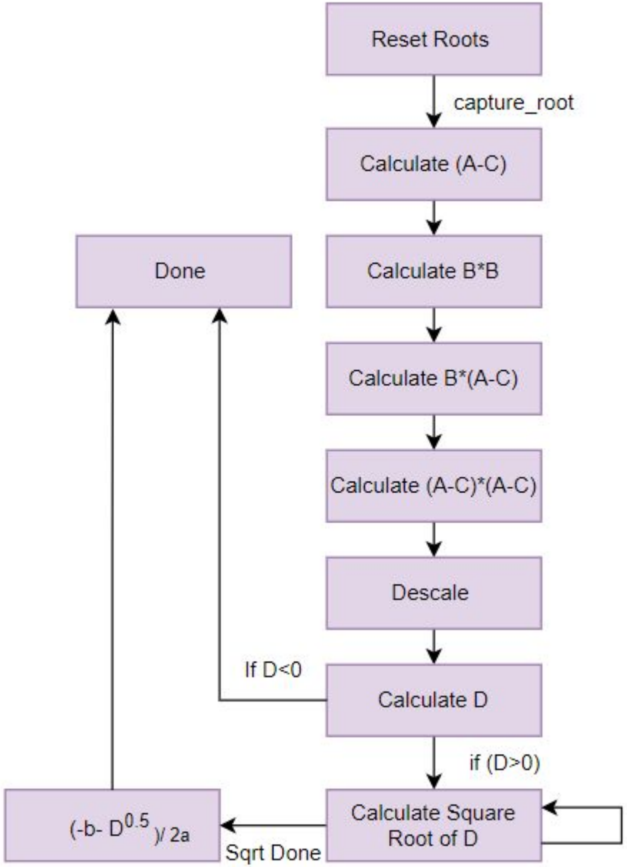
- Essentially calculates solution for-

$$t^2*(B\bullet B)+ 2t*((B)\bullet(A - C)) + (A - C)\bullet(A - C)$$

$$(-b- D^{0.5})/ 2a$$

Where $D= b^2 - 4a.c$

- Simplified FSM (Takes about 45 cycles (15+30) in the worst case).



Reset Roots

capture_root

Calculate (A-C)

Calculate B*B

Calculate B*(A-C)

Calculate (A-C)*(A-C)

Descale

Calculate D

If D<0

Done

if (D>0)

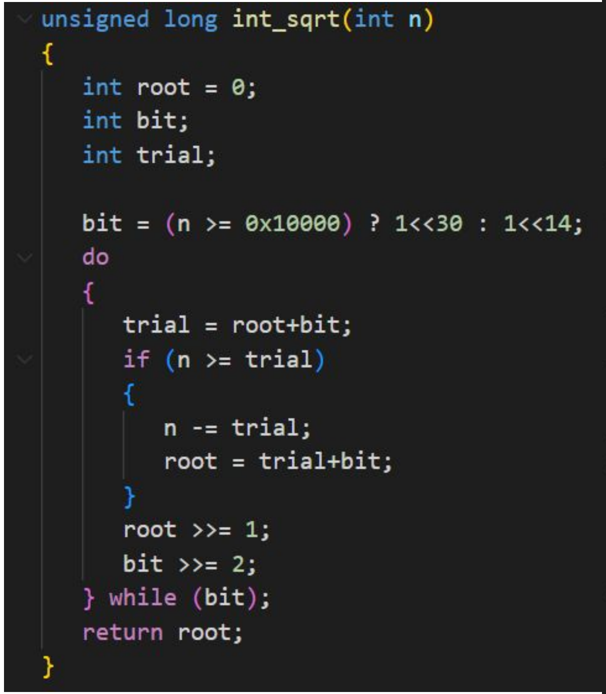
Calculate Square Root of D

Sqrt Done

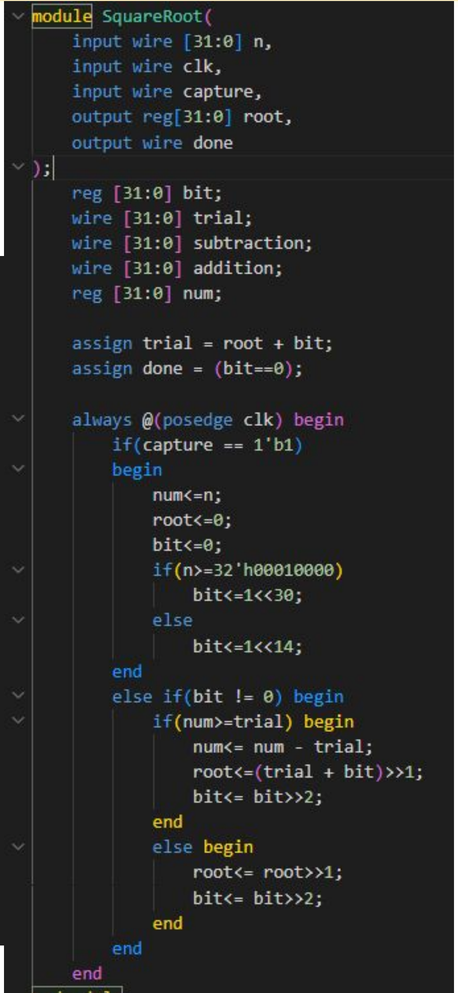
$(-b- D^{0.5})/ 2a$

Square Root Calculation

- C code provided by Dr.McDermott as part of Lab2
  - Converted C algorithm to Verilog.
- Takes about 15/30 cycles to calculate square root.



```c
unsigned long int_sqrt(int n)
{
    int root = 0;
    int bit;
    int trial;

    bit = (n >= 0x10000) ? 1<<30 : 1<<14;
    do
    {
        trial = root+bit;
        if (n >= trial)
        {
            n -= trial;
            root = trial+bit;
        }
        root >>= 1;
        bit >>= 2;
    } while (bit);
    return root;
}
```



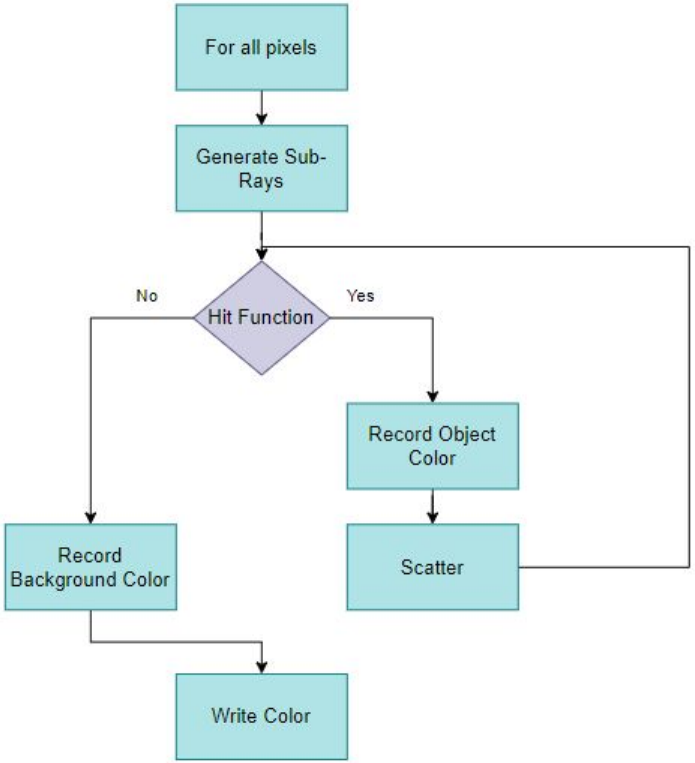
```verilog
module SquareRoot(
    input wire [31:0] n,
    input wire clk,
    input wire capture,
    output reg[31:0] root,
    output wire done
);
    reg [31:0] bit;
    wire [31:0] trial;
    wire [31:0] subtraction;
    wire [31:0] addition;
    reg [31:0] num;

    assign trial = root + bit;
    assign done = (bit==0);

    always @(posedge clk) begin
        if(capture == 1'b1)
        begin
            num<=n;
            root<=0;
            bit<=0;
            if(n>=32'h00010000)
                bit<=1<<30;
            else
                bit<=1<<14;
        end
        else if(bit != 0) begin
            if(num>=trial) begin
                num<= num - trial;
                root<=(trial + bit)>>1;
                bit<= bit>>2;
            end
            else begin
                root<= root>>1;
                bit<= bit>>2;
            end
        end
    end
```

Application Code

- Send scene, ray origin and direction information to HW.
- Detect whether a ray hits a particular object in HW. This is done iteratively for all objects and all rays in SW.
  - #Transfers/HW call = 11 integer inputs , 2 integer outputs
- Communicated with slave registers directly using mmap.
- Used Interrupts to detect completion of root calculation.
  - #Interrupts = 400*225*100*10*4 = 3.6e8 (worst case)!

Implementation-2

- Implemented a Quadratic Equation Solver in HLS (Single Object).
- Time- 634.02s
  - Note that this was synthesized at 100MHz.

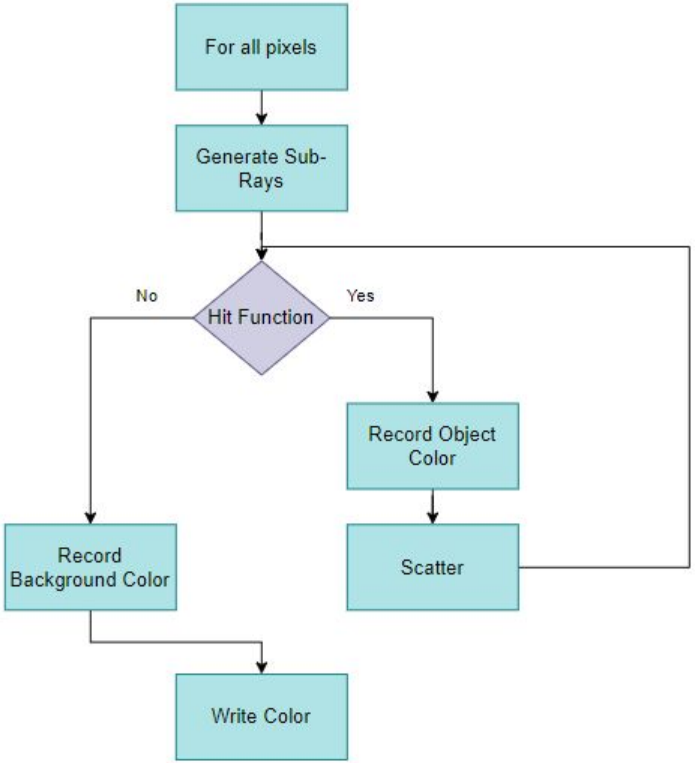


Interlude: High Level Synthesis

- Similar to writing C/C++ Code.
- Can use #pragmas to define AXI ports and perform code optimizations like loop unrolling and pipelining.
- C Simulation (with C/C++ testbench)- Used to verify correctness.
- C Synthesis- To generate verilog code.
- C/RTL Co-simulation- To test the synthesized code with the same C testbench.
- Export RTL- To export the final code as an IP block (for use in Vivado).

Implementation-3

- **Multi-Object** Quadratic Solver.
- Parallelized implementation at 250 MHz.
- Time- 176.73s(polling)



Salient Features

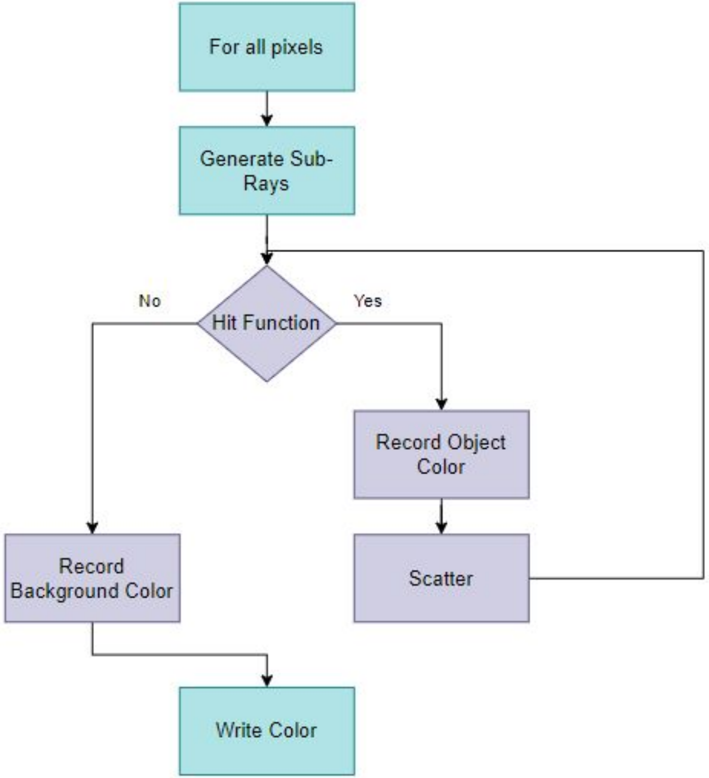
- Vivado design remains unchanged.
- The SW code had to be refactored, which slowed it down.
  - HW implementation still gives overall speedup.
- Reduced #Interrupts by a factor of 4.
- Reduced redundancy in data transfers (scene and geometry information).
- Experimented with polling - Since processor has no work.

Implementation-3 Time Summary

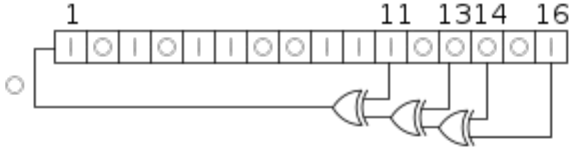
- Overall reducing the number of data transfers and interrupts led to a time of 273.92s
- Changing to polling reduced it to 176.73s

Implementation-4

- Offloaded Ray Scattering operation to HW.
- Time- 35.42s



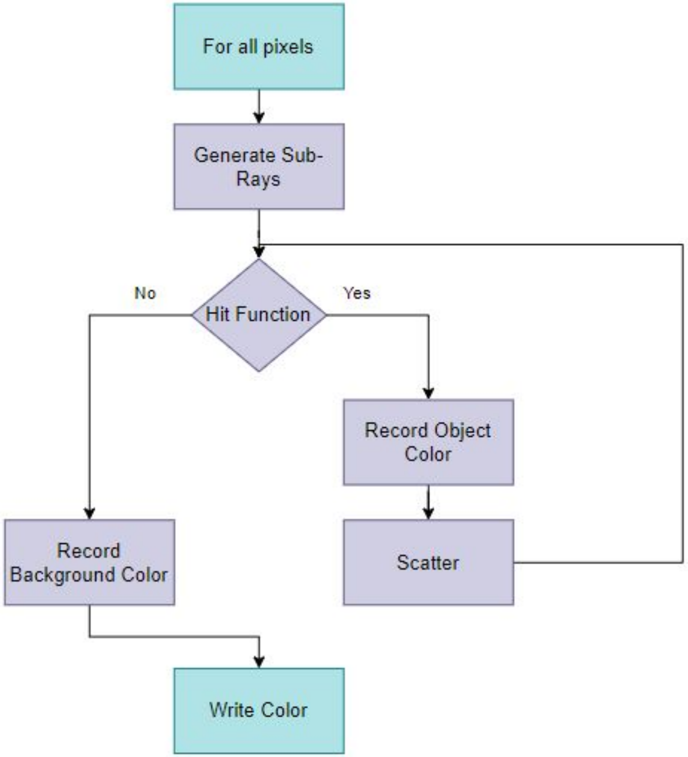
Salient Features



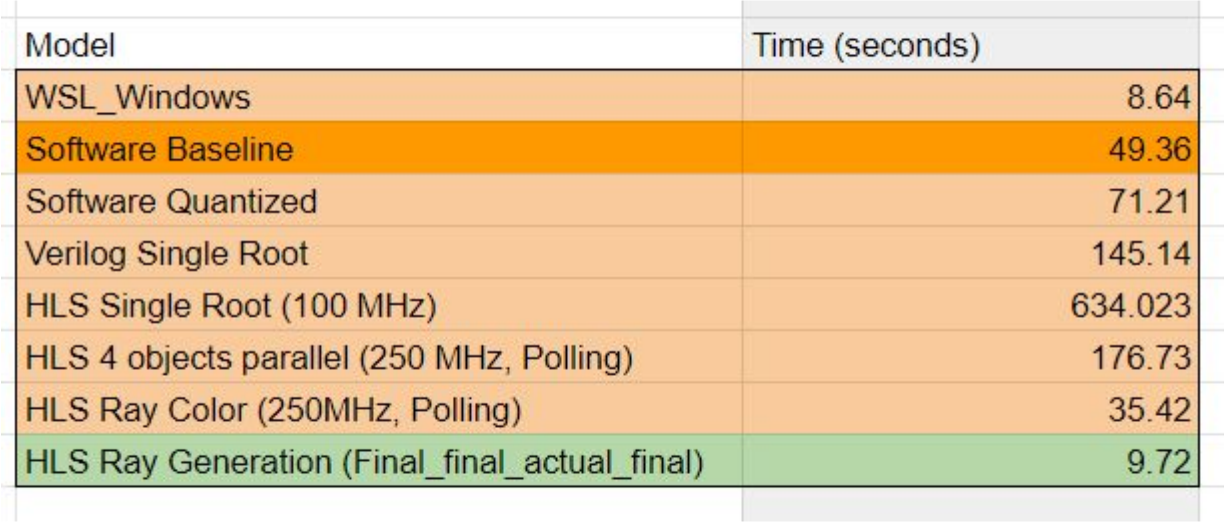
- Vivado design remains unchanged.
- Implemented a 32-bit LFSR to emulate random scattering in Lambertian surfaces.
- Represented colors using fixed point integers.
- Reduced #Interrupts by a factor of 10.
- Further reduced data transfers (ray origin information).

Implementation-5

- Generated sub-rays (100) in HW.
- Leveraged previously developed LFSR to generate sub-rays.
- Reduced #Interrupts by a factor of 100.
- Time- 9.72s

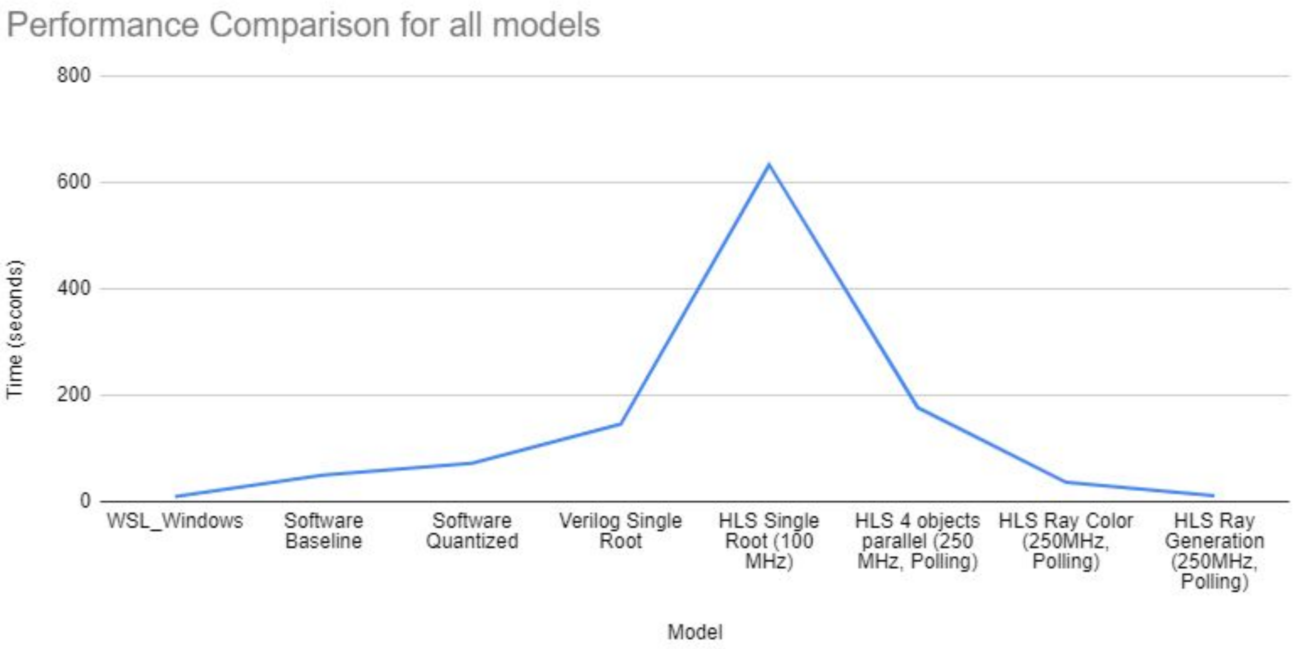


Performance Comparison



Model	Time (seconds)
WSL_Windows	8.64
Software Baseline	49.36
Software Quantized	71.21
Verilog Single Root	145.14
HLS Single Root (100 MHz)	634.023
HLS 4 objects parallel (250 MHz, Polling)	176.73
HLS Ray Color (250MHz, Polling)	35.42
HLS Ray Generation (Final_final_actual_final)	9.72

Performance Graph



Debug Techniques

- Verilog Debugging Techniques:
  - Used File Handling to verify functionality for a large number of test cases.
  - Registers for debug
- Modified HLS Files into a C++ Header File for functional verification.
- Visual Debugging
- GDB

Scope for Future Work

- Expand to cover more shapes/materials.
- Scale algorithm for more number of objects (more parallelization).
- Implementing techniques like BVH in hardware.
- Explore Texture Mapping.
- Parallelize the verilog code.

References

- https://raytracing.github.io/books/RayTracingInOneWeekend.html
- https://www.scratchapixel.com/index.html
- https://github.com/ssloy/tinyrenderer/wiki

- These are wonderful references to understand how graphics works from first principles, please do have a look at it!

Thank Yyou

- Prof. Mark McDermott
- TA- Abhijjith Venkkateshraj

Demo