Ray Tracing Acceleration on FPGA

Varun Saxena,
Santhana Bharathi Narasimmachari,
Sandilya Balemarthy

Course: ECE 8893 (Parallel Programming for FPGAs)
Advisor: Prof. Callie Hao

Georgia Tech

Goals

- Develop a ray-tracing accelerator for graphics rendering on Pynq Z2 FPGA.

- Main milestones:
  - Develop C++ model
  - Develop C model
  - Optimize for resource usage and latency using HLS for FPGA

Georgia Tech

Ray Tracing Introduction

- Ray Tracing is a rendering method used to generate realistic high-quality images by simulating interaction of light rays with objects in a scene.

- It can accurately portray advanced optical effects, such as reflections, refractions, and shadows, but at a greater computational cost and rendering time than other rendering methods.

- It is computationally intensive as ray-generation and intersection procedures are done for each individual pixel. This brings about a scope for parallelization opportunities.

Input Format

- .geo format

```
6
4 4 4 4 4 4
0 1 2 3 0 4 5 1 1 5 6 2 0 3 7 4 5 4 7 6 2 6 7 3
-1 1 1 1 1 1 1 1 -1 -1 1 -1 -1 -1 1 1 -1 1 1 -1 -1 -1 -1 -1
0 0 1 0 0 1 0 0 1 0 0 1 0 1 0 0 1 0 0 1 0 0 1 0 0 0 -1 0 0 -1 0 0 -1 0 0 -1 0 -1 0 0 -1 0 0
0.375 0.625 0.625 0.375 0.375 0.625 0.625 0.375 0.375 0.625 0.625 0.375 0.375 0.625 0.625 0.
```

Number of faces

Face Index Array

Vertex Index Array

Vertex Coordinates

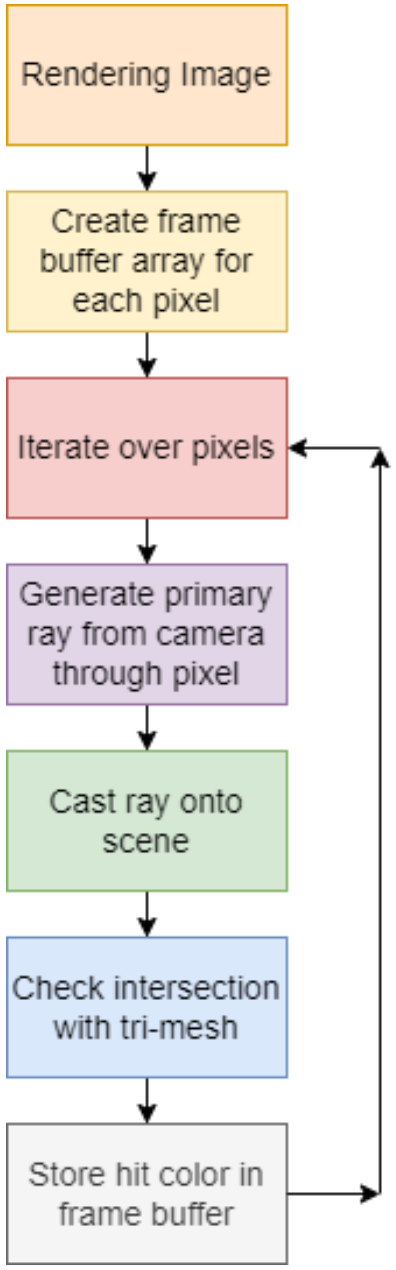
Normal direction per face

Texture Coordinates per face

Georgia Tech

C++ modelling

- Parse .geo file object and create mesh object
- Utilized classes and structures for storing vectors, 4x4 matrices, image objects as polygon meshes
- Rendering the image:
  - Create frame buffers for each pixel
  - Iterate over each pixel
  - Generate a primary ray passing through each pixel and from the camera
  - Cast Ray onto the scene
  - Check whether ray hits any points on/within the polygon mesh
  - Store the hit color into the framebuffer.



Reference: https://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-polygon-mesh/ray-tracing-polygon-mesh-part-2

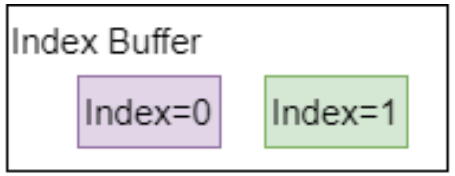
Georgia Tech

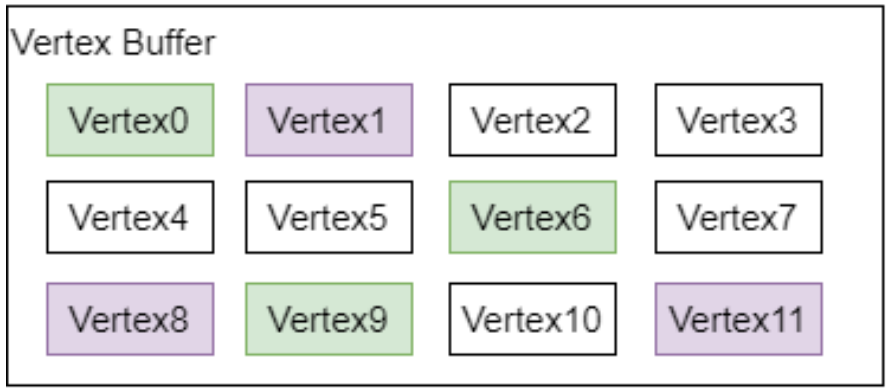
C modelling

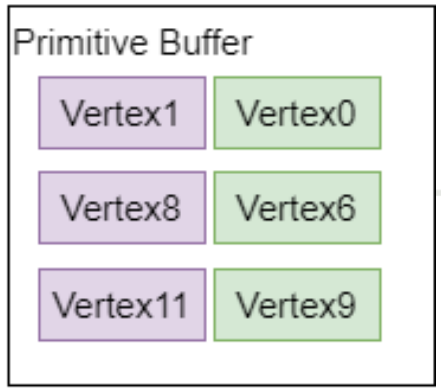
- Removed pointer to non-atomic data types (HLS Compliant)
- Partitioned to separate IPs (render function)
- Removed classes that stored vectors, objects, 4x4 matrices and converted them to float variables
- Added a custom 'fixed_t' data type that could be toggled for floating point and fixed point

Data Reordering

- Vertex buffer is optimized for data storage to reduce duplicate data
  - Con: Non-sequential memory access
- Primitive buffer is optimized for sequential storage
  - Con: Duplicate data
- HLS Arch: Convert vertex buffer into primitive buffer
  - Reason: Allows for efficient use of array partitioning techniques and possibilities for tiling are easier

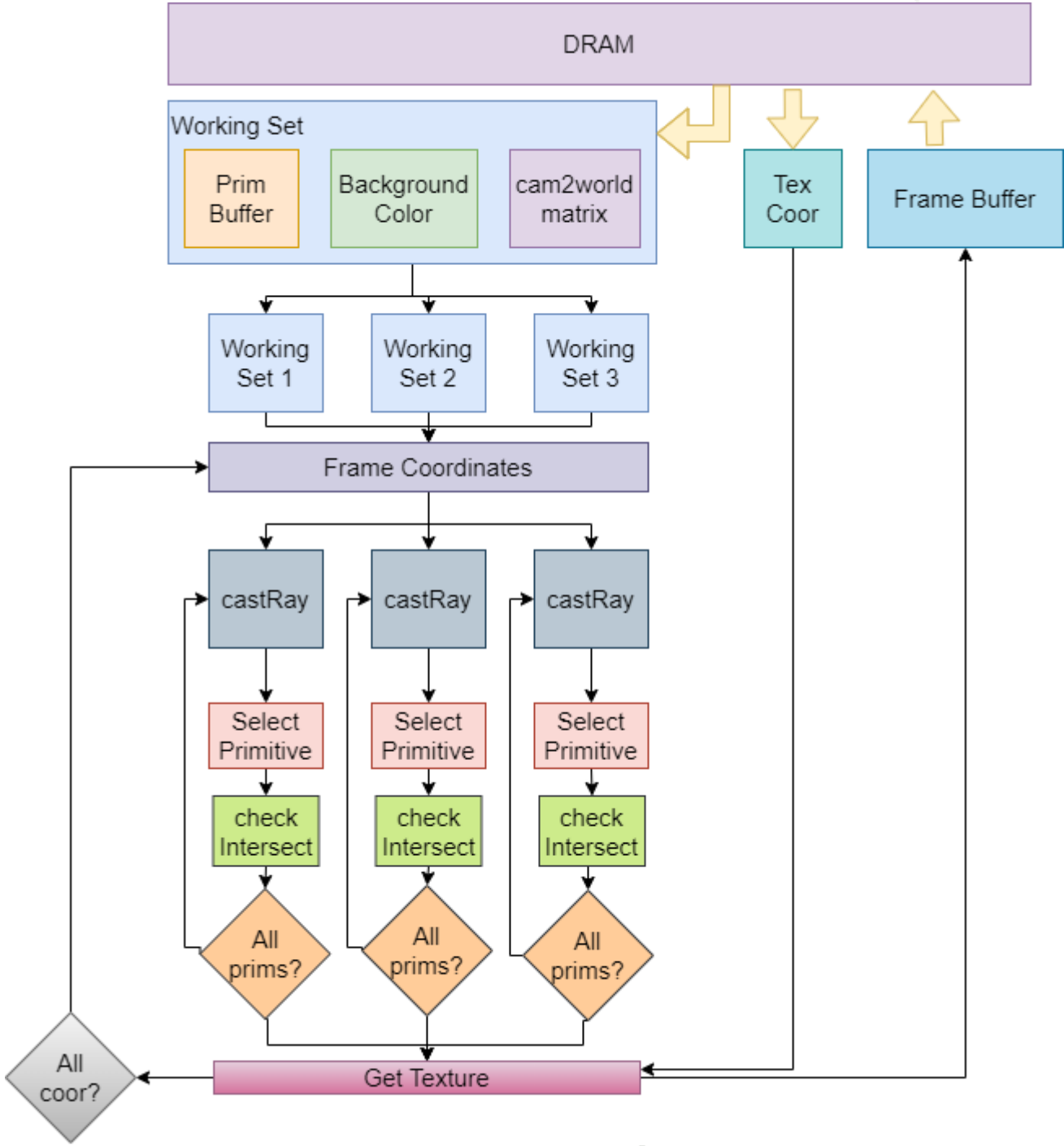






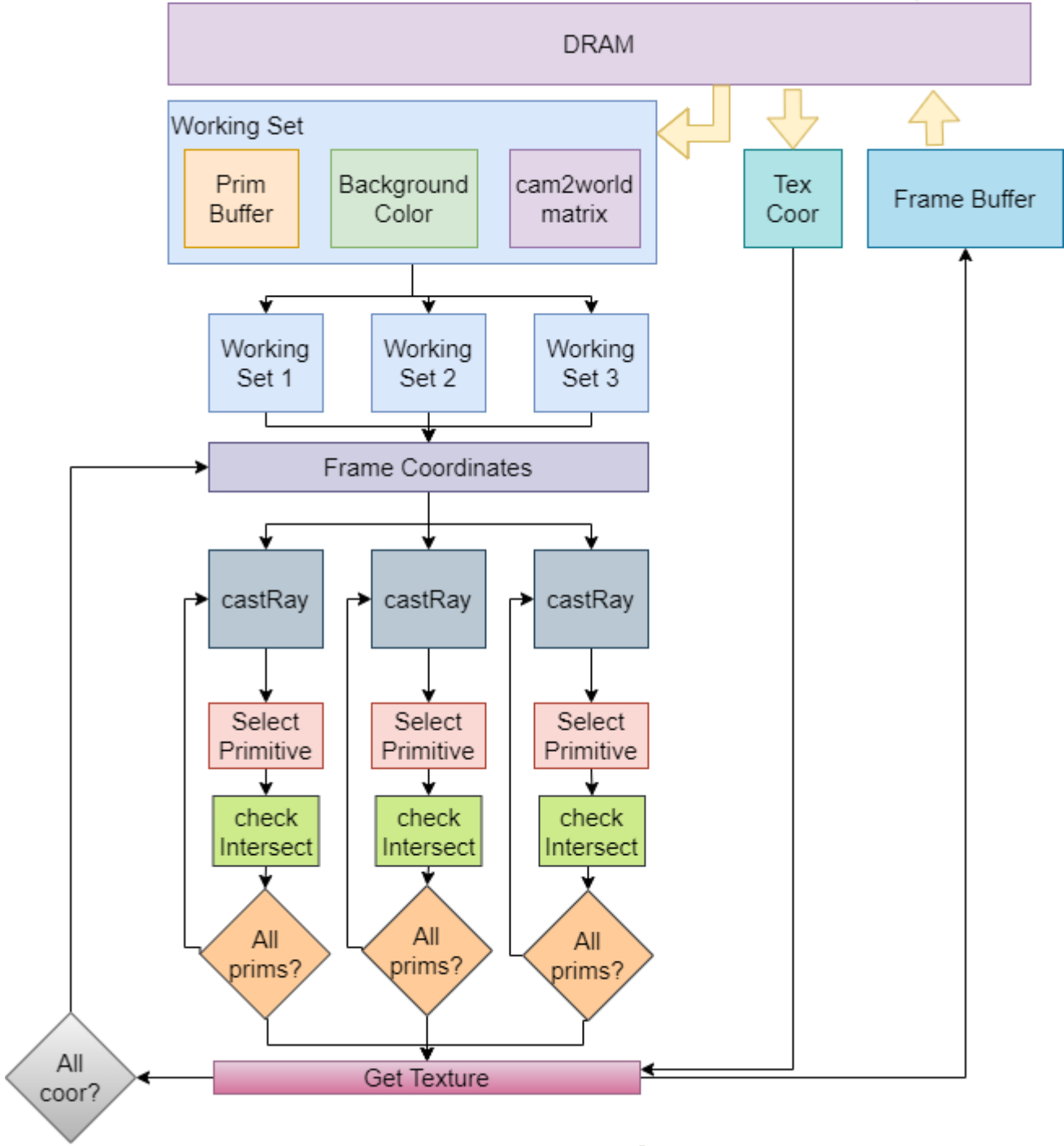
HLS Architecture (1)

- Steps:
  1. Read working set (WS) from DRAM
  2. Copy WS into 3 copies
     1. Choice of 3: Resource limits on Pynq Z2
  3. Iterate over frame coordinates
  4. Dispatch data into 3 lanes for ray tracing pipeline
  5. Cast ray for the coordinates onto frame



HLS Architecture (2)

- Steps:
  6. Iterate over primitives
  7. Check for intersections between cast ray and primitive
  8. Store intersection data and check next primitive (step 6)
  9. When all 3 lanes are done: Use the hit data to get hit color from the texture buffer.
     - Write data to frame buffer held in DRAM
  10. Check for next coordinate in frame (step 3)

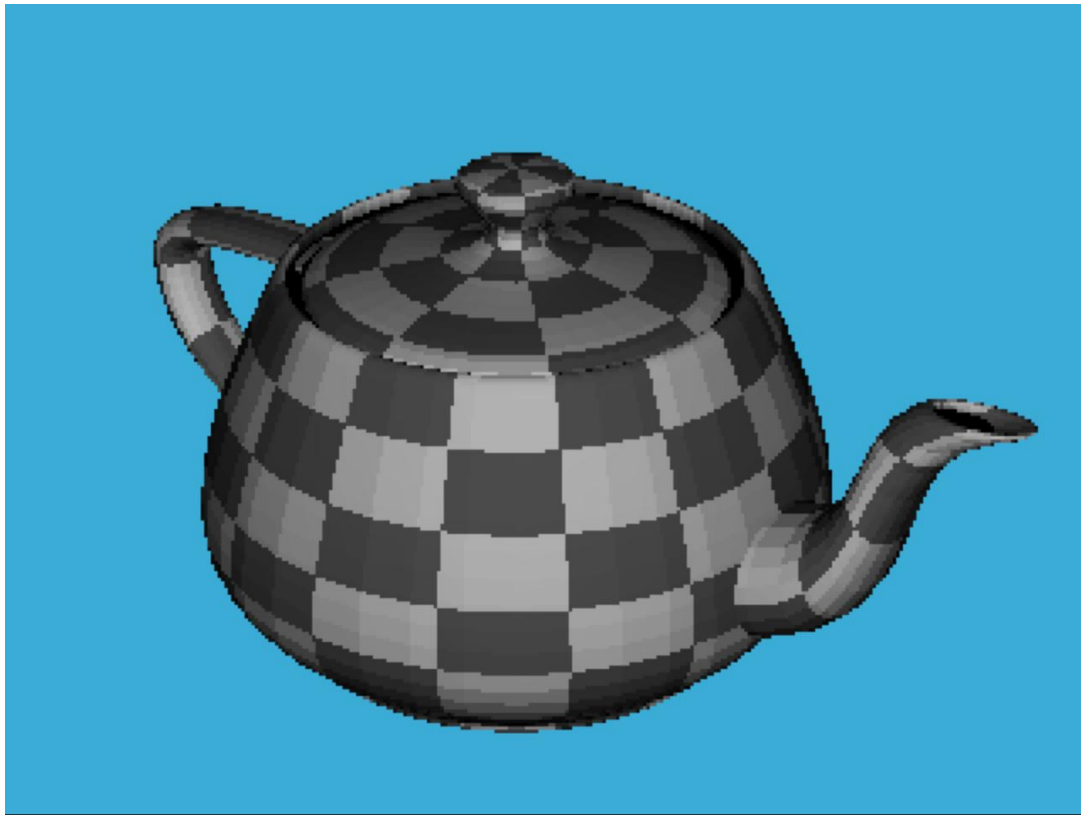


Design Choices/Challenges

- Sqrt support for fixed point
  - Not supported in library
  - Xilinx has a cordic algorithm for sqrt in their github repo
- Lack of precision issues in division
  - Generates 0 in the divisor leading to errors. Handled by wrapping the division to handle exceptions
- Over-use of DSP blocks due to loops unrolling
  - Controlled the unrolling amount in specific targets
- BRAM port limits for parallel casting
  - Created multiple physical copies for the BRAM data into arrays
  - Data specific to each cast is independent of the other
- Balancing BRAM vs DRAM usage
  - Stored the Primitive buffer in the BRAM
  - Kept the texCoordinates and FrameBuffer in the DRAM as accessed once per cast

Results – Rendered Image

- C code optimizations (runtime)
  - Initial setup: 75m 14.3s
  - Optimized C code: 21.886s
- Image Characteristics:
  - # faces: 3200
  - # triangles: 6320
  - Width: 320
  - Height: 240
- Data Characteristics:
  - Fixed point total width: 32 bits
  - Integer width: 16 bits



Georgia Tech

Optimization Techniques

Optimization	Latency	Speedup	BRAM_18K	DSP	FF	LUT
Floating point	2400s	-	0 %	52 %	25 %	33 %
Base: Fixed point	97.33s	-	0 %	8 %	9 %	26 %
• Store in BRAM • Array Partition #1	39.37s	2.47x	15%	25 %	29 %	61 %
• Optimized sqrt • Targeted loop unroll • Increased precision	6.651s	14.6x	23 %	136 %	22 %	42 %
• Targeted loop unroll • Array partitioning #2	4.993s	19.5x	31 %	142 %	19 %	37 %
• Inlining lib functions • Reordering BRAM structure • DSP usage: Removed modulo connections, removed unrolling • Array partitioning #3	5.012s	19.4x	46 %	93 %	23 %	42 %
• Reduced data width • Added techniques for sqrt and div handling for precision loss	4.964s	19.6x	32 %	34 %	15 %	26 %
• Dataflow optimization • Parallel casting of 3 rays	1.665s	58.46x	91 %	92 %	31 %	52 %

Q & A

Georgia Tech

Thank You

Georgia Tech