Evangelos Kolovos
Yuriy Osipyan
Supervisor: Mihalis Psarakis

Team Number: AOHW25_256

3D Software Renderer with FPGA Hardware Acceleration on Zynq-7000

Abstract

Real-time rendering on embedded SoCs is constrained by CPU performance and external memory bandwidth. We present a reusable 3D graphics pipeline on the ZYBO Z7-10 that combines a flexible software renderer on the ARM Cortex-A9 with FPGA accelerators for vertex and fragment processing. Our **Vertex Transform Unit (VTU)** offloads 4×4 matrix-vector transforms using a seven-stage pipelined microarchitecture; an in-progress **Fragment Unit** implements incremental half-space rasterization and span packing, with Z-test, fragment shading, and writeback postponed due to project deadline. On five benchmark meshes, the VTU delivers **~44.4 Mvectors/s**, a **6.9×** vertex-only speedup over software and an **average 1.62×** end-to-end speedup when rasterization remains in software. We analyze bandwidth limits, outline the remaining fragment stages, and propose tiling and span-based bursting to approach 720p–1080p real-time targets on ZYBO.

1. Project Overview

This project demonstrates the integration of a software renderer and hardware acceleration on the ZYBO Z7-10 SoC. The primary goal is to create a reusable realtime 3D graphics renderer by accelerating key stages of the graphics pipeline using custom hardware components: the **Vertex Transform Unit (VTU)** and **Fragment Unit**.

The system utilizes the **ARM Cortex-A9 processor** for managing rendering logic and mesh importing, while offloading computationally intensive tasks, such as **matrix-vector multiplication**, **rasterization**, and **fragment shading**, to the **FPGA**.

On **Zynq-class** devices, offloading linear-algebra-heavy stages (transforms, interpolation) to fabric IP often yields large speedups, but the overall throughput is ultimately constrained by DDR bandwidth and burst efficiency. Our design follows this principle while retaining a simple command/data interface.

2. Project Objectives

- **Software Rendering**: Develop a software renderer capable of parsing 3D models, managing 3D scenes, performing vertex transformations, and rendering models with basic shading techniques, such as *diffuse* and *specular* lighting using the **Blinn-Phong model**.
- **Vertex Processing with VTU**: Implement the ***Vertex Transform Unit*** to accelerate vertex processing by offloading computationally expensive matrix-vector multiplications from the ***ARM Cortex-A9 processor*** to the FPGA. This reduces CPU load and speeds up key operations like vertex positioning and normal transformations.
- **Fragment Processing with Fragment Unit**: Develop a ***Fragment Unit*** to hardware-accelerate rasterization, fragment shading, and texture sampling. This unit handles the pixel-level computations for *lighting*, *texture mapping*, and *color blending*, offloading the CPU and improving rendering performance.

3. Mesh Importer and Software Pipeline

The software renderer serves as the foundational component of the graphics pipeline, responsible for the basic operations required to render 3D scenes. While it provides flexibility and control, it is limited by the processing power of the CPU, making it unsuitable for real-time rendering in embedded systems with tight performance constraints.

Key tasks handled by the software renderer include:

- **3D Model Parsing**: The 3D model parsing process begins by importing **OBJ files**, which contain the model's vertex data and face definitions. The importer extracts and converts this data into an internal format for processing. While also handling triangulation, tangent generation for normal mapping, and index optimization for improved memory access. For the optimization step we used parts of the *meshoptimizer* library https://github.com/zeux/meshoptimizer.
- **Transformations**: After parsing the model data, the renderer applies 3D transformations to position the model within the scene. Transformations like *scaling*, *rotation*, *translation*, *camera view and projection* are performed on the model's vertices. These are represented as matrix operations, where each is a matrix multiplication. This process is computationally intensive when done in software, especially for complex models.
- **Rasterization**: After transforming the model, the rasterizer converts triangles into pixel data for display using the **incremental half-edge rasterization** technique. This highly parallelizable algorithm works by incrementally calculating edge equations and filling in the pixels of the triangle in a systematic way. It progressively steps across the edges, ensuring smooth interpolation of attributes like position, color, and texture coordinates.
- **Shading**: Following rasterization, shading is applied to simulate how light interacts with the model's surfaces. We used the **Blinn-Phong model** to compute both diffuse and specular reflections, considering the light source's position and the viewer's perspective to determine how light is scattered and reflected across the surface. Additionally, normal mapping and diffuse texturing are applied to enhance the surface detail.
- **Output**: The rendered output is displayed on an **HDMI monitor** using the **Digilent RGB2DVI module** and **dynamic clock** blocks.

While the software renderer is functional and provides the flexibility to implement various rendering algorithms, it is inherently limited by CPU performance. The computational load of handling large 3D models, complex lighting calculations, and real-time scene rendering is too heavy for the CPU to handle efficiently, especially in embedded systems with limited resources. As a result, the software renderer struggles to achieve the high frame rates required, leading to performance bottlenecks and delays.

Thus, while it serves as the base, hardware acceleration is needed to improve the overall performance and meet real-time rendering requirements.

4. Hardware Acceleration: Vertex Transform Unit (VTU)

The **VTU** plays a key role in accelerating the software renderer. **Vertex transformations** are a bottleneck in 3D rendering. By offloading these operations to hardware, the VTU accelerates the process.

4.1 VTU Design

The **VTU** is a **4×4 matrix-vector multiplication** unit that uses **AXI-Stream** and **AXI4-Lite** interfaces to communicate with the ARM processor.

- **AXI4-Lite Interface**: Loads the **4×4 transformation matrix** into the VTU. The matrix defines the transformations to be applied to the model's vertices.
- **AXI-Stream Interface**: Streams **vertex data** in **fixed point (q16.16)** representation to the VTU. The VTU performs the matrix-vector multiplication and outputs the transformed vertices in **32-bit floating point** format.

4.2 Seven-Stage Pipeline

The VTU operates through a **seven-stage pipeline**:

1. **Stage 1**: Parallel multiplication of matrix and vector elements.
2. **Stage 2a & 2b**: Row-wise partial summation followed by final row summation.
3. **Stage 3**: **Fixed-point resizing** to ensure data fits within the desired range.
4. **Stage 4/5/6**: Conversion from fixed-point to **32-bit floating-point format**, ensuring compatibility with further processing stages like shading.

This pipeline structure allows the hardware to efficiently process each **vertex transformation**, reducing the need for repeated floating-point operations in software.

4.3 Performance Characteristics

The **VTU pipeline** operates reliably at a **140 MHz clock frequency**, processing **1 vector per cycle** when the pipeline is full. Given this, the following theoretical performance metrics are calculated:

- **Theoretical Bandwidth**: The **HP AXI port** operates with a **64-bit data width**. At a **140 MHz clock frequency**, the data transfer rate is:

$$\text{Bandwidth} = 140\,\text{MHz} \times \left(\frac{64\,\text{bits}}{8}\right) = 1120\,\text{MB/s}$$

- **Compute Capability**: The VTU can perform 1 matrix-vector multiplication per cycle. With a 140 MHz clock, the system achieves a theoretical peak compute rate of:

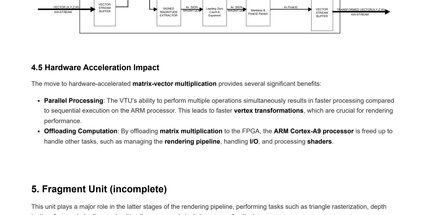
$$\text{Compute Capability} = 140\,\text{MHz} \times 1\,\text{vector/cycle} = 140\,\text{MVectors/s}$$

- **Bandwidth Limitation**: Despite the high compute capability of 140 million vectors per second, performance is still **bandwidth constrained**. However, by using **two AXI HP Ports**, we can double the available bandwidth. Alternatively we can reduce compute capability to match bandwidth capability to save FPGA resources (namely DSP's).

Resource Utilization (Zybo z7-10)

Resource	Usage
LUTs	2372
Registers	1984
Block RAMs	0
DSP Slices	64

Figure 1 — VTU Microarchitecture



4.5 Hardware Acceleration Impact

The move to hardware-accelerated **matrix-vector multiplication** provides several significant benefits:

- **Parallel Processing**: The VTU's ability to perform multiple operations simultaneously results in faster processing compared to sequential execution on the ARM processor. This leads to faster **vertex transformations**, which are crucial for rendering performance.
- **Offloading Computation**: By offloading **matrix multiplication** to the FPGA, the **ARM Cortex-A9 processor** is freed up to handle other tasks, such as managing the **rendering pipeline**, handling **I/O**, and processing **shaders**.

5. Fragment Unit (incomplete)

This unit plays a major role in the latter stages of the rendering pipeline, performing tasks such as triangle rasterization, depth testing, fragment shading, and writing the processed pixels to memory for display.

The Fragment Unit receives a stream of triangle descriptors from the ARM Processing System, processes them through several stages, and writes the final output to the framebuffer region in DDR.

5.1 Fragment Unit Design

Implemented

- **Rasterizer:** The Rasterizes receives triangle descriptors from the Arm Processing System and performs incremental half-edge stepping to convert the triangles into pixel data. This process ensures that each pixel within the triangle is correctly calculated and interpolated based on its barycentric geometric properties.
- **Row Packer I:** Packs scanline spans into BRAM-resident contiguous buffers for efficient memory bursts downstream.

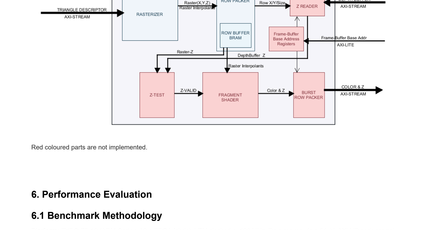
Remaining

- **ZTest:** Reads the old depth values from DDR Z-buffer region and compares them to the current Z values coming from the rasterizer. If the ZTest fails the raster is culled avoiding the heavy fragment processing stage.
- **Fragment Processing:** In this stage, shading and texture mapping are applied. The Fragment Processing stage handles the application of lighting and texture sampling to calculate the final color of each pixel.
- **Row Packer II:** Once the fragment has been processed, another Row Packer prepares the final data for efficient memory bursts.
- **Writeback:** Finally, the processed pixel data is written into the framebuffer DDR region for display using a datamover.

5.2 Progress and Remaining Work

Unfortunately, we were unable to complete the **Fragment Unit** for the competition due to half the team abandoning the project. We have implemented the Rasterizer and Row Packer modules, which handle the initial stages of the fragment processing pipeline. However, the remaining stages—*ZTest, Fragment Processing, Contiguous Burst Packer, and Writeback*—still need to be completed.

Figure 2 — FPU High Level Design



Red coloured parts are not implemented.

6. Performance Evaluation

6.1 Benchmark Methodology

Platform: ZYBO Z7-10 (*ARM Cortex-A9 + FPGA fabric*), VTU clocked at 100 MHz. Data moves via a 64-bit AXI HP port; control via AXI4-Lite. HDMI scanout was disabled for these microbenchmarks where noted to isolate compute and memory effects.

Workloads: Five meshes (*HOUSE, TEAPOT, KNIGHT, STATUE1, STATUE2*). For each we measured three configurations per frame:

- **SW_VERT:** software vertex transforms only (disabled rasterization)
- **SW_VERT/RASTER:** software vertex + software raster, *640x480 resolution*
- **VTU:** hardware vertex transforms only (disabled rasterization)

Method: We measure time in microseconds per frame; throughput is computed from vertex counts. Each run warms up, then averages multiple frames to reduce variance.

6.2 Results

Model	Vertices	SW_VERT (µs)	VTU (µs)	Vertex Speedup (×)	SW Total (µs) *(SW_VERT/RASTER)*	VTU + SW Raster (µs)	End-to-End Speedup (×)	Raster Share (SW)
TEAPOT	5,014	4,024	472	**8.53**	38,880 (6,977/31,903)	32,375	**1.20**	82.1%

Model	Vertices	SW_VERT (µs)	VTU (µs)	Vertex Speedup (×)	SW Total (µs) *(SW_VERT/RASTER)*	VTU + SW Raster (µs)	End-to-End Speedup (×)	Raster Share (SW)
HOUSE	12,632	5,549	1,143	**4.85**	65,549 *(8,525/57,024)*	58,167	**1.13**	87.0%
KNIGHT	41,061	31,314	3,646	**8.59**	95,601 *(53,220/42,381)*	46,027	**2.08**	44.3%
STATUE1	590,793	262,359	52,049	**5.04**	505,405 *(315,274/190,131)*	242,180	**2.09**	37.6%
STATUE2	240,621	159,674	21,219	**7.53**	569,035 *(241,143/327,892)*	349,111	**1.63**	57.6%

Averages: SW vertex throughput ~**1.72 Mverts/s**, VTU ~**11.13 Mverts/s** (vertex-only **6.9×**). End-to-end speedup averages **1.62×** with software rasterization held constant.

6.3 Takeaways

- **VTU is bandwidth-limited.** The stable *11 Mverts/s* - *(704 MB/s)* across meshes indicates AXI ceilings (800 MB/s) dominate over VTU compute.
- **Raster dominates in full SW.** Scenes with high raster share (HOUSE/TEAPOT) see modest end-to-end gains; when vertex cost is comparable, total speedup reaches ~2×.
- **Contention.** Software vertex inside the full SW path is ~1.5× slower than SW-only, consistent with cache/DDR contention. VTU removes this interference.

7. Conclusion

This project although incomplete demonstrates the significant performance gains achievable by integrating hardware acceleration into a software-based rendering pipeline. By offloading the matrix-vector multiplication required for vertex transformations to the Vertex Transform Unit (VTU) implemented on the FPGA, the system achieves real-time rendering with higher performance. The combination of software flexibility and hardware acceleration offers a viable solution for embedded systems requiring efficient 3D graphics rendering.

References

- **meshoptimizer** https://github.com/zeux/meshoptimizer
- **Digilent RGB2DVI/ DynClock** https://github.com/Digilent/vivado-library/tree/master