3D Hardware Rasterizer

Sathvik Rajampalli (vr2618), Shlok Desai (sbd2150), Aarush Agarwal (aa5763),
Gianna Belmont (gb2973), Srika Chagarlamudi (sc5800)

The goal of this project is to implement a hardware rasterizer on the DE1-SoC FPGA that is capable of rendering simple 3D models on a VGA display. The 3D object will be represented by a collection of vertices and triangles. By applying geometric transformations and projecting the vertices into screen coordinates, triangles can be drawn on the display to produce the final image. Updating these transformations every frame allows the object to rotate and be viewed from different angles.

The system is divided into two main components. Software is responsible for handling user input and performing the geometric computations needed to transform the 3D models. The hardware component of the project is responsible for drawing the triangles that make up the image. Once the triangle vertices have been projected onto the screen, the rasterizer determines which pixels lie inside each triangle and writes them to a framebuffer that is displayed through the VGA output.

Triangle filling will use edge-equation rasterization method described by Pineda [1]. In this approach, each edge of a triangle defines a linear function that indicates which side of the edge a pixel lies on. A pixel is considered part of the triangle if it lies on the correct side of all three edges. Since these functions are linear, their values can be updated incrementally from one pixel to the next, allowing the hardware to determine whether a pixel belongs to a triangle using only simple addition and comparison, which makes the approach well suited for hardware acceleration.

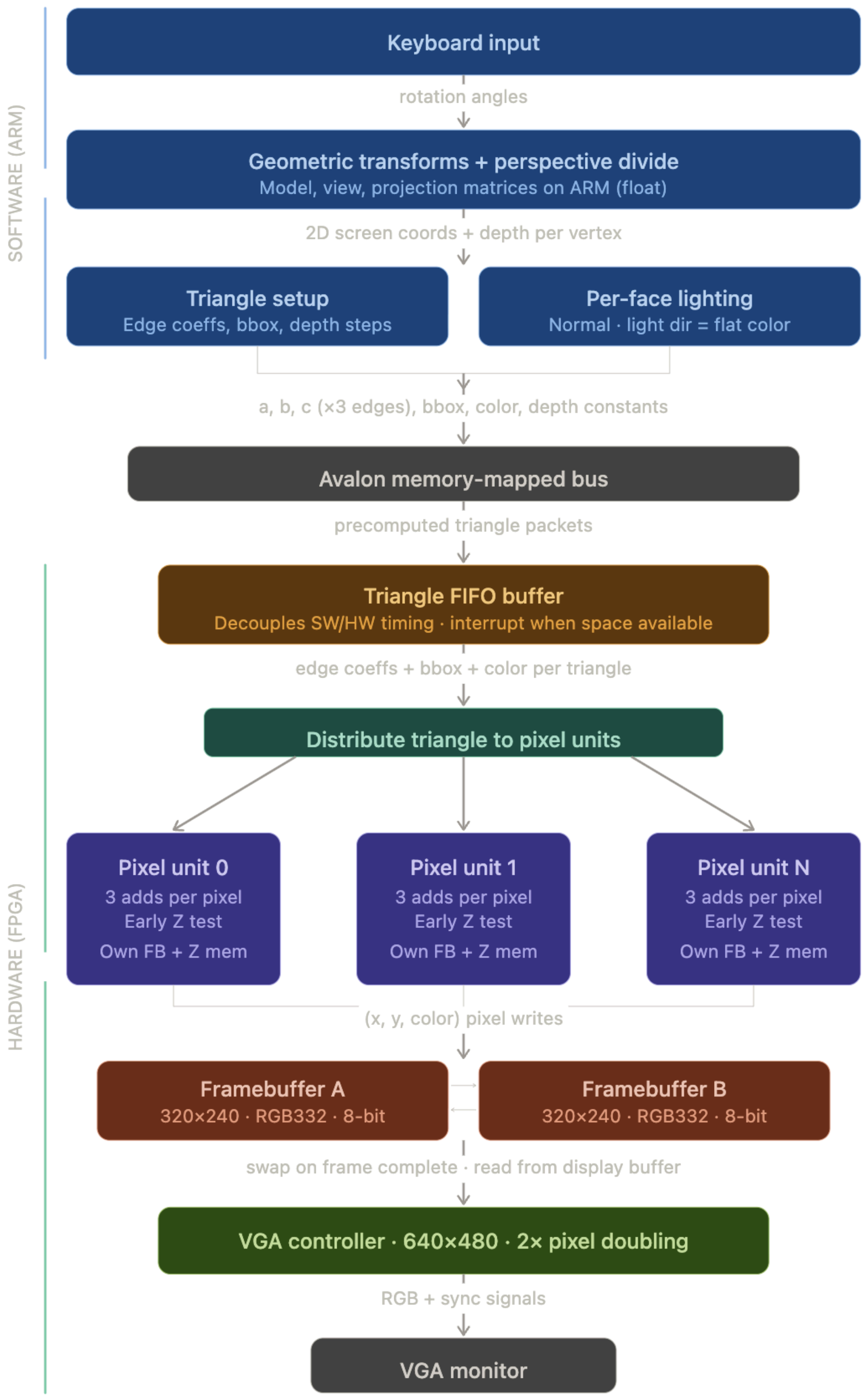
Hardware/Software Implementation:
The system is divided into a software component running on the ARM Cortex-A9 dual-core processor and a hardware component implemented on the FPGA.

Software is responsible for:

- **Model storage.** Software stores the 3D model as a triangle mesh made up of coordinates for vertices and faces.
- **User input.** Keyboard input is read by the ARM processor to control object rotation and view angle.
- **Geometric transformations.** Each frame, the processor applies model, view, and projection transformations to every vertex using floating-point arithmetic.
- **Perspective division.** Software performs perspective division to produce 2D screen coordinates. These calculations run once per vertex per frame.
- **Per-face lighting.** For each triangle, software computes a flat color based on the dot product of the face normal with a light direction vector. This gives different faces different brightness, producing visible 3D depth.
- **Triangle setup.** Software computes the edge function coefficients (a, b, c for each of the three edges), the bounding box (min/max x and y), initial edge function values at the

bounding box origin, and depth interpolation step values for each triangle. This moves all per-triangle math to software so the hardware only handles per-pixel work.

- **Triangle submission.** Precomputed triangle data is written to a hardware FIFO buffer over the Avalon memory-mapped bus. Software polls the FIFO status or responds to a low water mark interrupt to know when space is available, allowing software to prepare and submit triangles asynchronously while the hardware rasterizes previously submitted triangles.


SOFTWARE (ARM)
Keyboard input
rotation angles
Geometric transforms + perspective divide
Model, view, projection matrices on ARM (float)
2D screen coords + depth per vertex
Triangle setup
Edge coeffs, bbox, depth steps
Per-face lighting
Normal · light dir = flat color
a, b, c (×3 edges), bbox, color, depth constants
Avalon memory-mapped bus
precomputed triangle packets
HARDWARE (FPGA)
Triangle FIFO buffer
Decouples SW/HW timing · interrupt when space available
edge coeffs + bbox + color per triangle
Distribute triangle to pixel units
Pixel unit 0
3 adds per pixel
Early Z test
Own FB + Z mem
Pixel unit 1
3 adds per pixel
Early Z test
Own FB + Z mem
Pixel unit N
3 adds per pixel
Early Z test
Own FB + Z mem
(x, y, color) pixel writes
Framebuffer A
320×240 · RGB332 · 8-bit
Framebuffer B
320×240 · RGB332 · 8-bit
swap on frame complete · read from display buffer
VGA controller · 640×480 · 2x pixel doubling
RGB + sync signals
VGA monitor

The hardware component is responsible for rasterization.

- **Triangle FIFO buffer.** A small hardware FIFO sits between the Avalon bus and the rasterizer. Software pushes precomputed triangle data (edge coefficients, bounding box, color, depth constants) into the FIFO. The rasterizer pulls triangles from the other end. The FIFO issues an interrupt when it drains below a low water mark, signaling software that it can safely submit more triangles. This decouples software and hardware timing so neither side stalls waiting for the other.
- **Parallel pixel units.** Multiple pixel processing units operate simultaneously, each responsible for a partition of the screen. Each unit independently evaluates edge functions, performs depth testing, and writes to its own local framebuffer and Z-buffer memory. This avoids memory contention, no unit competes with another for memory access. The number of parallel units will be determined during design based on available FPGA resources.
- **Rasterizer.** Each pixel unit iterates over its portion of a triangle's bounding box. For each pixel, it evaluates the three edge functions using incremental addition, checks the sign of each to determine if the pixel is inside the triangle, and if so, proceeds to depth testing.
- **Z-buffer with early rejection.** A 16-bit Z-buffer stores the depth value for each pixel. The depth test is performed before the framebuffer write, if the new pixel is behind the existing pixel at that location, it is skipped immediately with no further work.
- **Framebuffer with double buffering.** The framebuffer is 320×240 with 8-bit direct color in RGB332 format (3 bits red, 3 bits green, 2 bits blue). Two framebuffers are maintained. The VGA controller reads from one while the rasterizer writes to the other. When a frame is complete, the buffers swap, ensuring the display always shows a fully rendered frame with no tearing or flicker.
- **VGA controller.** A VGA controller continuously reads the active framebuffer and generates a 640×480 VGA signal by doubling each framebuffer pixel in both dimensions.

Rendering Pipeline:
Software:

- Read user input and update rotation parameters
- Apply geometric transforms to all vertices (rotation, translation, projection)
- Perform perspective division to produce 2D screen coordinates
- For each triangle: compute edge function coefficients, bounding box, initial edge values, depth step values, and flat color from face normal lighting
- Push precomputed triangle data into the hardware FIFO

Hardware:

- Pull triangle data from the FIFO
- Distribute triangle to parallel pixel units
- Each pixel unit iterates over its portion of the bounding box
- For each pixel: incrementally evaluate edge functions (addition only), perform early depth test against local Z-buffer, write color to local framebuffer if pixel passes both tests

- VGA controller continuously scans the framebuffer to the display
- Swap framebuffers at end of frame

Milestones:

1. Software prototype in C/Python. Reference images and test vectors.
2. Single pixel unit rasterizer working in simulation. One triangle, flat shaded, verified against test vectors. Memory interface designed for partitioning from the start (one partition initially, but the address scheme supports N).
3. VGA controller with double-buffered, partition-aware framebuffer displaying test patterns. Verified independently.
4. Triangle FIFO and Avalon interface working in simulation. Software can push triangle packets, hardware can pull them.
5. End-to-end integration: software submits one triangle through the FIFO, hardware rasterizes it, triangle appears on screen.
6. Z-buffer with early rejection. Multiple triangles, correct occlusion. Rotating 3D object on screen with keyboard input.
7. Parallel pixel units. Additional units instantiated, each with its own memory partition. Fill rate improvement measured.
8. Demo polish: complex geometry, per-face lighting, performance characterization.

References:

[1] J. Pineda, “A Parallel Algorithm for Polygon Rasterization,” in Proc. SIGGRAPH ‘88, Computer Graphics, vol. 22, no. 4, Aug.1988, pp. 17-20.