

miniGPU: 3D Realtime Renderer

Final Report

1st Danny Villanueva

Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA, USA
mapleint@mit.edu

2nd Sadao Hirose

Department of Mechano-Informatics, Faculty of Engineering
University of Tokyo
Tokyo, Japan
sadao@mit.edu, hirose-sadao-71@g.ecc.u-tokyo.ac.jp

Abstract—We present the design of miniGPU, a fixed-function 3D graphics accelerator implemented entirely on an FPGA for real-time rendering of simple 3D objects such as a shaded rotating cube. The miniGPU implements the key stages of a classical 3D pipeline: model-view-projection transformations, projection, triangle setup, rasterization, and fragment shading. Vertices are represented in S16.16 fixed-point format and streamed between modules over AXI-Stream interfaces in a 83.33-MHz system clock domain, enabling predictable per-stage latency and high throughput. The pixel path is organized as a fixed-function rasterizer and fragment unit that writes interpolated color and depth values into a double-buffered framebuffer shared with an HDMI output controller, targeting tear-free real-time animation with Z-buffering. Although not all triangles are rendered correctly, the fundamental structure required for real-time rendering has been achieved. A key direction for future work is addressing the unexpected memory-bandwidth bottleneck caused by overdraw.

Index Terms—Digital systems, Fixed-function graphics pipelines, Real-time rendering

I. SYSTEM OVERVIEW

The miniGPU implements a fixed-function 3D graphics pipeline on an FPGA, organized into three major subsystems: the geometry pipeline, the rasterizer, and the fragment production and framebuffer subsystems. As shown in Fig. 1, all modules communicate through AXI-Stream interfaces and operate primarily in a 83.33-MHz system clock domain, while the HDMI output uses a separate pixel-clock domain for display timing.

The geometry pipeline performs all model-view-projection (MVP) transformations using S16.16 fixed-point arithmetic and converts object-space vertices into screen-space triangles. These screen-space primitives are passed to the rasterizer, which evaluates triangle edge functions to determine pixel coverage and produces fragment information, including interpolated color and depth. Downstream, the fragment subsystem applies shading and writes results into a double-buffered framebuffer with Z-buffering, enabling tear-free display and correct depth ordering. The completed frame is then streamed to the HDMI controller through FIFOs that bridge the pipeline to the display clock domain.

This modular design provides predictable per-stage latency, stable real-time rendering, and a clear separation between geometry processing, rasterization, and fragment output. It also

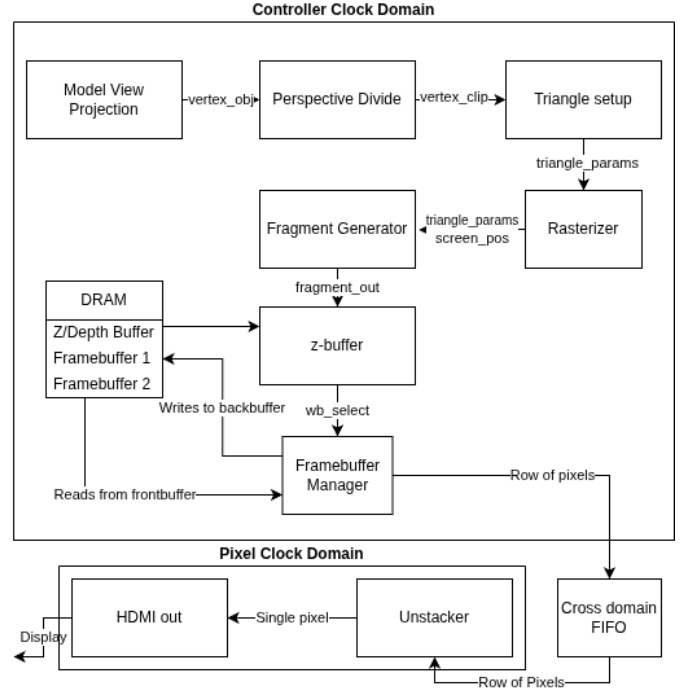


Fig. 1. Overall system block diagram

forms a scalable foundation for future extensions such as rotation, lighting, texture sampling, and tile-based rasterization.

II. GEOMETRY PIPELINE

Geometry pipeline mainly consists of three modules, including model-view-projection, perspective-divide, and triangle-setup modules. Because the divider isn't pipelined, the chain only emits one triangle roughly every 190 cycles.

A. Model View Projection

This module reads fixed-point 16.16 vertex data (position and RGB color) from BRAM, processes it through a *rotation* $\rightarrow$ *view matrix* $\rightarrow$ *projection matrix* pipeline to generate clip-space coordinates, and then normalizes the z value by the maximum z magnitude within the frame before outputting the final vertex values.

To render a rotating cube, the cube is rotated by 1° per frame around the z -axis of the world coordinate system by

incrementing the rotation angle. Let Θ be the rotation angle at a given frame. The rotation matrix $R(\Theta)$ can be written as follows. The sine and cosine values are precomputed for each 1° step and stored in BRAM.

$$R_z(\Theta) = \begin{bmatrix} \cos \Theta & -\sin \Theta & 0 & 0 \\ \sin \Theta & \cos \Theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Let the camera position be (e_x, e_y, e_z) , the target point be (t_x, t_y, t_z) , and the camera's up vector be (u_x, u_y, u_z) . Then the view direction vector $\mathbf{f}$, the right vector $\mathbf{r}$, and the updated up vector $\mathbf{u}'$ are defined as follows, and the view matrix can be constructed as:

$$\mathbf{f} = \frac{\mathbf{t} - \mathbf{e}}{\|\mathbf{t} - \mathbf{e}\|} \quad \mathbf{r} = \frac{\mathbf{f} \times \mathbf{u}}{\|\mathbf{f} \times \mathbf{u}\|} \quad \mathbf{u}' = \mathbf{r} \times \mathbf{f}$$

$$V = \begin{bmatrix} r_x & r_y & r_z & -\langle \mathbf{r}, \mathbf{e} \rangle \\ u'_x & u'_y & u'_z & -\langle \mathbf{u}', \mathbf{e} \rangle \\ -f_x & -f_y & -f_z & \langle \mathbf{f}, \mathbf{e} \rangle \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Let the field of view be Θ , the aspect ratio be aspect, the near clipping plane be near, and the far clipping plane be far. Using the auxiliary value f , the projection matrix can be written as follows:

$$f = \frac{1}{\tan(\Theta/2)}$$

$$P = \begin{bmatrix} \frac{f}{\text{aspect}} & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & \frac{\text{far} + \text{near}}{\text{near} - \text{far}} & \frac{2 \text{far} \text{near}}{\text{near} - \text{far}} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

Each matrix multiplication is followed by a normalization step for the z value to prepare it for subsequent processing. All vertices are first stored in a FIFO, and after determining the maximum z value, each vertex's z value is divided by this maximum before being output. For a cube with 36 vertices, it takes 160 cycles before the first vertex is produced, and 19 cycles per vertex thereafter. The 16-cycle latency of the divider dominates the processing time. Color fields are pipelined alongside the math so they stay aligned with their vertex through the pipeline.

B. Perspective Divide

Perspective Divide module takes a vertex's S16.16 (x, y, z, w) plus color, does perspective divide $(x/w, y/w)$, maps Normalized Device Coordinates to pixel coords via viewport scale/round/clamp, and streams out screen-x/screen-y with the original z and color using an iterative divider. Because this stage also uses the divider with its 16-cycle latency, it takes 37 cycles per vertex.

C. Triangle Setup

The triangle-setup module buffers three incoming screen-space vertices (x, y, z, color) , computes the 32-bit inverse area, and packs this value together with the three vertices into a single 416-bit AXI stream word $(\text{inv_area}, v0, v1, v2)$. With downstream ready, the module buffers three vertices, computes the triangle area, and then performs the division using the divider; as a result, it requires 22 cycles per vertex.

Given three vertices with screen-space coordinates (x_0, y_0) , (x_1, y_1) , and (x_2, y_2) , the inverse area used in the downstream pipeline can be computed as follows. The triangle area is

$$\text{area} = \frac{1}{2} |x_0(y_1 - y_2) + x_1(y_2 - y_0) + x_2(y_0 - y_1)|,$$

and the inverse area is therefore

$$\text{inv_area} = \frac{1}{2 \text{area}}.$$

D. Simulation result

Using the SystemVerilog file that implements only the geometric pipeline as the DUT in the testbench, we confirmed that the geometric pipeline operates correctly on each frame, as shown in the Fig 2.

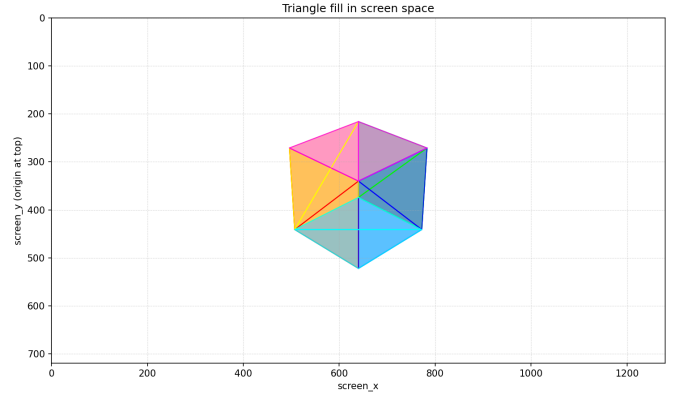


Fig. 2. Geometric pipeline DUT output

III. RASTERIZATION

We have transformed vertices three-dimensional vertices into the flat space of our screen, and we now need to color between the lines of the resulting triangle. Your average child do this just fine, but its surprisingly non-trivial to do with digital logic.

Take the edge $\overrightarrow{p_0 p_1}$. Where $p_0 = (x_0, y_0)$ and $p_1 = (x_1, y_1)$.

$$(a, b, c) = (y_0 - y_1, x_1 - x_0, c = x_0 y_1 - x_1 y_0)$$

$$E(X, Y) = aX + bY + c$$

One can verify that any $\forall(X, Y) \in \overline{p_0 p_1}. E(X, Y) = 0$. The important takeaway is that the functions are linear in X and Y , and as such, if we have $E(X, Y)$ we know that $E(X+i, Y+j) = E(X, Y) + ai + jb$. since we can precompute $ai + aj$ for small i, j , we can compute adjacent points' edge functions

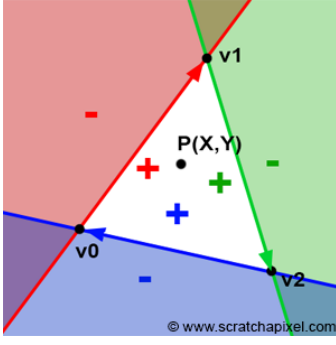


Fig. 3. Signed Area for triangle with CW winding[1]

for very cheap. If they're all positive for a given point, it is in the triangle! One caveat is that we are now computing for N by N tiles instead of single pixels, but this greatly increases throughput. For implementation reasons, N should be a small power of 2, we decide on 8, but will experiment with 4 and 16. We prove this insight as a state machine with 3 states: Idle, Setup, and Raster. Instead of walking over the entire display and computing this for all $X \in [0, 1280)$, $Y \in [0, 720)$, we can just compute it for the bounding box of the triangle, which is a massive save for smaller triangles. In addition, naively going over the bounding box is still not perfect, we start at the topmost vertex, marching a tile right until we exist the triangle entirely, then we go back to the center, and switch direction with the same logic before stepping down. This yields just under a 2x boost raster throughput.

Note that the edge inequality declares $\overrightarrow{p_0p_1} \neq \overleftarrow{p_0p_1}$. This is taken advantage by back-face-culling, dropping primitives that face away from us to save more cycles. As shown in Figure 2, we see signed area on the right side of a given edge. Thus, if the primitive has a counter clockwise winding, we safely conclude that it contributes nothing to the scene.

IV. FRAGMENT GENERATION

We use the term fragment to describe the output of this stage, which includes the color and depth values for a given pixel. The inputs are the metadata of the triangle from earlier and the barycentric coordinates of the point. They are passed through and produced by the rasterizer, respectively. As fig 4 shows, barycentric coordinates describe every point inside the triangle as a weighted average of the vertices, where the sum is 1. This allows smooth interpolation of depth and color.

We plan to implement fancier shading here, such as interpolating color between vertices with barycentric coordinates. Texture sampling, a stretch goal, would also be implemented here. We sample around the point pointed to in the texture, interpolating as necessary, and outputting the products with appropriate lighting and color multipliers.

V. FRAMEBUFFER

Up to this point, we have successfully produced the fragments of each triangle, the primary responsibilities of the framebuffer are to store the fragments, and facilitate reading

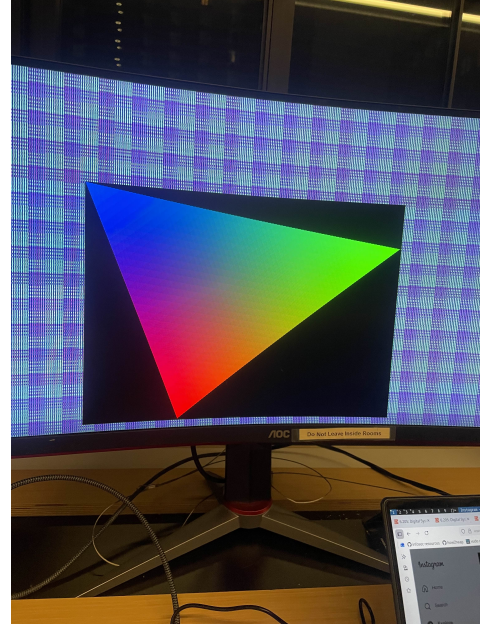


Fig. 4. Barycentric coordinates

for presentation via HDMI. There are a few key differences in the requirements of this framebuffer and traffic generator contrasted to that of Lab 6's:

- 1) If two fragments are mapped to the same pixel, whichever has a lower depth value is saved to the framebuffer for presentation. This allows us to render triangles in any order, with the closest one always being correctly drawn on top. This depth value has to be preserved somehow.
- 2) We mitigate screen-tearing by implementing 'double-buffering'. That is, we have two framebuffers. At any time, one is for drawing pixels (the 'back buffer'), the other is for presentation (the 'front buffer'). When a frame of a scene is done being drawn to the buffer and the HDMI v_sync signal is high (indicating that we also finished drawing a frame), we can 'swap' the buffers. Since we don't have access to this signal from the DRAM, we can approximate this by swapping the buffer pointers after the last HDMI read is issued.
- 3) Reads and writes are not in strictly the scanline order of top to bottom, left to right. While the presentation is still in the scanline order, rasterization should not be. In addition, fragments emitted in sets of 8, whereas the Camera sent single pixels to be stacked.

VI. TRAFFIC GENERATION AND THE Z-BUFFER

A. Traffic Generation

Traffic Generation maintains a few key data structures. A FIFO command queue, queueing when requests when placed and dequeuing on responses. The traffic generator has 3 items logically to worry about, framebuffer-0, framebuffer-1, and the z-buffer. The semantics of which framebuffer is for writing and which for reading is handled here as well, with simple additions in address space.

The interface is marginally different, the axis in gives fragment data (rgbd) for a tile. the implementation then caches this locally, issuing the writes necessary to implement z-buffering with the correct `wb_sel`'s in order to optimize away a framebuffer read.

As part of memory bandwidth optimization, we use a method called the hierarchical Z-buffer (Hi-z).

B. Hierarchical Z-Buffer

Hi-Z is a pretty simple idea: we can cache the greatest depth value of each tile. If the chunk's least depth is greater than the cached greatest depth, we can simply drop the chunk, avoiding any touching of DRAM at the cost of a substantial amount of BRAM and 2 cycles. The amount may sound prohibitive for the resolution, however we are only storing a single depth value for every N^2 pixels, so it isn't that bad. Any $N \geq 4$ results in this fitting into BRAM. However, we decide on $N = 8$ which fits quadratically more sparsely, taking less memory than popcat.

In the next section, it is made clear that trading latency for interfacing less with memory is the correct tradeoff to make.

VII. RESULT

At this stage, all modules pass their testbenches, but some triangles fail to render correctly, as shown in Fig 5, suggesting that there is an issue somewhere in synthesis. Color interpolation has also not yet been integrated. However, even with the missing parts, the results still show the cube rotating, indicating that the fundamental structure required for real-time rendering is in place.



Fig. 5. Current rendering result

VIII. PERFORMANCE EVALUATION

We evaluate the performance of rendering more and more complex scenes, while still maintaining the target framerate of

60FPS. Anything below 24FPS is seen as a failure, as that is the standard for motion pictures.

The geometric transformation modules achieve stable and predictable performance. Our current geometry pipeline outputs one triangle roughly every 190 cycles, allowing us to feed approximately 7,000 triangles per frame into the rasterizer. This throughput is sufficient for many realistic scenes, since a substantial amount of complex geometry can often be represented with only a few thousand primitives. Moreover, the execution time of all stages before the rasterizer is completely independent of the input data, resulting in highly consistent and deterministic behavior.

The rasterizer, however, is inconsistent. Rasterizing a larger triangle takes much more time than a small triangle. The solution of simply rasterizing many pixels per cycle mitigates this issue, but kicks the can down the road to the bottleneck of compute, memory. The DRAM chip on our board could be clocked up to 400MHz, with 16 wires connecting it to the FPGA, we get $400\text{MHz} \cdot 2 \cdot 16\text{bits} = 1.6\text{GBps}$ of bandwidth to the chip, which is an overestimate since DRAM has to refresh itself. A substantial amount of bandwidth to DRAM is used just to present the image

$$16\text{bits} \cdot 1280 \cdot 720 \cdot 60\text{hz} = 110.592\text{MBps}$$

Since each fragment is 16 bits, we may allow up to 13 full frames of fragments to be written. However, since each fragment must be first compared to a Z value (which must be read from DRAM) before being written, this really means only 6 frames of fragments can be saved to memory per frame. While this sounds like a lot, this also reveals that our design can be fully saturated by 6 triangles that cover the entire viewport. 20 adversarial triangles would push the framerate below 20FPS, hardly real-time. The generation of fragments is not the bottleneck; writing them is. Fortunately, the performance problem commonly referred to as overdraw, is largely mitigated by the fact that primitives are typically issued closest to furthest and that if a fragment fails a Z-test, it isn't written to memory. In addition, not all Z-tests necessitate a request to DRAM. Overall, the performance is fine for realistic workloads, and the key bottleneck is memory.

IX. WE'RE JUST A BIT STRAPPED FOR CACHE

One idea that came to us very late into the development of the project (and as such, was not reasonably implementable) was that what we've named sub-viewport-rendering (in industry this is called Tile-Based-Deferred-Rendering TBDR, and geometry binning). A lot of time was spent stressing heavily about the DRAM limitations. We never seriously considered rendering to BRAM. You can only store so much with a measly 2Mbits after all.

But what if we were not rendering a massive 720p image, but instead we were rendering 16 320x180 images? Now, instead of writing directly to DRAM, we can instead render to BRAM, which will be much faster. Not only is it disgustingly more efficient, but we could also now afford to have a higher quality image in the form of 24-bit color due to the bandwidth

savings. Sure, fragments overwrite each other, in BRAM. "Oh no, we lost 2 whole cycles", Much more preferable to dozens from DRAM, as well as we don't have to worry about starving the reader, which is another real problem we ran into on hardware that wasn't anticipated.

This also makes it so that the memory interface is almost the same as Lab06, where there is a producer of pixels and a writer. This doesn't remove the need for double buffering, but it does remove the memory bandwidths that we worried so heavily over. The implementation would require a little BRAM than just for the 'cache' and Zbuffer, as you need to store the screen-space triangles somewhere in BRAM to read them repeatedly. Though I would rather deal with 100 BRAM problems than 1 DRAM problem.

X. FUTURE WORK

The primary bottleneck - which we did not expect while designing, is overdraw and the toll it takes on memory bandwidth. A way to alleviate this is to split the pipeline. Instead of immediately rendering a primitive directly to the framebuffer, add the primitive to a table. Create a list of these primitives for each tile. Once all primitives for a frame have been transformed, one can coarsely rasterize - adding the primitive's ID to the tiles that it intersects. You may now rasterize the sections each primitive referenced by the given tile, caching results locally before flushing to DRAM.

This would substantially mitigate the memory pressure introduced from overdraw; additionally allowing parallel rasterization by having more tile-rasterizers. The complexity in this approach is substantial but could be worth it for the savings in memory bandwidth.

Other important, but optional and currently unimplemented features of the pipeline are alpha channels, texture sampling, and alternative lighting models.

XI. MEMBER CONTRIBUTION

Danny conducted the research, implementation, evaluation, and writing for the rasterizer, fragment shading, zbuffer, and double-buffer framebuffer. Sadao was responsible for the geometric pipeline. We would like to thank our instructor and advisor, Joe Steinmeyer, as well as the TAs for their valuable guidance.

REFERENCES

- [1] *The Rasterization Stage*." Scratchapixel, 2025. [Online]. Available: www.scratchapixel.com. [Accessed: Nov. 26, 2025].