Hardware-Accelerated Block Game Project Proposal

Josh Bernheisel (jcb2301) Wesley Maa (wm2505) Mihir Joshi (mnj2122)

Spring 2025

1 Overview

The goal of this project is to write a hardware-accelerated 3D rendering engine on the DE1-SoC that produces a real-time, interactive block game on a VGA display. The system splits work between the board's ARM processor running Linux and custom FPGA logic: the ARM handles all game logic, world representation, visibility computation, and per-quad setup in C, while the FPGA implements a fixed-function quadrilateral rasterizer with z-buffering and drives the VGA output. A Linux kernel device driver mediates communication between the user-space game application and the FPGA hardware accelerator.

The player will be able to move a camera through a small, procedurally generated block world composed of textured cubes, and place or remove blocks in real time. Since rasterization is massively parallel and arithmetic-regular, this is an ideal problem for dedicated hardware.

Because all geometry in a block world is inherently quadrilateral—every block face is a rectangle, and greedy meshing produces larger rectangles—we rasterize quadrilaterals.

2 Architecture

2.1 C Software

All complex, control-heavy computation runs on the ARM Cortex-A9. In particular, we will implement at least the following algorithms:

World Representation
The world is stored as a 3D array of block IDs organized into 16×16×16 chunks. With 8 visible chunks after frustum culling, total world storage is modest (a few hundred KB).

Hidden-Face Culling
For each solid block, check the six axis-aligned neighbors. Emit a face only when the neighbor is air. This eliminates the vast majority of interior geometry.

View-Frustum Culling
Reject entire chunks whose bounding boxes fall outside the camera frustum before any per-face work.

Vertex Transformation and Quad Setup
Multiply each quad's four vertices by a combined model-view-projection matrix using 32-bit fixed-point arithmetic. Perform perspective divide and viewport mapping to 640×480 screen coordinates. Precompute the four edge-function coefficients (A_i, B_i, C_i) for each quad edge, where $E_i(x, y) = A_i x + B_i y + C_i$, as well as depth gradient parameters and a

texture/color ID. Under perspective projection, a world-space rectangle becomes a general convex quadrilateral in screen space; the four-edge-function test handles this correctly as long as the quad remains convex.

Command Buffer Write

Write the resulting quad descriptors into a shared memory region accessible to the FPGA via the device driver and lightweight HPS-to-FPGA bridge. Each quad descriptor contains four edge equations, a screen-space bounding box, depth interpolation coefficients, and material data.

2.2 FPGA

The goal is to minimize the SystemVerilog surface area by having the ARM do as much preprocessing as possible, and to use the FPGA for the most parallel calculations. The hardware will be developed in SystemVerilog, synthesized with Quartus Prime, and verified using ModelSim and Verilator testbenches. We will implement the following algorithms:

Quad Rasterizer

Read quad commands from the shared buffer. For each quad, scan its bounding box and evaluate the four precomputed edge functions per pixel. A pixel is inside the quad when all four edge values are non-negative.

Z-Buffer

Maintain a 16-bit depth buffer at 640×480 (∼150 KB) in FPGA SDRAM. For each passing pixel, interpolate depth using the precomputed gradient and compare against the stored value; write only if the new fragment is closer.

Texturing

Store small tile textures in on-chip block RAM. Interpolate UV coordinates per pixel using the same incremental-add technique as edge functions and perform nearest-neighbor lookup.

Framebuffer and VGA Output

Double-buffered 640×480, 16-bit color framebuffers in FPGA SDRAM (∼300 KB total). The VGA timing controller displays the front buffer at 640×480 while the rasterizer writes to the back buffer.

2.3 Memory Map

Resource	**Location**	**Size**
World data	HPS DDR3	< 4 MB
Quad command buffer	FPGA-accessible shared region	∼256 KB
Framebuffer ×2 (640×480)	FPGA SDRAM	∼300 KB
Z-buffer (640×480)	FPGA SDRAM	∼150 KB
Texture tiles	FPGA on-chip block RAM	∼16 KB

3 User Interface

The player controls a first-person camera using a standard USB keyboard and mouse, both supported natively by the Linux USB HID subsystem. The keyboard handles movement and the mouse controls camera orientation: horizontal movement rotates the yaw and vertical movement adjusts the pitch. Mouse buttons will be mapped to block placement (right click) and block destruction (left click). The application reads input events from `/dev/input/event*` using the standard Linux evdev API, polling relative mouse motion events and key press/release events each frame.

4 Major Tasks and Milestones

1. **VGA + framebuffer baseline.** FPGA VGA timing controller displays a static test pattern from a 640×480 pixel buffer. ARM writes colored pixels over the link to confirm the communication path.

2. **Linux device driver.** Implement a kernel module that maps the FPGA bridge, exposes `/dev/voxel_gpu`, and supports writing command data from user space.

3. **Software renderer prototype.** Full C rendering pipeline in user space: world representation, meshing, transform, and software quad rasterization writing to the framebuffer through the driver.

4. **FPGA quad rasterizer.** Implement four-edge-function rasterizer in SystemVerilog; test with hardcoded quad descriptors via Verilator and ModelSim. Add z-buffer.

5. **Integration.** Connect the user-space game (via the driver) to the FPGA rasterizer. Render a small static scene of colored cubes using hardware acceleration.

6. **Camera and interaction.** First-person camera movement through the world with USB keyboard and mouse input. Block placement and destruction via mouse buttons.

7. **Texturing.** Add per-pixel texture lookup from block RAM.