FPGA 3Display

Nathaniel Felleke
Massachusetts Institute of Technology
Electrical Engineering & Computer Science
Cambridge, MA, USA
nfelleke@mit.edu

Liam Kronman
Massachusetts Institute of Technology
Computer Science & Engineering
Cambridge, MA, USA
lkronman@mit.edu

JT Markowitz
Massachusetts Institute of Technology
Computer Science & Engineering
Cambridge, MA, USA
jtma@mit.edu

***Abstract*—This paper introduces FPGA 3Display, a volumetric display system that uses a rotating HUB75-based 64x64 RGB LED matrix, driven by a CMOD-A7 FPGA to create dynamic 3D visualizations leveraging persistence of vision. Key innovations include a custom rotational frame buffer for real-time voxelized object rendering, integration of pre-computed Cartesian-to-cylindrical transformations, and robust synchronization using an infrared break-beam sensor for angular precision. The display is powered through a custom slip ring assembly, with visualizations spanning from geometric shapes (spheres, cubes) to complex 3D models (e.g., Stanford Bunny, TIE Fighter). We also demonstrate animations, such as a bouncing ball simulation, and user-defined .OBJ file support. Our modular SystemVerilog framework, alongside Cocotb simulation scripts, ensures both high performance and extendibility. Evaluation reveals efficient resource utilization and seamless operation at high rotational speeds, paving the way for future enhancements such as interactive 3D gaming and higher-resolution displays.**

***Index Terms*—FPGA, volumetric display, HUB75, persistence of vision, SystemVerilog, LED matrix, 3D visualization, voxelization, rotational synchronization, real-time rendering, interactive systems.**

I. Introduction

Why use a boring 2D display when you can use an FPGA 3Display?

This project is an exploration of driving RGB LED displays at high refresh rates with HUB75, a protocol (alternative to HDMI/VGA) that uses multiplexing to achieve fast display refresh. To demonstrate the power of this display, we have decided to build a spinning rig on which such a display will rotate, producing a full 3D image with a "persistence of vision" effect. The image will be composed of angular cross-sections updated at a rate defined by an infrared break beam sensor's determination of the rotational period. The resulting visual is considered a volumetric display, as it can be perceived as three-dimensional from any viewing angle.

We draw significant inspiration from James Brown's recently popular volumetric display projects (his 3D DOOM project and behind-the-scenes Mastodon posts). Before embarking on our project, we consulted Brown. He answered several of our questions in a Mastodon thread. Our setup is very similar to his: a 64x64 2mm pitch LED matrix display that we interface with using the HUB75 protocol through 14 DIP pins of a CMOD A7-35T FPGA, spun quickly (100s of RPM) within a sturdy, transparent structure. The mechanical aspects of this project are explained in Section II. In addition, a slip ring is used to supply the power and ground lines to the FPGA, which is mounted on a breadboard that is spinning with the panel.

After completing the physical construction, we first rendered a sphere (spinning a circle defined combinationally using the circle formula). Once we managed to get consistent signals from the infrared sensor, we implemented a hemisphere visualization, which quickly updates the display based on where it is in the revolution, to appear as a floating half-sphere. After developing a Fibonacci lattice-based addressing methodology (i.e., selectively updating single columns per rotational `theta`, as opposed to the whole screen every time, to achieve even pixel density), we implemented a cube visualization. With a stable cube visual (after some debouncing on the IR sensor signal), we developed a pipeline to voxelize any OBJ file into a 64x64x64 Cartesian coordinate system and store it in a `.mem` file within the CMOD's BRAM, which is then served via `rotational_frame_buffer.sv` into cylindrical coordinates for our `hub75_output.sv` module to send out to the display. This workflow enabled us to render exciting visuals like the Stanford Bunny (Fig [1]), TIE fighter from Star Wars (Fig [2]), Darth Vader's helmet, the Great Dome, a human skull, and even the STL of our 3Display Aluminum Extrusion Frame CAD (Fig [5])! Finally, we implemented a 3D "screen saver" of a small ball bouncing within a box.

In person, the system produces highly convincing 3D visual effects, but they are difficult to capture on camera. Our final project video showcases some of our visualizations here and [redacted].

II. Mechanical Aspects (Liam & Nathaniel)

The FPGA 3Display is made of seven primary physical components:

- A HUB75 64 x 64 2mm pitch LED matrix panel that acts as the main display for the project.
- A CMOD A7-35T FPGA that is used to control the display. We chose to use the CMOD A7-35T FPGA over the Urbana board for two main reasons: the onboard flash program memory, which would allow for programs to run without a direct connection to a computer, and its smaller size which would make it easier to mount on a spinning display. There are, of course, trade-offs with this choice.

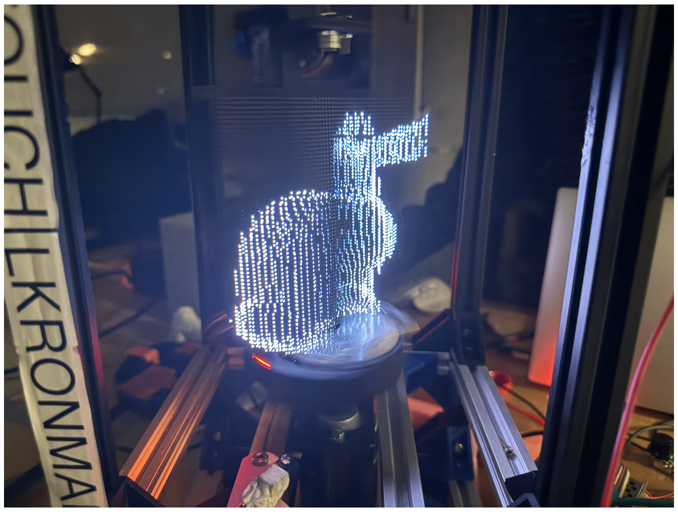


Fig. 1. Stanford Bunny visualization

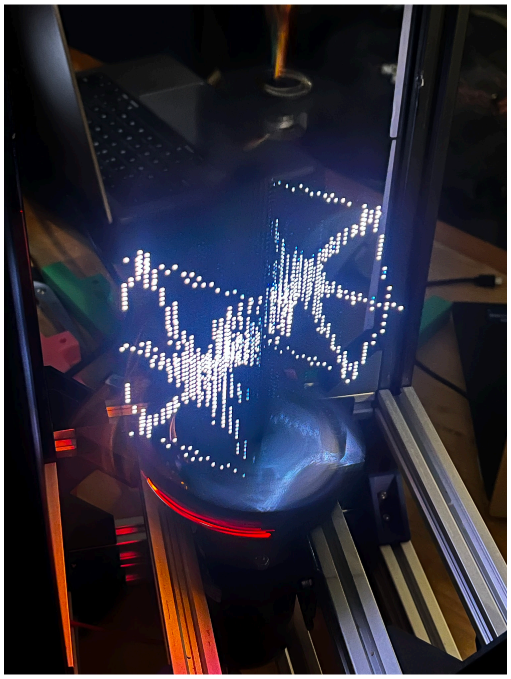


Fig. 2. Star Wars TIE Fighter visualization

For example, we only have access to the CMOD's BRAM and SRAM capabilities, which are on the order of KBs in size, while the Urbana boasts a 1Gbit-large DRAM. However, we discovered that this was, in fact, sufficient for storing our visualizations (see Section V).

- An 8-channel slip ring that is used to supply power and ground to the spinning display and FPGA.
- A custom 3D printed mount for the LED matrix panel, FPGA, and slip ring. It was completed over multiple iterations and includes front pocket counterweights like washer and nuts to increase rotational stability. It is designed such that the LED panel intersects with and is centered on the motor shaft's axis of rotation. (See Fig. 4.)
- An IR trip sensor mounted on the spinning display and an IR emitter on the stationary frame. The rate of trips of the sensor is used to estimate the rate of rotation of the display. The SystemVerilog implementation for this logic can be found in `detect_to_theta.sv` within the `hdl` folder of our code repository.
- A brushed motor and control electronics that are used to rotate the display.
- A custom frame built out of aluminum extrusions to hold and stabilize the brushed motor and the LED panel with the mount. (See Fig. 5.)

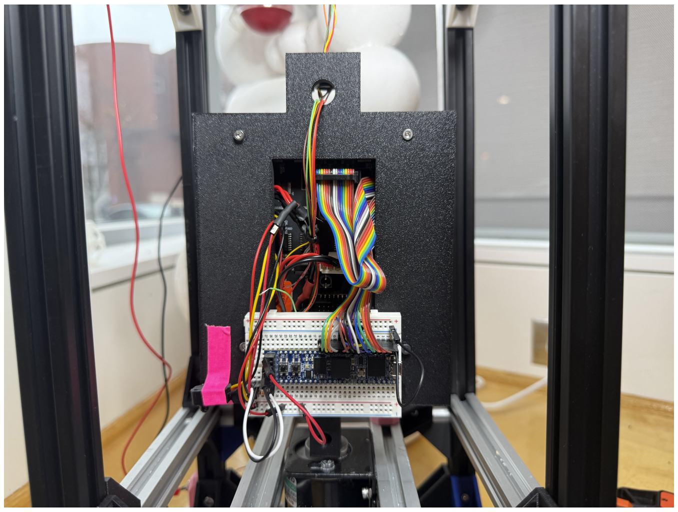


Fig. 3. FPGA 3Display Mount + Breadboard (Back)

Much of this work was done at the Cypress Engineering Design Studio in Building 38, with gracious assistance from instructors Anthony Pennes, Dave Lewis, and Alec Reduker. Although there is room for improvement and iteration, the structure holds together well in its current form, sufficient to sustain our desired graphics. Fig. 3 reveals the back of the 3D mount (including the CMOD board).

III. SystemVerilog Implementation

All of the SystemVerilog code we've written can be found in the `hdl` folder of our GitHub repository. Cocotb simulation scripts in Python can also be found there, under the `sim` folder. `top_level.sv` bridges `hub75_output.sv`, `frame_manager.sv`, and `detect_to_theta.sv`. Several modules (related to more complex visualizations like Boids) are being developed and tested separately. The way our SystemVerilog modules work together can be understood through our high-level block diagram (Fig 3.). Some of the components featured in that diagram have yet to be implemented to the extent they're planned to be.

A. `hub75_output.sv` (Nathaniel)

HUB75 displays allow for rapid refreshing of a 64 x 64 display by multiplexing rows (columns in physical setup) of the display. Two rows are addressed at the same time,

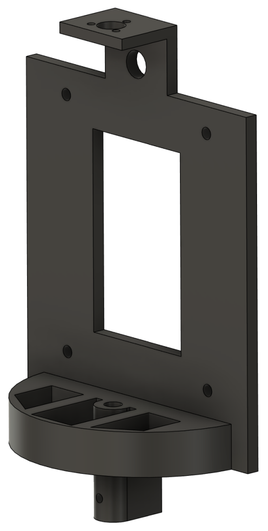


Fig. 4. 3D-Printed Display Mount (Fusion 360 CAD)

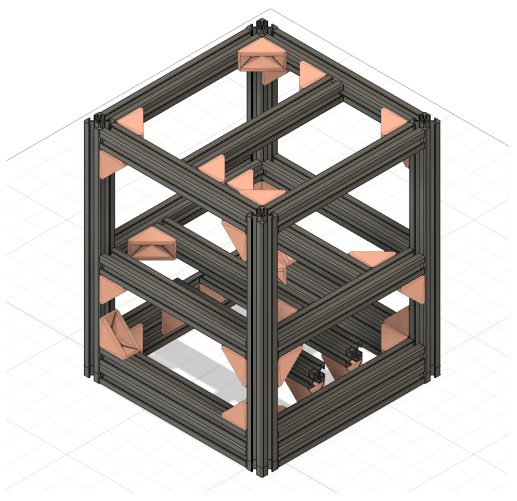


Fig. 5. 3Display Aluminum Extrusion Frame (Fusion 360 CAD)

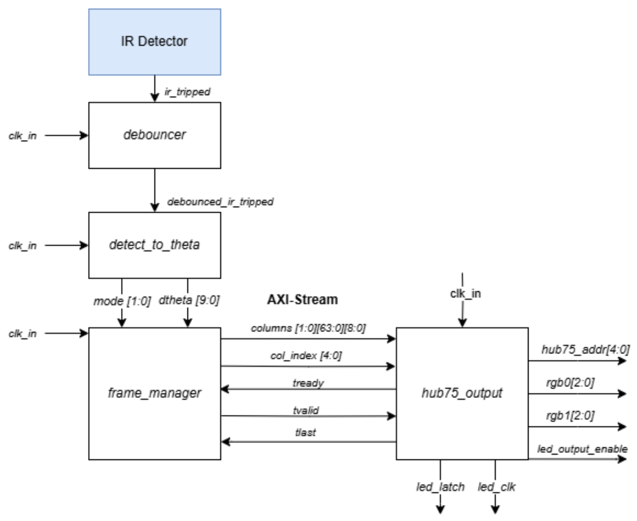


Fig. 6. Top Level Block Diagram

and the individual LEDs are set by clocking in the RGB values. To achieve more than one-bit color resolution with a HUB75 display, the columns must be set multiple times by the controller to modulate the LEDs. To interact with the display, the FPGA connects to five address lines, which are used to index into the 32-row groups, three control signals (CLK, Latch, and Output Enable), and two RGB ports.

Binary Count Modulation (also known as BCM) is used in place of PWM to achieve color resolution as BCM allows for a total of fewer writing periods (three instead of eight with PWM), even though it is the same number of cycles. Reducing the number of writing periods is essential as the display is turned off for the writing periods, and entire columns are addressed at the same time, making it so a change in one pixel's duty cycle requires rewriting 63 others. BCM works by representing brightness in binary numbers and weighting the time duration of each bit modulated proportional to its weight in binary form; with three bits of color resolution for each color, the least significant bit is held on the display for PERIOD cycles, while the most significant bit is held on the display for $2^3\cdot$ PERIOD cycles.

In SystemVerilog, the `hub75_output.sv` module acts as a hardware abstraction module to control the display. Operating similarly to an AXI-S communication—with ready, valid, and last signals—the module takes a column index and an RGB array (with 3 bits of resolution per RGB color) of the corresponding columns as inputs. It outputs the control and modulated RGB signals to the HUB75 display. The module is written as a finite-state machine with three states (IDLE, WRITING, and MODULATING). In the IDLE state, the module waits for the column and address data indicated by a data-valid signal. Once the data are acquired, it moves to the WRITING state. In the WRITING state, the module writes column data with one bit of color resolution to the proper address by clocking in the individual pixels and latching the columns. In the MODULATING state, the module counts up to the proportional number of clock cycles for the current binary bit in the BCM. If it has finished the third and last bit of color resolution in the BCM, the module moves back to the IDLE state for the next set of column data.

B. `frame_manager.sv` (Liam)

The `frame_manager.sv` module generates and streams pixel data for a volumetric display (to `hub75_output.sv`), handling two active columns at a time. It supports multiple rendering modes – sphere, cube, and other 3D assets – selectable via the mode input. Based on the rotational angle (`dtheta`), the module computes the appropriate columns using submodules and outputs the data to a HUB75 LED matrix controller. Synchronization is achieved through the `hub75_ready` signal, ensuring valid data is sent at the right

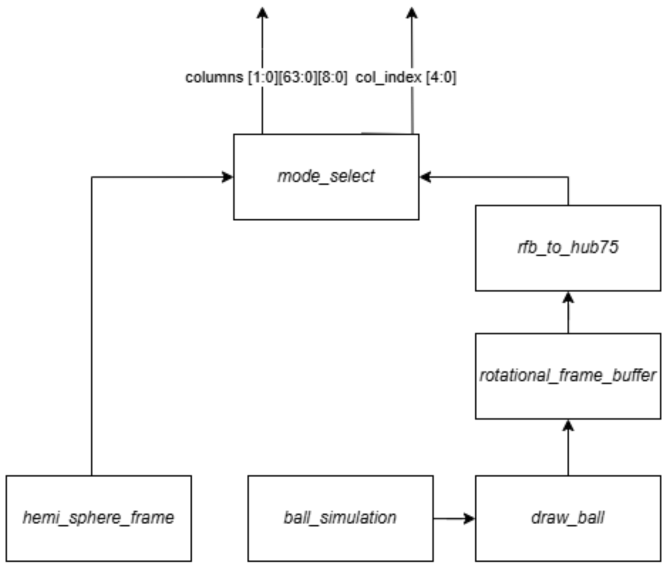


Fig. 7. Frame Manager High Level Diagram

time. A `data_valid` signal indicates when the computed pixel data is ready, and `hub75_last` indicates when the next period is ready. Fig. 7 illuminates how `frame_manager` multiplexes some submodules that serve frames.

C. `rot_frame_buffer.sv` (JT)

The job of the rotational frame buffer is essentially to supply a column of data given an angle of rotation. The job is a little more complex when you consider that any data displayed on angle θ, the angle $\theta + \pi$ will have the exact same data, but flipped across the origin axis.

Furthermore, the buffer requires independent read and write lines if we are going to do any live animations. The buffer should display a column of "0" (no data) if there is an ongoing write, and behave as a normal BRAM module if there is none. Writes are also complex, since entire columns are stored per address, so writing a point should "or" itself with existing points on its column.

Writes must also be accompanied by a "flushing" mechanism to prepare the buffer for the next frame. Otherwise, points would accumulate in the buffer during an animation/simulation, and the buffer would show all frames at once.

The Rotational Frame Buffer also includes the radius as the most significant bits, so that every column of information is concatenated with all information needed to display it, as it's assumed we know the angle given its the address of the column.

D. `rot_frame_buffer_to_hub75.sv` (Nathaniel)

Because of the way the columns are outputted out of `rot_frame_buffer`, with respect to the relative angle of the display instead of the multiplexed column index for HUB75 (i.e. two columns with different HUB75 indices are given at the same time), there is a need for a module that

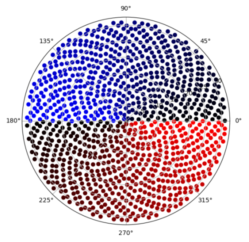


Fig. 8. Discretized Fibonacci Lattice

inputs data into the HUB75 one column at a time. The module acts as an FSM with three states (IDLE, WRITING_COLUMN0, and WRITING_COLUMN1). In the IDLE state, the module is always retrieving the most recent output from `rotational_frame_buffer` and transitions to WRITING_COLUMN0 if the tlast signal from HUB75 is true, getting ready to write once the display is ready. In the WRITING_COLUMN0 state, the module sends out the column and index for the left side of the display to the HUB75 controller and sets the column with the same index on the right side to zero. After the next tlast signal from Hub75, the module transitions to the WRITING_COLUMN1, which does the same for WRITING_COLUMN1 but for the right side of the display instead of the left. After the module is finished transferring both sets of column data and indices to the HUB75 controller module, the state returns to IDLE for the next set of data.

E. `cartesian_to_cylindrical.sv` (JT)

Converting between cartesian space to cylindrical space is arduous with a purely hardware implementation, so we decided to avoid the problem of implementing our own `atan_2(x, y)` and `distance_to_origin(x, y)` functions, by simply pre-computing a 64×64 space of inputs, then reducing our 256×256 simulation space to 64×64 (Fig 9).

Calling the space cylindrical is slightly misleading, since the atan lookup table returns numbers in a range from 0-1023, not only by their angle from the horizontal, but by their distance from a discretized Fibonacci Lattice (Fig 8, 9). Figure 9 has exaggeratedly low resolution in order to visualize the difference from a typical `atan_2()` function.

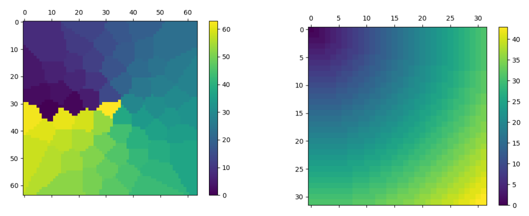


Fig. 9. Pre-computed `atan_2` and `distance` Table

F. *draw_ball.sv & ball_simulation.sv (JT)*

Because of the construction of rot_frame_buffer, as long as we flush the buffer, we don't have to worry about overwriting logic here, and can simply send points in as long as the timing between each is correct and they have been pipelined through cartesian_to_cylindrical. draw_ball.sv is parameterized by logic that will initiate a new frame, and logic that will draw a ball in x y z space.

We can then control this module by another FSM, ball_simulation.sv, which simply writes a new frame every time the IR sensor is tripped.

G. *detect_to_theta.sv (Liam)*

The detect_to_theta module measures the angular position (theta) and period of rotation for a system using an infrared (IR) sensor. It detects when the sensor is triggered (ir_tripped) and uses this to calculate the time since the last trigger, outputting the result as period. The module resets theta on each IR trigger and increments it on every clock cycle, effectively tracking the elapsed time between consecutive triggers.

However, the actual output of detect_to_theta is dtheta, which is a discretized version of theta, between 0 through 1023 (we've defined our ROTATIONAL_RES). That means there has to be some logic which converts the intermediate theta and period arrays into the normalized dtheta. This is done via an angle_counter variable, which increments up to (dtheta >> $clog2(ROTATIONAL_RES)). Essentially, whenever angle_counter overflows, dtheta is incremented by one, and since we are dependent on the last period, this will bound the dtheta between 0 and ROTATIONAL_RES - 1.

This module is designed for high-resolution angular tracking, with the THETA_RES parameter allowing customization of the resolution (width of 100 bits, calculated for a 24MHz clock and tuned experimentally). We also debounced the IR sensor reading, which can be seen in debouncer.sv, which is a variant of the module developed for class, but with a new CLK_PERIOD_NS of 42ns and DEBOUNCE_TIME_MS of 10ms, verified experimentally.

Future enhancements could include adjustments to mitigate the lag between period updates and real-time rotation tracking. A demonstration of the operation of the IR break beam sensor can be seen here, where LED1 of the CMOD turns on whenever the IR receiver reads a "high" signal.

H. *OBJ Pipeline (Liam)*

.OBJ is a standard file format to store 3D assets. An OBJ file includes information about vertices, textures, and other geometries that can be used to generate an asset in a display-specific format.

Converting OBJ geometries into a volumetric display-friendly format can be very different than displaying for traditional 2D screens like TV, which utilize graphics rendering pipelines to convert the 3D asset to a forced perspective using techniques such as ray tracing. We do not need to worry as much about projections and lighting (as we hope reality will fill in that info for us :)).

The voxelizer.py script within the sim folder of our code repository injests OBJ files, and, using the trimesh Python library, casts the vertices into a discretized version of a 64x64x64 Cartesian coordinate system. This vertex information is stored into .mem file within the data folder, and then can be loaded by rot_frame_buffer.sv into the cylindrical space for hub75_output.sv. This pipeline enables us to display virtually any OBJ file!

I. *Testing Philosophy*

Since this is very much a visual project with two extremely dependent parts: mechanical and software, our philosophy while testing is that modules must be both numerically and visually correct. An example of this is a mock display that allows allows us to quickly and intuitively understand unintended behavior of a module (Fig. 10).

As previously mentioned, our Cocotb simulation scripts can be found within the sim folder of our repository. These have been developed in tandem to our SystemVerilog modules, ensuring that they work correctly before being sent over to the FPGA.

IV. Evaluation

The latency of all of our modules is extremely small compared to what is required for a persistence of vision display. We found that the refresh rate of our display, while blazingly fast, was still no match for the speed of the CMOD A7 chip. We had to greatly increase the duty cycle of the PWM output module for coherent data to show up.

We are using 3 BRAM modules, with a utilization of around 20%. Since full color resolution is around 9 times the utility of the highest resourced BRAM, which itself is at $\approx$ 10%, we technically could display full color objects. This would be hard to improve without implementing intense mathematical operations on the fabric itself (arctan and distance).

Our design handles basic shapes, (sphere and cube), renders animations (bouncing ball inside double cube), and arbitrary

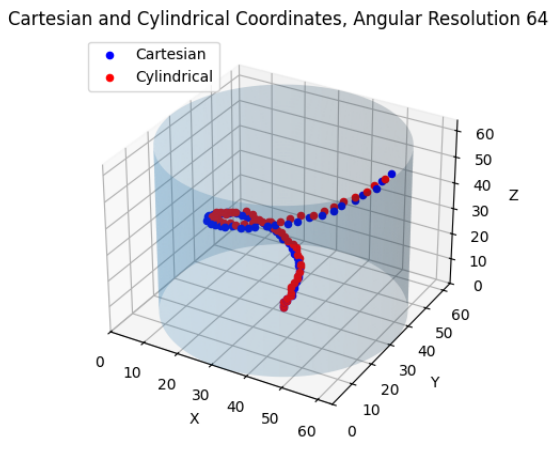
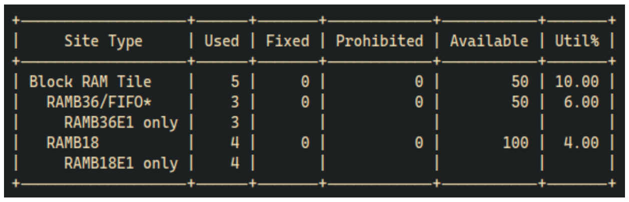


Fig. 10. Mock Display

.obj files. We hit all of our goals after pivoting between simulations. We wanted to also implement 3D pong, but the number of wires through our slip ring constrained us so that we could not implement controls.



```
+-------------------+------+-------+------------+-----------+-------+
|     Site Type     | Used | Fixed | Prohibited | Available | Util% |
+-------------------+------+-------+------------+-----------+-------+
| Block RAM Tile    |    5 |     0 |          0 |        50 | 10.00 |
|   RAMB36/FIFO*    |    3 |     0 |          0 |        50 |  6.00 |
|     RAMB36E1 only |    3 |       |            |           |       |
|   RAMB18          |    4 |     0 |          0 |       100 |  4.00 |
|     RAMB18E1 only |    4 |       |            |           |       |
+-------------------+------+-------+------------+-----------+-------+
```

Fig. 11. Block RAM Utilization Summary

V. Conclusion

FPGA 3Display successfully demonstrates a fully functional volumetric display system that combines mechanical precision with advanced digital design techniques. By leveraging the HUB75 protocol and a rotating RGB LED matrix, the system creates dynamic 3D visualizations that are both visually compelling and computationally efficient. We achieved real-time rendering of volumetric shapes, animations, and complex 3D models through a combination of hardware modules, such as the rotational frame buffer, Cartesian-to-cylindrical transformation pipelines, and the HUB75 controller. Key mechanical innovations, including a slip ring power delivery system and a stabilized mounting structure, ensured reliable high-speed operation.

Our evaluation highlights the efficient use of FPGA resources, including BRAM and DSPs, to meet the stringent requirements of persistence of vision displays. The project also underscores the importance of modular design, as our SystemVerilog framework allows for seamless integration of new visualizations and future interactive features, such as 3D gaming.

Author Contributions:

Nathaniel Felleke led the development of the HUB75 controller (`hub75_output.sv`), (`rot_frame_buffer_to_hub75.sv`) and ensured the mechanical stability of the spinning rig, including the slip ring integration.

Liam Kronman designed and implemented key rendering modules (`frame_manager.sv`, `detect_to_theta.sv`) and developed the OBJ voxelization pipeline. He also spearheaded the visualization efforts, including the integration of complex 3D models.

JT Markowitz focused on the rotational frame buffer (`rot_frame_buffer.sv`) and the Cartesian-to-cylindrical transformation module. He also developed and tested animations, such as the bouncing ball simulation. This collaborative effort has established a foundation for advancing volumetric displays. Future work could explore higher resolution matrices, more sophisticated visualizations, and interactive input devices, such as joysticks or gesture recognition systems, to further enhance user engagement.