TRACE FPGAinting: Three-Dimensional Rendering with an Augmented Camera Environment

Elaine Liu
Department of EECS
Massachusetts Institute of Technology
Cambridge, MA, USA
exliu@mit.edu

Tatiana Vassiliev
Department of EECS
Massachusetts Institute of Technology
Cambridge, MA, USA
tatianav@mit.edu

***Abstract*—We present an FPGA implementation of three-dimensional drawing and rendering of an object path. Our goal is to create a system that transfers occurrences in the physical world to the virtual world. This hardware networking stack interfaces with a 25 fps OV5640 camera and an HDMI 1280x720 monitor. We implement this design using custom object-capture, interpolation, and rendering modules on the Xilinx Spartan-7 XC7S50 chip, evaluate its performance and quality, and discuss potential areas for future expansion and improvement.**

I. Introduction

This project focuses on implementing real-time object path tracing and rendering. The system aims to achieve FPGA memory-efficient and effective rendering. The system will use center of mass tracking, filtering, and depth estimation to locate and track the marker, as well as rotation matrix transformations, linear interpolation, and perspective projection to achieve the desired effect of allowing users to view smooth paths from various perspectives.

Users will interface with a few different states as part of a finite state machine:

1) **IDLE**: Display direct camera input on the screen without modification.

2) **RECORD**: Begin recording the path of a 3D-printed bright orange ball on top of a stick. The user moves the ball to draw a path, while TRACE calculates its center of mass, recording the object's position in every frame.

3) **PLAYBACK**: Finish recording and begin rendering. A white path is displayed on a black screen, representing the ball's path. Four different buttons are available to rotate the path left, right, up, or down.

4) **CLEAR**: If the user presses a button that indicates rotation, clear the entire display screen.

5) **ROTATE**: Recalculate each of the points according to the desired rotation of the indicated button.

II. Path Tracing

The first step of tracing the user's path is capturing the marker's x-, y-, and z-coordinates. We implement this in two parts: color filtering with center of mass detection and depth estimation.

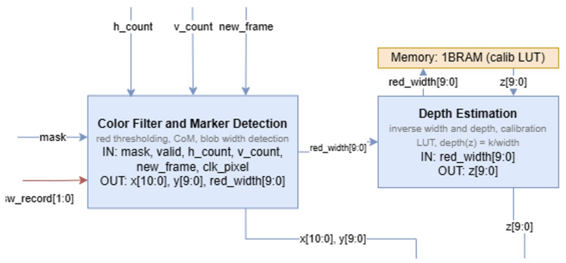
h_count
v_count
new_frame
Memory: 1BRAM (calib LUT)
red_width[9:0]
z[9:0]
Color Filter and Marker Detection
red thresholding, CoM, blob width detection
IN: mask, valid, h_count, v_count, new_frame, clk_pixel
OUT: x[10:0], y[9:0], red_width[9:0]
mask
w_record[1:0]
red_width[9:0]
Depth Estimation
inverse width and depth, calibration LUT, depth(z) = k/width
IN: red_width[9:0]
OUT: z[9:0]
z[9:0]
x[10:0], y[9:0]

Fig. 1. Color Filter, Marker Detection, and Object Depth in Block Diagram

A. Marker Capture: Color Filter and Marker Detection

To locate the marker in each camera frame, we employ a two-stage process: color-based segmentation followed by center-of-mass calculation.

The camera captures video at 25 fps in RGB565 format, providing 16-bit color values (5 bits red, 6 bits green, 5 bits blue) for each pixel. We first convert the RGB color space to YCrCb (luminance-chrominance) representation to provide color information that is more invariant to lighting conditions than raw RGB values. To isolate the red-orange marker, we apply threshold-based segmentation on the red and Cr channels. Each pixel is compared against empirically determined upper and lower bounds, generating a binary mask where pixels within the bounds are marked as 1 (marker) and those outside as 0 (background).

Once the binary mask is generated, we compute the marker's location using center-of-mass calculations on the thresholded pixels. The x- and y-coordinates of the marker are thus determined by:

$$(x_{\text{com}}, y_{\text{com}}) = (\sum \frac{x_i}{N}, \sum \frac{y_i}{N}) \tag{1}$$

where x_i and y_i are the pixel coordinates of the marker, and N is the total area of the marker.

B. Marker Capture: Depth Estimation

We use the spherical marker's depth relative to the camera for the marker's z-coordinate. The diameter of the object is fixed in real life at 50mm, and the depth can be established by the number of pixels the object takes up within the camera frame.

We exploit the inverse relationship between apparent size and distance: as an object moves farther from the camera, it appears smaller in the image. The depth is estimated using the formula:

$$\text{depth} = \frac{k}{diam_{\text{pixels}}} \tag{2}$$

where k is a calibration constant and $diam_{\text{pixels}}$ is the horizontal diameter of the marker sphere in the captured frame.

$$k = \text{diam} \times \text{focal length} \tag{3}$$

To measure marker width, we first apply red-orange color thresholding to isolate the marker pixels. We use the blob's center of mass (COM), as calculated above, then find the leftmost and rightmost red pixels at the COM's y-coordinate to measure its pixel diameter.

$$\text{depth} = x_{\text{right at CoM row}} - x_{\text{left at CoM row}} \tag{4}$$

where $x_{\text{right at CoM row}}$ is the rightmost x coordinate on the same line as the center of mass, and $x_{\text{left at CoM row}}$ is the leftmost x coordinate on the same line as the center of mass. Subtracting the two gives the diameter, as the marker is a sphere.

The calibration constant k is determined during system setup by placing the marker at a known distance and measuring its pixel width. This value is then stored in a lookup table (LUT) that maps pixel widths to depth values, eliminating the need for real-time division operations. The LUT contains 256 entries covering the expected range of marker sizes, with interpolation for intermediate values.

C. *Linear Interpolation*

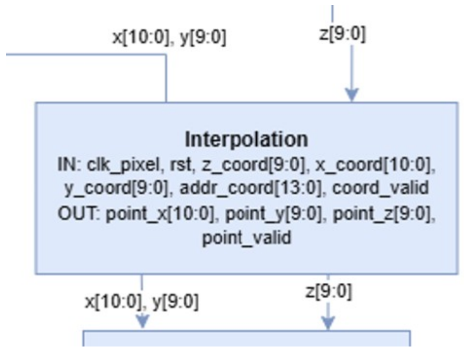
x[10:0], y[9:0]
z[9:0]
Interpolation
IN: clk_pixel, rst, z_coord[9:0], x_coord[10:0], y_coord[9:0], addr_coord[13:0], coord_valid
OUT: point_x[10:0], point_y[9:0], point_z[9:0], point_valid
x[10:0], y[9:0]
z[9:0]

Fig. 2. Interpolation Module in Block Diagram

To generate smooth trajectories from sparsely sampled marker positions captured at 25fps, the system implements real-time linear interpolation during the recording phase. Interpolation occurs before BRAM storage rather than during playback, simplifying the rendering pipeline and ensuring consistent visualization during both recording and rotation.

The interpolation module employs Manhattan distance-based motion between consecutive marker positions. After receiving a new center of mass coordinate $(x_{\text{new}}, y_{\text{new}}, z_{\text{new}})$ from the marker detection pipeline, the module generates intermediate points by incrementing or decrementing each coordinate by one pixel per clock cycle until reaching the target position:

$$x_{t+1} = \begin{cases} x_t + 1 & \text{if } x_{\text{new}} > x_t \\ x_t - 1 & \text{if } x_{\text{new}} < x_t \\ x_t & \text{if } x_{\text{new}} = x_t \end{cases} \tag{5}$$

and similar updates for y and z coordinates. This approach generates one interpolated point per clock cycle, with the total number of points determined by the Manhattan distance:

$$n_{\text{points}} = |x_{\text{new}} - x_{\text{curr}}| + |y_{\text{new}} - y_{\text{curr}}| + |z_{\text{new}} - z_{\text{curr}}| \tag{6}$$

Each interpolated coordinate is immediately processed through the graphics renderer (perspective projection) and stored in BRAM, consuming approximately 40 clock cycles per point. The interpolation completes when all three coordinates simultaneously equal their target values, at which point the system signals readiness for the next marker position.

This recording-time interpolation architecture offers several advantages: (1) eliminates complex state machines and buffering logic required for playback-time interpolation, (2) provides immediate visual feedback of the smoothed path during recording, (3) ensures rotated paths maintain interpolation quality without recomputation, and (4) naturally paces point generation to match the graphics renderer's throughput. The trade-off is increased BRAM consumption—interpolated paths require approximately 4× more storage than raw keyframes—necessitating switchable capacity modes (2,000 vs. 8,000 points) to accommodate different recording durations.

The Manhattan distance metric requires only comparisons and single-bit increments, avoiding costly multiplications or square root operations while producing visually smooth trajectories. For typical hand-drawn paths with marker speeds of 10-50 pixels per frame, this generates 30-150 interpolated points per captured position, sufficient to eliminate perceptible discontinuities in the rendered output.

III. Point Memory

The x-, y-, and z-coordinates of the ball's position at each frame are stored in a BRAM as a 31-bit number. The depth of the BRAM is constrained by the maximum allowed number of points for the system. Currently, we have tested up to 8,000 points (and have found that building fails at 10,000), but the maximum number of points we use is only constrained by the depth of the BRAM and the memory available. The points are stored sequentially in the BRAM by using an event counter which increments every time there is a new frame, meaning a point is captured and recorded in the BRAM every frames. This ensures there is chronological connection between the points, which will allow us to implement interpolation between contiguous points dependent on the order of points drawn.

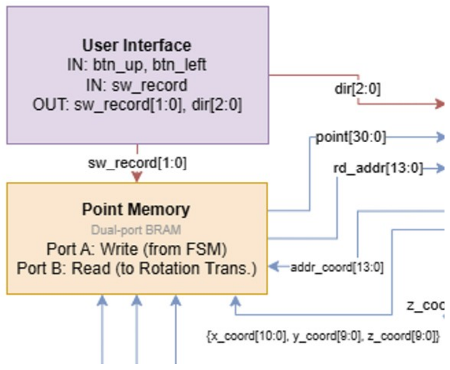

Fig. 3. Point Memory in Block Diagram

IV. Rendering

The second half of TRACE is to render the user-drawn path. This section involves rotating the path as desired, interpolating intermediary points for a smoother path, and performing a perspective projection onto a 2D screen (the HDMI monitor).

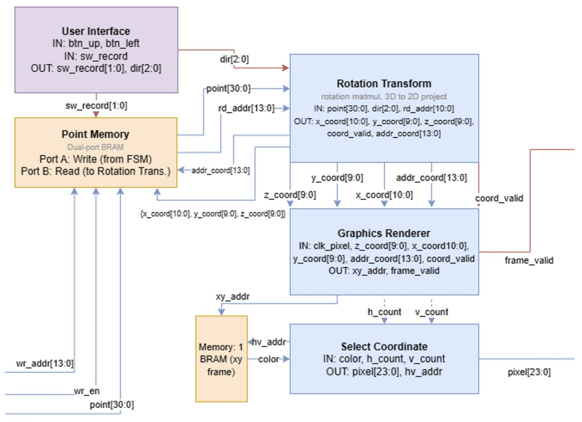
User Interface
IN: btn_up, btn_left
IN: sw_record
OUT: sw_record[1:0], dir[2:0]
dir[2:0]
point[30:0]
rd_addr[13:0]
sw_record[1:0]
Rotation Transform
rotation matmul, 3D to 2D project
IN: point[30:0], dir[2:0], rd_addr[10:0]
OUT: x_coord[10:0], y_coord[9:0], z_coord[9:0], coord_valid, addr_coord[13:0]
Point Memory
Dual-port BRAM
Port A: Write (from FSM)
Port B: Read (to Rotation Trans.)
addr_coord[13:0]
y_coord[9:0]
addr_coord[13:0]
z_coord[9:0]
x_coord[10:0]
coord_valid
{x_coord[10:0], y_coord[9:0], z_coord[9:0]}
Graphics Renderer
IN: clk_pixel, z_coord[9:0], x_coord10:0], y_coord[9:0], addr_coord[13:0], coord_valid
OUT: xy_addr, frame_valid
frame_valid
xy_addr
h_count
v_count
Memory: 1 BRAM (xy frame)
hv_addr
color
Select Coordinate
IN: color, h_count, v_count
OUT: pixel[23:0], hv_addr
pixel[23:0]
wr_addr[13:0]
wr_en
point[30:0]

Fig. 4. Enter Caption

A. Rotation Transform

To enable interactive visualization of the captured 3D path, we implement rotation transformations that allow users to view the trajectory from different perspectives using button controls. The rotation module applies 3D rotation matrices to the stored point coordinates before rendering.

We employ incremental rotation using fixed steps of 15 degrees, which allows users to rotate the path around three axis: pitch (vertical rotation), roll (clockwise rotation), and yaw (horizontal rotation). The rotation is performed around the center of the viewing space at coordinates $(640, 360, 0)$, ensuring that the path rotates naturally about its visual center.

For a point $\mathbf{P} = (x, y, z)$, we first translate it to the origin by subtracting the center coordinates:

$$\mathbf{P}_c = (x - 640, y - 360, z - 512) \tag{7}$$

We then apply rotation matrices.

For yaw (left/right rotation around the Y-axis):

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix} \begin{bmatrix} x_c \\ y_c \\ z_c \end{bmatrix} \tag{8}$$

For pitch (up/down rotation around the X-axis):

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x_c \\ y_c \\ z_c \end{bmatrix} \tag{9}$$

For roll (clockwise/counter-clockwise rotation around the Z-axis):

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_c \\ y_c \\ z_c \end{bmatrix} \tag{10}$$

Finally, we translate back to screen space:

$$\mathbf{P}_{\text{rotated}} = (x' + 640, y' + 360, z' + 512) \tag{11}$$

Furthermore, the user can also choose to implement zoom, which increments the z value upwards for zooming out, and decreases it for zooming out.

Each button press triggers a rotation operation that processes the stored points, applies the transformation, and writes the rotated coordinates back to point memory for subsequent rendering.

B. Graphics Renderer

When a full 31-bit point is captured, we must also translate this 3D point to its position on the screen. To do this, we use perspective projection so that the viewer can perceive depth of points on the screen. First, the x- and y-coordinates are shifted to determine their distance to the center of the screen (640, 360). Then, these distances are scaled by 256, which acts as a convenient focal length due to it being a power of 2. We also shift the z-coordinate by 256 to prevent exploding points at the front of the camera. We then have two dividers which divide the scaled x- and y-distances by the shifted z-coordinate, resulting in new flattened x- and y-coordinates. At this point, we re-shift the flattened coordinates to align with the coordinate system of our screen. We multiply this new y-coordinate by the width of the frame, and then add the x-coordinate which is then fed into the xybram as an address to record as having a point.

$$(X, Y) = (256 \times \frac{x}{z + 256}, 256 \times \frac{y}{z + 256}) \tag{12}$$

C. Select Coordinate

At each h_count and v_count generated by the video_sig_gen module, we determine if these count values are within the displayed frame. If they are, we convert these values to an address in the xybram by multiplying the v_count by the width of the frame, and then adding the h_count. This address is used to read from the xybram, which takes two cycles, and the output is whether there is a point in that address. Therefore, we pipeline the h_sync and v_sync signals to ensure the pixels align.

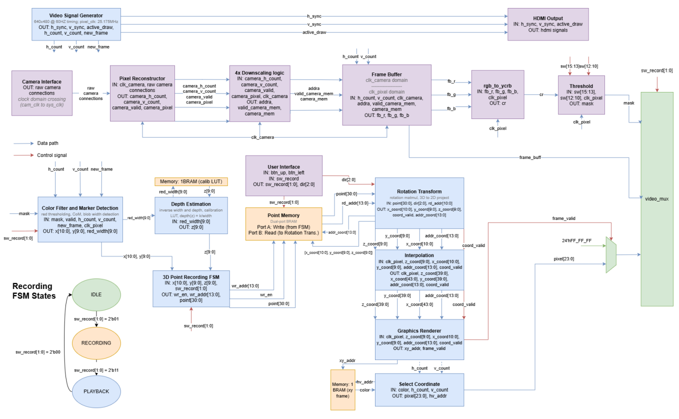
Video Signal Generator
640x480 @ 60HZ timing; pixel_clk: 25.175MHz
OUT: h_sync, v_sync, active_draw, h_count, v_count, new_frame
h_sync
v_sync
active_draw
HDMI Output
IN: h_sync, v_sync, active_draw
OUT: hdmi signals
Camera Interface
OUT: raw camera connections
clock domain crossing (cam_clk to sys_clk)
raw camera connections
Pixel Reconstructor
IN: clk_camera, raw camera connections
OUT: camera_h_count, camera_v_count, camera_valid, camera_pixel
4x Downscaling logic
IN: camera_h_count, camera_v_count, camera_valid, camera_pixel, clk_camera
OUT: addra, valid_camera_mem, camera_mem
Frame Buffer
clk_camera domain
clk_pixel domain
IN: h_count, v_count, clk_camera, addra, valid_camera_mem, camera_mem
OUT: fb_r, fb_g, fb_b
rgb_to_ycrb
IN: fb_r, fb_g, fb_b, clk_pixel
OUT: cr
Threshold
IN: sw[15:13], sw[12:10], clk_pixel
OUT: mask
sw_record[1:0]
clk_camera
frame_buff
video_mux
Data path
Control signal
Color Filter and Marker Detection
red thresholding, CoM, blob width detection
IN: mask, valid, h_count, v_count, new_frame, clk_pixel
OUT: x[10:0], y[9:0], red_width[9:0]
Memory: 1BRAM (calib LUT)
Depth Estimation
inverse width and depth, calibration LUT, depth(z) = k/width
IN: red_width[9:0]
OUT: z[9:0]
User Interface
IN: btn_up, btn_left
IN: sw_record
OUT: sw_record[1:0], dir[2:0]
Point Memory
Dual-port BRAM
Port A: Write (from FSM)
Port B: Read (to Rotation Trans.)
Rotation Transform
rotation matmul, 3D to 2D project
IN: point[30:0], dir[2:0], rd_addr[10:0]
OUT: x_coord[10:0], y_coord[9:0], z_coord[9:0], coord_valid, addr_coord[13:0]
Interpolation
IN: clk_pixel, z_coord[9:0], x_coord[10:0], y_coord[9:0], addr_coord[13:0], coord_valid
OUT: clk_pixel, z_coord[39:0], x_coord[43:0], y_coord[39:0], addr_coord[13:0], coord_valid
3D Point Recording FSM
IN: x[10:0], y[9:0], z[9:0], sw_record[1:0]
OUT: wr_en, wr_addr[13:0], point[30:0]
Graphics Renderer
IN: clk_pixel, z_coord[9:0], x_coord10:0], y_coord[9:0], addr_coord[13:0], coord_valid
OUT: xy_addr, frame_valid
frame_valid
24'hFF_FF_FF
pixel[23:0]
Recording FSM States
IDLE
sw_record[1:0] = 2'b01
RECORDING
sw_record[1:0] = 2'b11
PLAYBACK
sw_record[1:0] = 2'b00
Memory: 1 BRAM (xy frame)
hv_addr
color
Select Coordinate
IN: color, h_count, v_count
OUT: pixel[23:0], hv_addr

Fig. 5. Block diagram of entire TRACE system

V. Overall System Evaluation

A. Throughput, Latency, and Timing Requirements

Post-route timing analysis confirms successful timing closure with all user-specified constraints met. The design achieves Worst Negative Slack (WNS) of +2.122ns on the pixel clock domain, Total Negative Slack (TNS) of 0.000ns, and Worst Hold Slack (WHS) of +0.042ns. The critical path spans 10.854ns from BRAM read through rotation calculation, traversing 8 logic levels including DSP48E1 multiplication and carry chains against a 13.468ns requirement.

TABLE I
Clock Domain Timing Summary

Clock Domain	Frequency	Period (ns)	WNS (ns)
clk_pixel (HDMI)	74.25 MHz	13.468	+2.122
clk_camera	200 MHz	5.000	+0.519
clk_tmds (HDMI)	371.25 MHz	2.694	N/A

The marker detection pipeline completes center of mass calculation within one frame period (13.5ms at 74.25 MHz), including RGB to YCrCb conversion (1 cycle), thresholding (1 cycle), accumulation over 720 lines, and depth LUT lookup (2 cycles BRAM latency). Graphics rendering requires approximately 40 cycles per point for perspective division using successive subtraction, while rotation transforms complete in approximately 8 cycles using DSP48E1 pipelined multiplication. This yields estimated system throughput of 25 points/second in recording mode (camera-limited), 1.8M points/second for playback rendering (division-limited), and 9.3M points/second for rotation processing (DSP-limited).

Post-route power analysis reports total on-chip power of 0.526W (0.450W dynamic, 0.076W static). Clock generation consumes 0.206W (39% of dynamic), HDMI I/O requires 0.153W (29%), block RAM utilizes 0.050W (9.5%), and DSP slices consume 0.006W (1.1%). Junction temperature of 27.6°C with 82.4°C maximum ambient headroom confirms robust thermal performance.

B. Memory Usage

The system utilizes 64 block RAM tiles (85.33% of available 75), comprising 63 RAMB36E1 blocks and 2 RAMB18E1 blocks. This high utilization reflects memory-intensive operations for frame buffering, point storage, and display rendering.

TABLE II
FPGA Resource Utilization Summary

Resource Type	Used	Available	Util. (%)
Block RAM Tiles	64	75	85.33
Slice LUTs	1,578	32,600	4.84
Slice Registers	1,378	65,200	2.11
DSP48E1	12	120	10.00
IOBs	85	210	40.48

The camera frame buffer stores downsampled 320×180 frames in RGB565 format (57,600 entries, 16 bits wide). Point storage uses 31-bit packed coordinates (11-bit x, 10-bit y, 10-bit z). The screen buffer implements a sparse 1280×720 single-bit representation (921,600 entries) supporting dual-port access at 74.25 MHz. A 256-entry depth estimation LUT (10 bits wide) eliminates real-time division. Additional memory includes camera I2C configuration ROM (256×24 bits) and pipeline buffers. Memory breakdown estimates based on design architecture suggest approximately 1 RAMB36 for camera buffer, 8 for point storage, 29 for screen buffer, 1 RAMB18 for depth LUT, and remaining blocks for configuration and pipeline stages.

Logic resources remain below 5% utilization (1,578 LUTs, 1,378 registers), providing substantial headroom for future enhancements such as multi-object tracking or extended storage capacity. The 12 DSP48E1 slices (10% utilization) support multiplication in rotation transforms and YCrCb conversion. The modular architecture enables independent scaling of camera resolution, point storage, and display parameters.

C. Use Cases

Our design provides an efficient way to translate movements in the physical world to display and interaction in the virtual world. The maximum number of recordable points is adjustable, with our system successfully capturing up to 8,000 points. The initial project set out to achieve this with the purpose of enabling a new medium of artwork, one that can be created in the physical world and then adjusted and interacted with in the virtual world. With minimal changes to the design, our project could also accommodate an educational use case, where educators upload sets of 3D figures that are complicated to reproduce physically as point cloud arrays, and then students can interact with the figures and gain intuition for their properties through TRACE.

D. Checklist

Our project successfully achieves all of our commitments, as the camera accurately tracks the moving target, implements width-based depth perception, stores all of the 3D-marker positions with over 1500 points in the BRAM, displays these

3D points on the 2D screen as a recognizable "trace" with visible depth, and allows for button-controlled 3D rotation.

The project also achieved our goals, as we were able to implement continuous 3D rotation, though we opted to use the switches on the FPGA rather than a joystick. Furthermore, we successfully implemented interpolation and rotation in all three axis directions.

In terms of our stretch goals, we achieved two of the five targets, as we successfully implemented zooming in and out on our image, as well as including an "edit" state. In order to complete our stretch goals, we would need to implement visual enhancements of the point colors, as well as point cloud density visualization, and rotation speed adjustment.

E. Implementation Insights

Throughout the process of implementation, we encountered constraints that limited the development of the project. Firstly, when incorporating interpolation, we found that it was difficult to time the cycles of the module between the rotation module and the graphics renderer module. The compromise in doing Manhattan interpolation before rendering occurred involved a tradeoff in function and efficiency for graphic quality. The graphics renderer module relies on division, which takes an uncertain amount of cycles to complete. At the same time, the interpolation module provides 20x the number of points between two existing 3D coordinates. This meant that in order to process each point, we needed to spend significantly more cycles waiting for the division to occur, and creating issues in the feedback loop. We tackled this problem by incorporating interpolation directly into our main points BRAM. This meant that the amount of data we could collect from our marker was significantly limited, leading to short data collection times. Furthermore, an initial goal of interpolation was to maintain structural stability with angles that don't sum perfectly to 360. Since the points were directly written to the BRAM, they also experienced this skew. Further implementations of this design should determine how to incorporate interpolation without writing permanent points to the BRAM.

Secondly, we faced issues when re-centering our points for the graphic renderer. As mentioned earlier, the graphics renderer relies on division, which takes significant cycles. When we attempted to divide using signed integers, it took exponentially more cycles, which prevented us from seeing our image on the screen. We tackled this issue by creating a single bit that indicated whether a number was negative and allowed graphics renderer to use only unsigned integers. Future iterations should consider how to use a coordinate plane that is not reliant on (0,0) being the top left corner, reducing the need for sign values.

VI. Individual Contributions

A. Code

Tatiana focused on the FPGA modules for graphics rendering from 3D to 2D, selecting coordinates to be displayed on the screen through BRAM interaction, the main system FSM, and rotating the images shown on the screen. Elaine focused on the FPGA modules for capturing the marker, estimating the marker's depth, creating the interpolation module, and feeding points into the BRAM.

B. Report

Tatiana and Elaine collaborated on section I, section III part A, and section IV. Elaine wrote section II parts A and B, as well as section III part B and created the figures. Tatiana wrote section II part C, section III part C and D, and section V.

C. Other

Elaine 3D printed the marker we used for capturing data.

Code Repository

https://github.mit.edu/6205F25/fa25-6205-team23

Acknowledgment

We would like to extend our deepest gratitude to Professor Joe Steinmeyer and all of the 6.205 teaching assistants and lab assistants for their continued support and advice in the development of this project.