

Flock Herder Final Report

Jair Santana

Department of Electrical Engineering and Computer Science,

Sreeja Thipireddy

Department of Electrical Engineering and Computer Science

Abstract—We present a design for an interactive visual program where the user can see and herd "boids" —on-screen objects following a behavioral model designed to mimic bird flocks and fish schools —by having them move around and follow a tracked object in the user's hand in three-dimensional space. We implement tracking and depth detection of the tracked object via stereo vision, making use of two cameras each attached to a different FPGA. Using the three-dimensional coordinates of the tracked object, we evaluate the positions of the boids every frame and render them on screen. We implement this on a pair of Urbana FPGAs.

Index Terms—Digital systems, Field programmable gate arrays, Stereo vision, Boids, Rendering

I. INTRODUCTION

THIS report details the construction of an interactive visual program where the user can herd flocks of virtual birds known as "boids" —on-screen models which behave as birds in a flock, fish in a school, or mammals in a herd —which follow a simple behavioral model based on what each individual on-screen "boid" can perceive around them, aligning with nearby flock mates and travelling towards and around a target in the user's hand. Updating a singular boid's position requires the normalization of several vectors, which presents the need to develop a resource-efficient and low-latency module for vector normalizations. To acquire the three-dimensional position of the target, we implement stereo vision using a pair of FPGAs —each with their own camera —which communicate with each other through the SPI protocol. Using the target's positional information, we compute the positions of the boids every frame in three dimensions and use them to overlay them on the camera footage via a double frame buffer. The double frame buffer also presents a unique challenge since, due to overlaying the boids over the camera feed, there's a limited amount of additional pixels/DRAM requests than can be made every frame, amounting to roughly 25% of the screen's area.

For this project, we have committed to having multiple boids follow the target, moving every frame and being rendered at their computed positions. To demonstrate proper depth calculation, we modify every boid's size based on its depth.

II. PHYSICAL CONSTRUCTION

The system consists of:

- Two Urbana FPGA development boards made by Real Digital, built around a Spartan 7 Series FPGA

- Two 0V5640 cameras. Each FPGA will be equipped with one camera.
- Four jumper cables that connect the PMODB ports of both FPGAs. These wires will transmit data from one FPGA to the other.

A parallel physical system will be utilized, where both cameras are oriented in the same angle (Fig. 1).

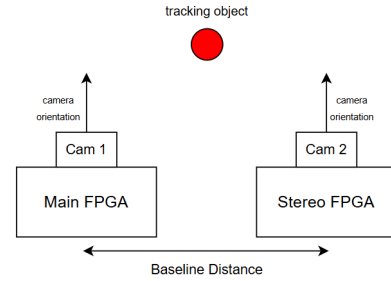


Fig. 1. Diagram of Physical System

III. INTER-FPGA COMMUNICATION

The two FPGAs are connected with jumper cables through their respective PMODB ports. The Stereo FPGA transmits the center of mass values from its attached camera to the Main FPGA through SPI protocol. The Stereo FPGA acts as the SPI controller, constructing DCLK and Chip Select, and the Main FPGA acts as the peripheral that receives the Stereo camera's COM data. This provides the Main FPGA with the necessary data, along with its own center of mass data, to compute the depth of the object and render the boids smaller or larger based on the distance of the object from the cameras.

The xdc and top level files for both FPGAs were modified to respectively output/input the DCLK, CS, COPI, and CIPO signals.

IV. DEPTH CALCULATION

Both the Stereo and Main FPGAs will compute their respective COM values using the modules from lab 5. The Stereo FPGA's purpose is only for calculating and transmitting this x-com value. As described in Section 3, the Stereo FPGA will transmit its COM data to the main FPGA. The Main FPGA will use the Stereo x-com value as well as its own x-com value to compute the depth of the object and do all boid calculations and rendering.

FPGA Stereo

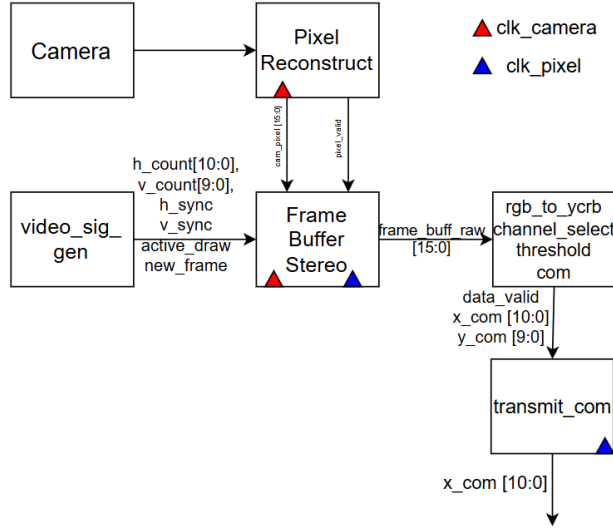


Fig. 2. Stereo FPGA Block Diagram

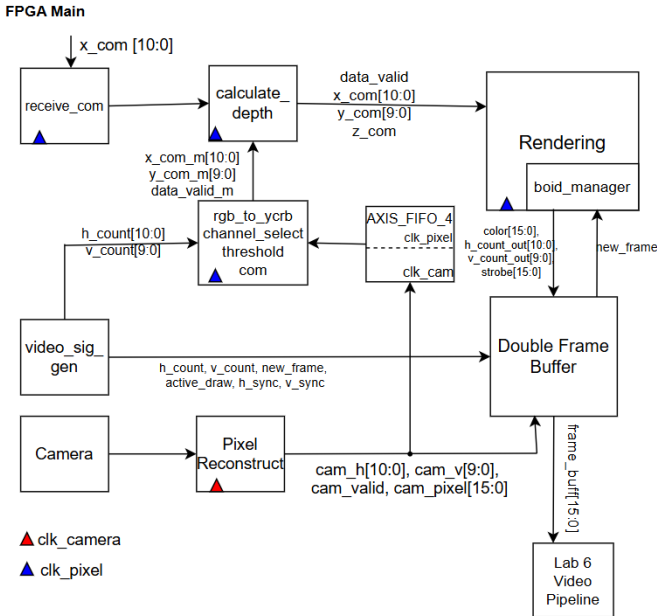


Fig. 3. Main FPGA Block Diagram

The depth is calculated using the following equation ¹

$$Z = \frac{f}{d} \cdot \frac{T}{D}$$

where disparity $D = |com_x_stereo - com_x_main|$, f is the focal length of the camera, d is the physical size of a pixel in camera sensor, and T is the baseline distance between the center of the Stereo and Main cameras. To compute $\frac{f}{d}$, we notice the following equalities:

¹Referenced from Medium's Stereo Vision: Depth Estimation between object and camera, Apar Garg

$$\frac{\text{real world size}}{\text{dist. to cam.}} = \frac{\text{size on sensor}}{\text{focal len.}} = \frac{\text{pix. measured} \cdot \text{pix. size}}{\text{focal len.}}$$

$$\frac{\text{focal len. (f)}}{\text{pix size (p)}} = \frac{\text{dist. to camera} \cdot \text{pix. measured}}{\text{size in real world}} = \frac{N_{px} \cdot D}{L_{mm}}$$

Using a ruler and measuring pad in lab, two measurements of this value were taken. For the first, a 1cm segment (L_{mm}) was placed 15mm away from the camera (D). On the monitor, the segment measured a 15cm length of the screen. The screen itself was measured at 37.8 cm, which gives $N_{px} = \frac{15}{37.8} \cdot 1280px = 507.9px$. These values result in a $\frac{f}{p}$ value of 76.19. For the second measurement, we used a 1cm segment placed 2cm away from the camera, which yielded a 9cm length segment on the monitor screen. Using the same calculation method, this resulted in a $\frac{f}{p}$ value of 60.95. These two measurements were then averaged to 68 in order to minimize noise. The baseline distance between the cameras was set to 20cm. Therefore, depth is simplified to the following calculation ²:

$$Z = 68 \cdot \frac{2000mm}{D} = \frac{204000}{|com_x_stereo - com_x_main|}$$

The values for depth can therefore range from 0 to 204000. In the future, the renderer will likely define a set number of sizes the boids can possibly have. A specific portion of this depth range will correspond to a size. For example if the depth of the tracking object is in the range [200000, 204000], the rendered boids could have a size of 16 pixels. If the depth is much lower, the size of the boids will increase.

V. VECTOR NORMALIZATION (JAIR)

To compute the necessary vectors in the boid computations, it is necessary to normalize vectors. To normalize a vector, it's sufficient to multiply the vector's components by the reciprocal of the vector's magnitude.

We compute an initial approximation to the reciprocal of the square root via the position of the number's most significant bit. Observe that

$$x \approx 2^b \tag{1}$$

where b is the most significant bit of the number x .

Note that

$$b = \text{BIT_COUNT}(x) - 1 - \text{CLZ}(x)$$

, where $\text{CLZ}(x)$ is the count of leading zeroes in x . We've implemented a custom module with a tree-based approach which computes the value combinationally. The result for any n -bit number can be derived from the result of its left and right halves:

Using this, we approximate the reciprocal of the square root as

$$\frac{1}{\sqrt{x}} \approx 2^{(-\frac{b}{2})} \tag{2}$$

²A video of the working depth calculation can be found in the Appendix. As the tracking object moves closer/further from the cameras, the depth value that is displayed on the seven segment display changes accordingly.

We use the following approximation to the reciprocal of the square root in fixed point format

$$\frac{1}{\sqrt{x}} \approx 2^{\left(\frac{3FP_SHIFT-b}{2}\right)} \quad (3)$$

where FP_SHIFT is the amount of precision bits used. The extra $3FP_SHIFT$ factor stems from the need to adjust the number to the fixed point format.

To improve this initial guess, we compute several iterations of the following Newton-Raphson approximation scheme

$$y_{n+1} = \frac{3}{2} - xy_n^2 \quad (4)$$

derived from the Taylor-Series expansion of $\frac{1}{\sqrt{x}}$. Through our C simulation we've found that 4 iterations is enough to achieve visually satisfactory results.

When using an 11 bit integer range and an FP_SHIFT of 12, the normalizer has very accurate results, and the computed vector is always within 10% of the desired vector, which is enough accuracy for our application.

When synthesized for an 11 bit integer range and 12 bits of precision, the normalizer consumes 48 DSP slices. The latency of the normalizer is of $3 + 3I$, where I is the number of iterations used for the inverse square root solver. While it is fully pipelined, all multiplications are assumed to take one cycle, which limits the range and precision of the input vectors due to timing constraints.

VI. BOID COMPUTATIONS (JAIR)

Every boid is associated with a position and velocity vector. Every frame, we add the velocity of the vector multiplied by a small Δt to its position. We also calculate the acceleration of the boid based on

- 1) Normalized offset to the target's position. Used in computing the target force.

$$\frac{target_pos - boid_pos}{||target_pos - boid_pos||} \quad (5)$$

- 2) Average direction of other boids in the perception radius. Also known as the average flock heading. Used in computing the alignment force.

$$\sum_{j \in FLOCK_i} \hat{v}_j \quad (6)$$

where $FLOCK_i$ is the set of indices of boids within the perception radius of boid i , and $\hat{v}_j$ is the normalized velocity vector for boid j .

- 3) Normalized offset to the average position of other boids in the perception radius. Also known as the flock center. Used in computing the cohesion force.

$$o = \left(\sum_{j \in FLOCK_i} pos_j \right) - pos_i \quad (7)$$

- 4) Average of the negative offset to the position of other boids. Also known as the separation heading. In essence,

this force makes sure that boids don't fly over each other. Used in computing the separation force.

$$\sum_{j \in S_i} pos_j - pos_i \quad (8)$$

where S_i is the set of indices of boids within the avoidance radius of boid i .

and multiply it by Δt before adding it to the boid's velocity.

Each of the above results is computed into a steering force as

$$\vec{a}_i = \vec{p}_i - \vec{v}_i \quad (9)$$

where $\vec{p}$ is one of the influencing vectors computed above. These results are then clamped to have a maximum magnitude equal to a constant maximum steering force.

These contributions to the acceleration are then summed and added to the boid's velocity as follows

$$\vec{v}'_i = \vec{v}_i + \vec{a}_{i\text{target}} + \vec{a}_{i\text{align}} + \vec{a}_{i\text{cohesion}} + \vec{a}_{i\text{separation}} \quad (10)$$

The velocity's magnitude is then clamped between a constant minimum and maximum speed. This new velocity is then used next frame to update the boid's position.³

The *boid_update* module computes the boid's new velocity, which is passed back to the *boid_manager* module. For this report, it only implements the first of the four steering forces, which is responsible for leading the boid towards the target. To do this, it implements an FSM comprised of the following states:

- 1) *IDLE*
- 2) *NORMALIZE_OFFSET*
- 3) *GET_STEER*
- 4) *NORMALIZE_STEER_TARGET*
- 5) *CLAMP_STEER*
- 6) *GET_ACCELERATION*
- 7) *GET_VELOCITY*
- 8) *NORMALIZE_VELOCITY*
- 9) *CLAMP_VELOCITY*

Of relevance here is that, to clamp a vector between given minimum and maximum magnitudes, it is necessary to normalize it beforehand. Due to resource limitations, it is impossible to instantiate a separate normalizer module for each of these operations, so instead we have opted for a blocking implementation during which the *boid_update* module changes the inputs to its *normalizer* sub-module and waits for results. To implement the rest of the steering forces, it would then be necessary to add stages for the computation of the necessary data as well as the necessary normalization stages before each steer force computation and subsequent clamping of the output value.

The current implementation of the *boid_update* module has a 51-cycle latency. Within a single frame and running on the pixel clock, we can compute and update at least 19000 boid positions, even when accounting for *boid_manager* overhead.

The *boid_manager* module responsible for passing the necessary parameters to *boid_update* for every boid implements an FSM comprised of the following states.

³Boid computation formulas referenced from Sebastian Lague's "Coding Adventure: Boids" youtube video

- 1) *IDLE*
- 2) *POS_UDPATE*
- 3) *POS_UDPATE2*
- 4) *UPDATE_START*
- 5) *UPDATE_WAIT*

During the *IDLE* state, the module awaits the *new_frame* signal before saving the currently inputted target coordinates and transitioning to the *POS_UDPATE* state. In this state, the module updates all boid positions by adding their respective velocities scaled by a small positive number. Then, during *POS_UPDATE_2*, all positions are boundary-checked and wrapped around the screen if necessary. During *UPDATE_START*, we give the *data_in_valid* signal to the *boid_update* module and then transition to the waiting state, where the current boid index is incremented after the *boid_update* module's *busy* signal goes low. This continues until all boids velocities have been updated, at which point the module returns to the *IDLE* state.

VII. RENDERER

The renderer is responsible for sending pixels to the frame buffer to be written into memory to reflect current boid positions. In order to only send one tlast signal from the renderer per frame, we need a one-cycle high signal per frame that the renderer can use to know when to start sending that frame's boid pixels.

A. Generating a start signal

To generate a clean, single-cycle pulse in the pixel clock domain once per frame, we use the camera's *h_count* and *v_count* values to detect when the camera has reached a specific frame region (we chose close to the end of the frame). Because the camera clock and pixel clock are asynchronous, a one-cycle pulse generated in the camera domain might be missed in the pixel domain. To avoid this, we define a short window high the last in-frame line (when *camera_v_count* == 719 and *camera_h_count* is between 1200 and 1279). This window spans 80 camera-clock cycles at 200MHz, guaranteeing that it will be observed at least once by the pixel clock at 74.25 MHz. Since pixel clock is around a third of the speed of camera clock, an 80 cycle window should be long enough for pixel clock to pick up the signal. When the pixel domain logic sees this window and is not already in the countdown period, it produces a one-cycle pulse and starts a countdown timer. The countdown timer prevents multiple pulses from being generated while the window is active, ensuring that exactly one high signal is emitted per frame. The renderer uses this pulse as a reliable "start of frame" signal to begin sending the boid data to the frame buffer.

B. Depth to Size Conversion

The renderer uses the center of mass data from the previous frame to pass into the boid manager, which computes the new boid locations in the current frame. The output from the boid manager tells the renderer what *h_count* and *v_count* values to send to the frame buffer. The depth value is used to determine

the size of each boid, where the depth and boid size are inversely related. In our current implementation, the depth of the tracking object, *d*, is scaled from 1 to *BOID_DEPTH*. The larger *BOID_DEPTH*, the more noticeable the size difference of the boid is as we move further away from the camera.

C. Sending Write Requests

Since the DDR3 controller only accepts 128-bit chunks, sending render data pixel by pixel causes extensive overhead since we are only sending and receiving 1 pixel's data rather than the 8 we are allowed. We are rendering boxes that start at an 8-pixel chunk on the screen (*h_count* [2:0] = 0). If the boids' *x* positions, received from *boid_manager*, don't align with the start of an 8-pixel chunk, we round down the positions down to the closest multiple of 8 subsequently round up the width of each boid to the nearest multiple of 8. In doing so, we make only an eighth of the number of dram memory requests.

In the main FPGA, an AXI FIFO is used outside the frame buffer to cross clock domains from camera clock to pixel clock. This FIFO supplies the current frame's pixel data from the camera directly to the center of mass module instead of through the frame buffer as in previous labs. The resulting COM is therefore be up to date rather than have a 1-frame delay. The value will be used to compute the boids' new positions in the next frame. Since the system cannot compute COM until a full frame has been written, the pipeline delays the boid calculations by one frame. Therefore, frame *N*'s COM is computed immediately after writing Frame *N* and this COM is used by the renderer for boid overlay data in Frame *N+1*. When rendering the first frame, a hard-coded COM value will be used as there is no previous COM value. Using the previous frame's COM to calculate the current frame's boid positions provides the v-sync period of the previous frame and the writing period of the current frame to complete the computations necessary to overlay boids onto the current frame.

VIII. DOUBLE FRAME BUFFER

A. Renderer Integration

The double frame buffer module handles all memory reads and writes, both from the camera and the renderer, into memory.

Camera pixels are received in raster order and passed through an 8-pixel stacker module forming 128-bit chunks. These chunks are transferred through the camera-specific clock-domain-crossing FIFO into the memory controller domain. The camera stream always writes first to the write frame buffer, ensuring the background frame is fully populated before the boids are overlaid. The renderer detailed in the previous section runs in the pixel clock domain and computes the boid overlays. The rendered boid pixels are sent to the frame buffer in multiples of 8, where all 8 pixels are contiguous. This data passed through the render-specific AXI FIFO to cross into the controller clock domain. Both camera and render FIFOs pass their respective data to the traffic generator. Since boids are not contiguous on the screen, the frame buffer also

Module	LUTs	FFs	DSPs
<i>spi_per</i>	21	61	0
<i>calculate_depth</i>	511	317	0
<i>normalizer3</i>	463	340	48
<i>boid_update</i>	1106	1011	60

TABLE I

RESOURCE CONSUMPTION INCLUDES SUBMODULES.

calculates the 128-bit chunk index in memory of the 8-pixel chunk received from the renderer as follows:

$$\text{idx} = 160 \cdot v_count + \frac{h_count}{8}$$

This index is passed through the traffic generator as the memory request address instead of relying on an event counter like the camera write requests.

B. Switching Between Buffers

Addressing will be handled in the traffic generator using event counters and offset registers. The two frame buffers are implemented with two base addresses. The write frame buffer's base address in DRAM is initially set to 0 and the read frame buffer's base address is initially set to 115200 (both buffers are contiguous in memory). The traffic generator first sends the camera write requests to memory. Once the current frame's camera data is written, the traffic generator allows the render data to be written into memory, overlaying the camera data. This order ensures correct overlaying of boids onto the camera image and thus, the render data has only the vsync period of the camera to be written into memory in order for the next frame's camera data to be sustained. The traffic generator swaps the read and write buffers once the ray data has been written into memory, and the video pipeline reads from the frame we previously wrote into.

The camera and renderer write to physical addresses in DRAM, computed as $\text{write_buffer_base} + \text{write_chunk_index}$. The write index for the boids are computed as described previously. HDMI reads use $\text{read_buffer_base} + \text{read_request_chunk_index}$. A command FIFO records each issued request so that it can identify reads and correctly route them to the video pipeline.

Similar to Lab 6 the ddr3 controller returns 128-bit chunks from the READ frame buffer to the read FIFO, where they are unstacked back into 16-bit pixels for the video pipeline to consume. Read requests proceed concurrently with the write requests.

IX. DESIGN EVALUATION

A. Double Frame Buffer and Rendering Bottleneck

The objective of the current implementation is to be able to render as many boids as possible and have them change sizes appropriately. Writing camera pixels into the frame buffer consumes most of the frame time, and the system cannot write an amount of boid pixels that surpasses the amount of cycles in the remaining frame time. This amount of pixels is

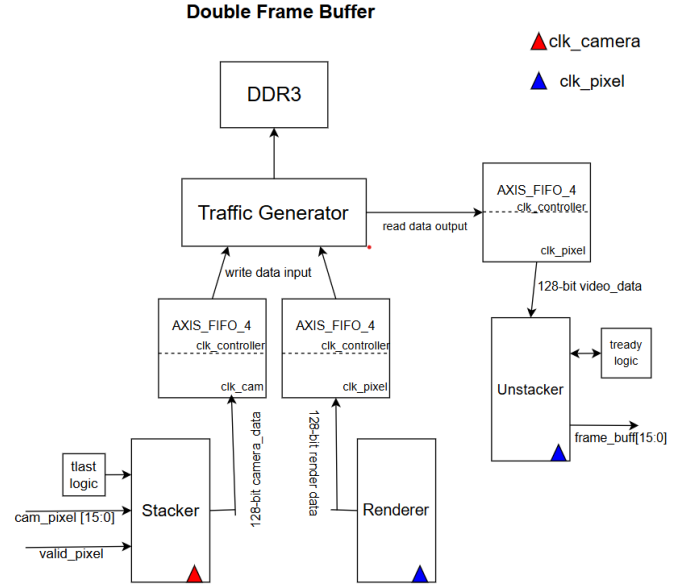


Fig. 4. Double Frame Buffer Diagram

roughly equal to the amount of cycles in the v-sync period. This upper bounds the size and quantity of the boids by the amount of pixels for each boid when they're closest to the screen. We've found that we can comfortably render 16 boids as 96x96 squares—or 147456 pixels—whilst remaining under the frame time constraint. As will be evident from the following evaluations, this is the design's current bottleneck and is the first area to be improved upon in the next iteration as both the depth calculation and boid updates can handle increased throughput.

Additionally, we could cut off the additional pixels sent via the 8-pixel packing by modifying the strobe signal to the DDR3 controller to only write pixels included in the square region. This would remove the current rendering artifact where box widths change only in increments of 8.

B. Communication Protocol and Depth Calculation

Since the communication protocol only needs to communicate 22 bits for the COM information every frame regardless of any changes to the rest of the design, it will not need any changes or improvements. Similarly, the depth calculation module deals with known constants and is mostly a wrapper for a fixed point divide. Even if a different camera were to be used, it would only require constant values and/or bit widths to be changed in this module.

C. Vector Normalization and Boid Updates

The biggest source of resource allocation in the design comes from the normalizer instanced by the *boid_update* module. Forty-eight of the sixty DSP slices in the module come from the normalizer instance. Most of these come from the pipelined Newton-Raphson iterations, which instance multiplications between numbers with result bit-widths of up to $4FP_RANGE + 3FP_SHIFT + 2$. This results in timing and resource utilization issues for higher integer ranges and

precision widths. This problem could be solved by splitting each multiplication into different steps —each of them fitting into a single DSP slice. This would drastically reduce resource utilization and improve throughput at the cost of increased latency. Increased latency, however, directly influences the amount of boids that can be updated every frame due to the *boid_manager*'s blocking implementation of updates and the choice to instance only one normalizer for *boid_update*.

X. COLLABORATION

The report was written by both authors, with Sreeja Thipireddy being responsible for the Physical Construction, Inter-FPGA Communication, Depth Calculation, Renderer, and Conclusion sections, and Jair Santana being responsible for the Introduction, Vector Normalization, Boid Computation, and Design Evaluation sections. The rest of the sections were written in collaboration.

Sreeja Thipireddy was responsible for the implementation of the inter-FPGA communication protocol, depth calculation module, double frame buffer, and modifications to the traffic generator. Jair Santana was responsible for the implementation of the normalizer, boid manager and boid update modules. The renderer module was implemented in collaboration.

Sreeja Thipireddy was responsible for editing the demonstration video submitted along this report.

XI. CONCLUSION

This project presents a complete FPGA-based system for real-time stereo tracking and interactive boid herding. By integrating dual-camera depth calculations, fixed-point boid computations, and rendering through a double frame buffer, this project aims to demonstrate a functional pipeline capable of tracking a user-controlled object in 3D-space and driving accurate boid movement on the screen. Future work will focus on implementing the rest of the forces for the boid position computations to achieve a more visually satisfying simulation, as well as improving the rendering technique for the boids.

APPENDIX A DEMONSTRATIONS

- 1) Video link of working depth calculation:
<https://youtube.com/shorts/44YEDf6AcTw?feature=share>
- 2) Link to final video: <https://www.youtube.com/watch?v=L-vBwgjFy7U>

APPENDIX B SOURCE CODE

Source code for the project can be found at <https://github.mit.edu/6205F25/fa25-6205-team28>. Building the project requires vivado; running the included testbenches requires icarus and coco-tb.

ACKNOWLEDGMENT

The authors would like to thank Joe Steinmeyer and Hasan Zeki Yildiz for their invaluable feedback regarding project direction and design decisions.