

CORNEA

Nigel Norman

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

Thu Le

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

Abstract—This project simulates an event-based camera from standard luminosity-formatted video to facilitate low-latency ego-motion estimation and optical flow vector field display. Brightness changes between frames are converted into asynchronous events, which generate a time surface encoding recent motion activity. Spatial and temporal gradients of this surface produce a vector field of local motion, from which global ego-motion is estimated and visualized.

Index Terms—Field Programmable Gate Arrays, Neuromorphic, Ego-motion, Vector fields, Image Processing, Optical Flow

I. INTRODUCTION

Modern robotic and vision-based systems require the ability to interpret motion from the captured scene in real time. This task can involve both the motion of external entities, a process referred to as optical flow, and the motion of the system itself, or ego-motion. As it stands, both approximations remain a computational challenge due to the sheer number of high-frequency calculations required for every pixel while under timing, resource, and hardware constraints. These difficulties are further exacerbated in an environment with limited texture or range of visual information, dramatic changes in lighting, or loud visual noise. As such, the difficulty of motion estimation stems from both architectural limitations and ambiguity from the sensor data.

We put forth a design that implements ego-motion approximation in real time, utilizing the FPGA and its capabilities to parallelize calculations and operate with an asynchronous data stream.

A. Overall System - High Level

The system uses the OV5640 camera module and configures it to output luminosity values. These pixels are transmitted serially, but downstream calculations require an asynchronous stream of events. Here, the ego-motion pipeline begins by thresholding the difference in luminosity values for every pixel it receives to generate events. With this, gradients with respect to the x-y axes are calculated, encoding both spatial and temporal information. The frame is then partitioned into a grid of blocks. using the previously calculated gradients to produce an optical flow vector field. This vector field is then convolved to calculate the camera's movement. The system displays this vector field as a matrix of arrows whose color corresponds with magnitude and direction indicated by the arrowhead.

II. CAMERA

As the downstream ego-motion calculations depend on the luminosity values of the visual input, the OV5640 Camera can be configured to output pixel data in 8-bit luminosity (Y) formatting. The default configuration of the OV5640 sends 16-bit RGB image data through two sequential byte packets over 8 parallel 1-bit wires. By transitioning to 8-bit Y Data, the module processing the camera's output signals can be simplified. To supplement raw pixel data, the processing module will also extract coordinate, image data, and pixel validity information.

A. Video Stream - Thu

Given that the camera operates on a 50MHz clock, while the system operates at 74.25MHz, a FIFO is used to perform clock domain crossing early on in the pipeline. The system's frame of consideration is a cropped 256x256 region in the center of the camera, not a down-sampled version of the frame. As such, all downstream calculations only receive data if it is within this region. This is to work with a simplified resolution while considering memory limitations, and to avoid anti-aliasing with down-sampling.

The video displayed by the system can be broken down into two components: the background and the overlay. The existing background options are the raw camera footage and the event generation mask, while the overlay is another mask provided by the arrow-generation pipeline. Both components are stored in their respective frame buffers, which are read using the same signals during the HDMI's raster scan. A MUX then transmits the appropriate background or overlay pixel, depending on user selection, to the TMDS bus.

III. EGO-MOTION

A. Event Generation - Thu

The camera's serialized stream of pixels is processed to produce an asynchronous stream of event data. Given a pixel with position (x, y) , report an event if the change in its luminosity, at the current time t and the time it last reported an event t_0 , exceeds an arbitrary threshold:

$$E(x, y, t) = |L(x, y, t) - L(x, y, t_0)| \geq \delta$$

Implementing this on the FPGA, a BRAM frame buffer maintains a history of each pixel's luminosity value at its most

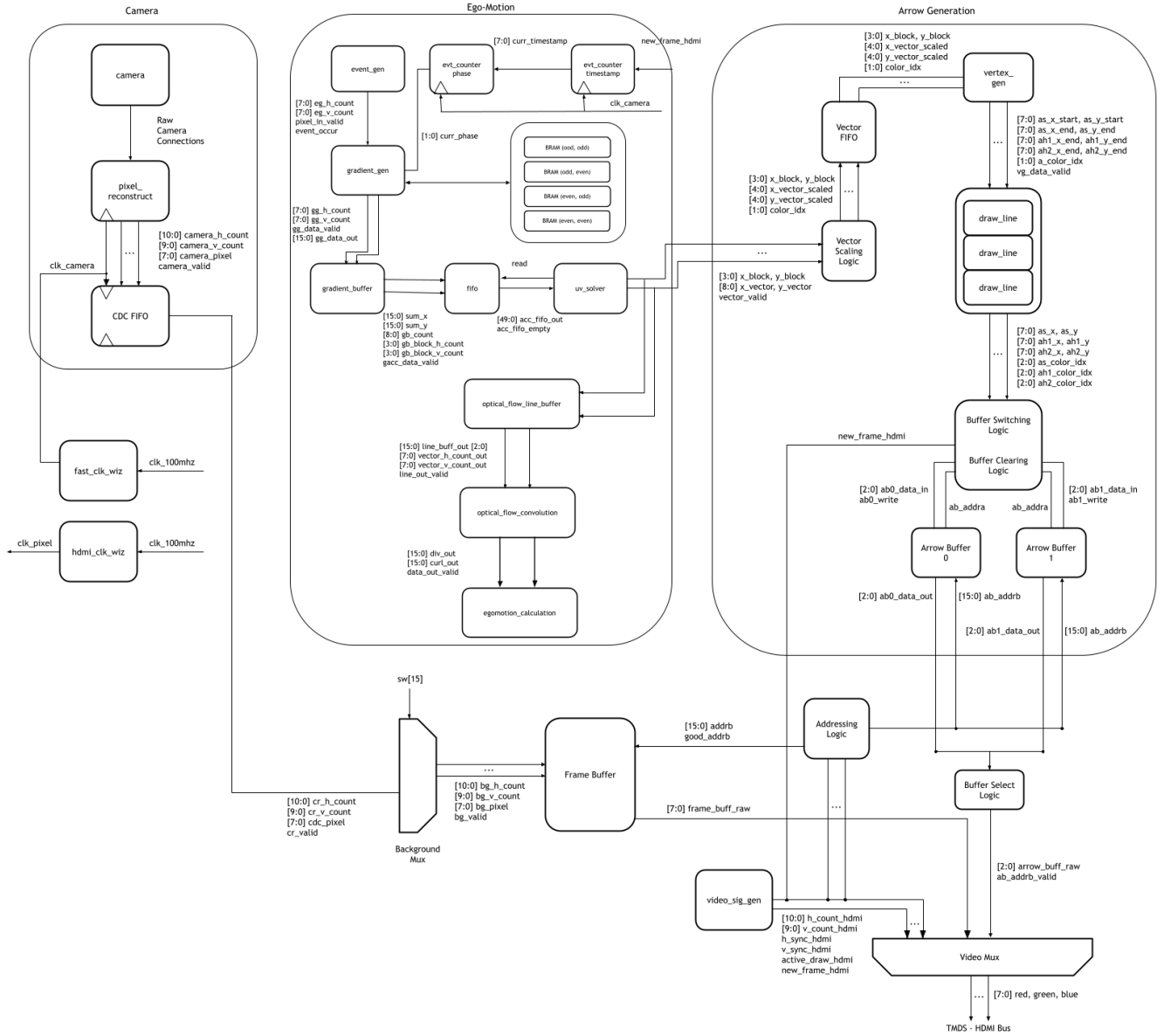


Fig. 1. Block Diagram - Overall System (Simplified)

recent event. The serial input stream allows a simple event-triggered counter to handle the addressing logic, sending a read request for every valid input. Once the BRAM outputs the desired pixel, the above calculation is then performed. If an event occurs, the module updates the frame buffer with the new luminosity value and outputs the corresponding pixel with an event flag.

B. Gradient Generation - Nigel

Events, non-events, and their respective locations are streamed into this module for proper gradient generation. To keep track of motion over time, the system stores timestamps of events when they occur. This effectively creates a surface

of event occurrences across the 256 x 256 pixel space. By referencing neighboring pixels, the system can interpret spatial information to derive direction and magnitude of movement in both x and y directions. Given a pixel at location (x, y) , its timestamp is $T(x, y)$:

$$T_x = T(x + 1, y) - T(x - 1, y) \quad (1)$$

$$T_y = T(x, y + 1) - T(x, y - 1) \quad (2)$$

The relevance and relationship of the timestamps to each other are created through a sliding window. Stored timestamps are represented by two values:

- phase: 2-bit counter representing the frame block

- `curr_timestamp`: 6-bit counter representing the frame number

The timestamps themselves are generated with an event-triggered counter based on the `new_frame` signal from the `pixel_reconstruct` module. When the counter reaches its max value, the phase bit is incremented.

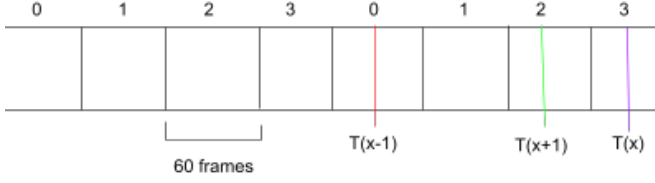


Fig. 2. Phases described by text on top of boxes, $T(x)$ is timestamp of center pixel where event has happened, while $T(x+1)$ and $T(x-1)$ are the timestamps to the right and left of center pixel. Gradient calculated by using phase and timestamps as measures for distance

The phase bits allow proper comparisons with the current timestamp to the one residing in BRAM. If the distance is deemed too far, approximately two timestamp cycles, then the system records a "do not touch" value outside of the range of a normal timestamp. When calculating a gradient that includes this flag, output a zero.

For every pixel that has an event, calculate its corresponding gradient. This involves four read requests for the neighboring timestamps and one write request for the current time stamp at the pixel location. If the center pixel is located outside the window of consideration, the system needs to rewrite it in BRAM.

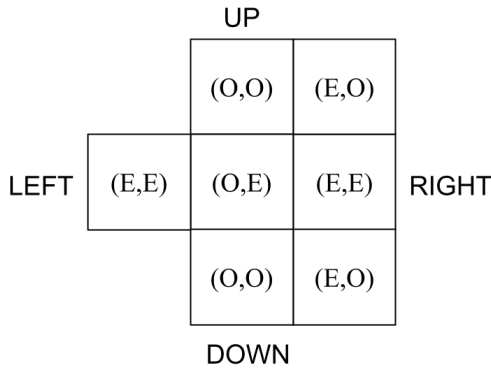


Fig. 3. Each pixel has a pair (h_parity , v_parity), representing which BRAM it will be addressed to. Left and right are addressed to the same BRAM, as well as pixels labeled up and down.

The design implements four BRAMs with depth 16384 and width 10, each for a different group of pixels determined by the combination of h_count and v_count parity. In this manner, it is possible to send four read requests in parallel with only two BRAMs. The other two BRAMs are then free

to write the current timestamp if there is an event (done in the 'A' port), read if there is no event, and/or use the second (B) port of a BRAM to rewrite a "do not touch" value if its value was previously determined as irrelevant. This works solely because of the fact that the w/ parity BRAMS we use never write and read from the same BRAM in the same cycle. When timed correctly, writes and reads do not overlap.

As such, the sub-system's pipeline only has 3 cycles of latency to compute gradient.

C. Block Gradient Accumulation - Nigel



Fig. 4. Reduced Gradient Processing Block Diagram

For each, is assigned a 16 x 16 pixel block to contribute to based on its h_count and v_count . Within the block, every pixel's x and y gradient is aggregated towards two accumulators.

$$S_y = \sum_{i=1}^{256} T_{i,y}, \quad S_x = \sum_{i=1}^{256} T_{i,x}, \quad (3)$$

To save resources, only pixels that have events within the current frame are relevant to accumulation because they are where motion occurs.

When the end of the block is reached, the module then writes component sums to a FIFO along with block location and the number of events within the block. The FIFO is necessary because we valued modularity. On the last row of the block the aggregator outputs every 16 cycles, but may take longer/shorter if we are to change resolution or scale.

D. UV Solver and FIFOs - Nigel

This module simply grabs the accumulators from the FIFO and runs a division and normalization pipeline to get the resulting aggregate x and y gradient direction and magnitude.

$$u = \frac{S_x}{C}, \quad v = \frac{S_y}{C}, \quad \text{where } C \text{ is the \# of events per block}$$

We used two signed radix-2 division modules to complete the calculations for both the x-component and the y-components. Because both u and v are 16 bit integers, there is a maximum of 16 cycles to generate proper u and v for each block.

Intuitively, large u and v components should mean faster motion, yet large time gradients suggest slow motion since there is more time between neighboring event timestamps. So, magnitudes of the u and v components must be linearly inverted compared to time-surface gradients, where small time-surface gradients turn into large u and v components, and vice-versa. This is done simply by returning 128 - magnitude, for each component.

E. Ego-motion calculation - Nigel

Block vectors are written into series of 2 line buffers of depth 16 and width 16, implemented in registers, which hold both the u and v component for each pixel on 2 subsequent rows. The incoming u and v on the third row is added to make a three row, one column outputs for both u and v components.

We use this line buffer output to create a 3 x 3 window for each vector component (u and v). Operations are performed on each component window to find divergence and curl. These are then summed across each block vector within the frame as well as their respective components. The relative magnitudes describe direction and magnitude of camera motion. For reference, divergence and curl are calculated like so:

$$\text{div} = u(x+1, y) - u(x-1, y) + v(x, y+1) - v(x, y-1), \quad (4)$$

$$\text{curl} = v(x+1, y) - v(x-1, y) - (u(x, y+1) - u(x, y-1)). \quad (5)$$

Through experimentation, a weighted sum is used to classify whether planar rotation, axial translation, or planar translation is occurring.

IV. ARROW GENERATION

The arrow generation sub-system is powered by Bresenham's Algorithm. Given the start and end vertices of a line, the algorithm serially outputs the coordinates of all the pixels on the specified line. The following logic is designed to adapt to this data flow. The ego-motion pipeline outputs the $\langle x, y \rangle$ components of the flow vector, which is processed to extract the start and end vertices of three lines: the arrow shaft, one line of the arrowhead, and the other line of the arrowhead. The output of Bresenham is stored into a buffer and treated as a mask, where downstream logic decides whether or not to display the pixel. To manage handshake agreements between the FIFO, vertex generation, and drawing engines, an FSM-approach is used.

A. FIFO - Thu

Buffering the read and write interactions between the ego-motion pipeline and the arrow generation pipeline, a FIFO is used as a cautionary measure as read and write commands are issued at different, varying rates.

From the ego-motion pipeline, scale the provided vector to fit within a 16x16 pixel block. To color-code the arrow based off the vector's magnitude, add the x-y components to gain a relative sense of magnitude and threshold the sum. Note this method only allows for a limited range of colors. For a given vector $\langle x, y \rangle$, determine its color as:

$$\text{color}(\langle x, y \rangle) = \begin{cases} \text{blue} & \text{if } x + y < \sigma_S \\ \text{green} & \text{if } x + y < \sigma_M \\ \text{red} & \text{else} \end{cases}$$

The thresholds σ_S and σ_M can be arbitrarily set based off the maximum possible value of $x + y$. With this, pass the

scaled vector, its block coordinates, and its color index to the FIFO.

B. Vertex Generation - Thu

To display one arrow, the sub-system draws three lines to represent the arrow shaft and arrowhead. The drawing engines assume these input vertices are in global coordinates for a 256x256 pixel frame. However, the vertex calculations work in coordinates local to a single 16x16 pixel block, and are translated to global for the final output. In order to initiate the drawing process, the start and end vertices must be provided for each line. Define these values as the following:

- $\langle \text{as_x_start}, \text{as_y_start} \rangle$: The start vertex of the arrow shaft
- $\langle \text{as_x_end}, \text{as_y_end} \rangle$: The end vertex of the arrow shaft
- $\langle \text{ah1_x_end}, \text{ah1_y_end} \rangle$: The end vertex of the arrowhead line
- $\langle \text{ah2_x_end}, \text{ah2_y_end} \rangle$: The end vertex of the other arrowhead line

The end vertex for the arrow shaft is treated as the start vertex for both arrowheads.

1) *Arrow Shaft - Thu*: The most straightforward set of vertices to compute. Given a vector, the system centers the resulting arrow to make full use of the 16x16 pixel block. Centering offsets for both x-y directions are determined by dividing the x-y components in half, which are then applied to the center pixel of a block.

$$\text{as_x_start} = x_{\text{center}} - \frac{x}{2}$$

$$\text{as_y_start} = y_{\text{center}} + \frac{y}{2}$$

The end vertex is then simply found by adding the original $\langle x, y \rangle$ components.

$$\text{as_x_end} = \text{as_x_start} - \frac{x}{2}$$

$$\text{as_y_end} = \text{as_y_start} - \frac{y}{2}$$

The division can be simplified with bit-shifting.

2) *Arrowhead - Thu*: The system defines eight possible orientations for the arrowhead in the cardinal directions and their sub-directions. It determines the corresponding orientation of a given vector using the absolute x-y components to define three signals: horizontal, vertical, and diagonal.

$$\text{horizontal} = 2|y| < |x|$$

$$\text{vertical} = 2|x| < y$$

$$\text{diagonal} = \overline{\text{horizontal}} \cdot \overline{\text{vertical}}$$

To derive the horizontal signal, define θ_H to provide a range of angles $(-\theta_H, \theta_H)$, where any vector whose angle falls inside this range is considered horizontal. Out of symmetry, define $\theta_H = 22.5^\circ$ and for a given vector $\langle x, y \rangle$:

$$\frac{|y|}{|x|} = \tan(\theta_H)$$

$$|y| < |x| \tan(\theta_H)$$

$$|y| < 0.4|x|$$

$$2|y| < |x|$$

To simplify the math on the FPGA, we approximate 0.4 as 0.5, allowing us to use bit-shifting. The vertical signal is derived in a similar manner based off a threshold angle θ_V . If neither of these signals are true, the vector falls outside of the defined ranges and is considered diagonal.

From this information, the system uses the signs of the x-y components to judge which of the eight directions (E, NE, N, NW, W, SW, S, SE) the vector is facing.

Each orientation has associated offsets to compute the arrowhead vertices:

Direction	x_1	y_1	x_2	y_2
E	-1	1	-1	-1
NE	-1	0	0	-1
N	-1	-1	1	-1
NW	0	-1	1	0
W	1	-1	1	1
SW	0	1	1	0
S	-1	1	1	1
SE	-1	0	0	1

Use the end vertex of the arrow shaft as the start vertex of the arrowheads. Therefore, to compute the end vertices of the arrowheads:

$$\begin{aligned} \text{ah1_x_end} &= \text{as_x_end} + \text{ah1_x_offset} \\ \text{ah1_y_end} &= \text{as_y_end} + \text{ah1_y_offset} \\ \text{ah2_x_end} &= \text{as_x_end} + \text{ah2_x_offset} \\ \text{ah2_y_end} &= \text{as_y_end} + \text{ah2_y_offset} \end{aligned}$$

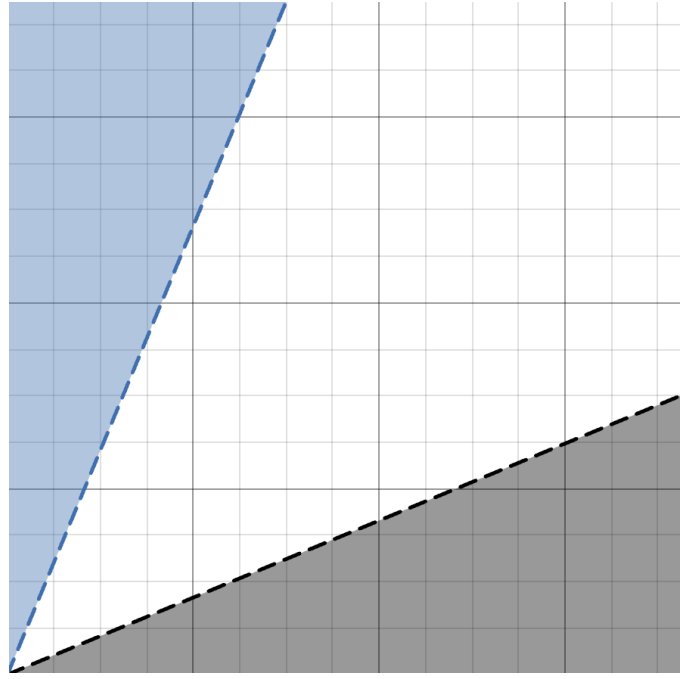


Fig. 5. The definitions for the horizontal, vertical, and diagonal signal the system uses represent threshold angles $\theta_H = \theta_V = 26.5^\circ$. The blue region represents angles considered vertical, and the gray region represents horizontal angles. Outside is the diagonal region.

Given a local coordinate (x_L, y_L) and the block coordinates (x_B, y_B) , find the global coordinates (x_G, y_G) :

$$x_G = x_L + (x_B \cdot 16)$$

$$y_G = y_L + (x_B \cdot 16)$$

C. Drawing Engines - Thu

The drawing engines implement Bresenham's line drawing algorithm as a Finite State Machine (FSM) that incrementally generates integer pixel coordinates along a straight line between two endpoints. The key idea of Bresenham's method is to approximate a line using only integer additions, subtractions, and comparisons. The algorithm achieves this by maintaining an accumulated "error" that tracks how far the current pixel is from the "true" line.

1) *Idle*: In this state, the FSM waits for valid incoming data. When it does, the FSM pipelines the input data for later outputting, records the sign of the x-y components (sx, sy), and extracts the absolute value of the x-y components ($|x|, |y|$).

2) *Start*: The module initializes an `error` variable, which dictates the stepping logic of the line.

$$\text{error} = |x| + |y|$$

3) *Draw*: Every clock cycle in this state, the FSM outputs the coordinate of a pixel belonging on the line. If the current pixel matches the end vertex, the algorithm terminates and transitions to idle state. Otherwise, the module uses `error`

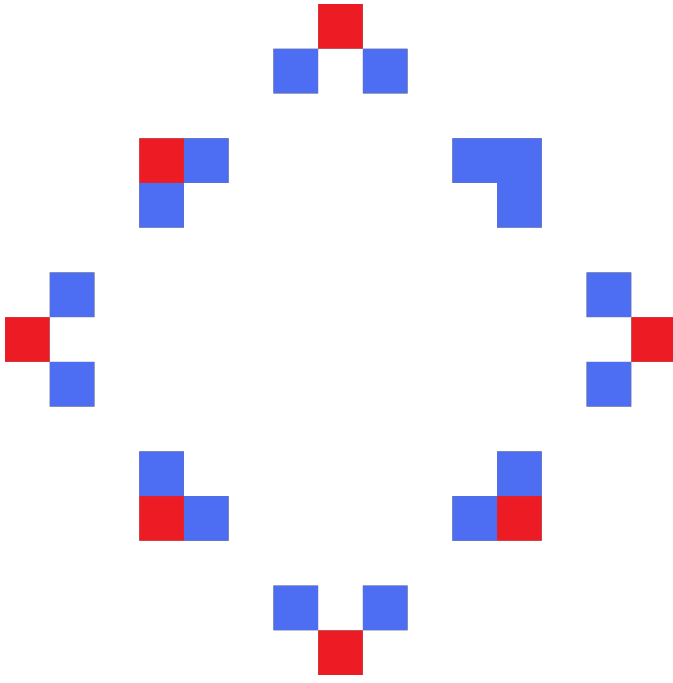


Fig. 6. The arrowhead configurations as specified by the offsets for all eight directions. The red pixel represents the start of the arrowhead.

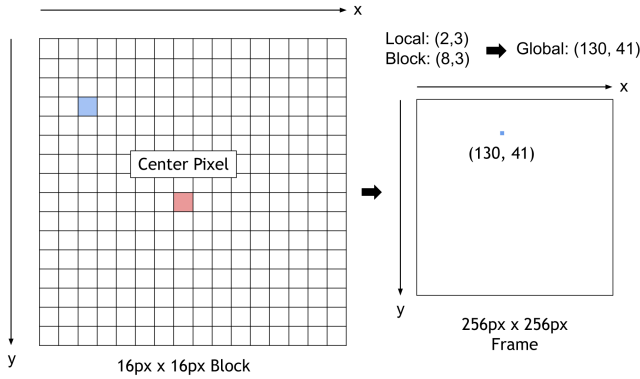


Fig. 7. Converting a pixel from local to global coordinates

to determine how the next pixel steps in the x-y directions. Given the current pixel (x_0, y_0) , determine the next (x_1, y_1) :

$$x_1 = \begin{cases} x_0 + sx & \text{if } 2 \cdot \text{error} \geq |y| \\ x_0 & \text{else} \end{cases}$$

$$y_1 = \begin{cases} y_0 + sy & \text{if } 2 \cdot \text{error} \leq |x| \\ y_0 & \text{else} \end{cases}$$

D. Arrow Buffers - Thu

The pipeline maintains two buffers that store a pixel mask provided by the drawing engines. The system clears primary buffer of data from the previous frame before receiving data from the engines. At the same time, the HDMI sync signals read from the secondary buffer in the same raster-scan order

as the background pixels, displaying the arrow pixels as an overlay.

On a new frame, the primary and secondary buffers switch roles. This double buffering is necessary, as a buffer cannot be cleared while HDMI reads from it.

V. VIDEO STREAM

The system will display visualizations for the intermediate stages of the ego-motion calculations.

1) *Event Generation - Thu*: Given an asynchronous stream of events, color a given pixel white if it reports an event, otherwise color it black. See Figures 9 and 10.

2) *Vector Field - Thu*: For each block in the frame, the system displays an arrow representing the block's aggregate x and y gradients through its angle and magnitude. See Figure 11.

VI. DESIGN EVALUATION

Our latency of the modules are small in comparison to the size of the frame we are creating. The only major delay is waiting for line buffers to fill, which delays our egomotion output on convolution by 1 row (or 256 cycles) compared to the start of the event stream. This doesn't affect the visualization though. Most of our processing includes parallel use of h and v counts from hdmi stream. The highest latency any one of our module produces is 5 cycles.

BRAM is used only for frame buffer, arrow buffers, and timestamp storage. Our design only uses 31.5% of total BRAM capacity.

Also the design meets timing requirements having a positive slack of 4.8 ns.

VII. CONCLUSION

Our project meets the commitment and targets laid out earlier in the semester, with the exception of the time surface decay map. The system implements a simplified time surface for efficiency and improved performance, but as a result, does not encode the information originally intended to be visualized. Furthermore, our team was not able to meet our stretch goal of implementing full-resolution. Instead, the utility of the project comes from the optical flow vectors it computes. The visualization allows users to personally interpret motion, while the FPGA outputs exact motion information for a given block. As such, this finds relevance in any visual system that needs to interpret motion.

VIII. CODE REPOSITORY

<https://github.mit.edu/6205F25/fa25-6205-team61>

REFERENCES

- [1] Chiang, B. & Bohg, J., "Optical and Scene Flow," CS231A Course Notes, Stanford University, Feb. 14, 2022., The Technical Writer's Handbook. Mill Valley, CA: University Science, 1989.

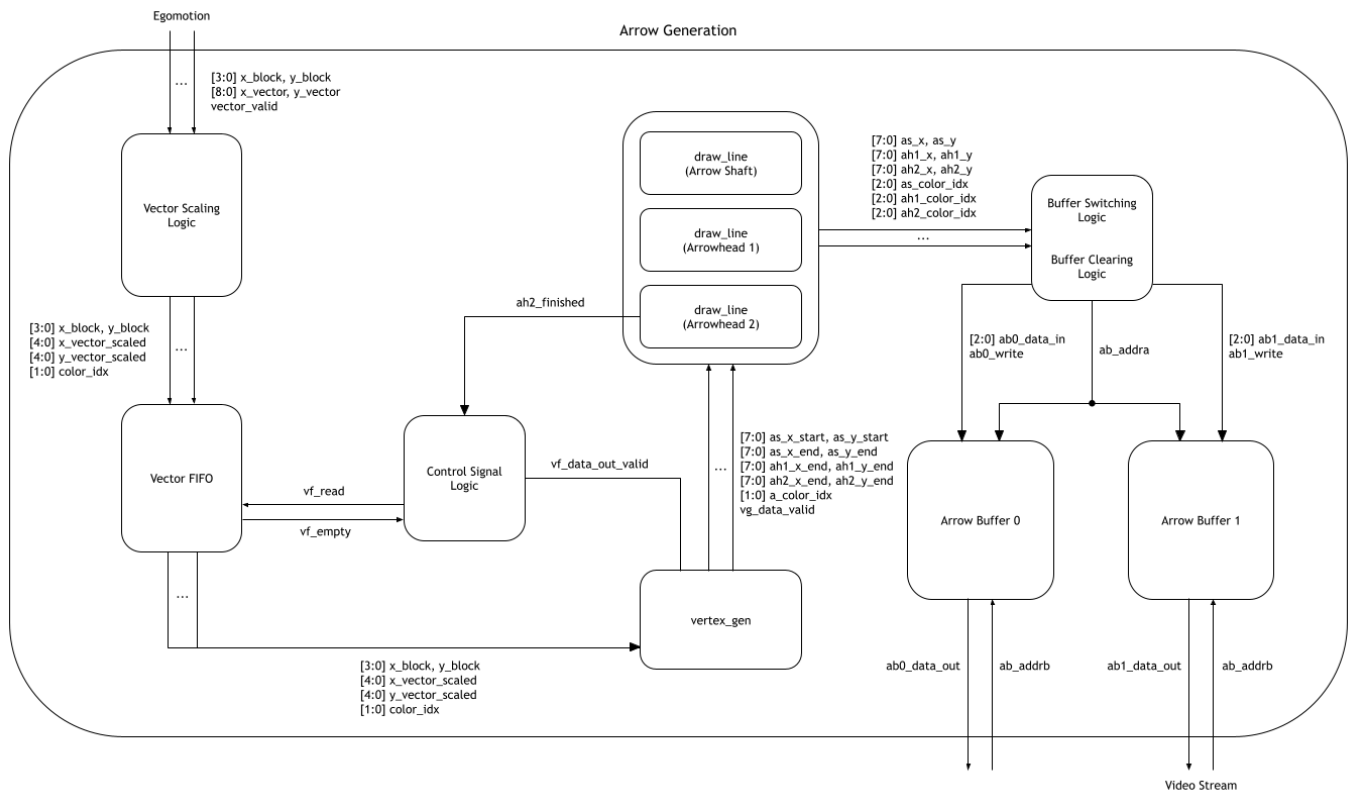


Fig. 8. Block Diagram - Arrow Generation



Fig. 9. Raw camera footage in grayscale formatting.

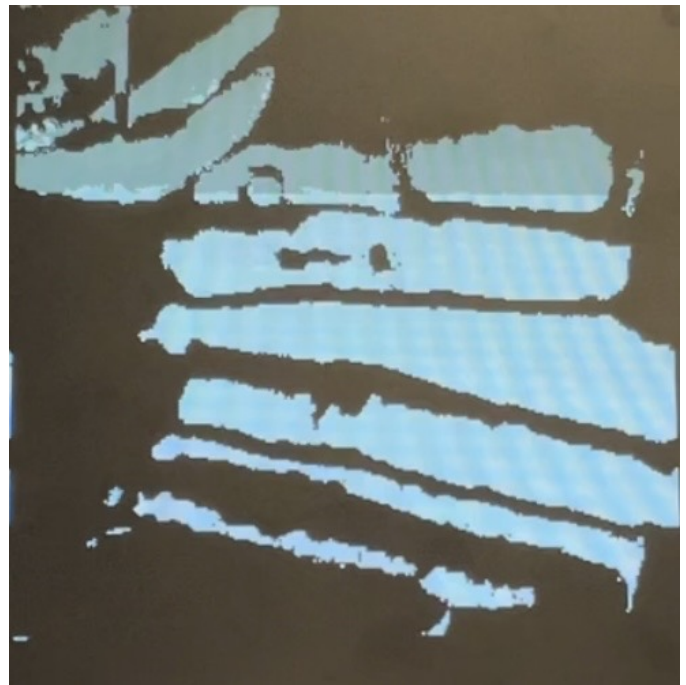


Fig. 10. Visualization of the event generation. A pixel is colored white if it reports an event.

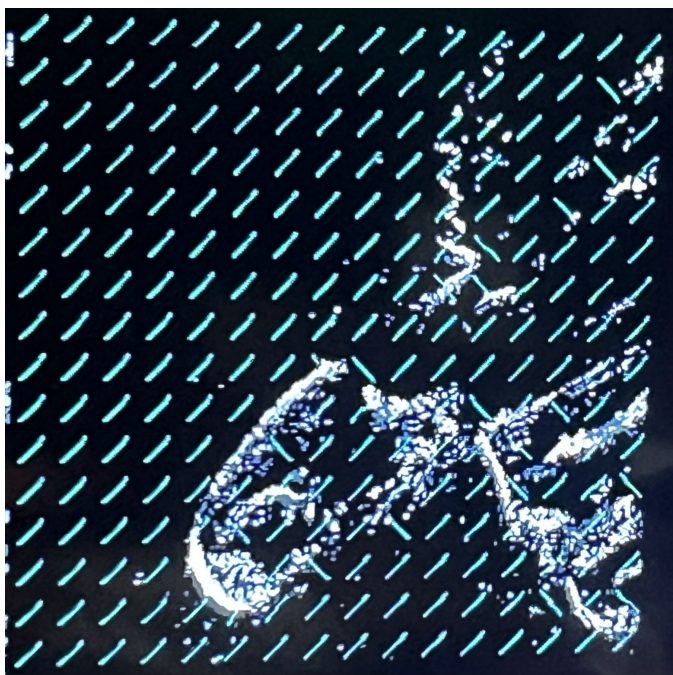


Fig. 11. The optical flow vectors of each block as influenced by the motion of the hand.