Optical FlowPGA Final Report

Maggie Shi
Department of EECS
MIT
manxishi@mit.edu

Christy Li
Department of EECS
MIT
ckl@mit.edu

***Abstract*—We present Optical FlowPGA, an FPGA-based motion tracking system for real-time detection and visualization of moving feature points in video streams with minimal latency. The system implements Harris corner detection in hardware to continuously identify and track salient feature points on moving objects across consecutive frames. By computing spatial and temporal gradients at these corner locations, the system also provides all necessary intermediate computations for optical flow estimation to enable real-time visualization of feature point trajectories overlaid on the original video feed. While a Lucas–Kanade optical flow module was developed and validated against a Python reference implementation during testing, it was not integrated into the final system; however, the existing architecture includes all downstream visualization capabilities and intermediate gradient computations necessary for future integration of Lucas–Kanade or other optical flow algorithms. By exploiting the FPGA's inherent parallelism and a low-latency pipelined architecture, Optical FlowPGA targets real-time computer vision applications such as autonomous driving, robotics, and motion detection systems.**

I. Overview

The primary goal of Optical FlowPGA is to achieve real-time processing of camera footage while maintaining accurate tracking of salient feature points on moving objects. The system processes video in a continuous loop: capturing frames, computing spatial gradients via Sobel convolution, computing temporal gradients through frame differencing with DRAM-stored previous frames, accumulating gradient products over 16×16 windows, identifying corner features through Harris corner detection, and outputting visualizations that track multiple feature points overlaid on the original video feed.

Optical FlowPGA employs a sparse feature tracking approach that focuses computational resources on corners, i.e. the most informative and trackable points in the image, identified through the Harris corner detection algorithm [1]–[3]. The system first computes spatial gradients (I_x, I_y) using 3×3 Sobel filters applied to incoming camera luminance data, then computes temporal gradients (I_t) by subtracting synchronized current and previous frame pixels retrieved from DDR3 DRAM. A multiply-accumulate (MAC) module with a 16×16 cache processes these gradients to compute windowed sums ($\sum I_x^2$, $\sum I_y^2$, $\sum I_x I_y$, $\sum I_x I_t$, $\sum I_y I_t$) over 16×16 pixel regions. These windowed sums serve dual purposes: they feed into the Harris corner detector to identify the top 10 corner features that are spatially well-separated, and they provide the exact intermediate values required for Lucas–Kanade optical flow computation.

This sparse approach exploits the FPGA's parallel processing architecture through pipelined modules that simultaneously compute gradients, accumulate products, and identify trackable features, improving efficiency while maintaining tracking accuracy for moving objects. A Lucas–Kanade optical flow module was implemented and validated against a Python reference implementation but was not incorporated into the final system; however, the existing infrastructure computes all necessary gradient products and maintains the data structures required for future optical flow integration. A top level diagram of the complete system is shown in Fig. 2.

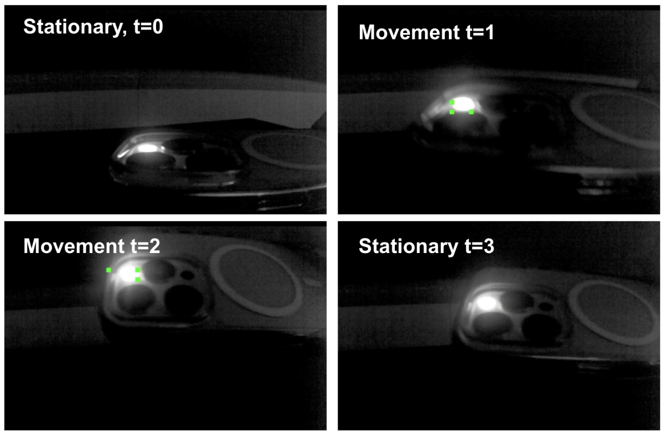

Fig. 1. Continuous streaming detection of the top-k most spatially and temporally salient feature points in a live video feed.

II. Computation

A. Spatial Gradients via Convolution

Similar to our convolution lab, this module takes in 3 vertically stacked pixels, specifically their luminance values. These pixels come in streamed from the camera input data path. Then it constructs 3x3 windows and computes Sobel X and Sobel Y. Every pixel produces data_valid, h_count, v_count, and I_x, I_y out for the next MAC module to use. One issue we ran across was that reusing our exact implementation from the lab caused the convolution to be the longest path and our build to violate timing. We found that the bottleneck was a long adder chain of 9 sequential additions. We overcame this challenge by splitting up the additions into 3 groups of 3 and then summing the groups, creating a much more balanced

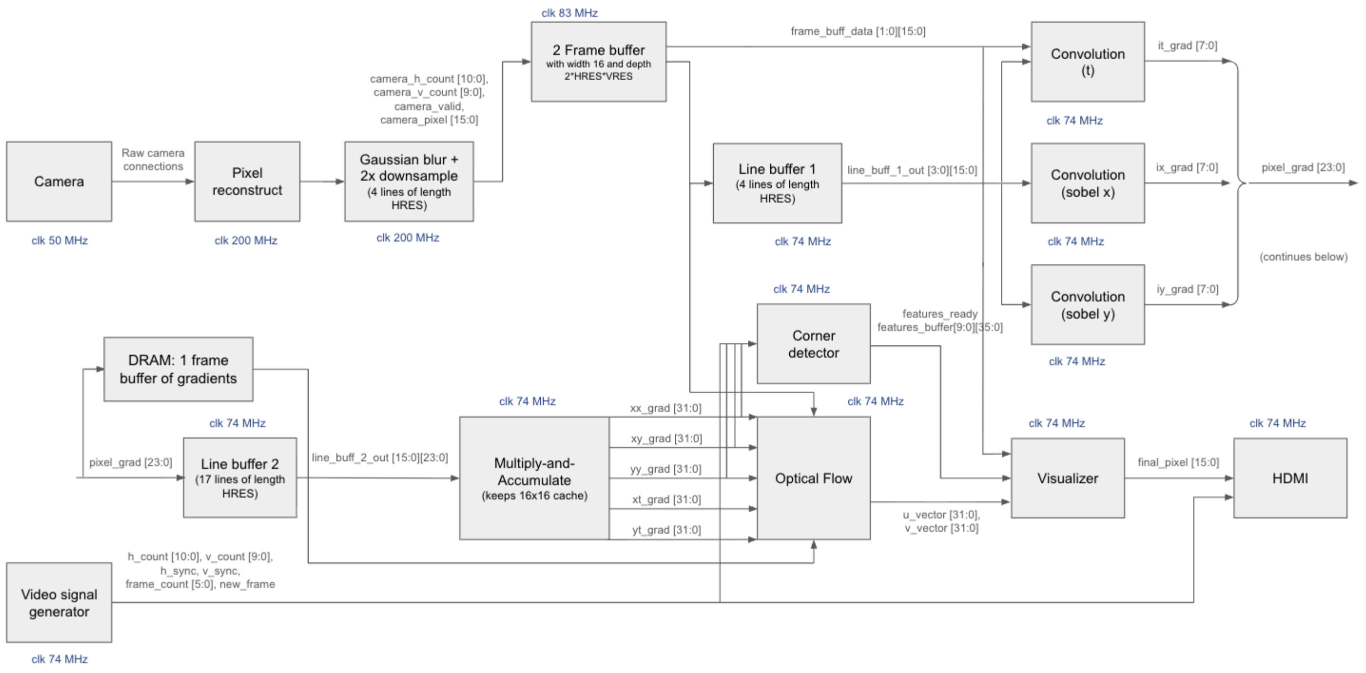
clk 83 MHz
2 Frame buffer
with width 16 and depth 2*HRES*VRES
frame_buff_data [1:0][15:0]
Convolution (t)
it_grad [7:0]
clk 74 MHz
camera_h_count [10:0], camera_v_count [9:0], camera_valid, camera_pixel [15:0]
Camera
Raw camera connections
Pixel reconstruct
Gaussian blur + 2x downsample (4 lines of length HRES)
Line buffer 1 (4 lines of length HRES)
line_buff_1_out [3:0][15:0]
Convolution (sobel x)
ix_grad [7:0]
pixel_grad [23:0]
clk 50 MHz
clk 200 MHz
clk 200 MHz
clk 74 MHz
clk 74 MHz
(continues below)
clk 74 MHz
Convolution (sobel y)
iy_grad [7:0]
features_ready
features_buffer[9:0][35:0]
Corner detector
clk 74 MHz
DRAM: 1 frame buffer of gradients
clk 74 MHz
clk 74 MHz
clk 74 MHz
clk 74 MHz
clk 74 MHz
pixel_grad [23:0]
Line buffer 2 (17 lines of length HRES)
line_buff_2_out [15:0][23:0]
Multiply-and-Accumulate (keeps 16x16 cache)
xx_grad [31:0]
xy_grad [31:0]
yy_grad [31:0]
xt_grad [31:0]
yt_grad [31:0]
Optical Flow
Visualizer
final_pixel [15:0]
HDMI
u_vector [31:0], v_vector [31:0]
h_count [10:0], v_count [9:0], h_sync, v_sync, frame_count [5:0], new_frame
Video signal generator
clk 74 MHz

Fig. 2. Top level block diagram of Optical FlowPGA.

adder tree and allowing us to comfortably meet timing without using any additional pipeline stages.

B. Temporal Gradients via Subtraction

In order to detect motion in specific feature points, we compute I_t the time gradient as current pixel - previous pixel of two consecutive frames. The current pixel is streamed in from the camera data path and the previous pixel is pulled from DRAM. The same current pixels were previously used in the convolution module. These live camera and DRAM pixels need to be synchronized at the same h_count and v_count. In detail, the two distinct paths to computing temporal gradient are:

1) Camera $\rightarrow$ pixel reconstruct $\rightarrow$ RGB to luminance $\rightarrow$ downsample $\rightarrow$ CDC $\rightarrow$ 3-line buffer
2) Request $\rightarrow$ traffic generator $\rightarrow$ DDR3 DRAM fifo $\rightarrow$ CDC $\rightarrow$ unstacker

Some key signals are: When does DRAM start reading? cam_frame_start (camera frame begins). When does prefill complete? After dram_prev_valid is high for 10 cycles. When does MAC start consuming? dram_read_enable = lb_valid and dram_prefill_done. The prefill means that DRAM starts read requests at the camera frame start, and when the FIFO is full for 10 cycles, this means the DRAM data is ready to read. By this time, the line buffer holding camera data has been filled as well, and upon the valid of both data from DRAM and the line buffer, they are ready to be used for a temporal gradient. This is just a simple subtraction of two pixels, but the challenge was aligning them.

C. MAC

The multiply accumulate module holds a 16x16 cache, which is made up of data from 16 line buffers. One thing to note is this is not exactly a ”cache” in that the data is not reused between computations, since this is a windowed multiply where every pixel in the window gets the same computed value. The values stored in the cache are the concatenated I_x, I_y, I_t streamed in from the previous convolution block. The purpose of this module is to compute $\sum_i I_x^2$, $\sum_i I_y^2$, $\sum_i I_x I_y$, $\sum_i I_x I_t$, and $\sum_i I_y I_t$, which are values summed over the 16x16 window. These outputs are valid at the bottom left corner of a window, since the entire window has the same sum. These outputs are passed to both the corner detector module and the optical flow module.

D. Corner Detection

The corner detection module implements a streaming top-k with an absolute distance check in the x and y directions, to give multiple corners that are not too close together. This module runs on initialization and reset, since we only need to find corners once at the beginning. It computes the Harris corner response

$$R = \sum I_x^2 \sum I_y^2 - \left(\sum I_x I_y\right)^2 - k\left(\sum I_x^2 + \sum I_y^2\right)^2 \quad (1)$$

with streamed sums from the MAC module and keeps the k windows with the largest response. The output of this module is a length k buffer with `h_count` and `v_count` coordinates of the top left of the corner windows. This module is meant as an approximate means to find corners, but it does not necessarily have to find the exact k corners with the largest response. Sometimes, because of the motion threshold, it may not find up to k moving corners, in which case it will output 0, 0 for the coordinates of this corner. A potentially more accurate version of corner detection was the Shi-Tomasi corner detector, but because the computed corner response required a square-root, we decided on this corner response calculation isntead. In our implementation, we used $k = 3$ but this module should generalize.

E. Optical Flow

The optical flow module implements the Lucas-Kanade algorithm to compute motion vectors (u, v) representing pixel displacement between consecutive frames. The Lucas-Kanade method assumes brightness constancy and small motion between frames, leading to the optical flow constraint equation $I_x u + I_y v + I_t = 0$ [4]. Since this single equation has two unknowns, we aggregate the constraint over a local neighborhood (our 16×16 window) and solve the system in a least-squares sense.

Summing over all pixels in the window, we obtain the system:

$$\begin{bmatrix} \sum_i I_x^2 & \sum_i I_x I_y \\ \sum_i I_x I_y & \sum_i I_y^2 \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} = - \begin{bmatrix} \sum_i I_x I_t \\ \sum_i I_y I_t \end{bmatrix} \tag{2}$$

The five summations on the left and right sides are provided directly from the multiply-and-accumulate module described in Section II-C. To solve for the motion vector, we invert the 2×2 matrix on the left hand side. The solution is given by:

$$u = \frac{(\sum_i I_y^2)(\sum_i I_x I_t) - (\sum_i I_x I_y)(\sum_i I_y I_t)}{\sum_i I_x^2 \cdot \sum_i I_y^2 - (\sum_i I_x I_y)^2} \tag{3}$$

$$v = \frac{(\sum_i I_x^2)(\sum_i I_y I_t) - (\sum_i I_x I_y)(\sum_i I_x I_t)}{\sum_i I_x^2 \cdot \sum_i I_y^2 - (\sum_i I_x I_y)^2} \tag{4}$$

The optical flow module computes the numerators and denominators through multiplication and accumulation operations on the input terms, performing all arithmetic using signed fixed-point representation to maintain precision throughout the computation pipeline. Division is implemented using a custom double-pipelined divider module designed to handle 48-bit signed numbers, which was necessary to maintain sufficient fixed-point accuracy for the motion vector calculations. The complete module is pipelined to 28 stages, with 24 of these stages dedicated to the custom divider itself, enabling high-throughput operations.

Although this optical flow module was not integrated into the final system due to time constraints, it was thoroughly tested and validated against a Python reference implementation of the Lucas-Kanade algorithm. The hardware implementation matched the Python results to high precision across various test sequences, as demonstrated in 3. The module remains fully functional and can be readily integrated into the existing system architecture, as all required input values ($\sum I_x^2$, $\sum I_y^2$, $\sum I_x I_y$, $\sum I_x I_t$, $\sum I_y I_t$) are already computed and available from the MAC module.

III. Memory

The DRAM uses a triple-buffered rotation to hold downsampled 640x360 luminance frames so that one buffer is written by the camera stream while two independent read paths fetch the previous frame. One is aligned to HDMI vsync and the other is aligned to camera vsync, without risking overwrite. On each detected `tlast`, `write_frame_index` advances modulo 3 to select the next buffer base address, giving a guard buffer that prevents frame t-1 from being overwritten by t+1 if processing or display ever slips past a single vsync interval. The traffic generator round robins through 3 operations, 2 reads (frame t-1 for display and frame t-1 for computation) and 1 write (frame t). The reason we have two read paths is one goes to HDMI and uses HDMI's vsync, while the other is synced with the camera's vsync, in order to be aligned with the pixel out of the line buffer that the camera fed into. At any moment, in the traffic generator, the display and calculation path have pointers that point to and increment different read addresses. The design assumes the full downsampling occurs before DRAM so all stored frames are 640x360, and the available bandwidth plus FIFO depth are sufficient to service both read streams and the write stream within each frame period; otherwise the guard buffer absorbs timing slips. Proper operation depends on robust `tlast` handling to rotate indices correctly, per-frame pointer initialization for both read domains, and arbitration that prevents the HDMI and camera synced reads from overrunning or diverging within the selected frame.

IV. Visualization

The first visualization we implemented was simply detecting and plotting the computed moving corners continuously each frame, resulting in a visualization like the one in 1. When all objects in a scene are stationary, i.e. have a small time gradient below a motion threshold, no corners are plotted no matter the magnitude of their spatial gradients. When there is motion and a significant spatial gradient in luminance, corners are detected and plotted.

The second visualization was a trail buffer module plots the positions of each of the k feature points over frames, using a distinct color for each point and maintaining continuity of point colors across all frames, implemented as an FSM. It accomplishes this by keeping a "trail buffer" frame in a dual-port BRAM. The trail buffer is the size of a video frame and stores "pixels" with values 0 through k, where each a value 1 through k represents the ID of the feature point that was last located at that pixel and 0 represents no feature point having ever been located at that pixel.

At initialization and after the corner detector has determined the features to track, the module will assign a unique ID from 1 through k to each feature point and write their ID into the trail buffer at their location. For each subsequent frame, it takes as input the current and predicted locations of a particular feature point returned by the optical flow module. Port A of the BRAM is responsible for updating the trails by performing a sequential read and then write to the trail buffer to retrieve the ID of the point at the current location and write the same

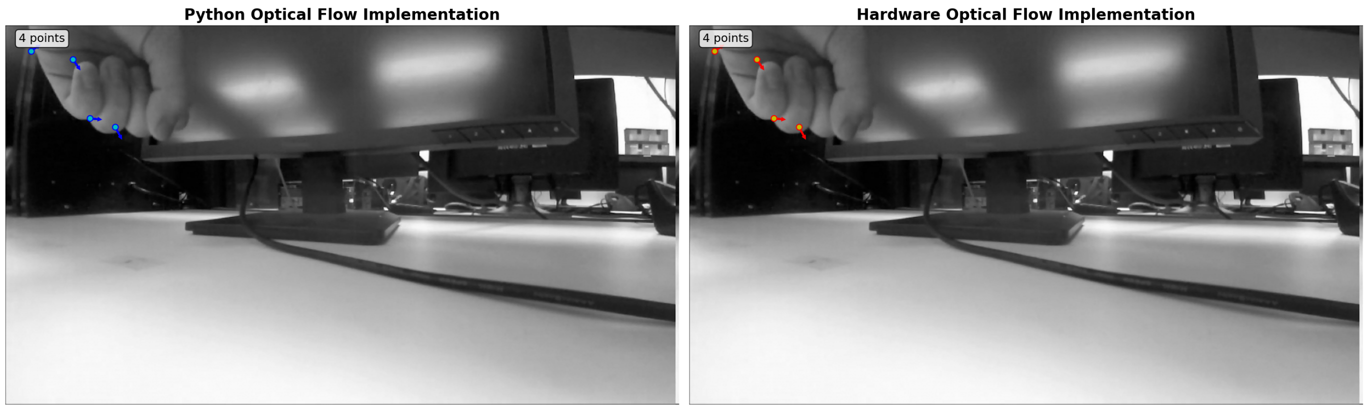
Python Optical Flow Implementation
4 points
Hardware Optical Flow Implementation
4 points

Fig. 3. We validated the hardware optical flow implementation in simulation against a Python reference and found that it matched to high accuracy in the features it discovers and the motion vectors it computes.

ID to the predicted location. This functionality is implemented with a three-stage pipeline, since a read from BRAM take two stages while a write take one.

Port B is used for HDMI readout from the trail buffer at a given address. The resulting trail pixel from this read (some value from 0 to k) is passed to a video mux along with the current camera pixel. If the trail pixel is 0, the video mux returns the camera pixel. Otherwise, it returns a distinct color based on the trail pixel value. The muxed pixel is then fed to the HDMI pipeline for visualization, resulting in the desired effect of distinctly colored motion trails overlaid with camera footage.

An example of the visualization is shown in Fig. 4 using real-time detected moving corners but synthetic motion data (continuous motion down and to the right). We believe this visualization scheme lends itself to being easily compatible with future integration of Lucas-Kanade or other optical flow algorithms.

V. Top Level Integration

The top level ties the pipeline from camera input, to DRAM storage, to computation, and HDMI visualization. Data from the camera and data from DRAM come together into the compute path and drives `mac_wrapper`. Corner results are latched in the top level and are either 1. converted into initialization coordinates that either seed the trail buffer or 2. simply overlayed on HDMI in continous corner detection mode. Switch 0 toggles between option 1 and 2. For both, corner detection is initialized on a button press. For display, the DRAM read output is upsampled 2x2 locally, muxed with the trail overlay, and augmented with corner markers; then video_sig_gen provides HDMI timing and TMDS encoders/serializers drive the differential HDMI outputs.

VI. Final Result

Due to time constraints, we did not incorporate optical flow into the top level dataflow. Instead, we opted for continuous moving corner detection. This means that on every frame, new moving corners will be detected. The detected corners are shown on HDMI by overlaying green dots on the live video. Additionally, we have another version of the visualization which is a singular moving corner detection upon startup and trail buffers drawn starting from those corners. For this trail buffer, as described in the section above, we have fake generated next pixel locations, which would have been replaced by calculated predicted next locations had we incorporated optical flow. We found that tracking a light on a dark background worked the best because of the contrast. Additionally, tracking our hand moving also worked well.

In conclusion, we successfully implemented a real-time hardware-accelerated feature tracking system that identifies and visualizes salient corner features on moving objects with minimal latency. While the system does not currently predict corner motion vectors, it establishes a complete computational pipeline, from gradient computation through corner detection and visualization, that provides all intermediate values necessary for future optical flow integration. The modular architecture and validated Lucas-Kanade implementation demonstrate that extending the system to full motion vector prediction is achievable, requiring only integration of the existing optical flow module into the dataflow pipeline. The system successfully demonstrates real-time sparse feature tracking capabilities suitable for computer vision applications requiring low-latency object motion detection.

VII. Evaluation

A. Timing

In post route timing, we find a WNS of 0.098 ns, with the longest path in high_definition_frame_buffer. After pipelining most modules, we successfully met timing.

B. Resources

Looking at `post_place_util.rpt`, we see 17 DSP blocks used. Our project is not BRAM heavy, since we use DRAM for frame buffer storage. A couple line buffers from convolution and MAC reside in BRAM and a frame buffer for trail visualization is also in BRAM. As calculated in our presentation, this totals to 135 KB of BRAM.

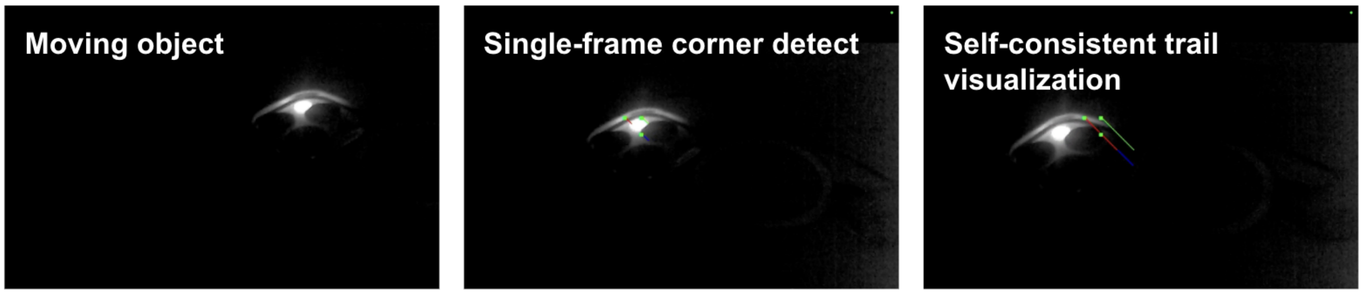
Moving object
Single-frame corner detect
Self-consistent trail visualization

Fig. 4. Single-frame corner detection on a button press that launches a motion trail visualization, currently implemented with synthetic data.

C. Checklist

We successfully accomplished our commitments, which were to have well-pipelined modules that pass local testbenches and working corner detection and trail drawing on moving objects. The build meets timing, and we consistently find moving corners. While we were able to implement optical flow and testbench it, we did not have time to integrate this module into our top level. Where it should have gone was after the MAC module and before the trail buffer module, where it should have fed in updated coordinates (instead of manually incremented coordinates for the sake of visualization). Thus, we have a solid chunk of our goals completed. Our stretch goal involved iterative refinement, which is just a figment of our dreams now.

VIII. Implementation Insights

We found debug LEDs very helpful throughout the process. For example, some signals that helped us debug were: if DRAM was feeding valid data, or if MAC outputs were valid, if corners were latched in top level. Synchronizing modules in top level was nontrivial, even when we had working, well-testbenched individual modules. We also spent a considerable amount of time thinking and planning out system features that we did not end up having time to implement, so perhaps for future work, we will spread out our thinking/planning and start with the most basic implementations.

IX. Code Repository

Our GitHub repository is available at https://github.mit.edu/6205F25/fa25-6205-team66

X. Individual Contributions

Maggie implemented the DRAM memory subsystem, convolution, and MAC. Christy implemented the HDMI visualization subsystem, trail buffers, and optical flow. We collaborated on implementing corner detection, pipelining completed modules, and connecting modules in top level.

XI. Acknowledgment

We thank Professor Joe Steinmeyer, Kiran Vuksanaj, and the 6.205 teaching staff for their advice and assistance on completing this project.

References

[1] OpenCV Documentation, “Optical Flow.” [Online]. Available: https://docs.opencv.org/3.4/d4/dee/tutorial_optical_flow.html
[2] OpenCV Documentation, “Harris Corner Detection.” [Online]. Available: https://docs.opencv.org/4.x/dc/d0d/tutorial_py_features_harris.html
[3] viso.ai, “Optical Flow.” [Online]. Available: https://viso.ai/deep-learning/optical-flow/
[4] B. Chiang, “Optical flow,” Stanford University CS231A Course Notes, Stanford, CA, USA. [Online]. Available: https://web.stanford.edu/class/cs231a/course_notes/09-optical-flow.pdf