Glow Trails on FPGA

Kiran Vuksanaj
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge MA, United States
kiranv@mit.edu

Eugeniya Artemova
Department of Mathematics
Massachusetts Institute of Technology
Cambridge MA, United States
giniya@mit.edu

***Abstract*—We present a design to utilize an FPGA for live digital effects video processing, for use while performing or practicing with flow arts props. The video effect applied by the system creates trails behind bright lights, so the previous locations props have moved through, and the patterns they have created, become visible on camera recording, mimicking what is seen when watching glow and fire prop performances in person. We present a novel IIR-based algorithm to create this effect with a small memory footprint and high data throughput to process image data in real time. We implement this algorithm, control of a high-framerate camera sensor, and an HDMI output of the live results of the algorithm. We evaluate the success of our algorithm at displaying glow trails and doing so in real time.**

***Index Terms*—Digital Systems, Field-Programmable Gate Arrays, Digital Video Effects, Real-time Computation, Flow and Fire Arts**

I. Introduction

When watching live performances using glow and fire props, audience members are able to see an ephemeral effect of a "trail" behind a fast-moving, bright object. The art of performing with these props takes advantage of this effect to create the shapes and patterns in the air that define many styles of flow arts performance. However, since the exposure of a camera works differently than the human eye, only the momentary position of the prop's bright head is visible when captured on video; video footage fails to capture the glow trail behind it. It's common to use post-processing video effects tools like After Effects to simulate this effect, but the process is slow when running on standard computers. The unique ability of Field-Programmable Gate Arrays to process data with high throughput presents the possibility of applying digital video effects to a video input with no delay perceivable by a human.

Utilizing an FPGA for high-throughput video data processing and a new algorithm for generating the glow trails video effect, we propose Glow Trails on FPGA, a system designed to enable a flow arts performer to see the desired video effects on camera in real-time. We prioritize the experience of the user by aiming for high quality video, with a framerate that can properly capture fast-moving objects. We aimed to have as high quality of an image as possible, and low enough latency that a performer wouldn't practically notice the delay in video output while performing in front of the camera. In order to enable these design goals while working within the constraints of the components available to us, we aim to minimize extraneous memory usage and maximize the throughput of the algorithm, and utilize the necessary memory components to store necessary data.

In this paper, we implement the Glow Trails on FPGA system. Section II outlines the specific technical challenges that come with this implementation, and tasks we set out to complete in order to achieve our implementation. Section III outlines the components used and the modules they are part of that are necessary to generate the output we desire. Sections IV through VI outline the design goals, choices, and implementations of each of our major modules, including the novel algorithm based on an Infinite Impulse Response (IIR) matrix to store past frame data as compactly as possible. Finally, in sections VII and VIII we evaluate the success of these goals, explore the use cases of our system, and reflect on potential future work to expand the project.

II. Goals

In order to create the implementation of our video interaction interface, our two most significant hurdles we needed to design for were limiting memory usage as much as possible, and ensuring an algorithm design where all components can work with high throughput. In order to process camera data that comes in at a constant, rapid rate, the algorithms must be able to work at a fast clock cycle and be able to always accept new data. In order to store data for a full frame, algorithms must be designed to require as few bits of data per pixel as possible–no excess copies of pixel data can be stored if a full frame of 240p, 16-bit color data is being stored; camera input must immediately be modified into output data, rather than storing the camera input for later comparison. In order to accomplish our intended beahvior and design for these constraints, we aimed to complete the following tasks:

- Design an IIR-based algorithm module, which requires only an IIR history value and new camera value to determine the effect-enhanced output pixel value with the desired effect.
- Implement an interface with which the necessary reads and writes from BRAM memory for the IIR algorithm and video output can be performed.
- Implement modules to read from this BRAM memory as needed for an HDMI signal to display the current state of the IIR matrix.
- Design microcontroller code manage the settings of the OV5640 camera sensor, which has the capacity for high

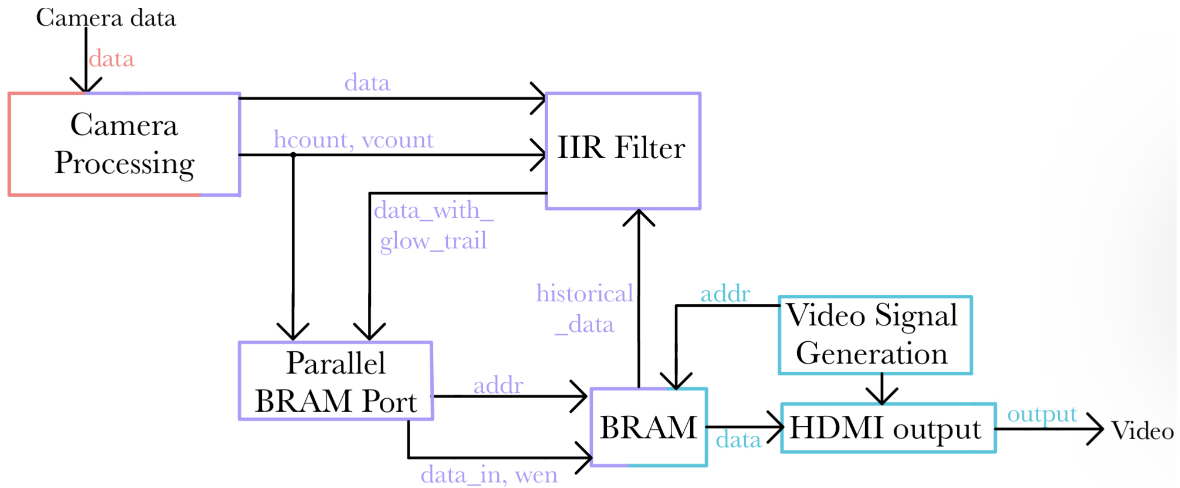

Fig. 1. Top Level Diagram with Red Camera Clock Domain, Puple IIR Clock Domain and Blue HDMI Clock Domain

framerate output and many options of settings for frame-size output.

- Design FPGA modules to read the data input from the OV5640 and cross clock domains to the domain utilized by the IIR algorithm, and determine the coordinate of pixel data in the frame as it is passed in.
- Implement the Memory Interface generator to access onboard DDR3 memory, and replace BRAM storage with this larger storage solution. Write modules to ensure necessary data is always drawn from DDR3 memory and available in BRAM when it is necessary for any algorithms. Switch to using 720p output from the camera, utilizing the additional space available in DDR3 memory.

Our final implementation accomplishes all of these tasks except for the final task of DDR3 memory incorporation. We successfully figured out how to get 720p or 240p data out of the camera, but without DDR3 memory we can't store a frame of 720p data. As a result, we successfully display glow trail output over HDMI, using the OV5640 camera, but we do so at 240p image quality. Since we're using the OV5640 camera, the sensor does a better job than the OV7670 camera does at capturing true-color images, and captures input at 120fps.

III. Physical Components

The project is made up of a number of physical components, consisting of:

- A Spartan 7 FPGA, contained within the Urbana Board. The image processing occurs here, exploiting the speed and parallel-processing capabilities of an FPGA. This board also has a BRAM with 2,700 block, each with a kilobit of space for a total of 2,700 kilobits of memory.
- An OV5640 camera. This camera has 720p resolution with shutter control, providing us with a high-quality image and proper shutter control to capture the entirety of the path of a fast-moving bright object.
- An Arduino XIAO SAMD21 microcontroller. This microcontroller is responsible for programming the OV5640 settings via I2C protocol, but handles none of the image data.
- An adapter PCB, made by Joe Steinmeyer. Originally made for the OV7670 camera, this PCB has outputs that allow us to power the camera and microcontroller, connect the microcontroller to the camera control pins, and connec the camera data pins to the FPGA.
- An HDMI monitor. Connected to the FPGA board, this displays the final video with the glow trails effect applied.

Through these components, 4 modules are implemented which handle different components of the system tasks and share results with one another; the camera handler, the glow trails algorithm, the memory interface, and the HDMI output. The camera handler module comprises of the camera, the SAMD21 microcontroller and the FPGA. The microcontroller drives the camera, setting specific parameters. The data from the camera comes back to the FPGA where it is processed, clock domains are crossed. The glow trails algorithm uses the current and previous camera data to create the glow trail effect. The memory interface handles interactions with the BRAM, making sure that the read requests from the IIR filter and the BRAM, as well as the write requests from the IIR filter do not occur at the same time. The HDMI output module uses values from the BRAM and a video signal generation module to generate an image that can be outputted to a display.

IV. Infinite Impulse Response (IIR) Trail Generation (Kiran)

To generate the glow trails video effect we use a novel algorithm utilizing an IIR filter. This filter adds the decayed brightness of historic pixel data to current pixel data, in order to create a decaying trail following a bright spot. This algorithm minimizes the necessary memory to generate such an effect; rather than having to store any number of camera input frames, only the output frame is stored, and the same frame buffer used for displaying output is used to calculate the next decayed frame.

Fig. 2. Results of the IIR filter glow trail algorithm. On the left, a frame from a video of a glow performance shows the present position of the poi head, but doesn't capture the trail a viewer would see. On the right, the IIR algorithm modifies the frame with the influence of the bright pixels of the most recent frames, revealing the trail behind the poi head. Generated in Python code written to verify the IIR algorithm.

A. Algorithm

The IIR filter takes in historical pixel $H = G_{t-1}$ and current camera pixel C and outputs an updated pixel G_t. Both of these are stored as 5-bit red, 6-bit green and 5-bit blue. The luminance of the historical pixel is calculated, using a fomula for conversion from RGB to YCrCb.

The algorithm has two adjustable parameters, the luminance threshold t and the decay factor d. If the historic pixel data is brighter than t and the camera pixel, the new pixel is equal to the historic pixel, with a brightness decayed by d. Otherwise, the pixel is replaced with the new camera data. We can adjust t based on the brightness of the room and d to change the length of the trails.

Let the luminance for a pixel X to be calculated by, $X_Y = 0.299X_R + 0.587X_G + 0.115X_B$. Then,

$$G_t(H,C) = \begin{cases} \{dH_R, dH_G, dH_B\} & \text{if } H_Y > t,\ H_Y > C_Y \\ C & \text{else.} \end{cases}$$

The decay calculation is done by multiplying an 8-bit color value by a 24-bit number to get a 32-bit number. Then, we take the most significant 8 bits. This lets us approximate multiplying by a float fairly well while having much shorter computation times.

This algorithm is applied to each pixel coordinate independently, producing a full image influenced by the brightest pieces of the most recent frames and the new camera input. Since it only requires the data of a single pixel position, a pipelined system can continuously process pixels without relying on unavailable data, allowing for high throughput. The algorithm minimizes memory footprint and maximizes throughput, enabling the high image quality and real-time processing features desired from our implementation.

B. Python Implementation

In order to demonstrate the feasibility of the IIR glow trails algorithm, we implemented the algorithm in Python before writing it for the FPGA. A sample of the results from this simulation are shown in Figure 2; the algorithm was shown to create the intended visual result intended. We found that in the python code a threshold value t of approximately 85% brightness and a decay value of 98% produced desirable results. When implementing the algorithm on FPGA, we discovered that these default values were not as good, and having an adjustable cutoff range was better given the variable lighting.

C. Board Implementation

We can see the IIR filter in Figure 1. The IIR filter interfaces with the camera output of an OV5640 camera to get new pixel data and the BRAM to get the historical pixel data and write the updated pixel data. It also interfaces with the parallel BRAM port module to handle the timing of the read and write requests to the BRAM.

V. Camera Interface (Kiran)

In order to improve the video quality that we can achieve for end user experience for the Glow Trails system, we chose to incorporate an OV5640 sensor. As opposed to the OV7670 sensor used in labs for 6.205, the OV5640 has capacity to take much higher quality images and have more complete control over sensor options like shutter speed. It has the ability to transmit video data at 720p (HD) video quality at 60fps, and transmit 240p (QVGA) video quality at 120fps. Experimentally, we have also found the realism of the camera output to be much higher quality from the OV5640 camera as opposed to the OV7670, regardless of which setting we use.

In order to power on and configure the camera, an sequence of I2C messages must be sent to the camera. These messages set configuration registers within the camera, controlling frame size output, white balance, shutter cycle, PLL clock manipulation for the data output, and many other components of how the sensor functions. We implement this sequence of register writes using a microcontroller, the Seeeduino XIAO SAMD21, which connects to the camera's SCL and SCK pins to communicate. The code on the microcontroller is written in C, implementing only basic I2C register write commands. It is based on the structure of Joe Steinmeyer's C code [1] to control the OV7670 camera in a similar fashion. There exist modules in CircuitPython [2] and C [3] to control the OV5640, but they require a microcontroller with larger flash storage and more pinouts to operate. However, their code is open source, and are the most successful source we found for determining what must be set on the camera for it to successfully run. The register datasheet for the OV5640 is helpful in places, but is far from complete and skips over defining key registers which are necessary for the camera to even turn on, let alone provide meaningful data. By examining what the CircuitPython source code would write over I2C, we determined what registers to set using the Seeeduino XIAO; we generated two sets of register settings, one to receive 240p120fps video output and one to receive 720p60fps video output.

The camera operates off of a 24MHz internal crystal clock, but uses a PLL to change the clock speed it outputs data at, depending on how much data it is transmitting; this can get to be as high as 96MHz. Valid data is transmitted with a clock wire and 10 parallel data wires; 2 wires are used for the horizontal and vertical sync, and 8 wires are used to send a bit in parallel. Data wires are held stable for the entire high portion of the data clock cycle, and may

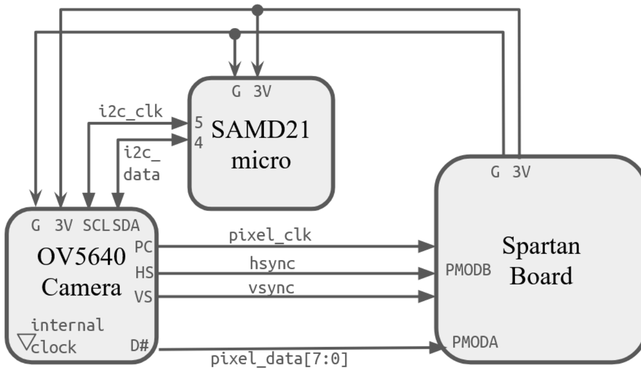

Fig. 3. Diagram of power and data connections to the SAMD21 XIAO Microcontroller and OV5640 camera. These connections are wired in the PCB adapter, built by Joe Steinmeyer.

change during the low portion of the clock cycle. In order to reliably poll the clock for rising edges, we must have a clock period shorter or equal in length to a half-cycle of the data clock, so we reliably see both a low edge and a high edge of the clock. Thus, we read our data on a 192MHz clock, as generated by the FPGA. This is, by extension, the clock cycle on which we operate the IIR algorithm. We first buffer the input to our IIR clock (the purple clock domain in Fig 1) for reliable reading of data. Two sequential bytes of pixel data make one pixel, transmitted in RGB565 format. Using the hsync and vsync signals, we determine when valid pixel data is entering, when a new row begins, and when a new frame begins. Thus we determine the full 16-bit pixel value, and its coordinate, for use as the new camera input in the IIR algorithm. The clock domain crossing is inspired by the 'camera.sv' and 'recover.sv' code from Lab 05, but is rewritten to work more reliably with different framerates and clock speeds, into our 'camera_bare.sv' and 'camera_coord.sv' modules. Through our work to handle camera input from the OV5640, we achieved our goal of better visual output for the Glow Trails system.

VI. Memory Handler (Eugeniya)

We used the BRAM units within the Spartan 7 FPGA to store the frame buffer for our IIR filter. The on-chip BRAM units are able to handle one full copy of a 240p frame, with 240*320 pixels, each holding 16 bits of color data. Block RAM has the capacity for exactly two ports, each with read/write capacity and each operating on different clock cycles. Through this, we are able to cross clock domains between the IIR clock and the HDMI output pixel clock. However, the IIR clock's port needs to be able to both read a pixel's data to provide the IIR algorithm with a history value, and write a pixel's data as updated by the IIR. Thus, we created a module with a state machine to manage both the read and the write.

This module, and its corresponding state machine, rely on the assumption that memory access is never needed by either the write job or the read job on two adjacent clock cycles. We can assume this because we chose a fast enough clock cycle that camera input data will always have one clock cycle reading as low in between valid data, so rising edges can be detected. Thus, if a valid read request and write request happen at the same time, there is a guaranteed cycle with no requests immediately following–so both requests can be fulfilled before any new requests enter.

We built a simple state machine module to enable all requests to successfully complete; in the Idle state, request addresses are passed through whenever a valid signal is read for a write or read request. If a valid signal happens at the same time, the read request is sent first; this guarantees a constant response time for read requests. The module then stores the valid data of the write request, and enters the Write Pending state. In this state, the write data is immediately sent, ignoring valid signals, which by are assumption will never come while in this state. Through this, all requests are sent to the BRAM over one port with no throughput decreases. This (very simple) state machine can be seen in Figure 4.

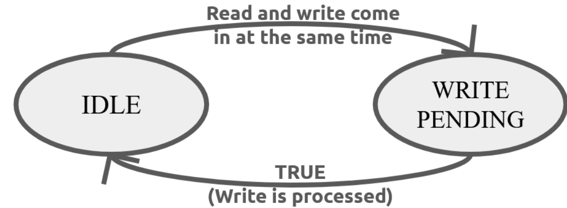

Fig. 4. Memory Handling Finite State Machine

Originally, we were meant to implement the storage of our frame buffer in DDR3 memory, as this would have allowed us to store the 720p video output from the camera, and achieve better image quality. We hoped to build FIFOs which hold the relevant data for each of our three read/write tasks, maintaining data availability in the FIFOs for high-throughput usage in every other module. We were not able to make this version of memory storage work before our deadline.

VII. Evaluation

Our primary method for evaluating our success were the latency of our image output, our total memory unit usage, and the visual quality of the output that we generate. In this section, we evaluate our degree of success on each of these fronts, and then explore potential further use cases of the Glow Trails system we built with limited further design.

The latency of the IIR algorithm is consistently exactly 9 clock cycles from pixel data entering the FPGA; two clock cycles sync the input to our clock domain and determine the coordinate of data received, one clock cycle to send a read request through our BRAM port handler, two clock cycles read the relevant history data from block RAM, two cycles are used to calculate the IIR algorithm output, and 2 clock cycles write the output data to memory. At 192MHz, this is a latency of 47ns. This is imperceptible to the human eye, and it is even imperceptible to the cycle on which HDMI output

is read, so there is practically no delay in the video output of our algorithm.

Operating on a 192MHz clock, but in specifically the BRAM interaction needing two clock cycles to complete successfully (our assumption of valid data only every other cycle), our IIR algorithm and writing to memory can accept new data on a 96MHz cadence and process in a constant number of clock cycles. Thus, the throughput of the algorithm is 96MHz. This exceeds the throughput of data coming in parallel from the OV5640 camera at either QVGA or HD quality, so it is successfully able to process all 120fps of QVGA data it works with. This framerate is higher than the 720p60fps output to HDMI, so our IIR algorithm writes data in excess of what is needed by the HDMI output.

By achieving this higher framerate and working with this data consistently for the IIR, combined with a high quality camera and a well-timed shutter cycle, the visual output of the glow trails look very satisfactorily continuous and produce an image result we're happy with.

By storing the 240p IIR frame buffer in Block RAM, we use (240*320*16)=1228.8KBit of BRAM memory on just our frame buffer, out of a total of 2700KBit available. Since we were unable to utilize DDR3 memory, we used much more Block RAM than we hoped and were unable to store a 720p frame buffer, but no excess memory was needed in order to operate the algorithm beyond the base requirements to display HDMI video from a frame buffer.

Our project handles the use case we cared about in our project specification. Specifically, given a person spinning a glow prop, it creates a trail showing where the prop previously was. Unfortunately, it does not use the 720p camera output which means that it is not as high an image quality as we were hoping. We have reached our minimum goal (glow trails with BRAM and the OV7670 camera), and progressed beyond it towards our ideal goal, since we integrated the OV5640 camera. However, we didn't reach our ideal goal, which was to use both the OV5640 camera and the DDR3 memory in order to be able to have 720p images. We also definitely didn't get to our stretch goal, which was to add an SD card where we could save recordings of the images captured by the camera.

We had some ideas for additional use cases for our project. As with the glow trails, these use cases would be nicer with the 720p camera output, but will work with the 240p camera output as well. Use cases include:

- Capturing the output with HDMI capture to be able to create a not real time recording of the glow trails. HDMI captures already exist, so this should be fairly easy to achieve with no modifications to the FPGA code.
- We could find the 'glow trail' of old CRT TVs. On old CRT TVs you can see imperfections in the image from the screen loading/reloading the pixels. Using the glow trail, you would be able to get rid of this, as dark pixels would be overwritten by the lingering trail.
- This could be used as an art piece, similar to the ones in the stata display! This would probably require changing the threshold at which it draws glow trails (since most people do not walk around with glow props). However, the threshold is currently variable, so setting it lower such that it creates a glow trail wherever there is moving human skin, or generally lighter colors, or pretty much everything it sees should be easy and would create a fun art display. (Especially since the background is stationary, so setting a lower threshold would not cause any glow trails to appear there.)

VIII. Reflection

There were a couple of things we weren't able to achieve, that we were interested in further pursuing:

- We want to be able to access the DDR3 to be able to use a 720p camera output for a higher quality image. For this, we want to use MIG, which we tried to do during the project, but weren't able to achieve.
- We want to use an SD card to be able to store a copy of the output so that you can record yourself and then play it back at a later time rather than as you spin.
- We want to use data on an SD card as an input video (such as a performance recording) to be able to add the trails to pre-recorded videos.
- The camera doesn't perform that well under low-light conditions, and doesn't capture the color of the prop that well. Using a better camera, or potentially changing the settings might make the image color nicer.

In hindsight there were a number of things we learnt throughout the process:

- Camera datasheets are hard to read. Using pre-written modules, at least to understand things like registers, is tremendously helpful. We thought that switching from the OV7670 camera to the OV5640 camera would not be too hard, but we ended up working on the camera throughout all six weeks of the project. The datasheets for the two cameras looked very different and both were quite hard to decipher. It turned out that looking at existing camera interfaces, like espcamera, was tremendously helpful. While the SAMD21 microcontroller we were using could not run the espcamera module, looking at the source code for it let us understand what different registers were doing and allowed us to program the SAMD21. If you can use a pre-existing solution use it. Datasheets are dense and not designed to be easy to find the relevant information, but more so to contain all the possible information somewhere.
- Make sure you budget enough time for all parts of the project, even if you get stuck on a specific part. One of the things we couldn't get working was MIG. Largely this was because it took us much longer than we expected to get the camera working. We did not leave enough time to work with MIG and be able to get the IP and all the helper function for it working. Budgeting our time a little better (especially since both of us had other final projects) might have increased the chance that we would have been able to get MIG working.

- Don't use code you don't understand, especially if you can write it yourself easily. We ran into a lot of issues with our RGB to YCrCb/YCrCb to RGB code. At the time we were storing all our data as YCrCb and were struggling to understand why we were getting a grayscale output. We believe this was because we had taken code from the internet which was causing problems. Writing code ourselves, or only taking code we can sanity check, would have saved us a lot of time, especially since this was fairly simple code. Instead we spent a while staring at a somewhat convoluted set of code which used some Vivado features we didn't understand and trying to debug it.
- Manta simulations are great for testing hardware-dependant code, in our case the camera. We didn't have the option to test a lot of our code off-hardware since we were trying to debug the camera and the output it was giving itself. Probing with the scope and testing with the manta logic analyzer proved to be invaluable in this case.
- Using testbenches is good. We probably didn't do this enough, and probably should have done more of this. There were times where we were testing things on the FPGA and trying to debug. This seemed especially odd when we were using the FPGA's seven segment display + LEDs to show debugging output which would have been easy to see on a testbench.
- Lighting conditions dramatically change what threshold one might want to use for determining which pixels are 'bright enough'. Specifically, we used the python code to find a good threshold, but found it didn't work as well on the FPGA, and we wanted a much lower threshold, due to the brighter conditions. Additionally, we found that lower thresholds weren't much of a problem because if the algorithm was trying to display a glow trail on a object that didn't move, you could not actually tell and everything still looked normal.

IX. Thanks

Many thanks to the people who helped us along the way with our debugging and providing reference code that was invaluable to our development and exploration in this project:

- Our instructor, Joe Steinmmeyer, for his base microcontroller code, lab code, help in understanding the OV5640 camera, very helpful lectures, and a very enjoyable class. Thanks :)
- Our TA, Joseph Feld, for his invaluable help when we were debugging unknowable problems and his help in focusing and guiding our project as we began our design process and as we got lost in the code.
- Fellow student Andrew Weinfeld, whose MIG base code helped us explore the usage of DDR3 memory, even though we didn't end up getting it working for ourselves in time.
- The developers of CircuitPython modules that took us into the realm of succeeding with getting meaningful data out of the OV5640 cameras.

And for their lovely help in creating our project video:

- Rory Knight, our videographer
- Jonathan Anziani, our guest performer with a really cool LED Staff (the Flowtoys Vision staff)
- Topaz and Q-Tip, cameo cats

References

[1] Steinmeyer, J (2023) OV7670 Camera Board [source code]. https://fpga.mit.edu/6205/F23/documentation/ov7670

[2] Adafruit (2023) Adafruit CircuitPython OV5640 [source code]. https://github.com/adafruit/Adafruit_CircuitPython_OV5640

[3] Espressif (2021) ESP32 Camera [source code]. https://github.com/espressif/esp32-camera/

[4] Omnivision (2011) Datasheet Product Specification: OV5640 [Online] https://cdn.sparkfun.com/datasheets/Sensors/LightImaging/OV5640_datasheet.pdf