Intelligent Dash Cam

A Design Project Report

Presented to the School of Electrical and Computer Engineering of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering, Electrical and Computer Engineering

Submitted by

David Pirogovsky

MEng Field Advisor: V. Hunter Adams

Degree Date: May 2022

Abstract

Master of Engineering Program

School of Electrical and Computer Engineering

Cornell University

Design Project Report

Project Title: Intelligent Dash Cam

Author: David Pirogovsky

Abstract: Road trips are a core part of American culture, but recording them efficiently can be very difficult. One approach is to use a dashboard-mounted camera (dash cam) to record footage, but recording full-length videos in real time requires excessive storage and post-processing to capture highlights. Garmin's solution to this problem is the "Travelapse" feature, essentially a moving timelapse with a fixed frame rate, but it is ineffective at creating smooth footage in varying conditions and fails to differentiate between interesting and boring scenery. This project aims to use a variable frame rate to create a more interesting, smoother timelapse — using feature detection and matching techniques — to better capture highlights. The concept has been proven using OpenCV implementations of the ORB feature detector (Oriented FAST and Rotated BRIEF) and brute force matcher. Each feature that is matched has an associated distance, and this distance is used to determine whether a match is "good". Although other methods were explored first, the approach of looking at the proportion of "good" matches to overall matches over a window of frames was determined to be the best way to scale the frame rate, successfully creating a smoother, more interesting timelapse.

Executive Summary

Road trips are a beautiful subject for a timelapse, but there is no readily available method for capturing them effectively. Garmin's dash cams offer one implementation, a "Travelapse" feature which uses a fixed frame rate to condense hours of footage into a short, but choppy, video. Regardless of whether you're stopped at a traffic light, driving through a Kansas prairie, rock crawling off-road, or winding through a scenic canyon, the frame rate remains constant, yielding a video with more boring, choppy sections than smooth, interesting ones. Storing raw footage for manual post-processing is time consuming and impractical due to storage constraints. This project demonstrates a method of producing shortened videos with a variable frame rate for smoother output over changing conditions

Given a set of sequential frames, an appropriate frame rate for recording can be determined based on many different real-world inputs: GPS location, accelerometer, and/or gyroscope inputs, to name a few, which can be used to determine velocity. Alternatively, computer vision can be used to interpret and decide how "interesting" a scene is. The second approach requires less finicky hardware while also providing more valuable data, which is conducive to the goal of creating a timelapse that is smooth and interesting! OpenCV provides a variety of feature extraction primitives, and ORB (Oriented FAST Rotated BRIEF) Feature Detection is paired with Brute Force matching in order to quantify the quality of matches.

Scaling the recording frame rate by the proportion of good matches to overall matches proved to be an effective technique for creating a smooth, interesting timelapse. The core logic currently runs at around 20 FPS on 1440p video with no GPU acceleration, so the initial goal of implementing a real-time dash cam on the Jetson Nano with a 1080p camera should not require more optimization than compiling OpenCV with CUDA enabled.

Problem Statement and Requirements

Based on a continuous input of image frames in real-time, a system needed to be created to determine how many frames to save to a video. This means that all processing must occur at a minimum of 30 frames per second (FPS). The frames must also be High Definition, at least 1080p, and the camera used should be at a reasonable price point, or less than about $50. The camera should also have a wide-angle lens to enable the capture of more scenery. The board used should have some video processing capabilities and be easy to interface with a camera.

The board selection was initially between the DE1-SoC FPGA, a Raspberry Pi 4, and the Jetson Nano. The FPGA was the frontrunner until realizing that the analog video input would only support a resolution of 640x480. This would make the project relatively useless. After looking for more cameras, one was found which fit all the parameters but was designed for use with the Jetson Nano. The Raspberry Pi was mainly preferred due to familiarity with it and strong community support, but after a little research, the Jetson Nano seemed a much better choice for any kind of video processing application, so the camera and board were chosen.

Potential Solutions

One potential solutions to the problem is Garmin's Travelapse feature, which uses a fixed frame rate to condense hours of driving into a short highlight reel. It is concise but jerky, and yields almost unusable results in many offroad environments, creating the desire to develop an alternative in the form of this project. Using sensors like GPS, accelerometers, and gyroscopes, position, velocity, and acceleration (lateral and rotational) could be used to make decisions on frame rate, which could yield slightly more interesting and smooth results, and would likely provide better capture in offroad environments, but would lack the context of the visuals of the real-world. For this to work effectively, the GPS location would have to be cross-

referenced with maps and corresponding images of locations to make better decisions. Instead, computer vision could be used to try to glean the same information about motion while having the benefit of seeing the scene being captured.

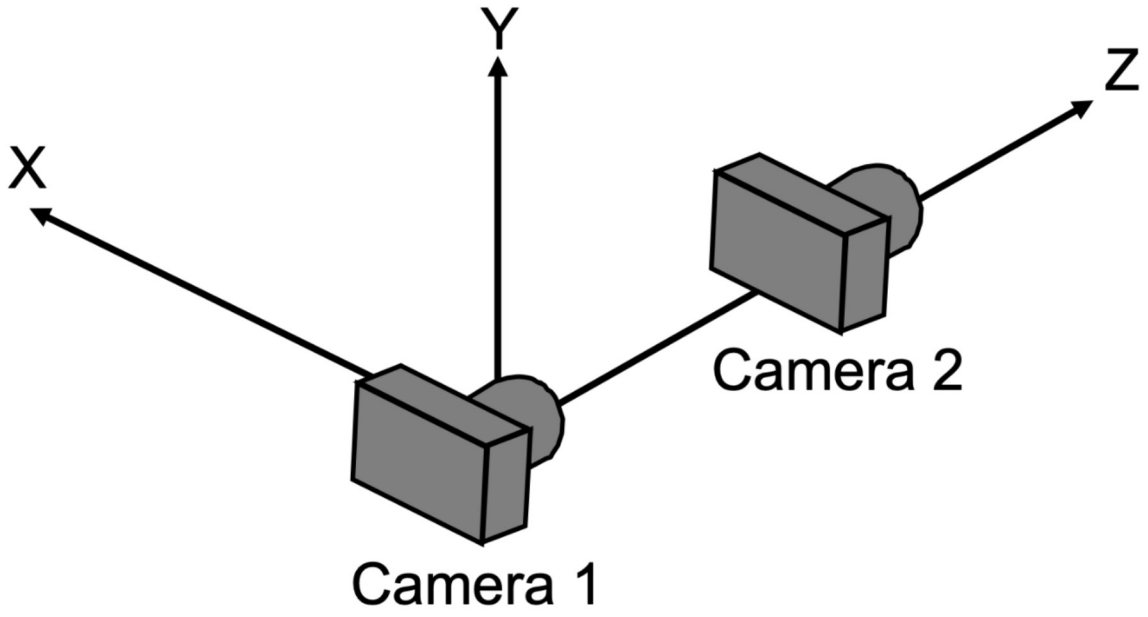

Figure 1: Camera Translation over Time [2.]

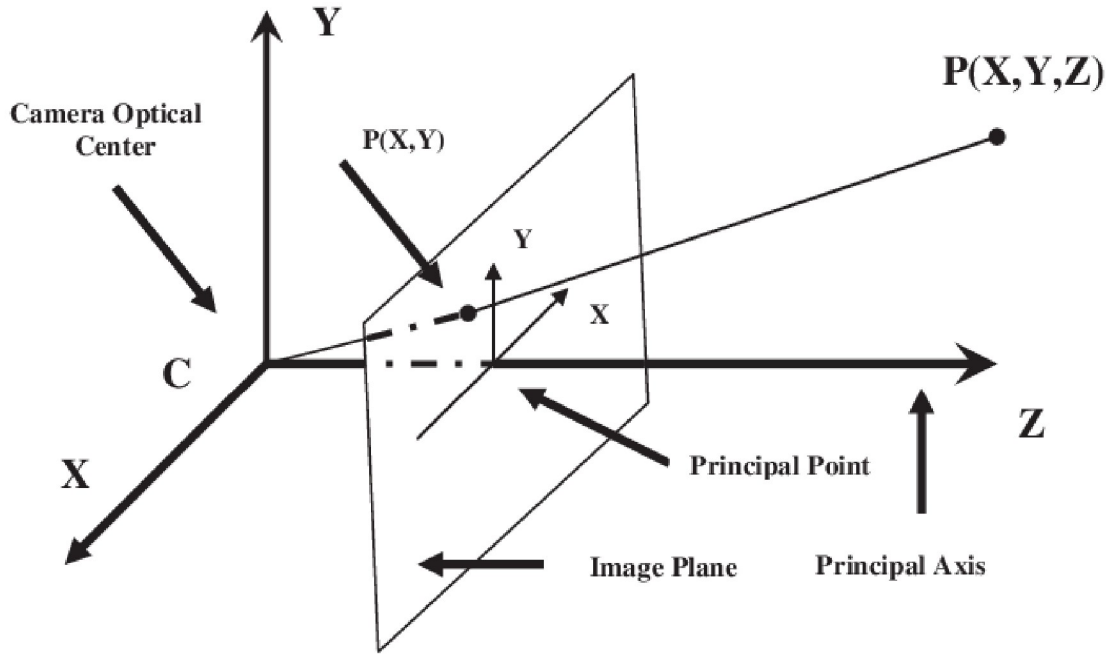

Figure 2: Camera Image Plane [5.]

Figure 1 is shown to help give an understanding of the problem parameters, as it shows the dash cam at time 1 and the expected position of the dash cam at time 2. In the most typical case, the dash cam will have experienced no rotation, and simply translated along the principal, or z, axis of the camera; into the camera's image plane. An image plane, a two-dimensional plane representing the three-dimensional world in the camera's FoV, will produce the image, and the procession of these planes can be imagined overlayed following this paradigm. The two image planes will then have their correspondences computed as described in the following paragraph.

The initial approach with computer vision was to try to determine the overall movement of features detected between two frames. Doing this by brute force yielded very noisy data which was hard to interpret, largely because objects move radially away from the center of the image, in all different directions, and without knowledge of the location or size of real-world objects, it is impossible to normalize these values to any useful measurement. Estimating the essential matrix from matched features would allow extraction of the rotation matrix and translation vector, with the translation vector giving a relative scale. While the rotation matrix would help determine if the vehicle is turning, since translation of a vehicle is almost entirely along one axis, the translation vector would provide almost no useful information. Using a structure from motion pipeline might yield more useful results, but they would still have to be appropriately interpreted. Even if this approach were capable of producing valuable data, it would still lead to a very large decision tree, where an overwhelming amount of edge cases would need decisions made.

With classic computer vision techniques yielding no obvious path forward, the problem was reconsidered. The number of good matches detected varied greatly depending on footage, and the length of time between matched frames. Features are some level of indication how interesting a scene is, and the more features move, the more their descriptors will change, leading to poorer matches, and indicating that more frames need to be captured. Thus the

accepted solution to the problem is to rely on the quality of matches between frames to scale the frame rate.

Implementation

My design prototype, summarized in the flowchart in Figure 3, relies on OpenCV4 in Python3, using only algorithms in the free portion of OpenCV's library. Video frames are streamed in using an OpenCV video capture object, which can either decode existing videos, such as those recorded on my Garmin 66W on previous road trips, or live input from an attached camera device. The input frame rate depends on the device or video used. The frame is then fed to OpenCV's ORB feature detector, which detects keypoints, then computes their descriptors. For the videos I was using, the bottom portion of the frame is always the hood of the truck, skewing the match statistics since these keypoints shouldn't move between frames, so the bottom of each frame is not passed into the ORB detector. Matches between the previous frame and current frame are then computed using OpenCV's Brute Force Matcher, and the distance of the matches is stored. This distance is then compared against a threshold, and the percentage of matches below this threshold compared to overall matches is stored. This percentage is written to a buffer of length 30 whose average is taken for frame rate decisions. This serves as a low-pass filter so that the output isn't overly sensitive to large changes between only two frames. The frame rate of the output video will always be 30 FPS, but the compression factor determined by the average match quality will determine how many frames are dropped before writing another frame to the output. For example, if the compression factor is 5, every fifth frame will be written to the output video. The compression factor is bounded between 5 and 40, and it is directly scaled between these values by multiplication with the average match quality. While the low-pass filter is being initialized with real data, the compression factor is set to 5. After the initialization period, the compression factor will be

recomputed using the following equation. This will be recomputed every time another frame is written to the output, which occurs every compression_factor frames.

$$\text{compression\_factor} = \text{int}(\text{match\_quality}*35+5)$$

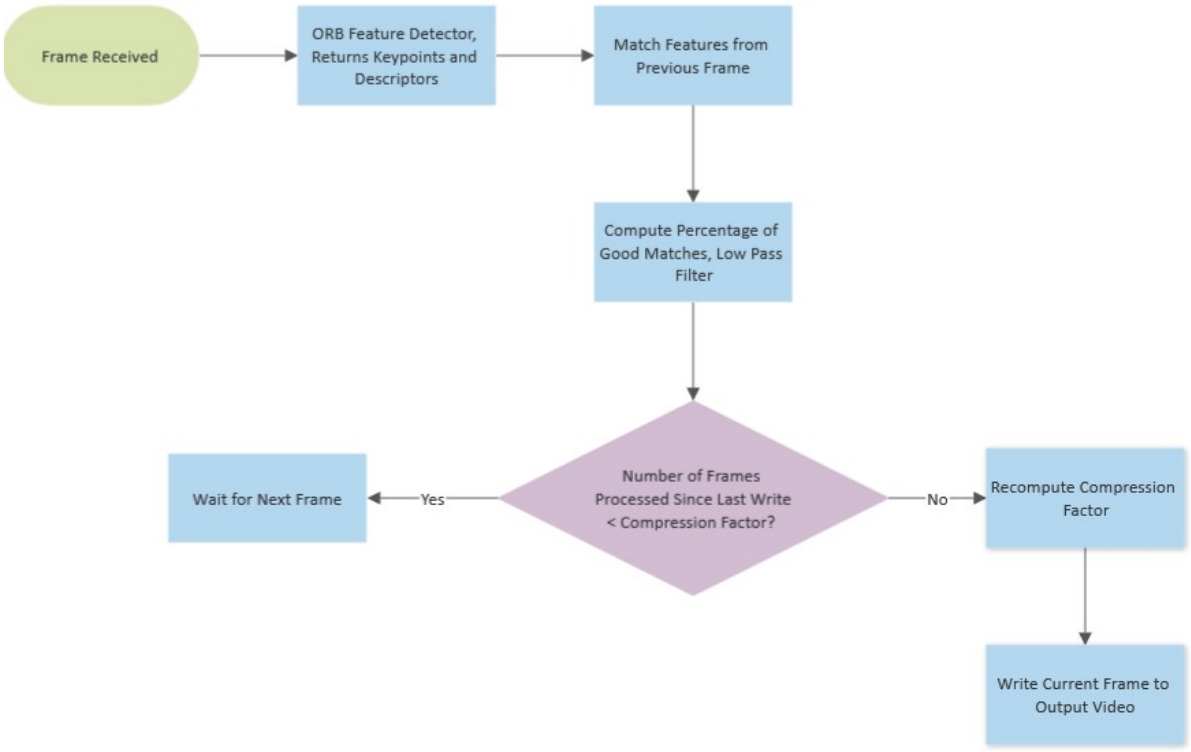

Figure 3: Flowchart of Core Logic

The feature detection portion of the ORB feature detector, oriented FAST (Features from Accelerated Segment Test), takes a circle of pixels around a specified pixel, and if a certain number of those pixels are brighter or dimmer than the center pixel by a threshold, the center pixel is marked as a feature – shown in the right image in Figure 4. This feature is then given an orientation depending on the center of mass of a patch, to allow for some rotation invariance (if the image is rotated, the same features should be detected). ORB also runs feature detection on a Gaussian pyramid of the images – shown in the left image in Figure 4 – giving some scale invariance, since different features will be detected as the image changes size. The Rotated BRIEF (Binary Robust Independent Elementary Feature) portion creates a binary descriptor, in

this case holding the outcomes of intensity tests between 256 pairs of points, producing an efficient and unique descriptor which is easy to compare. The descriptors are compared using Brute Force matching, and the distance measured between the descriptors is Hamming distance – the sum of the XOR'd bits of two feature descriptors. If the Hamming distance is less than 35, the match is considered good.

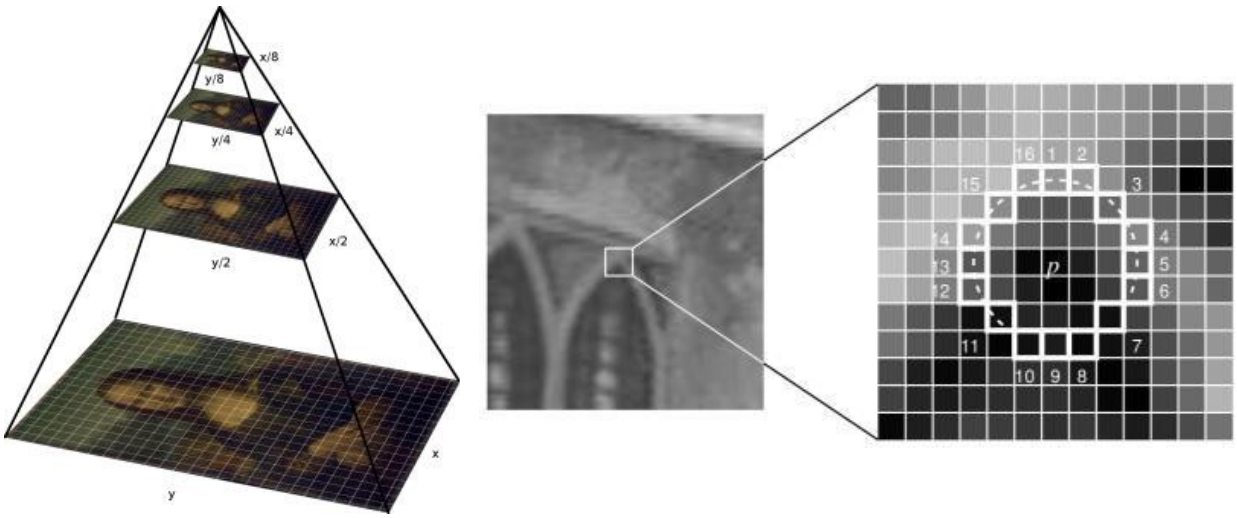

Figure 4: Image Pyramid and FAST

This method takes the characterization of an interesting scene to a high level, avoiding the pitfalls of creating different outcomes for many different cases. An example of a poor decision could be: the vehicle has stopped moving, so the compression factor should be infinite until we start to move again. A good counterexample to this can be seen around 1:15 in this off-road video captured in Tucson, where there are several cows in the middle of the road that cause me to stop for about a minute waiting for them to move. Instead of the video abruptly stopping when the vehicle comes to a stop and restarting once I am able to drive again, we see the cows moving around for about two seconds, which keeps the video interesting. Another scenario where this upper bound is useful is in the capture of a weather event, such as a sunrise, sunset, or dust storm, where the dash cam may remain running, but stationary, for a period of time.

For convenience in the user interface, a directory containing mp4 files can be specified, and every mp4 in the directory will be processed into a timelapse and stored as a separate file. FFMPEG will then be used to concatenate all the videos into one output video, and to compress it, since OpenCV's video writer is very simple and produces huge output files.

NVIDIA provides many tutorials for the Jetson Nano, which are helpful when everything works as intended. The tutorials provide very little explanation for failures or other modes of use, so it is easy to get stuck when the device doesn't work as expected. Several weeks were needed before the debug was complete and the system was plug-and-play, as it was supposed to be. The first step was downloading NVIDIA's provided image, formatting a microSD card, and flashing the provided image to the microSD card, which could then be used to boot the Jetson Nano. The Jetson Nano has a DisplayPort connector on it, and since I was working with the equipment available in Phillips 238, I found a cable and tried it with a monitor I found with a DisplayPort input. Confusingly, this worked only once — with significant visual artifacts, possibly due to aliasing — and allowed me to get through the basic Ubuntu install. After selecting a setting to delete the unneeded partitions, the screen went dark and I couldn't get any output to be driven by the Jetson Nano ever again. After several more time consuming formats and reflashes, I hypothesized that there may be something wrong with the DisplayPort driver, and decided to find an HDMI cable and monitor I could use. Once I connected to HDMI, all of those initial issues were solved, and I could finally begin using the Jetson Nano.

The first step was to take a course from the NVIDIA Deep Learning Institute, which helped familiarize me with some of the libraries and capabilities of the Jetson Nano. This, however, didn't work as intended either. The Nano supports SSH over USB, and all of the courses are run over Docker containers over SSH. I couldn't get a stable WiFi connection to the Nano with the installed USB WiFi adapter, but this ended up being an SSH problem from the side of my laptop, which took about half a day for me to realize. In order for SSH over USB to work, a jumper must be connected to pins J25 so that power will be provided through the barrel

jack connector instead of over USB. Most of the tutorial went over pipelines that can be used from the command line rather than actual code implementation, and proved not to be too useful. I started off by using OpenCV 4.1.1 in Python3, which allowed for more rapid prototyping, although the video processing itself was slow.

Results

Initially, the goal was to implement a standalone dash cam using the Jetson Nano and LI-IMX219 which would produce smooth timelapse videos. The first month and a half of the project was spent choosing components and bringing up the system, after which algorithmic development was started. The image produced by the chosen camera did not have as wide a field-of-view (FoV) as my Garmin 66W, and produced a somewhat noisier image, so it was unlikely that I would use it as a true replacement, more as a proof of concept. Due to time constraints, as the algorithmic development was wrapping up towards the end of the project, there was no time to implement the working algorithm on the Jetson Nano, and focus was instead shifted to creating output videos using the algorithm.

The produced video output exceeds my expectations in terms of smoothness and creation of interesting footage, but the video processing time does not meet my original expectations, due to a lack of the originally intended GPU acceleration. This YouTube link (https://www.youtube.com/playlist?list=PL8Dej7LQfMgPqRznYZVdm9o-YK462m3-J) shows the outputs produced via the method described here from various saved footage I had, while this link (https://www.youtube.com/playlist?list=PL8Dej7LQfMgP2n9tGeyLuysqb4MeyPIba) shows the footage produced by Garmin's Travelapse feature.

As shown in figure 5, the quality of matches changes dramatically between on-road and off-road driving, especially with the difference in the proximity of the surroundings. Each plot shows two images, one frame, five frames, ten frames, and thirty frames apart, with all of the

good matches between the images drawn on. The last plot also shows all of the features detected in the image, for comparison to see how few features are good matches.

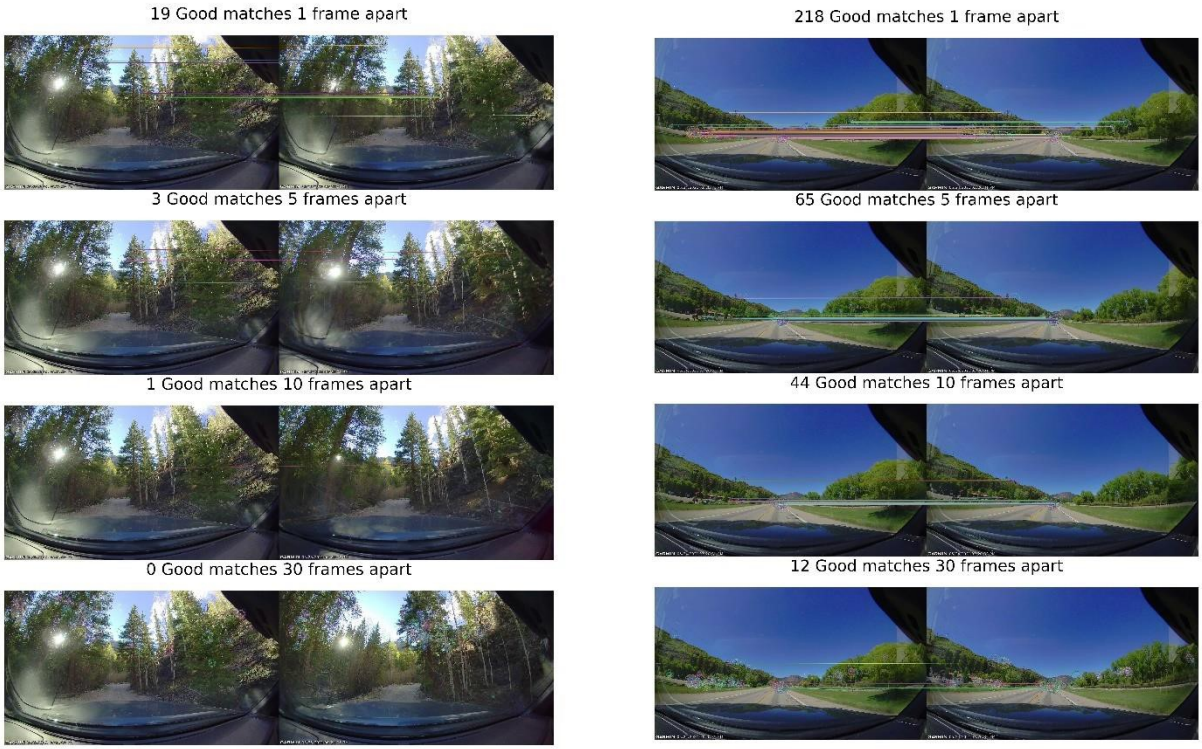

Figure 5: Off Road vs On Road Match Samples at Different Frame Separations

Code Appendix

https://github.com/dap263/jetson

References

1. *Getting started with Jetson Nano Developer Kit*. NVIDIA Developer. (2022, January 29). Retrieved May 18, 2022, from https://developer.nvidia.com/embedded/learn/get-started-jetson-nano-devkit

2. Hariharan, B. (2022, March). CS 5670 Homework 2. Ithaca, NY.

3. *OpenCV modules*. OpenCV. (n.d.). Retrieved May 18, 2022, from https://docs.opencv.org/4.5.5/

4. Tyagi, D. (2020, April 7). *Introduction to ORB (oriented FAST and rotated BRIEF)*. Medium. Retrieved May 18, 2022, from https://medium.com/data-breach/introduction-to-orb-oriented-fast-and-rotated-brief-4220e8ec40cf

5. *Vision-based calibration of a Hexa parallel robot*. Scientific Figure on ResearchGate. Retrieved May 18, 2022, from https://www.researchgate.net/figure/llustration-of-the-pinhole-camera-model-image-plane-in-front-of-the-lens-to-simplify_fig26_265969691