DEVELOPING A LOW-COST OPEN-SOURCE TRAFFIC COUNTER FOR RURAL AREAS (CTRA)

FINAL PROJECT REPORT

by

Dr. Nathan Belz
University of Alaska Fairbanks

Daniel Eagan
University of Alaska Fairbanks

for

Center for Safety Equity in Transportation (CSET)
USDOT Tier 1 University Transportation Center
University of Alaska Fairbanks
ELIF Suite 240, 1764 Tanana Drive
Fairbanks, AK 99775-5910

In cooperation with U.S. Department of Transportation,
Research and Innovative Technology Administration (RITA)

DISCLAIMER

The contents of this report reflect the views of the authors, who are responsible for the facts and the accuracy of the information presented herein. This document is disseminated under the sponsorship of the U.S. Department of Transportation's University Transportation Centers Program, in the interest of information exchange. The Center for Safety Equity in Transportation, the U.S. Government and matching sponsor assume no liability for the contents or use thereof.

ACKNOWLEDGMENTS

The authors would like to thank Dr. Jonathan Metzgar for his contributions to the coding work on this project. Additionally, the authors would like to thank Tristan Sayre for his initial work on the video processing code. Without the help of these two individuals, this project would not have been successful.

TECHNICAL REPORT DOCUMENTATION PAGE

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.

4. Title and Subtitle	5. Report Date
Developing a Low-Cost Open-Source Traffic Counter for Rural Areas (CTRA)	06/25/2024
	6. Performing Organization Code

7. Author(s) and Affiliations	8. Performing Organization Report No.
Dr. Nathan Belz Daniel Eagan	INE/CSET 24.08

9. Performing Organization Name and Address	10. Work Unit No. (TRAIS)
Center for Safety Equity in Transportation ELIF Building Room 240, 1760 Tanana Drive Fairbanks, AK 99775-5910	**11. Contract or Grant No.**

12. Sponsoring Organization Name and Address	13. Type of Report and Period Covered
United States Department of Transportation Research and Innovative Technology Administration 1200 New Jersey Avenue, SE Washington, DC 20590	Final Report
	14. Sponsoring Agency Code

15. Supplementary Notes

Report uploaded to:

16. Abstract

This research addresses operational disconnects and knowledge gaps related to traffic data collection in rural areas by developing a low-cost 3D-printed and open-source traffic counter (CTRA). Conventional pneumatic tube-based systems, which are still in use by transportation agencies across the United States because of their affordability, simply do not work on gravel roads and have difficulty counting non-motorized users and differentiating non-traditional vehicles from conventional motor vehicles. CTRA was developed and field tested at the University of Alaska Fairbanks and designed to provide a video-based data collection system that overcomes the limitations of other traffic counting devices. A count rate of 100% was achieved during the calibration process. Other than electronic hardware, most pieces of hardware can be printed on a 3D printer to form a simple and robust case and mounting system and only straps are needed to secure the counter to a fixed object. Because of its relatively simple and affordable design, CTRA could also be used for STEM and educational activities in schools and other related programs.

17. Key Words	18. Distribution Statement
Traffic counting, transportation planning, open-source	

19. Security Classification (of this report)	20. Security Classification (of this page)	21. No. of Pages	22. Price
Unclassified.	Unclassified.	28	N/A

Form DOT F 1700.7 (8-72) **Reproduction of completed page authorized.**

SI* (MODERN METRIC) CONVERSION FACTORS

APPROXIMATE CONVERSIONS TO SI UNITS				
Symbol	**When You Know**	**Multiply By**	**To Find**	**Symbol**
		LENGTH		
in	inches	25.4	millimeters	mm
ft	feet	0.305	meters	m
yd	yards	0.914	meters	m
mi	miles	1.61	kilometers	km
		AREA		
in^2	square inches	645.2	square millimeters	mm^2
ft^2	square feet	0.093	square meters	m^2
yd^2	square yard	0.836	square meters	m^2
ac	acres	0.405	hectares	ha
mi^2	square miles	2.59	square kilometers	km^2
		VOLUME		
fl oz	fluid ounces	29.57	milliliters	mL
gal	gallons	3.785	liters	L
ft^3	cubic feet	0.028	cubic meters	m^3
yd^3	cubic yards	0.765	cubic meters	m^3
	NOTE: volumes greater than 1000 L shall be shown in m^3			
		MASS		
oz	ounces	28.35	grams	g
lb	pounds	0.454	kilograms	kg
T	short tons (2000 lb)	0.907	megagrams (or "metric ton")	Mg (or "t")
		TEMPERATURE (exact degrees)		
°F	Fahrenheit	5 (F-32)/9 or (F-32)/1.8	Celsius	°C
		ILLUMINATION		
fc	foot-candles	10.76	lux	lx
fl	foot-Lamberts	3.426	$candela/m^2$	cd/m^2
		FORCE and PRESSURE or STRESS		
lbf	poundforce	4.45	newtons	N
lbf/in^2	poundforce per square inch	6.89	kilopascals	kPa

APPROXIMATE CONVERSIONS FROM SI UNITS				
Symbol	**When You Know**	**Multiply By**	**To Find**	**Symbol**
		LENGTH		
mm	millimeters	0.039	inches	in
m	meters	3.28	feet	ft
m	meters	1.09	yards	yd
km	kilometers	0.621	miles	mi
		AREA		
mm^2	square millimeters	0.0016	square inches	in^2
m^2	square meters	10.764	square feet	ft^2
m^2	square meters	1.195	square yards	yd^2
ha	hectares	2.47	acres	ac
km^2	square kilometers	0.386	square miles	mi^2
		VOLUME		
mL	milliliters	0.034	fluid ounces	fl oz
L	liters	0.264	gallons	gal
m^3	cubic meters	35.314	cubic feet	ft^3
m^3	cubic meters	1.307	cubic yards	yd^3
		MASS		
g	grams	0.035	ounces	oz
kg	kilograms	2.202	pounds	lb
Mg (or "t")	megagrams (or "metric ton")	1.103	short tons (2000 lb)	T
		TEMPERATURE (exact degrees)		
°C	Celsius	1.8C+32	Fahrenheit	°F
		ILLUMINATION		
lx	lux	0.0929	foot-candles	fc
cd/m^2	$candela/m^2$	0.2919	foot-Lamberts	fl
		FORCE and PRESSURE or STRESS		
N	newtons	0.225	poundforce	lbf
kPa	kilopascals	0.145	poundforce per square inch	lbf/in^2

*SI is the symbol for the International System of Units. Appropriate rounding should be made to comply with Section 4 of ASTM E380.
(Revised March 2003)

TABLE OF CONTENTS

LIST OF FIGURES

EXECUTIVE SUMMARY

The research presented herein addresses operational disconnect and knowledge gaps related to traffic data collection in rural areas by developing a low-cost 3D-printed and open-source traffic counter (CTRA). Conventional pneumatic tube-based systems, which are still in use by transportation agencies across the United States because of their affordability, simply do not work on gravel roads and have difficulty counting non-motorized users and differentiating non-traditional vehicles from conventional motor vehicles. CTRA was developed and field tested at the University of Alaska Fairbanks and designed to provide a video-based data collection system that overcomes the limitations of other traffic counting devices. A count rate of 100% was achieved during the calibration process. Other than electronic hardware, most pieces of hardware can be printed on a 3D printer to form a simple and robust case and mounting system and only straps are needed to secure the counter to a fixed object. Because of its relatively simple and affordable design, CTRA could also be used for STEM and educational activities in schools and other related programs.

CHAPTER 1. BACKGROUND

The research presented herein was an attempt to address an operational disconnect and knowledge gaps related to traffic data collection in rural areas. These disconnects and gaps have significant implications for transportation planning efforts which rely on accurate and timely data. The lack of a skilled or properly trained workforce, resources geared toward paved and modernized systems, and limited funding makes it difficult to sustain long term data collection or to maintain a comprehensive data management plan. Furthermore, the lack and poor quality of data for non-traditional and non-motorized modes for RITI communities creates difficulty in knowing true distributions across road classes and area types which hinders strategic planning and research. This situation is of particular concern considering approximately 25 percent of pedestrian and bicycle fatal and injury crashes occur on rural roadways nationwide (FHWA, 2007). In addition, fatal and injury-related crashes involving non-traditional vehicles (e.g., all-terrain and off highway vehicles) on rural public roads have been as high as double that of urban (Belz, 2019).

Without good, or in some cases any, data or data management systems, communities are not eligible to receive funding for safety improvements. Limited funding and lack of personnel requires that solutions for RITI communities be simple, affordable, and easy to maintain. To that end, this research project sought to develop a low-cost and low-energy consumption traffic counter that could be deployed in rural areas and powered off battery or solar power. Conventional pneumatic tube-based systems, which are still in use by transportation agencies across the United States because of their affordability, simply do not work on gravel roads and have difficulty counting non-motorized users and differentiating non-traditional vehicles from conventional motor vehicles based on local experience.

1.1. Previous Efforts

Previous research efforts by Belz and Sayre (2018) successfully developed a video-based field data collection methodology (see Figure 1 and Figure 2) for unconventional vehicle types like all-terrain vehicles and snowmachines. The equipment used for this, although relatively inexpensive compared to other traffic counting solutions, was relatively large and required a significant amount of post-processing.

Figure 1. Video-based field detection and counting system (a), video camera mounted on power pole (b), and post-processing graphical user interface (c) for the AKDOT&PF off-highway vehicle study.

(a) (b)

Figure 2. Post-processed video showing an off-highway vehicle on the shoulder (a) and an off-highway vehicle using the paved roadway (b) in Healy, AK.

1.2. Organization of Report

The Approach and Methods are presented in Chapter 2. This includes notes on the development of the case and the coding as well as the field-testing process. Chapter 3 provides the Usage Guide, including hardware setup, the 3D printing process, and case assembly. A few concluding remarks, recommendations, and notes on areas for improvement are presented in Chapter 4. Lastly, Appendix A and Appendix B provide the Python Programming code for the video processing.

CHAPTER 2. APPROACH AND METHODS

2.1. Code Development Criteria

The Raspberry Pi Traffic Counter Module, termed here a Traffic Counter for Rural Areas (CTRA) was designed to be a small, portable, DIY system for basic traffic counting services in rural areas and at test sites with unconventional or nonexistent infrastructure near the roadways. This module is based on the code developed by Belz and Sayre (2018) and utilizes a Raspberry Pi 4+ single-board computer to collect traffic video footage with an attached 8-megapixel camera, which is then processed by a Python-based program. The current software version requires user input through a GUI displayed on the Raspberry Pi computer. Due to the open-source, DIY nature of the project, a medium degree of user involvement is required to properly set up, adjust, and operate the traffic counting system. For a complete novice with access to the required materials, the estimated setup time is about 2 hours minus printing time. Additional information on the required materials, proper hardware and software setup, and field usage is in Chapter 3.

2.2. Case Development Criteria

The criteria for the case design are based primarily on the ability to 3D print as much of the shell and hardware as possible, with the exception of any straps or hardware needed for mounting it in the field. Other than electronic hardware, most pieces of hardware can be printed on a 3D printer to form a simple and robust case and mounting system (see Figure 3). More details on the case and 3D printing settings can be found in Chapter 3.

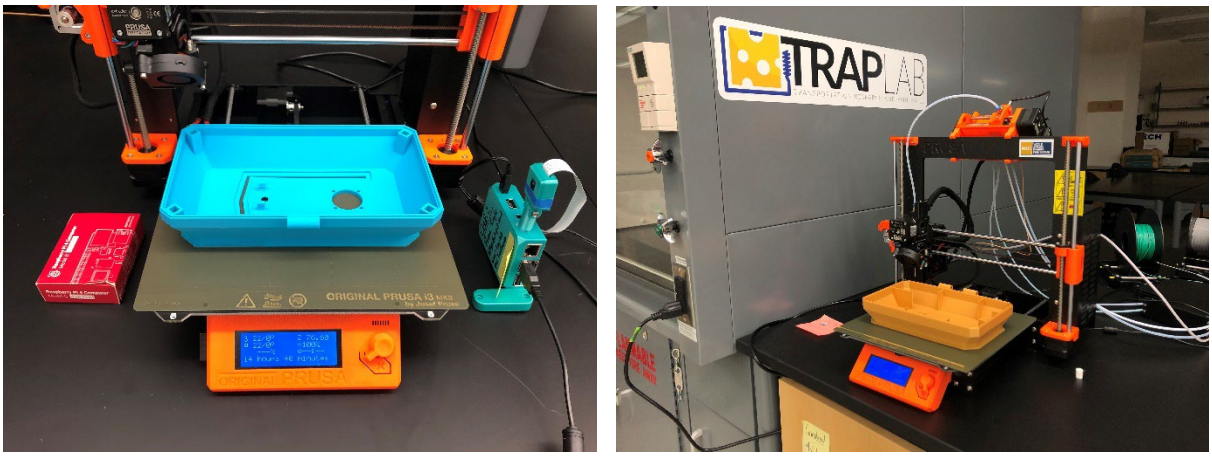


Figure 3. 3D printing process of the CTRA (a) shell front and (b) shell back.

2.3. Field Testing

Field testing occurred primarily on the University of Alaska Fairbanks campus. The CTRA module was attached to a light pole along Tanana Loop. Parameters were adjusted in the code in order to calibrate the settings that would provide accurate counting results from the test video until 100% counting rate with no “ghost vehicles” e.g., trees waving or clouds moving by in the background. On cold days, a small hand warmer was placed inside the case to help retain battery life.

Figure 4. Field testing the CTRA

Additionally, the following parameters and coding were changed to improve performance:

- In the config.json file
  - set “diff_flag” to false, disabling user input for the dividing line
  - increased “min_area” to 3000
  - set “limit” value to 75 and changed the description of its function (more on this below)
- In the traffic_counter.py file
  - modified the parameters passed to DirectionCounter in line 149 so that the dividing line was in the middle of the video frame, rather than based on the x_offset and y_offset values.
  - changed the logic in lines 207 and 213 to only consider objects whose centroid was within a specified distance of either side of the counting line. The limit value used was the same “limit” parameter from the config file. As it was written, the limit value only worked on one side of the dividing line, which led to ghost detections of background trees. e.g. “abs(x - W//2) > conf[“limit”]" instead of “ x < conf[“limit”]"
- Changed the cv2.line code in lines 243 and 248 to draw the dividing line in the middle of the screen — matching the actual counting line changed in the DirectionCounter parameters.

CHAPTER 3. USAGE GUIDE

3.1. Hardware Setup

As the current version of the traffic counter module requires some user input during the software installation/setup phase and for analysis of collected traffic footage, a Raspberry Pi workstation is required — if you use a desktop computer, you'll likely already have most of the required equipment to set this up, but it may be worth sourcing some duplicate equipment so that you have a dedicated Raspberry Pi work station, depending on the amount that you plan to use the module. For the hardware required to set up this traffic counter in the field, a list is provided below. All pieces of the plastic case were 3D printed using a Prusa i3 Mk3), but any 3D printer with a minimum print area of 230 x 150 x 50 mm (width x depth x height) should be able to print this version. Because the case was developed using a Prusa printer, this guide will be based around Prusa's performance and software.

Required Equipment and Supplies:

- Raspberry Pi workstation
- Computer monitor with HDMI plugin
- HDMI-to-mini-HDMI cord
- USB keyboard
- USB mouse
- 3D printer with minimum bed size of blank (we used Prusa i3 Mk3 to develop)
- (2) 1 kg spool of PLA filament
- Miscellaneous supplies
- Superglue

Required Materials (Traffic Counter Module):

- (1) Raspberry Pi 4+, 4GB
- (1) Raspberry Pi Camera
- (1) Raspberry Pi 5V fan
- (1) 1 ft USB-C to USB power cord
- (1) Belkin 20k Portable Power Bank
- (1) USB wall plug
- (1) 32 GB Micro-SD card
- (2) 1"x12" cinch straps

3.1.1. 3D-Printing Notes

This manual assumes user familiarity with the process of 3D printing, and to teach the whole process is beyond the scope of this brief guide. If you're unfamiliar with the process of transforming CAD files into 3D printed parts, Prusa has an excellent guide[1], and many others are available online. The basic process includes creating a 3D model of the part using a CAD program, exporting the model as an .stl file, "slice" that .stl file into layers that the 3D printer can print, and transferring this final, printable code to the printer. Most of these steps have already been done for you — with all pieces of the case provided for

[1] Original Prusa i3 MK3S+ User Guide: https://www.prusa3d.com/new-user-mk3/

you as .stl files in the “CAD Files” folder of the installation package, you’ll just need to use the PrusaSlicer (or other, if you prefer) software to slice the files into .gcode files that can be printed on your 3D printer. The names and quantity of the pieces to be printed are:

- (1) Case Bottom
- (1) Case Top
- (3) Clip
- (3) Clip Hinge Pin
- (1) Front Hood
- (1) Main Hinge Pin
- (1) Main Hinge Pin Nut
- (1) Mounting Insert

After much experimentation, we had the best success by modifying the following print settings, which can be accessed from the “Print Settings” tab of the PrusaSlicer program:

- Layer height: 0.2mm
- Vertical shells: 2
- Horizontal shells, solid layers: 4
- Fill density: 10%
- Fill Pattern: Honeycomb
- Brim: 8 mm for large, flat pieces (case top/bottom, mounting insert), 4 mm for all others
- Supports: Auto-generated or Off
- First layer speed: 25%

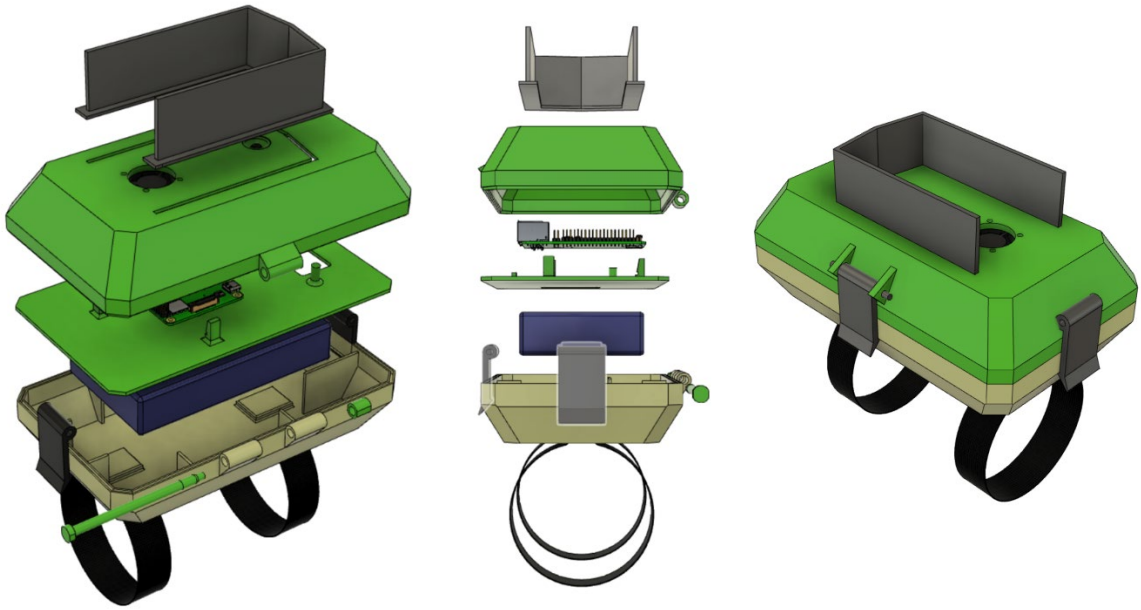
Feel free to experiment with the settings to achieve optimal results for your printing setup.

To print all case pieces, roughly 145 m of PLA filament is required. A 1 kg spool of PLA contains about 330 m of filament, so two full case sets can be printed from one spool. For different colors, extra material in case of failed prints, or modified parts, at least one additional spool will be required. The wide 8 mm brim required for the larger pieces requires trimming, but greatly enhances the print quality of those pieces, which tend to “warp” (pull up from the print bed) without it. The brim can be pulled off by hand or trimmed with a utility knife. For best printing results, clean the print bed with alcohol in between each print and avoid touching the print surface with bare hands.

3.1.2. Case Assembly

With all materials sourced, begin by assembling the printer case. First, slide the front hood into place from the inside of the case top, and superglue the hood flange against its mounting location and let dry. Depending on your intended usage, an extra bead of superglue or silicone can be applied to the outer, upper seam of the front hood to prevent water infiltration. Then, using Figure 3 as a reference, close the case halves together and insert the main hinge pin, securing it by snapping the main hinge pin nut onto the end. Next, use the smaller clip hinge pins to attach all three clips to the case top. Thread the bare end of both 1” x 12” cinch straps through the mounting strap holes on the back of the case bottom — this may require a paperclip or similar to help the end pop out the other side depending on your print quality. Take care to orient the strap so that the cinching clasp will be positioned correctly to tighten when you

wrap the straps around an upright pole. After setting the mounting insert into the case bottom and fastening the halves together shut with all three clips, your case is now assembled.



(a) (b) (c)

Figure 5. (a) exploded isometric view of case assembly; (b) exploded profile view of case assembly; (c) final assembly of traffic counter case assembly.

3.1.3. Software Setup

This traffic counter module utilizes a python-based code running on the Raspberry Pi to analyze the traffic footage. To properly set up the software, follow these steps:

- Use the Raspberry Pi documentation guide[2] to flash the latest Raspian software to the micro-SD card using the Raspberry Pi Imager application.
- Connect your Raspberry Pi to power, keyboard, mouse, and monitor.
- Connect your Raspberry Pi to the Internet using the WiFi interface or by ethernet cable.
- Navigate to https://github.com/uaf-cs/uaf-ctra on the Raspberry Pi and download the uaf-ctra package using the green "Code" dropdown button.
- To install the necessary software to run the traffic counting code, follow the README document in the code download or run the "inst.sh" script on your Raspberry Pi before following the rest of the README.
- At this point, the Raspberry Pi should be ready to run the traffic counting code.

[2] Getting started with your Raspberry Pi: https://www.raspberrypi.com/documentation/computers/getting-started.html

3.1.4. Performing a Field Test (Collecting Video Footage)

To perform a traffic counting test, several things must be done. First, traffic footage must be collected. (Add software part when code is streamlined, still relying on manual video capture now).

To capture video, the Raspberry Pi must be attached to the mounting insert of the case with the snap-fit clips, and the Raspberry Pi Camera should also fit into its snap-fit clips with the lens poking through the hole in the case. Connect the ribbon cable from the camera into its spot on the Raspberry Pi board (this can be a bit awkward as the case needs to be partially closed to have enough cable to connect). Place the portable battery in the bottom of the case, underneath the mounting insert, and connect the power cord from the battery to the Raspberry Pi. Any excess cord length should be wrapped or looped around the post on the mounting insert to provide for a tidier setup. Once the case halves are clipped together, use the straps to securely mount the case in its test location on a light pole, power pole, sign pole, etc. For best results, the camera should be positioned at least 6 ft away from the driving surface, mounted around 6 ft above grade, and pointed at a slight (~30˚) angle to the roadway.

3.1.5. Data Analysis and Software Usage

To count the traffic in your collected video footage, connect your Raspberry Pi to your workstation and open a new terminal window. Type “workon ctra” to enter the virtual environment. Navigate to the project directory by typing “cd uaf-ctra/traffic_counter”. Then, drag and drop your newly collected video (which should be on the Desktop) into the “Videos” folder. To watch and count your video, type into terminal “python3 traffic_counter.py -c config/config.json -m horizontal -i Videos/yourvideoname.h264”. The video will show a vertical “counting” line in the middle of the screen, and a counter showing left and right-traveling vehicles will pop up in the upper left of the screen after the first vehicle is detected crossing the line. The video will close when it is done playing.

3.2. Modifications

3.2.1. Case Design

The case system was designed using Autodesk’s free Fusion 360 software. The Fusion 360 project file has been provided in the “CAD” folder of the installation package so that you can modify the design if necessary or desired, but it will require familiarity with the software and won’t be covered in this guide. However, there are several parameters that can more easily be modified for novice users by opening the project file and pressing the shift and C buttons, which brings up the parameter table. The parameters have been labeled in the “comments” column. The values for the parameters can be adjusted as needed, and the model should auto-update. The parts would then need to be re-saved as .stl files to use in PrusaSlicer before printing. If you adjust CAD parameters for your needs (to accommodate battery size, etc), make sure to:

- Include hand-labeled renderings of parameter options
- Copy adjustment notes at bottom of work log

3.2.2. Software

Depending on the conditions of your test site, modifications may be required to gain accurate counting results. Configuration parameters can be edited in the “config.json” file in the traffic_counter folder, which allows you to adjust detection distance, size limits, etc. For more in-depth code adjustments, look

to the the comments in the python script, the PyImageSearch blog post on an OpenCV People Counter[3] (this code relied heavily on the code presented in the article), and the README document in the uaf-ctra folder.

3.2.3. Miscellaneous

If you find your recorded videos to be blurry to the point of impacting the counting accuracy of the code, follow our recommended guide[4] on adjusting the focus of the Raspberry Pi Camera.

[3] OpenCV People Counter: https://pyimagesearch.com/2018/08/13/opencv-people-counter/

[4] Fixing the blurry focus on some Raspberry Pi Camera v2 models: https://www.jeffgeerling.com/blog/2017/fixing-blurry-focus-on-some-raspberry-pi-camera-v2-models#:~:text=To%20adjust%20focus%2C%20pinch%20the,to%20confirm%20the%20correct%20focus.

CHAPTER 4. CONCLUSIONS AND RECOMMENDATIONS

The research efforts presented in this report demonstrate a low-cost and 3D printable traffic counting device with general applicability for transportation planning and system performance monitoring, especially applicable in rural areas. Considering that the video was both collected and processed using a Raspberry Pi, powered by a small battery, and mounted in a 3D-printed case, demonstrates an acceptable level of success for the project. Additional work could be completed to fine-tune and adjust the parameters to meet the needs of other traffic counting applications.

Additionally, the final iteration of the code could be modified to reduce power consumption by turning off Wi-Fi/Bluetooth, turning off HDMI output, turning off unused USB ports, etc., which all consume a considerable amount of power even when not being actively used.

The coding approach is after identifying moving objects within the video frame, that object is then detected to be moving left and is on the left side of the counting line is counted, and vice versa for the right side. To eliminate false positives, the code was modified to only detect objects within a certain "limit" or distance of the center line. This was because any tree movement that was detected on the left side of the screen and was moving left at any moment was counted, even though it had not crossed the center line of the screen and was just moving back and forth. There may be a way to only count vehicles as they cross the dividing line in the frame since we are not necessarily concerned with objects that are moving in the frame but only those that cross the counting line. This may provide a simplified approach to reduce processing demands on the Raspberry Pi unit. The code does currently employ a "power saving" script that turns Wi-Fi/Bluetooth and the USB chip off while recording video to reduce overall power consumption. This could also be incorporated into the video counting code while processing in real time and reduce the demands on the battery bank. The test videos were recorded at 6 FPS, which seemed to be sufficient for tracking purposes and ran the code much faster. It might be possible to reduce the frame rate of the video stream input and further save on power and memory consumption.

Other possible areas for improvement include:

- Non-GUI reliant with an external button to start and stop video recording.
- Easier way to test power consumption (e.g. track elapsed time and save continuously to file, then run until battery dies and read the file with the saved time values).
- Left and right counting numbers saved in file externally, not just displayed on screen.
- Fan operable mode only if CPU temp requires it (currently fan was not coded to run since it was only tested in cold temperatures).
- Option to manually mask out sections of the video while it's being processed.
- LED screen mounted on case showing current counts for those running tests to quickly view current data without connecting monitor.

REFERENCES

Belz, N.P. & Sayre, T. (October 2018). *Quantifying Unlawful Use of Off-Highway Vehicles on Public Facilities in Alaska*, 2018 Region 10 Transportation Conference, Pacific Northwest Transportation Consortium and Center for Safety Equity in Transportation, Fairbanks, AK.

Belz, N.P. (2019). *Safety Evaluation of Statewide Off-Highway Vehicle Use in Alaska - Crash Review, On- And Off-Road System Use, and Conflict Evaluation*. Alaska Department of Transportation and Public Facilities. DOT&PF Report Number HFHWY00082, USDOT Report Number 4000180.

Federal Highway Administration (FHWA) (January 2007). *Factors Contributing to Pedestrian and Bicycle Crashes on Rural Highways*, https://highways.dot.gov/research/publications/safety/hsis/Factors-Contributing-to-Pedestrian-and-Bicycle-Crashes. Retrieved July 15, 2022.

APPENDIX A VIDEO RECORDING CODE

```
# Python Script for Raspberry Pi:
# Will capture video for number of seconds specified in line 47.
# Will save video file to RPi desktop without overwriting previous video
files.

# To run program on startup, modify rc.local with sudo nano
#copy python filepath before exit 0
# full approach detailed here: https://learn.sparkfun.com/tutorials/how-to-
run-a-raspberry-pi-program-on-startup/all
# sudo nano /etc/rc.local

from picamera import PiCamera
import math
import time
from datetime import datetime
from dateutil import parser
from gpiozero import OutputDevice
import subprocess #to call power saving bash script
import os


# Start Power Saving
#subprocess.call("./startpowersave.sh")
subprocess.call("/home/pi/Desktop/startpowersave.sh")

# Initialize variables and objects
GPIO_PIN = 17 # 5V hot pin to power fan, ground pin is 6
recordtime = 2 #seconds

fan = OutputDevice(GPIO_PIN)
camera = PiCamera(framerate=7)

# Check current directory to avoid overwriting file, change file name if
needed

filename = "testvideo"
i = 0

while os.path.exists(filename+".h264"):
    i = i+1
    filename = filename + str(i)


#Start actions
#fan.on() #sends power to fan GPIO pins
camera.start_preview()
camera.start_recording("/home/pi/Desktop/" + filename + ".h264")


camera.wait_recording(500)
camera.stop_recording()
```

```
camera.stop_preview #can remove preview options once running on RPi on
startup without GUI
camera.close()
#fan.off()

# Stop Power Saving
subprocess.call("/home/pi/Desktop/stoppowersave.sh")
```

APPENDIX B VIDEO PROCESSING AND USAGE CODE

```
 # USAGE
# To read and write back out to video:
# python traffic_counter.py --conf config/config.json \
#     --mode vertical --input videos/highway-resized-short.avi \
#     --output output/highway-resized-short.avi
#
# To read from webcam and write back out to disk:
# python traffic_counter.py --conf config/config.json \
#     --mode vertical --output output/webcam_output.avi

# import the necessary packages
from pyimagesearch.directioncounter import DirectionCounter
from pyimagesearch.centroidtracker import CentroidTracker
from pyimagesearch.trackableobject import TrackableObject
from pyimagesearch.utils import Conf
from multiprocessing import Process
from multiprocessing import Queue
from multiprocessing import Value
from imutils.video import VideoStream
from imutils.video import FPS
import numpy as np
import argparse
import imutils
import time
import cv2

def set_points(event, x, y, flags, param):
      # declare a global variable to store difference point
      global diffPt

      # check if a left button down event has occurred
      if event == cv2.EVENT_LBUTTONDOWN:
            # if the direction is set as vertical, set the difference
            # point as the x-coordinates, otherwise set it as the
            # y-coordinate
            diffPt = x if param[0] == "vertical" else y

def write_video(output, writeVideo, frameQueue, W, H):
      # initialize the FourCC and video writer object
      fourcc = cv2.VideoWriter_fourcc(*"MJPG")
      writer = cv2.VideoWriter(output, fourcc, 30,
            (W, H), True)

      # loop while the write flag is set or the output frame queue is
      # not empty
      while writeVideo.value or not frameQueue.empty():
            # check if the output frame queue is not empty
            if not frameQueue.empty():
                  # get the frame from the queue and write the frame
                  frame = frameQueue.get()
                  writer.write(frame)

      # release the video writer object
      writer.release()
```

```
# construct the argument parser and parse the arguments
ap = argparse.ArgumentParser()
ap.add_argument("-c", "--conf", required=True,
      help="Path to the input configuration file")
ap.add_argument("-m", "--mode", type=str, required=True,
      choices=["horizontal", "vertical"],
      help="direction in which vehicles will be moving")
ap.add_argument("-i", "--input", type=str,
      help="path to optional input video file")
ap.add_argument("-o", "--output", type=str,
      help="path to optional output video file")
args = vars(ap.parse_args())

# load the configuration file
conf = Conf(args["conf"])

# initialize the MOG foreground background subtractor object
mog = cv2.bgsegm.createBackgroundSubtractorMOG()

# initialize and define the dilation kernel
dKernel = cv2.getStructuringElement(cv2.MORPH_RECT, (3, 3))

# initialize the video writer process
writerProcess = None

# initialize the frame dimensions (we'll set them as soon as we read
# the first frame from the video)
W = None
H = None

# instantiate our centroid tracker and initialize a dictionary to
# map each unique object ID to a trackable object
ct = CentroidTracker(conf["max_disappeared"], conf["max_distance"])
trackableObjects = {}

# if a video path was not supplied, grab a reference to the webcam
if not args.get("input", False):
      print("[INFO] starting video stream...")
      # vs = VideoStream(src=0).start()
      vs = VideoStream(usePiCamera=True).start()
      time.sleep(2.0)

# otherwise, grab a reference to the video file
else:
      print("[INFO] opening video file...")
      vs = cv2.VideoCapture(args["input"])

# check if the user wants to use the difference flag feature
if conf["diff_flag"]:
      # initialize the start counting flag and mouse click callback
      start = False
      cv2.namedWindow("set_points")
      cv2.setMouseCallback("set_points", set_points,
            [args["mode"]])

# otherwise, the user does not want to use it
```

```
else:
	# set the start flag as true indicating to start traffic counting
	start = True

# initialize the direction info variable (used to store information
# such as up/down or left/right vehicle count) and the difference
# point (used to differentiate between left and right lanes)
directionInfo = None
diffPt = None

# loop over frames from the video stream
while True:
	# grab the next frame and handle if we are reading from either
	# VideoCapture or VideoStream
	frame = vs.read()
	frame = frame[1] if args.get("input", False) else frame

	# if we are viewing a video and we did not grab a frame then we
	# have reached the end of the video
	if args["input"] is not None and frame is None:
		break

	# check if the start flag is set, if so, we will start traffic
	# counting
	if start:
		# if the frame dimensions are empty, grab the frame
		# dimensions, instantiate the direction counter, and set the
		# centroid tracker direction
		if W is None or H is None:
			# start the frames per second throughput estimator
			fps = FPS().start()

			(H, W) = frame.shape[:2]

			#Original code, modified and rewritten below
#			dc = DirectionCounter(args["mode"],
#				W - conf["x_offset"], H - conf["y_offset"])
			dc = DirectionCounter(args["mode"],
				W//2, H//2)
			ct.direction = args["mode"]



			# check if the difference point is set, if it is, then
			# set it in the centroid tracker object
			if diffPt is not None:
				ct.diffPt = diffPt

		# begin writing the video to disk if required
		if args["output"] is not None and writerProcess is None:
			# set the value of the write flag (used to communicate when
			# to stop the process)
			writeVideo = Value('i', 1)

			# initialize a shared queue for the exchange frames,
			# initialize a process, and start the process
			frameQueue = Queue()
```

```
                writerProcess = Process(target=write_video, args=(
                        args["output"], writeVideo, frameQueue, W, H))
                writerProcess.start()

        # initialize a list to store the bounding box rectangles
        # returned by background subtraction model
        rects = []

        # convert the frame to grayscale image and then blur it
        gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
        gray = cv2.GaussianBlur(gray, (5, 5), 0)

        # apply the MOG background subtraction model which returns
        # a mask
        mask = mog.apply(gray)

        # apply dilation
        dilation = cv2.dilate(mask, dKernel, iterations=2)

        # find contours in the mask
        cnts = cv2.findContours(dilation.copy(), cv2.RETR_EXTERNAL,
                cv2.CHAIN_APPROX_SIMPLE)
        cnts = imutils.grab_contours(cnts)

        # loop over each contour
        for c in cnts:
                # if the contour area is less than the minimum area
                # required then ignore the object
                if cv2.contourArea(c) < conf["min_area"]:
                        continue

                # get the (x, y)-coordinates of the contour, along with
                # height and width
                (x, y, w, h) = cv2.boundingRect(c)

                # check if direction is *vertical and the vehicle is
                # further away from the line, if so then, no need to
                # detect it
                if args["mode"] == "vertical" and abs(y - H//2) >
conf["limit"]:
                        continue

                # otherwise, check if direction is horizontal and the
                # vehicle is further away from the line, if so then,
                # no need to detect it
                elif args["mode"] == "horizontal" and abs(x - W//2) >
conf["limit"]:
                        continue

                # add the bounding box coordinates to the rectangles list
                rects.append((x, y, x + w, y + h))

        # check if the direction is vertical
        if args["mode"] == "vertical":
                # draw a horizontal line in the frame -- once an object
                # crosses this line we will determine whether they were
                # moving 'up' or 'down'
```

```
		cv2.line(frame, (0, H - conf["y_offset"]),
			(W, H - conf["y_offset"]), (0, 255, 255), 2)

		# check if a difference point has been set, if so, draw
		# a line diving the two lanes
		if diffPt is not None:
			cv2.line(frame, (diffPt, 0), (diffPt, H),
				(255, 0, 0), 2)

	# otherwise, the direction is horizontal
	else:
		# draw a vertical line in the frame -- once an object
		# crosses this line we will determine whether they were
		# moving 'left' or 'right'

		#This was the original code to draw diving line
		#cv2.line(frame, (W - conf["x_offset"], 0),
		#	(W - conf["x_offset"], H), (0, 255, 255), 2)

		cv2.line(frame, (W//2, 0), (W//2, H), (0, 255, 255), 2)

		# check if a difference point has been set, if so, draw a
		# line dividing the two lanes
		if diffPt is not None:
			cv2.line(frame, (0, diffPt), (W, diffPt),
				(255, 0, 0), 2)

	# use the centroid tracker to associate the (1) old object
	# centroids with (2) the newly computed object centroids
	objects = ct.update(rects)

	# loop over the tracked objects
	for (objectID, centroid) in objects.items():
		# check to see if a trackable object exists for the
		# current object ID and initialize the color
		to = trackableObjects.get(objectID, None)
		color = (0, 0, 255)

		# create a new trackable object if needed
		if to is None:
			to = TrackableObject(objectID, centroid)

		# otherwise, there is a trackable object so we can
		# utilize it to determine direction
		else:
			# find the direction and update the list of centroids
			dc.find_direction(to, centroid)
			to.centroids.append(centroid)

			# check to see if the object has been counted or not
			if not to.counted:
				# find the direction of motion of the vehicles
				directionInfo = dc.count_object(to, centroid)

			# otherwise, the object has been counted and set the
			# color to green indicate it has been counted
			else:
```

```
                        color = (0, 255, 0)

                # store the trackable object in our dictionary
                trackableObjects[objectID] = to

                # draw both the ID of the object and the centroid of the
                # object on the output frame
                text = "ID {}".format(objectID)
                cv2.putText(frame, text, (centroid[0] - 10,
                      centroid[1] - 10), cv2.FONT_HERSHEY_SIMPLEX, 0.5,
                      color, 2)
                cv2.circle(frame, (centroid[0], centroid[1]), 4, color,
                      -1)

          # extract the traffic counts and write/draw them
          if directionInfo is not None:
                for (i, (k, v)) in enumerate(directionInfo):
                      text = "{}: {}".format(k, v)
                      cv2.putText(frame, text, (10, ((i * 20) + 20)),
                            cv2.FONT_HERSHEY_SIMPLEX, 0.6, (0, 0, 255), 2)

          # put frame into the shared queue for video writing
          if writerProcess is not None:
                frameQueue.put(frame)

          # show the output frame
          cv2.imshow("Frame", frame)
          key = cv2.waitKey(1) & 0xFF

          # if the `q` key was pressed, break from the loop
          if key == ord("q"):
                break

          # update the FPS counter
          fps.update()

    # otherwise, the user has to select a difference point
    else:
          # show the output frame
          cv2.imshow("set_points", frame)
          key = cv2.waitKey(1) & 0xFF

          # if the `s` key was pressed, start traffic counting
          if key == ord("s"):
                # begin counting and eliminate the informational window
                start = True
                cv2.destroyWindow("set_points")

# stop the timer and display FPS information
fps.stop()
print("[INFO] elapsed time: {:.2f}".format(fps.elapsed()))
print("[INFO] approx. FPS: {:.2f}".format(fps.fps()))


# terminate the video writer process
if writerProcess is not None:
      writeVideo.value = 0
```

```
	writerProcess.join()

# if we are not using a video file, stop the camera video stream
if not args.get("input", False):
	vs.stop()

# otherwise, release the video file pointer
else:
	vs.release()

# close any open windows
cv2.destroyAllWindows()
```