Traffic Camera Dangerous Driver Detection TCD^3

Contextually Aware Heuristic Feature & OFA Density-Based Computer Vision with Inertial Movement Machine Learning Analysis of Live Streaming Traffic Camera Footage to Identify Anomalous & Dangerous Driving

www.drunkdriverdetection.com

Intel International Science & Engineering Fair 2016

Project Researcher: Vidur Prasad

Project Advisors: Dr. Nilesh Powar & Dr. Robert Williams

AIR FORCE RESEARCH LABORATORY

UDRI UNIVERSITY of DAYTON RESEARCH INSTITUTE

DISCOVERY LAB

TecEdge Wright Brothers Institute Innovation and Collaboration Center

IDCAST

Traffic Camera Dangerous Driver Detection

Vidur Prasad
TCD^3
12th Grade
Dayton Regional STEM School
Dayton, Ohio, United States
vidurtprasad@gmail.com

Dr. Nilesh Powar
University of Dayton Research Institute
Institute for the Development and Commercialization of Advanced Sensor Technologies
Dayton, Ohio, United States
Nilesh.Powar@udri.udayton.edu

ABSTRACT

The goal of TCD^3 project is to identify anomalous driving pattern from traffic camera feeds. Successful execution can improve road safety by assisting law enforcement catch dangerous drivers, who text while driving or drink and drive. TCD^3 overcomes several technical challenges such as detecting vehicles under different lighting conditions, tracking vehicles in different frames, and distinguishing random variations in a vehicle's path due to normal driving from anomalous variations due to distracted driving. TCD^3 C++ script runs on a server, receiving live streaming traffic camera feed. A heuristic Computer Vision algorithm, that uses custom morphological-based operators, performs computer vision to reliably determine vehicle positions. Image registration allows a vehicle's path to be analyzed through multiple frames. A test suite of traffic camera footage was used to evaluate vehicle detection. Frames were doctored and drunk drivers were simulated to test the system. Machine learning was used for historical and active comparative analyses of vehicle paths to identify anomaly. The system is contextually aware and is robust with respect to normal irregularities in traffic patterns such as from red lights. Permission for large scale testing of our prototype on actual high fidelity traffic camera footage has been requested. Upon detection, relevant video clip will be extracted and sent to law enforcement for further action. To increase affordability, processing speed, and scalability, a multi-node networked Spark-based supercomputing architecture is being investigated. TCD^3 is multi-threaded for maximum resource allocation.

INTRODUCTION

The TCD^3 project was conceptualized from the growing need to have automated active analyses of drivers on roads to detect distracted or drunk drivers.

The aim of the project was to use open source, low-level languages to be create Computer Vision and machine learning algorithms to actively analyze traffic camera footage to identify drunk or distracted drivers. This paper will outline the different foundational algorithms that were used, the reasoning and justification for their selection, there importance, and their implementation.

DRUNK & DISTRACTED DRIVING

Drunk and distracted driving is a problem that is growing in our society, and is a problem that affects us all. The current paradigm for solving this pressing problem is fundamentally outdated:

convince drivers to not drive drunk or distracted, or use human resources to identify these drivers. Unfortunately, this solution is flawed as it has been concretely proven that drunk and distracted driving is not on a decline, even with the advent of commercials urging people to drive safely, and with recent law enforcement budget cuts, it is becoming more difficult to deploy human assets to observe and find these drivers.

The current push to use technology to solve the problem is also flawed for two reasons, one, it requires new hardware and investments to pay for it, and two, many of these methods call for drivers to manually opt-in to self-monitor, which is not a permanent nor a viable solution.

There are over 600,000[1] distracted drivers on the road on any daylight hour in the United States. There are also over, on average, 300,000 injuries and deaths resulting from just drunk driving every year.

After analyzing this problem, it is clear that the only long-term, and financially, as well as technically, feasible solution is to use existing technology and have a pervasive and active monitoring system.

TECHNICAL GOAL

The goal of this project is to create a hardware and software package that is able to take in live streams of traffic camera footage and process these streams to determine the position of the cars in the feed. The cars' positions are then intelligently analyzed to determine if the cars are exhibiting anomalous behavior commonly shown by distracted or drunk drivers. If a car is flagged as being suspected of being driven by a drunk or distracted driver, then the clip of video containing the car, and its license plate[2], is sent to law enforcement for final validation, and for further investigation.

This system will help law enforcement target its efforts by leveraging the current visual sensing infrastructure to identify suspicious drivers.

PREVIOUS WORK

TCD[3] was initially created using Octave, a language based on Matlab, by Mathworks, and used simple blob analysis and manual background subtraction to identify cars. Morphological operators were used to refine detects, and the grassfire algorithm was utilized to identify and register discrete objects. A system to record the x and y displacement of the coordinates was created to perform rudimentary threshold to determine when the cars were moving outside of the normal zone. This work was instrumental in understanding the complexity of Computer Vision, and developing a paradigm that is used in the current iteration of TCD[3].

[1] Conservative estimate by CDC

[2] Image-Analysis software to read license plates already exists using OCR

Another foundational project was Evalu8: Autonomous Image Analysis of Drone Footage to Evaluate Visual Perception Load & Clutter Using C++ & OpenCV; from which the TCD^3 architecture was adapted, and the proper usage of OpenCV APIs. Evalu8 performed multiple feature extraction and Optical Flow Analysis methods, tasks which are essential for TCD^3 to effectively and reliably detect cars.

C++

C++, long considered the main language to be used for Computer Vision, alongside Python, was chosen for use for its broad support with various Machine Learning and Computer Vision APIs, as well as for its low-level access to optimize systems. TCD^3 is written in C++ as OpenCV, the primary library used, is natively written in C++, and other versions, such as Python, operate with wrappers. C++ offered some obstacles in regards to memory leaks when working with larges sets of frames, but efficient use of pointers and careful memory checking was able to reduce RAM usage to approximately 2 GB.

OPENCV

OpenCV is a library that contains more than 2500 algorithms for use in Computer Vision and Machine learning. OpenCV provides a Matrix data structure that has broad support across multiple other libraries, such as BGSLibrary. OpenCV, as it is completely implemented in C++, allows all of its algorithms to be edited and tweaked for the specific use case. It is also protected under the BSD license, making it ideal for use in Commercial Applications such as TCD^3.

STEP 1: READING IMAGES

TCD^3 reads video files, of standard MP4 or MOV type, and converts the files into individual frame that are then passed through the system to be processed.

An image is a stack of matrices; one for each of the three colors, red, green, and blue, the components of RGB. Images are read and saved into a vector of Mat objects with pointers to this vector passed through the system.

This architecture allows all frames to be stored once, reducing RAM load, and allows all frames to be saved only once, reducing IO usage. Client methods can additionally clone data from global vector if analysis will edit actual frames.

On startup, a buffer of frames are loaded into the vector, usually around 100, and the architecture is in place to allow methods to access the buffer as it is being built to begin initializing their systems.

The buffer is also created to ensure that no systems ever cause an out of bounds error as there is sufficient memory. The size of the buffer controls multiple events as it effects the time it takes to train, and effects the timing of initialization as many steps must be completed in sequence, as they cannot be done in parallel, and proper relational timing is kept to ensure that no matter the size of the buffer memory, there are no runtime errors.

The size of the buffer, also, as expected, affects the quality of the Background Detection, as many operations, such as the median background generator, rely on large data sets to reliably estimate and separate the background from the foreground objects.

An additional vector of gray scale frames is also maintained as certain algorithms do not require color.

The raw frames that are read and kept for conversion are deleted at the end of every cycle, after one frame has been processed, and are scrubbed to reduce RAM usage and ensure that there are not memory leaks.

PHASE 1: COMPUTER VISION

STAGE 1: GENERATE BACKGROUND MODEL

STEP 2: BACKGROUND MODEL MEDIAN

The first background subtraction method that is used is the median model, or BMM. TCD^3 can generate BMMs, and can read a BMM, stored as an image and containing the median value for each pixel, and perform analysis. A BMM is generated by stepping through the entire buffer memory and creating a vector of integers of every pixel value in history for every specific position, and calculating the median value.

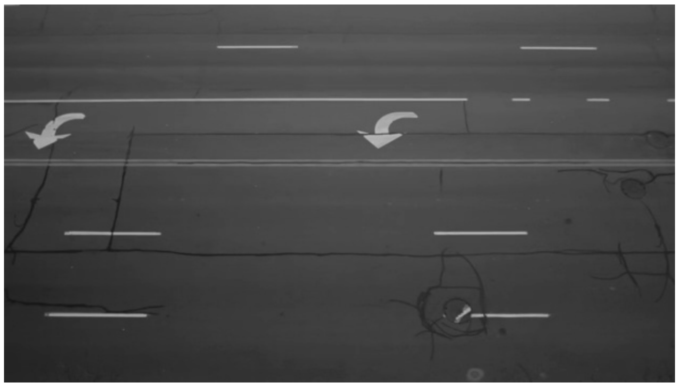
A mean based model was investigated, but an accurate reading required a very large buffer memory, and removing ghosting is impossible, as variations will constantly affect the learned average.

Another pro to using a median based model is that with advanced sorting methods such as Tim Sort, a median calculation is highly efficient.

The BMM generation runs on its own thread, and has the ability to perform simultaneous operations by separate the image and process using multiple threads to decrease compute time. As the current testing rig has only 8 cores, further branching of threads will not increase speed, but it is an option for the future.

The BMM model is generated every 1.5 minutes to account for changing weather conditions, as well as general lighting changes. The thread silently works in the background, and does not slow runtime as the new model is used only when the BMM is successfully generated.

A sample of a BMM generated model is shown below, at no point in the test footage was a picture of the road without cars ever shown, proving that the BMM model is able to effectively generate a background model. The BMM model shown was generated with a buffer memory of 100 frames. The image also shows no ghosting, a common artifact of background model generation.



All BMM models are generated in gray scale for speed, as a RGB model would require 3X time, and as background subtraction is more efficient and clean using gray scale images.

The BMM model is also used for display purpose as all detected cars and objects are laid on the frame.

TCD^3 save all BMMs as JPEG so that the system can read a BMM without having to recalculate at runtime.

STEP 3: BACKGROUND MODEL GAUSSIAN MODEL (Kaewtrakulpong & Bowden Approach)

Multiple background subtraction methods exist that use techniques such as BMM, however, these methods, especially BMM, suffer as a result of lighting changes and are forced to recalculate a model periodically to remove these changes.

The Kaewtrakulpong & Bowden approach is designed to use a Gaussian Mixture Model in conjunction with revised model update equations to allow for more accurate lighting adjustment, as well as to allow for shadow removal in the model. The approach, referred to in the OpenCV library as the MOG1 approach, utilizes this approach to reduce the training time to detect the objects.

TCD^3 utilizes MOG1 and trains during run time by feeding color images to fill the GMM. This model also uses a kernel estimator for each pixel that allows regular movements, such as in trees in the image, to be ignored. Other systems utilized by TCD^3, such as Optical Flow Analysis, are also used to further crosscheck the models, and increase overall accuracy.

MOG1 runs on its own separate thread, and is one of the slowest of the Computer Vision algorithms used by TCD^3 for real time computation.

STEP 4: BACKGROUND MODEL GAUSSIAN MODEL (Zivkovic Approach)

The second Gaussian approach utilizes recursive equations to actively update the parameter of the model, such as the number of Gaussian mixture models, and size of the memory for each pixel. The Zivkovic approach is also referred to as MOG2 in OpenCV.

This adaptive method offers better response to objects such as wildlife or shadows that normally mess up other background subtraction methods.

The primary purpose of running both the Kaewtrakulpong & Bowden and the Zivkovic approach was to adapt the model to the conditions, allowing for a system that is able to update and reflect the scene as accurately as possible. Running both models also allows multiple guesses to be made, which increases the accuracy of detection.

This also furthers the central heuristic paradigm of TCD^3 to use a variety of methods to increase the chance of the detects, and to use redundancy in detect to calculate detect confidence.

This model also furthers the paradigm that the system should be able to adapt to any situation with minimal initial input, as it is able to adapt to lighting and weather changes without requiring outside sensor input.

MOG2 runs on its own separate thread, and is one of the fastest methods for real time computation.

STEP 5: BACKGROUND MODEL GAUSSIAN TRADITIONAL MIXTURE MODEL

TCD^3 also uses a traditional GMM model, using three mixtures, to adjust and deal with changes in the lighting conditions. The traditional GMM model does not work as well as MOG1 or MOG2 as it is not able to adapt to changing conditions as well, and is also not able to intelligently develop a GMM.

After multiple tests, it was determined that using a three mixture model gave the most accurate detect, but due to the variance in the number of mixtures actually found in the sensor data, the traditional GMM model has many false-positives. To help reverse this, the buffer memory for the GMM is tripled.

Of the detection methods, the traditional GMM is weighted the lowest in the final confidence detector, and future versions will quite likely stop using the traditional GMM and rely solely on MOG1 and MOG2 for a GMM based background mode.

STEP 6: BACKGROUND MODEL ViBe

ViBe is a commercial background subtraction algorithm designed to perform analysis over multiple frames to create a background model.

ViBe works by analyzing the values of a pixel over a period and then calculating the probability density function. This method cause ViBe to require a long training time, about seven times the buffer memory, to get a proper understanding of the background model.

ViBe, through its use of advanced statistical analysis of pixel histories, is able to create a highly accurate model of the background. Given enough time, ViBe returns a model of the cars better than both MOG1 & MOG2.

TCD^3 runs ViBe even after real-time analysis begins, and continues training and applies ViBe after an appropriate training period, about seven times the buffer memory.

ViBe runs on a separate thread, and is able to run real time during both the training and the run phase.

STAGE 2: DETECT CARS

STEP 7: BACKGROUND SUBTRACTION

The next step in TCD^3 is to use the generated background models to perform subtraction with the current frame to determine the difference, or objects in the frame.

Background subtraction starts with the Median background frame, which is subtracted from the raw gray scale frame. This returns a Matrix where the amplitude of the difference correlates with the actual difference, allowing for thresholds to be applied.

After subtraction, a binary image is generated with every value above 50, eliminating small differences due to shadows, saved as one. Using a binary image also greatly decreases compute time.

The various models also generate binary masks that show the difference, calculated through various methods, to identify objects.

STEP 8: OPTICAL FLOW ANALYSIS OBJECT DETECTION

During the planning stages of TCD^3, research work was conducted to gain an understanding how humans understand images. One of the main things that was gleaned from this research was that humans use a multi-faceted approach, using different "Computer Vision" like methods to create a composite model, from which a final decision to identify and recognize an object is done.

To emulate this method, TCD^3 utilizes multiple background subtraction methods to understand an image. However, to add redundancy, an Optical Flow based method was created to detect objects.

The Farneback Dense Optical Flow model was used to generate a model, for each pixel, that showed its movement frame over frame.

Using this Optical Flow model, a threshold is run to detect every pixel that has movement. From this, the Sliding Window Neighbor Detector is used to clean the image, and develop a complete model of all large motion in the image.

This is an effective method as all the cars, no matter how optically complex and similar to the background, can be caught through motion.

The actual Farneback Optical Flow algorithm creates an image of motion vectors, from which a heat map of motion is generated. This then goes through the

The optical flow analysis and the object-refining portion are performed on their own thread.

STAGE 2: CLEAN DATA

STEP 9: SLIDING WINDOW NEIGHBOR DETECTOR

Various morphological approaches were tried to attempt to repair holes and other issues with the results of background subtraction, but required too much compute time, as well as did not do a good job. One of the biggest issues was that running repeated morphological operators degraded the quality of the detect by causing multiple cars close together to morph into one, as well as were not adequate at removing noise.

To address this problem, a custom recursive algorithm was created to read in raw background subtracted day and perform image cleaning in a manner reminiscent of morphological operators.

The Sliding Window Neighbor Detector, or SWND, is designed to determine a pixels value based on the density of its neighbors, and therefore fill holes and remove noise, while retaining the quality of an outline of a well define object. The algorithm works by sliding through each pixel while observing the area around the pixel to determine if the pixel is part of well-defined object.

A pixel will be determined as existing if over 25% of its neighbors exist. This ensures that holes inside of images will be filled, as they will have non-adjacent relatively close neighbors that cause the SWND to detect a pixel. SWND is also able to remove noise, as it is able to identify a pixel as being a loner as its neighborhood does not contain many other pixels.

The property of SWND that makes it powerful, and quite accurate, is its ability to work recursively by starting with relatively small sectors to larger and larger sectors that carefully remove more noise, fill larger holes, and, more importantly, careful knead the objects into cleaner, more ovular shapes. This approach drastically reduces noise, and almost perfects object detections.

The SWND is usually run three times with quadrupling sector size to get a proper detect.

SWND is used for all data sources, and is one of the primary innovations in the Computer Vision Phase as it allows for all sensor values to be carefully analyzed, and equalized as they all pass through the same processer.

Equalization is critical as it is important that if not every object receiving a similar detect, a single car must keep a similar detect through its time to ensure that all deviations are caused by the cars path, not by inaccurate detects.

SWND is optimized to move through the image in the most efficient path as possible, and does the minimum movement. SWND also attempts to scrub temporary variables as quickly as possible to reduce RAM usage.

As SWND is imperative for a detect to be considered finished, it does not open a new thread, and instead is launched from each thread running a computer vision algorithm.

STEP 10: CANNY CONTOUR DETECTOR

The next step in the process is to take the raw binary images and determine the center point of all of the cars. To detect the individual cars as discrete objects, the Canny contour detector is used to identify just the outlines. A raw threshold is then used to remove objects too small to be a car. Objects that are on the outsides, where a complete detect is not yet possible, are thrown out. This is to ensure that all of the anomaly detection happens after a proper solid detect is acquired.

After the canny contours are detected, the center points of the contours are saved into a global vector of points.

This method allows every car detected, through each of the different Computer Vision methods, to be saved in one vector.

PHASE 2: TRACKING MACHINE LEARNING

STAGE 1: PROCESS COORDINATES

STEP 1: AVERAGE DETECTED POINTS

After all of the different Computer Vision methods run, a method to cluster the points was devised. K-Means is the optimal method, but it requires that one knows the number of cars before running the clustering algorithm.

The averaging system works by identifying all the points close to each other, and then averaging them and finding the center point of the cars. The distance threshold is set so that it is large enough to detect all points that are within a car, but small enough to not detect two adjacent cars.

A system to have different distances threshold for the x and y since cars have different sizes for each dimension.

The final detected points are then saved into the same vector of detected points.

STEP 2: LEARNING AREA SECTOR MODEL

One of the central qualities of TCD^3 is the ability to be able to adapt and work in a diverse array of highway situations. One of the models used is an area sector model, where an image is divided up into sectors of 7 pixels by the size of one lane’s dimension.

This requires the number of lanes to be entered into the system. A grid is created where a car is classified into the lane that it is traveling in, and then every general area, 7 pixels across, to have an updated model of where most cars are when they travel through each sector. This allows the model to continuously update and become more accurate the longer it is used.

This model is generated so that thresholds can be applied to determine if a car is outside of the safe range.

This model is also useful so that scenes with multiple different roads and different directions and speeds will be able to intelligently apply correct models to appropriate cars.

The Learning Area Sector Model will eventually be modified to include angular and movement data, in addition to position data, for each of the individual sectors. The LASM is useful as it is an infrastructure that is able to utilize any metric and save it into sectors. LASM, and LASM Enforcement, are designed to use raw thresholds, as well as learned models.

LASM & LASME are generated using a vector of positions that is entered by the user, which shows the boundaries between each lane.

LASM & LASME both run on the main thread.

STEP 3: MATCHING COORDINATES

The next step, prior to running deeper analysis, is to determine the x and y displacement for each car. This is done by saving all of the coordinates of the cars into a vector of vectors of coordinates, and the comparing the current frame of detects to the previous frame of detects. A matching algorithm was created to allow cars to be tracked frame over frame, and from this, the x and y displacement of each car was extrapolated.

STAGE 2: MOVEMENT PROPERTY DETECTION

STEP 4: LEARNING X AND Y NORMAL

The first machine-learning algorithm that is run is a simple averaging system that accumulates the x and y movements for every car, and then average by the number of detects, thereby building an accurate model of the movement.

The x and y movements are useful as they can be used to identify excessive movement, as well as changes in acceleration, which is a main sign of drunk or distracted driving.

The movement property calculations are all calculated on the main thread, after the Computer Vision stage runs.

STEP 5: LEARNING ANGULAR MOVEMENT

The second machine-learning algorithm is to determine the normal range of angular movement that cars exhibit.

The angular movement is calculated by computing the inverse tangent of the x and y movement. The angular movement is an excellent indicator to identify anomalous behavior as it gives a clear indication of the direction of acceleration, and shows if someone is swerving.

The angular movement is learned so that the limit of normal movement can be learned. It is from this metric that the threshold value for anomalous behavior can be calculated. In addition, a hard stop of 60 degrees is also implemented, unless the road scene specifically has a tight turn.

STEP 6: LEARNING SPEED

The next step, designed to also perform the job of speed cameras, is to use the Pythagorean Theorem to calculate total movement each frame. The normal displacement, as it is calculated frame over frame, yields the speed.

STAGE 3: DETECTING ANOMALIES

STEP 8: HARD THRESHOLD DETECTION

One method to detect anomalies is to use hard thresholds for each of the different properties. The hard threshold method is used to catch anomalous cars even if the learning stage has not created a reliable model of drivers yet.

The first hard threshold that is set is for the individual x and y displacement. This is useful to find users making quick swerve movements. The thresholds are calibrated based on the situation, and are applied to all of the different paths. The x and y displacement is calculated with an absolute value, so that it works for both direction of lanes. In some situations, taking the inverse of the x and y displacement is useful for some situations that have roads going in different directions.

The issue with the hard thresholds, and the reason that the learning system is given a higher weight during actual run, is that it is difficult to craft thresholds that work for each of the different lanes and parts of the road.

This is one of the primary reasons that the learning area sector model is used.

STEP 9: LEARNED MODEL DETECTION

The primary method that TCD^3 uses to detect anomalies is by comparing active movement with the learned model. For each of the metrics, a proportional, as well as absolute distance based method is used to recognize anomalies. For the proportional system, if the anomaly exceeds a 25% safety zone, for any of the metrics, then it is considered anomalous. However, in some cases, the numbers may be large enough that a 25% zone may not be exited, but the amplitude of the cars movement will be greater than the learned model, in which case subtraction is done to find the difference.

Any time an anomaly is detected, it is written to a frame of anomalies. This frame of anomalies is cleared after a set time, the time it takes for a car to go across the frame, after the first anomaly is detected. If more than one anomaly is detected in the same frame, a separate frame of anomalies is opened and saved to.

The confidence of an anomaly detect can also be ascertained by counting the number of movement properties that exceed limits, the amount of time that the cars are anomalous, as well as the amplitude with which the car exceed safe limits.

If these frames have more than the required number of anomaly detects, usually at least 10, corresponding to 2/3 of second of anomalous behavior, at 15 FPS, the corresponding video is tagged as anomalous. This method ensures that one time detects, or small movements are not considered anomalous, and a long registered set of anomalies across the life cycle of a cars movement across the frame is detected.

This method ensures that small variances are not caught, and only large deviations are caught that occur over an extended period of time. This is critical in removing false positives, and removing instances where the system incorrectly identifies one or two movements as anomalous. A long series of anomaly detects therefore shows, with high probability that the driver is indeed anomalous.

This system runs on the main thread, and is the final step in the process of detecting anomalies.

PHASE 3: PROCESSING ENHANCEMENTS

STAGE 1: CPU OPTIMIZATION

STEP 1: MULTI-THREADING

One of the main methods used to optimize TCD^3 was to multi-thread the system to spread processing load across multiple cores. The pthread library is used to open threads, check successful thread completion, and close threads after the task finished.

For each of the major tasks listed above, a new thread is opened and global variables are used to pass data into the threads. The threads all poll the same global vectors that contain the frames.

The paradigm for processing and controlling the multiple threads is to utilize thread handlers that each handle a set of threads and report completion back to the main thread. Each of the major steps has a thread handler that launches all necessary threads to finish a step.

A more abstract thread handler then waits for its child threads to finish, before terminating completion and reporting back to the main thread, where a while loop executes until all major Computer Vision steps are complete. This system is designed specifically for scalability and to be able to add more steps to the system without increasing compute time.

This system of abstraction also makes it easier for more methods to be added, allowing the system to be continually updated, and work with the existing infrastructure.

STEP 2: SAVING MODELS

Efforts were also made to lower compute time by reducing the amount of things that needed to be calculated every frame, and therefore allow the system to work faster. Some of the models, such as the BMM, can be saved and read during run time to avoid recalculation.

To reduce speed, models such as BMM are also actively recalculated, during runtime to get a more accurate model, in the background until they are finished so that no steps have to wait for completion.

This is part of the primary processing paradigm of TCD^3 all steps should be executed in series if and only if there is no way to perform the steps in parallel. This paradigm has allowed the system to be optimized as much as possible, and will allow more powerful hardware to significantly decrease compute time.

STAGE 2: LATENCY ISSUES

Another major issue with this system is the latency involved into analyzing large amounts of sensor data in real time. Because of the sheer vast amount of frames that have to be analyzed, one of the longest steps in the process is when the media is read into the RAM for further analysis. Several techniques were used to reduce the latency, such as using internal SSDs with no other file transfer occurring while TCD^3 was running.

STAGE 3: RASBERRY PI EXPERIMENTATION

Another solution that was considered, using networked Raspberry Pis, was conceptualized, and preliminary implementation has occurred. This method involved networking multiple Raspberry Pis, initially trialed with two, but will eventually be implemented with around 20, and manually assigning each Raspberry Pi a different piece of the execution sequence, so that they could then each process the data and return their results to the host Raspberry Pi. This would bring the application closer to a scalable system as it would allow for more or less Raspberry Pis to be used based on the size of the application. The Raspberry Pis can be networked, as they use Arch Linux using Python, into an interface where commands can be sent to any Raspberry Pi using the host. This system will be the final implementation of TCD^3, and will also incorporate the previous cooperative GPU and CPU processing paradigm.

In conclusion, many different methods have been conceptualized and solidified to efficiently execute the code, through better resource allocation, and through multi-threading, to achieve a system that is able to analyze video intelligently in real time.

PHASE 4: CONCLUSIONS

STAGE 1: TESTING

One of the challenges to creating a system such as TCD^3 is that there is a lack of footage to test and see if the system is able to detect drunk or distracted drivers. To rectify this, it was decided that drunk or distracted driving footage would be artificially generated.

To do this, iMovie was used to edit in cars with tracks that emulated drunk drivers. Using the path-editing tool, a car was edited in to show it swerving through the frame, as well as speeding through the frame.

After creating multiple different videos, they were run through the system to ensure that TCD^3 would be able to detect a variety of anomalous movements accurately.

Of all of the different anomaly detectors, the angle detector was the most accurate of the anomaly detector, as it had the ability to identify even small motions as they lead up to a large anomalous motion as it could identify a car moving toward the edge.

The system was also tested by running footage of cars changing lanes, and making small anomalous movements. This was designed to ensure that TCD^3 was smart enough to avoid false positives.

This system of creating test footage to ensure that the different parts of TCD^3 all work effectively is critical to ensuring that it is ready to work in reality.

STAGE 2: FUTURE WORK

The future of this project is very bright as there is a strong foundation currently implemented and tested. In addition, the entirety of the project has been conceptually created, with prototyping already having occurred for a large part of the project that has not been implemented yet.

In addition, testing has not been able to be completed on the implemented portions of the Machine Learning, as high-fidelity traffic camera data is extremely difficult to find for free. To attempt to get actual high resolution, high frame rate traffic camera footage, requests have been made to the Dayton Police Department and San Francisco Police Department. The University of Dayton Research Institute also has access to traffic camera footage in Dayton, Ohio, and efforts are being made to test with using this footage to test the system. In the future, collaboration will also occur with MADD to make it easier for research institutions to get traffic camera footage. Until these requests are processed, full scale testing of the entire system will not be able to be completed.

After this, the system to facilitate faster processing will be completed, on the hardware and the software side, to allow for the system to be used in actual applications.

STAGE 3: IMPACT

This project will have a massive impact on the way that we deal with drunk and distracted driving, as it will finally allow us to move our enforcement of DUI laws into the 21st century. The solution that was designed is crafted for scalability, and will therefore make it possible for use in a variety of cities of different sizes. The system utilizes many different algorithms that are designed to maximize efficiency to increase accuracy. By having a system that is able to utilize the current resources and leverage current technology to help law enforcement, we have eliminated the need to consistently use human resources to monitor the roads. The system is also intrinsically private, as the license plates of cars are only recorded, and the data is only given to humans when a car's path is flagged as exhibiting anomalous behavior. In conclusion, the technical advancements that we have made throughout this project have allowed for a system that is able to efficiently deal with the pressing issue of distracted and drunk drivers.

ACKNOWLEDGMENTS

I would like to thank Dr. Nilesh Powar, Mr. Kelly Riggins, Mr. Kevin Klawon, Mr. Josh Gold & Mr. Larrell Walters at the University of Dayton Research Institutes' Institute for the Development and Commercialization of Advanced Sensor Technology for their administrative support and funding for this project, and for creating an workspace that fostered technical innovation.

I would also like to thank Dr. Robert Williams for giving me the opportunity to intern at the Air Force Research Lab's Discovery Lab. An environment that encouraged innovation, and allowed students to pursue their passion, was essential in ensuring that research on TCD^3 continued at the pace that it did.

I would also like to thank the wonderful open source community around C++ & OpenCV at StackOverflow that helped accelerate development.

I would also like to thank the open source community around GNU Octave that made it possible to use propriety algorithms implemented in Matlab in an open source manner.

I would like to dedicate this project to the countless numbers of innocent people who have been injured or killed as a result of drunk or distracted driving.

TCD^3

```
//====================================================================================
// Name        : TrafficCameraDistractedDriverDetection.cpp
// Author      : Vidur Prasad
// Version     : 0.5.0
// Copyright   : Institute for the Development and Commercialization of Advanced Sensor Technology Inc.
// Description : Detect Drunk, Distracted, and Anomalous Driving Using Traffic Cameras
//====================================================================================


//include opencv library files
#include "opencv2/highgui/highgui.hpp"
#include <opencv2/objdetect/objdetect.hpp>
#include <opencv2/imgproc/imgproc.hpp>
#include <opencv2/video/tracking.hpp>
#include "opencv2/nonfree/nonfree.hpp"
#include "opencv2/gpu/gpu.hpp"
#include <opencv2/nonfree/ocl.hpp>
#include <opencv/cv.h>
#include <opencv2/video/background_segm.hpp>
#include <opencv2/core/core.hpp>
#include "bgfg_vibe.hpp"

//include c++ files
#include <iostream>
#include <fstream>
#include <ctime>
#include <time.h>
#include <thread>
#include <chrono>
#include <stdio.h>
#include <stdlib.h>
#include <limits>
#include <math.h>
#include <algorithm>
#include <vector>
#include <pthread.h>
#include <cstdlib>

//include CMT
#include "CMT.h"

//include CCV
#include <ccv.h>

#define PI 3.14159265

//namespaces for convenience
using namespace cv;
using namespace std;

////global variables////

//multithreading global variables
```

```
vector <Mat> globalFrames;
vector <Mat> globalGrayFrames;

vector <Point> detectedCoordinates;

CvHaarClassifierCascade *cascade;
CvMemStorage  *storage;

//vibe constructors
bgfg_vibe bgfg;
BackgroundSubtractorMOG bckSubMOG; // (200,  1, .7, 15);

//global frame properties
int FRAME_HEIGHT;
int FRAME_WIDTH;
int FRAME_RATE;

//global counter
int i = 0;

//global completion variables for multithreading
int medianImageCompletion = 0;
int medianColorImageCompletion = 0;
int opticalFlowThreadCompletion = 0;
int opticalFlowAnalysisObjectDetectionThreadCompletion = 0;
int gaussianMixtureModelCompletion = 0;
int vibeDetectionGlobalFrameCompletion = 0;
int mogDetection1GlobalFrameCompletion = 0;
int mogDetection2GlobalFrameCompletion = 0;
int medianDetectionGlobalFrameCompletion = 0;

//background subtraction models
Ptr<BackgroundSubtractorGMG> backgroundSubtractorGMM =
Algorithm::create<BackgroundSubtractorGMG>("BackgroundSubtractor.GMG");
Ptr<BackgroundSubtractorMOG2> pMOG2 = new BackgroundSubtractorMOG2(500, 64, true);
Ptr<BackgroundSubtractorMOG2> pMOG2Shadow = new BackgroundSubtractorMOG2(500, 64, false);

//matrix holding temporary frame after threshold
Mat thresholdFrameOFA;

//matrix to hold GMM frame
Mat gmmFrame;

//matrix holding vibe frame
Mat vibeBckFrame;
Mat vibeDetectionGlobalFrame;

//matrix storing GMM canny
Mat cannyGMM;

//matrix storing OFA thresh operations
Mat ofaThreshFrame;

//Mat to hold Mog1 frame
Mat mogDetection1GlobalFrame;
```

```
//Mat to hold Mog2 frame
Mat mogDetection2GlobalFrame;

//Mat objects to hold background frames
Mat backgroundFrameMedian;
Mat backgroundFrameMedianColor;
Mat medianDetectionGlobalFrame;

//Mat for color background frame
Mat backgroundFrameColorMedian;

//Mat to hold temp GMM models
Mat gmmFrameRaw, binaryGMMFrame, gmmTempSegmentFrame;

Mat finalTrackingFrame;
Mat drawAnomalyCar;

//Mat for optical flow
Mat flow;
Mat cflow;
Mat optFlow;

Mat ofaGlobalHeatMap;

vector <vector <Point> > carCoordinates;

vector <vector <Point> >  vectorOfDetectedCars;

vector <Point> globalDetectedCoordinates;

bool objectOFAFirstTime = true;

//optical flow density
int opticalFlowDensityDisplay = 5;

//Buffer memory size
const int bufferMemory = 95;

//boolean to decide if preprocessed median should be used
bool readMedianImg = true;

//controls all displayFrame statements
bool debug = false;

//controls if median is used
bool useMedians = true;

double xAverageMovement = 0;
double xAverageCounter = 0;
double xLearnedMovement = 0;

double yAverageMovement = 0;
double yAverageCounter = 0;
double yLearnedMovement = 0;

double learnedAggregate = 0;
double learnedAngle = 0;

double currentSpeed = 0;
double learnedSpeed = 0;
double learnedSpeedAverage = 0;

double currentDistance = 0;
double learnedDistance = 0;
double learnedDistanceAverage = 0;
double learnedDistanceCounter = 0;

const int mlBuffer = 3;

String fileTime;

vector <int> lanePositions;

vector <vector<Point> > coordinateMemory;

vector<vector <Point> >learnedCoordinates;

vector <vector <Point> > accessTimes;

vector <vector <int> > accessTimesInt;

vector <double> distanceFromNormal;
vector<Point> distanceFromNormalPoints;

//setting constant filename to read form
//const char* filename = "assets/testRecordingSystemTCD3TCheck.mp4";
//const char* filename = "assets/ElginHighWayTCheck.mp4";
//const char* filename = "assets/SCDOTTestFootageTCheck.mov";
//const char* filename = "assets/sussexGardensPaddingtonLondonShortElongtedTCheck.mp4";
//const char* filename = "assets/sussexGardenPaddingtonLondonFullTCheck.mp4";
//const char* filename = "assets/sussexGardenPaddingtonLondonFullEFPSTCheck.mp4";
//const char* filename = "assets/sussexGardenPaddingtonLondonFPS15TCheck.mp4";
//const char* filename = "assets/sussexGardenPaddingtonLondonFPS10TCheck.mp4";
//const char* filename = "assets/genericHighWayYoutubeStockFootage720OrangeHDCom.mp4";
//const char* filename = "assets/xmlTrainingVideoSet2.mp4";
//const char* filename = "assets/videoAndrewGithub.mp4";
//const char* filename = "assets/froggerHighwayTCheck.mp4";
//const char* filename = "assets/OrangeHDStockFootageHighway72010FPSTCheck.mp4";
//const char* filename = "assets/trafficStockCountryRoad16Seconds30FPSTCheck.mp4";
//const char* filename = "assets/cityCarsTraffic3LanesStreetRoadJunctionPond5TCheck15FPSTCheck.mp4";
//const char* filename = "assets/froggerHighwayDrunk.mp4";
//const char* filename = "assets/froggerHighwayDrunkShort.mp4";
//const char* filename = "assets/froggerHighwayDrunkV2.mp4";
//const char* filename = "assets/froggerHighwayDrunkV4TCheck.mp4";
const char* filename = "assets/froggerHighwayLaneChangeV4.mp4";

//defining format of data sent to threads
struct thread_data{
  //include int for data passing
  int data;
};
```

```
//function prototypes
Mat slidingWindowNeighborDetector(Mat srcFrame, int numRowSections, int numColumnSections);
Mat cannyContourDetector(Mat srcFrame);
void fillCoordinates(vector <Point2f> detectedCoordinates);

//method to display frame
void displayFrame(string filename, Mat matToDisplay)
{
    //if in debug mode and Mat is not empty
    if(debug && matToDisplay.size[0] != 0)
    {imshow(filename, matToDisplay);}

    else if(matToDisplay.size[0] == 0)
    {
        cout << filename << " is empty, cannot be displayed." << endl;
    }
}

//method to display frame overriding debug
void displayFrame(string filename, Mat matToDisplay, bool override)
{
    //if override and Mat is not emptys
    if(override && matToDisplay.size[0] != 0 && filename != "Welcome"){ imshow(filename, matToDisplay);}
    else if(override && matToDisplay.size[0] != 0){namedWindow(filename); imshow(filename, matToDisplay);}
    else if(matToDisplay.size[0] == 0)
    {
        cout << filename << " is empty, cannot be displayed." << endl;
    }
}

//method to draw optical flow, only should be called during demos
static void drawOptFlowMap(const Mat& flow, Mat& cflowmap,
                double, const Scalar& color)
{
    //iterating through each pixel and drawing vector
   for(int y = 0; y < cflowmap.rows; y += opticalFlowDensityDisplay)
   {
    for(int x = 0; x < cflowmap.cols; x += opticalFlowDensityDisplay)
      {
         const Point2f& fxy = flow.at<Point2f>(y, x);
         line(cflowmap, Point(x,y), Point(cvRound(x+fxy.x), cvRound(y+fxy.y)),
              color);
         circle(cflowmap, Point(x,y), 0, color, -1);
      }
    }
   //display optical flow map
   displayFrame("RFDOFA", cflowmap);
   imshow("RFDOFA", cflowmap);
}

//method that returns date and time as a string to tag txt files
const string currentDateTime()
{
    //creating time object that reads current time
   time_t now = time(0);
```

```
    //creating time structure
    struct tm tstruct;

    //creating a character buffer of 80 characters
    char buf[80];

    //checking current local time
    tstruct = *localtime(&now);

    //writing time to string
    strftime(buf, sizeof(buf), "%Y-%m-%d.%X", &tstruct);

    fileTime = buf;

    //returning the string with the time
    return buf;
}

//method to apply morphology
Mat morph(Mat sourceFrame, int amplitude, string type)
{
     //using default values
     double morph_size = .5;

     //performing two iterations
     const int iterations = 2;

     //constructing manipulation Mat
     Mat element = getStructuringElement(MORPH_RECT, Size( 2*morph_size + 1, 2*morph_size+1 ), Point( morph_size,
morph_size ) );

     //if performing morphological closing
     if(type == "closing")
     {
          //repeat for increased effect
        for(int v = 0; v < amplitude; v++)
          {
               morphologyEx(sourceFrame, sourceFrame, MORPH_CLOSE, element,
                         Point(-1,-1), iterations, BORDER_CONSTANT, morphologyDefaultBorderValue());
          }
     }

     //if performing morphological opening
     else if(type == "opening")
     {
          for(int v = 0; v < amplitude; v++)
          {
               //repeat for increased effect
               morphologyEx(sourceFrame, sourceFrame, MORPH_OPEN, element,
                         Point(-1,-1), iterations, BORDER_CONSTANT, morphologyDefaultBorderValue());

          }
     }

     else if(type == "erode")
     {
        erode(sourceFrame, sourceFrame,  element);
    }

    //if performing morphological gradient
    else if(type == "gradient")
    {
        //repeat for increased effect
        for(int v = 0; v < amplitude; v++)
        {
            morphologyEx(sourceFrame, sourceFrame, MORPH_GRADIENT, element,
                    Point(-1,-1), iterations, BORDER_CONSTANT, morphologyDefaultBorderValue());
        }
    }

    //if performing morphological tophat
    else if(type == "tophat")
    {
        //repeat for increased effect
        for(int v = 0; v < amplitude; v++)
        {
            morphologyEx(sourceFrame, sourceFrame, MORPH_TOPHAT, element,
                    Point(-1,-1), iterations, BORDER_CONSTANT, morphologyDefaultBorderValue());
        }
    }

    //if performing morphological blackhat
    else if(type == "blackhat")
    {
        //repeat for increased effect
        for(int v = 0; v < amplitude; v++)
        {
        morphologyEx(sourceFrame, sourceFrame, MORPH_BLACKHAT, element,
                        Point(-1,-1), iterations, BORDER_CONSTANT, morphologyDefaultBorderValue());
        }
    }

    //if current morph operation is not availble
    else
    {
        //report cannot be done
        if(debug)
            cout << type <<  " type of morphology not implemented yet" << endl;
    }

    //return edited frame
   return sourceFrame;
}

//method to calculate center point of contour
vector <int> centerPoint(vector <Point> contours)
{
    //initializing coordinate variables
    int xTotal = 0;
    int yTotal = 0;

    //vector to hold center point
    vector <int> centerPoint;

    //iterating through each contour
    for(int v = 0 ; v < contours.size() ; v++)
    {
        Point p = contours.at(v);

        xTotal += p.x;
        yTotal += p.y;
    }

    //averaging points
    centerPoint.push_back(xTotal / (contours.size()/2));
    centerPoint.push_back(yTotal / (contours.size()/2));

    //return both center points
    return centerPoint;
}

//method to blur Mat using custom kernel size
Mat blurFrame(string blurType, Mat sourceDiffFrame, int blurSize)
{
    //Mat to hold blurred frame
    Mat blurredFrame;

    //if gaussian blur
    if(blurType == "gaussian")
    {
        //blur frame using custom kernel size
        blur(sourceDiffFrame, blurredFrame, Size (blurSize,blurSize), Point(-1,-1));

        //display blurred frame
        //displayFrame("Gauss Frame", blurredFrame);

        //return blurred frame
        return blurredFrame;
    }

    //if blur type not implemented
    else
    {
        //report not implemented
        if(debug)
            cout << blurType <<  " type of blur not implemented yet" << endl;

        //return original frame
        return sourceDiffFrame;
    }

}

//method to perform OFA threshold on Mat
void *computeOpticalFlowAnalysisObjectDetection(void *threadarg)
{
    //reading in data sent to thread into local variable
    struct opticalFlowThreadData *data;
    data = (struct opticalFlowThreadData *) threadarg;

    Mat ofaObjectDetection;

    //deep copy grayscale frame
    globalGrayFrames.at(i-1).copyTo(ofaObjectDetection);

    //set threshold
    const double threshold = 10000;

    //iterating through OFA pixels
    for(int j = 0; j < cflow.rows; j++)
    {
        for (int a = 0 ; a < cflow.cols; a++)
        {
            const Point2f& fxy = flow.at<Point2f>(j, a);

            //if movement is greater than threshold
            if((sqrt((abs(fxy.x) * abs(fxy.y))) * 10000) > threshold)
            {
                //write to binary image
                ofaObjectDetection.at<uchar>(j,a) = 255;
            }
            else
            {
                //write to binary image
                ofaObjectDetection.at<uchar>(j,a) = 0;
            }
        }
    }

    //performing sWND
    displayFrame("OFAOBJ pre" , ofaObjectDetection );
    ofaObjectDetection = slidingWindowNeighborDetector(ofaObjectDetection, ofaObjectDetection.rows / 10,
ofaObjectDetection.cols / 20);
    displayFrame("sWNDFrame1" , ofaObjectDetection );
    ofaObjectDetection = slidingWindowNeighborDetector(ofaObjectDetection, ofaObjectDetection.rows / 20,
ofaObjectDetection.cols / 40);
    displayFrame("sWNDFrame2" , ofaObjectDetection );
    ofaObjectDetection = slidingWindowNeighborDetector(ofaObjectDetection, ofaObjectDetection.rows / 30,
ofaObjectDetection.cols / 60);
    displayFrame("sWNDFrame3" , ofaObjectDetection);

    ofaObjectDetection.copyTo(ofaGlobalHeatMap);

    thresholdFrameOFA = cannyContourDetector(ofaObjectDetection);
    displayFrame("sWNDFrameCanny" , thresholdFrameOFA);

    //signal thread completion
    opticalFlowAnalysisObjectDetectionThreadCompletion = 1;
}

//method to handle OFA threshold on Mat thread
void opticalFlowAnalysisObjectDetection(Mat& cflowmap, Mat& flow)
{
    //instantiating multithread object
    pthread_t opticalFlowAnalysisObjectDetectionThread;

    //instantiating multithread Data object
    struct thread_data threadData;

    //saving data to pass
    threadData.data = i;

    //creating optical flow object thread
    pthread_create(&opticalFlowAnalysisObjectDetectionThread, NULL, computeOpticalFlowAnalysisObjectDetection,
(void *)&threadData);

}

//method to perform optical flow analysis
void *computeOpticalFlowAnalysisThread(void *threadarg)
{
    //reading in data sent to thread into local variable
    struct thread_data *data;
    data = (struct thread_data *) threadarg;
    int temp = data->data;

    //defining local variables for FDOFA
    Mat prevFrame, currFrame;
    Mat gray, prevGray;

    if(i > 5)
    {
        prevFrame = globalFrames[i-1];
        currFrame = globalFrames[i];
    }

    else
    {
        prevFrame = globalFrames[i];
        currFrame = globalFrames[i];
    }

    displayFrame("Pre blur", currFrame);
    currFrame = blurFrame("gaussian" , currFrame, 15);
    displayFrame("Post blur", currFrame);
    prevFrame = blurFrame("gaussian", prevFrame, 15);

    //converting to grayscale
    cvtColor(currFrame, gray,COLOR_BGR2GRAY);
    cvtColor(prevFrame, prevGray, COLOR_BGR2GRAY);

    //calculating optical flow
    calcOpticalFlowFarneback(prevGray, gray, flow, 0.5, 3, 15, 3, 5, 1.2, 0);

    //converting to display format
    cvtColor(prevGray, cflow, COLOR_GRAY2BGR);

    //perform OFA threshold
   opticalFlowAnalysisObjectDetection(flow, cflow);
   //draw optical flow map
    if(debug)
    {
        //drawing optical flow vectors
        drawOptFlowMap(flow, cflow, 1.5, Scalar(0, 0, 255));
    }

    //wait for completion
    while(opticalFlowAnalysisObjectDetectionThreadCompletion != 1){}

    //wait for completion
    opticalFlowAnalysisObjectDetectionThreadCompletion = 0;

    opticalFlowThreadCompletion = 1;
}


//method to do background subtraction with MOG2
Mat bgMog2(bool buffer)
{

    //instantiating Mat objects
    Mat fgmaskShadow;
    Mat frameToResizeShadow;

    //copying into tmp variable
    globalFrames[i].copyTo(frameToResizeShadow);

    //performing background subtraction
    pMOG2Shadow->operator()(frameToResizeShadow , fgmaskShadow, .01);

    //performing sWND
    displayFrame("fgmaskShadow" , fgmaskShadow);
    Mat fgmaskShadowSWND = slidingWindowNeighborDetector(fgmaskShadow, fgmaskShadow.rows/10,
fgmaskShadow.cols/20);
    displayFrame("fgmaskShadowSWND", fgmaskShadowSWND);

    fgmaskShadowSWND = slidingWindowNeighborDetector(fgmaskShadowSWND, fgmaskShadowSWND.rows/20,
fgmaskShadowSWND.cols/40);
    displayFrame("fgmaskShadowSWND2", fgmaskShadowSWND);

    //performing canny
    Mat fgMaskShadowSWNDCanny = cannyContourDetector(fgmaskShadowSWND);
    displayFrame("fgMaskShadowSWNDCanny2", fgMaskShadowSWNDCanny);

    //returning processed frame
    return fgMaskShadowSWNDCanny;
}

//method to perform vibe background subtraction
Mat vibeBackgroundSubtraction(bool buffer)
{
    //instantiating Mat frame object
    Mat sWNDVibeCanny;

    //if done buffering
    if(i == bufferMemory)
    {
        //instantiating Mat frame object
        Mat resizedFrame;

        //saving current frame
        globalFrames[i].copyTo(resizedFrame);

        //initializing model
        bgfg.init_model(resizedFrame);

        //return tmp frame
        return resizedFrame;
    }

    else
    {

        //instantiating Mat frame object
        Mat resizedFrame;

        //saving current frame
        globalFrames[i].copyTo(resizedFrame);

        //processing model
        vibeBckFrame = *bgfg.fg(resizedFrame);

        displayFrame("vibeBckFrame", vibeBckFrame);

        //performing sWND
        Mat sWNDVibe = slidingWindowNeighborDetector(vibeBckFrame, vibeBckFrame.rows/10, vibeBckFrame.cols/20);
        displayFrame("sWNDVibe1", sWNDVibe);

        //performing sWND
        sWNDVibe = slidingWindowNeighborDetector(vibeBckFrame, vibeBckFrame.rows/20, vibeBckFrame.cols/40);
        displayFrame("sWNDVibe2", sWNDVibe);

        //performing canny
        Mat sWNDVibeCanny = cannyContourDetector(sWNDVibe);
        displayFrame("sWNDVibeCannycanny2", sWNDVibeCanny);
    }

    //returning processed frame
    return sWNDVibeCanny;
}


//method to do background subtraction with MOG 1
Mat bgMog(bool buffer)
{

    //instantiating Mat objects
    Mat fgmask;
    Mat bck;
    Mat fgMaskSWNDCanny;
    Mat fgmaskSWND;
    //performing background subtraction
   bckSubMOG.operator()(globalFrames.at(i), fgmask, .01); //1.0 / 200);

    if(!buffer)
    {
        displayFrame("MOG Fg MAsk", fgmask);
        //displayFrame("RCFrame", globalFrames[i]);

        cout << " df" << endl;
        //performing sWND
        fgmaskSWND = slidingWindowNeighborDetector(fgmask, fgmask.rows/10, fgmask.cols/20);
        //displayFrame("fgmaskSWND", fgmaskSWND);
            cout << "here" << endl;

        fgmaskSWND = slidingWindowNeighborDetector(fgmaskSWND, fgmaskSWND.rows/20, fgmaskSWND.cols/40);
        //displayFrame("fgmaskSWNDSWND2", fgmaskSWND);

        fgmaskSWND = slidingWindowNeighborDetector(fgmaskSWND, fgmaskSWND.rows/30, fgmaskSWND.cols/60);
        //displayFrame("fgmaskSWNDSWND3", fgmaskSWND);

        //performing canny
        fgMaskSWNDCanny = cannyContourDetector(fgmaskSWND);
        //displayFrame("fgMaskSWNDCanny2", fgMaskSWNDCanny);
    }



    //return canny
    return fgMaskSWNDCanny;
}


//method to handle OFA thread
Mat opticalFlowFarneback()
{
    //cout << "ENTERING OFF" << endl;
    //instantiate thread object
    pthread_t opticalFlowFarneback;

    //instantiating multithread Data object
    struct thread_data threadData;

    //saving data to pass
    threadData.data = i;

    //create OFA thread
    pthread_create(&opticalFlowFarneback, NULL, computeOpticalFlowAnalysisThread, (void *)&threadData);

    while(opticalFlowThreadCompletion != 1){}

    opticalFlowThreadCompletion = 0;

    return thresholdFrameOFA;
}
```

```
//write initial statistics about the video
void writeInitialStats(int NUMBER_OF_FRAMES, int FRAME_RATE, int FRAME_WIDTH, int FRAME_HEIGHT, const
char* filename)
{
    ////writing stats to txt file
    //initiating write stream
    ofstream writeToFile;

    //creating filename  ending
    string filenameAppend = "Stats.txt";

    //concanating and creating file name string
    string strFilename = filename + currentDateTime() + filenameAppend;

    //open file stream and begin writing file
    writeToFile.open (strFilename);

    //write video statistics
    writeToFile << "Stats on video >> There are = " << NUMBER_OF_FRAMES << " frames. The frame rate is " <<
FRAME_RATE
    << " frames per second. Resolution is " << FRAME_WIDTH << " X " << FRAME_HEIGHT;

    //close file stream
    writeToFile.close();

    if(debug)
    {
        //display video statistics
        cout << "Stats on video >> There are = " << NUMBER_OF_FRAMES << " frames. The frame rate is " <<
FRAME_RATE
                << " frames per second. Resolution is " << FRAME_WIDTH << " X " << FRAME_HEIGHT << endl;;
    }
}

//display welcome message and splash screen
void welcome()
{
    if(i < bufferMemory * 2)
    {
        Mat img = imread("assets/TCD3.png");
        putText(img, "Initializing; V. Prasad 2015 All Rights Reserved"
            , cvPoint(0,30),CV_FONT_HERSHEY_SIMPLEX, 1, cvScalar(255,255,0), 1, CV_AA, false);

        //display welcome images
        displayFrame("Welcome", img, true);
    }

    else
    {
        //close welcome image
        destroyWindow("Welcome");
    }
}

//display welcome message and splash screen
void welcome(String text)
```

```
{
    Mat img = imread("assets/TCD3.png");
    putText(img, text
        , cvPoint(0,30),CV_FONT_HERSHEY_SIMPLEX, .8, cvScalar(255,255,0), 1, CV_AA, false);

    //display welcome images
    displayFrame("Welcome", img, true);
}

//calculate time for each iteration
double calculateFPS(clock_t tStart, clock_t tFinal)
{
    //return frames per second
    return 1/((((float)tFinal-(float)tStart) / CLOCKS_PER_SEC));
}

//method to calculate runtime
void computeRunTime(clock_t t1, clock_t t2, int framesRead)
{
    //subtract from start time
    float diff ((float)t2-(float)t1);

    //calculate frames per second
    double frameRateProcessing = (framesRead / diff) * CLOCKS_PER_SEC;

    //display amount of time for run time
    cout << (diff / CLOCKS_PER_SEC) << " seconds of run time." << endl;

    //display number of frames processed per second
    cout << frameRateProcessing << " frames processed per second." << endl;
    cout << framesRead << " frames read." << endl;
}

//method to calculate median of vector of integers
double calcMedian(vector<int> integers)
{
    //double to store non-int median
    double median;

    //read size of vector
    size_t size = integers.size();

    //sort array
   sort(integers.begin(), integers.end());

   //if even number of elements
    if (size % 2 == 0)
    {
        //median is middle elements averaged
        median = (integers[size / 2 - 1] + integers[size / 2]) / 2;
    }

    //if odd number of elements
    else
    {
        //median is middle element
        median = integers[size / 2];
    }

    //return the median value
    return median;
}

//method to calculate mean of vector of integers
double calcMean(vector <int> integers)
{
    //total of all elements
    int total = 0;

    //step through vector
    for (int v = 0; v < integers.size(); v++)
    {
        //total all values
        total += integers.at(v);
    }

    //return mean value
    return total/integers.size();
}

//method to identify type of Mat based on identifier
string type2str(int type) {

    //string to return type of mat
    string r;

    //stats about frame
    uchar depth = type & CV_MAT_DEPTH_MASK;
    uchar chans = 1 + (type >> CV_CN_SHIFT);

    //switch to determine Mat type
    switch ( depth ) {
        case CV_8U:  r = "8U"; break;
        case CV_8S:  r = "8S"; break;
        case CV_16U: r = "16U"; break;
        case CV_16S: r = "16S"; break;
        case CV_32S: r = "32S"; break;
        case CV_32F: r = "32F"; break;
        case CV_64F: r = "64F"; break;
        default:     r = "User"; break;
   }

    //append formatting
    r += "C";
    r += (chans+'0');

    //return Mat type
    return r;
}

//thread to calculate median of image
void *calcMedianImage(void *threadarg)
```

```
{
    //defining data structure to read in info to new thread
    struct thread_data *data;
    data = (struct thread_data *) threadarg;

    //performing deep copy
    globalGrayFrames[i].copyTo(backgroundFrameMedian);

    //variables to display completion
    double displayPercentageCounter = 0;
    double activeCounter = 0;

    //calculating number of runs
    for(int j=0;j<backgroundFrameMedian.rows;j++)
    {
        for (int a=0;a<backgroundFrameMedian.cols;a++)
        {
            for (int t = (i - bufferMemory); t < i ; t++)
            {
                displayPercentageCounter++;
            }
        }
    }

    //stepping through all pixels
    for(int j=0;j<backgroundFrameMedian.rows;j++)
    {
        for (int a=0;a<backgroundFrameMedian.cols;a++)
        {
            //saving all pixel values
            vector <int> pixelHistory;

            //moving through all frames stored in buffer
            for (int t = (i - bufferMemory); t < i ; t++)
            {
                //Mat to store current frame to process
                Mat currentFrameForMedianBackground;

                //copy current frame
                globalGrayFrames.at(i-t).copyTo(currentFrameForMedianBackground);

                //save pixel into pixel history
                pixelHistory.push_back(currentFrameForMedianBackground.at<uchar>(j,a));

                //increment for load calculations
                activeCounter++;
            }

            //calculate median value and store in background image
            backgroundFrameMedian.at<uchar>(j,a) = calcMedian(pixelHistory);
      }

      //display percentage completed
      cout << ((activeCounter / displayPercentageCounter) * 100) << "% Median Image Scanned" << endl;

    }
    backgroundFrameMedian.copyTo(finalTrackingFrame);
    backgroundFrameMedian.copyTo(drawAnomalyCar);
    backgroundFrameMedian.copyTo(backgroundFrameMedianColor);

    //signal thread completion
   medianImageCompletion = 1;
}

//calculate max value in frame for debug
int maxMat(Mat sourceFrame)
{
    //variable for current max
    int currMax = INT_MIN;

    //step through pixels
    for(int j=0;j<sourceFrame.rows;j++)
    {
       for (int a=0;a<sourceFrame.cols;a++)
       {
        //if current value is larger than previous max
        if(sourceFrame.at<uchar>(j,a) > currMax)
        {
            //store current value as new max
            currMax = sourceFrame.at<uchar>(j,a);
        }
       }
    }

    //return max value in matrix
    return currMax;
}

//method to threshold standard frame
Mat thresholdFrame(Mat sourceDiffFrame, const int threshold)
{
    //Mat to hold frame
    Mat thresholdFrame;

    //perform deep copy into destination Mat
    sourceDiffFrame.copyTo(thresholdFrame);

    //steping through pixels
    for(int j=0;j<sourceDiffFrame.rows;j++)
    {
       for (int a=0;a<sourceDiffFrame.cols;a++)
       {
        //if pixel value greater than threshold
        if(sourceDiffFrame.at<uchar>(j,a) > threshold)
        {
            //write to binary image
            thresholdFrame.at<uchar>(j,a) = 255;
        }
        else
        {
            //write to binary image
```

```
                thresholdFrame.at<uchar>(j,a) = 0;
            }
          }
      }

      //perform morphology
      //thresholdFrame = morph(thresholdFrame, 1, "closing");

      //return thresholded frame
      return thresholdFrame;
}

//method to perform simple image subtraction
Mat imageSubtraction()
{
      //subtract frames
      Mat tmpStore  =  globalGrayFrames[i] - backgroundFrameMedian;

      displayFrame("Raw imgSub", tmpStore);
      //threshold frames
      tmpStore = thresholdFrame(tmpStore, 50);
      displayFrame("Thresh imgSub", tmpStore);

      //perform sWND
      tmpStore =  slidingWindowNeighborDetector(tmpStore, tmpStore.rows / 5, tmpStore.cols / 10);
      displayFrame("SWD",tmpStore);
      tmpStore = slidingWindowNeighborDetector(tmpStore, tmpStore.rows / 10, tmpStore.cols / 20);
      displayFrame("SWD2", tmpStore);
      tmpStore = slidingWindowNeighborDetector(tmpStore, tmpStore.rows / 20, tmpStore.cols / 40);
      displayFrame("SWD3", tmpStore);

      //perform canny
      tmpStore = cannyContourDetector(tmpStore);
      displayFrame("Canny Contour", tmpStore);

      //return frame
      return tmpStore;
}

//method to perform median on grayscale images
void grayScaleFrameMedian()
{
      if(debug)
            cout << "Entered gray scale median" << endl;

      //instantiating multithread object
      pthread_t medianImageThread;

      //instantiating multithread Data object
      struct thread_data threadData;

      //saving data into multithread
      threadData.data = i;

      //creating thread to calculate median of image
      pthread_create(&medianImageThread, NULL, calcMedianImage, (void *)&threadData);
```

```
    //save median image
    imwrite((currentDateTime() + "medianBackgroundImage.jpg"), backgroundFrameMedian);
}

//method to calculate Gaussian image difference
void *calcGaussianMixtureModel(void *threadarg)
{
    //perform deep copy
    globalFrames[i].copyTo(gmmFrameRaw);

    //update model
    (*backgroundSubtractorGMM)(gmmFrameRaw, binaryGMMFrame);

    //save into tmp frame
    gmmFrameRaw.copyTo(gmmTempSegmentFrame);

    //add movement mask
    add(gmmFrameRaw, Scalar(0, 255, 0), gmmTempSegmentFrame, binaryGMMFrame);

    //save into display file
    gmmFrame = gmmTempSegmentFrame;

    //display frame
    displayFrame("GMM Frame", gmmFrame);

    //save mask as main gmmFrame
    gmmFrame = binaryGMMFrame;

    displayFrame("GMM Binary Frame" , binaryGMMFrame);

    //if buffer built
    if(i > bufferMemory * 2)
    {
        //perform sWND
        gmmFrame = slidingWindowNeighborDetector(binaryGMMFrame, gmmFrame.rows / 5, gmmFrame.cols / 10);
        displayFrame("sWDNs GMM Frame 1", gmmFrame);

        gmmFrame = slidingWindowNeighborDetector(gmmFrame, gmmFrame.rows / 10, gmmFrame.cols / 20);
        displayFrame("sWDNs GMM Frame 2", gmmFrame);

        gmmFrame = slidingWindowNeighborDetector(gmmFrame, gmmFrame.rows / 20, gmmFrame.cols / 40);
        displayFrame("sWDNs GMM Frame 3", gmmFrame);

        Mat gmmFrameSWNDCanny = gmmFrame;

        if(i > bufferMemory * 3 -1)
        {
            //perform Canny
            gmmFrameSWNDCanny = cannyContourDetector(gmmFrame);
            displayFrame("CannyGMM", gmmFrameSWNDCanny);
        }

        //save into canny
        cannyGMM = gmmFrameSWNDCanny;
    }
    //signal thread completion
    gaussianMixtureModelCompletion = 1;
}

//method to handle GMM thread
Mat gaussianMixtureModel()
{

    //instantiate thread object
    pthread_t gaussianMixtureModelThread;

    //instantiating multithread Data object
    struct thread_data threadData;

    //save i data
    threadData.data = i;

    //create thread
    pthread_create(&gaussianMixtureModelThread, NULL, calcGaussianMixtureModel, (void *)&threadData);

    //return processed frame if completed
    if(i > bufferMemory * 2)
        return cannyGMM;
    //return tmp frame if not finished
    else
        return globalFrames[i];
}


//method to handle all background image generation
void generateBackgroundImage(int FRAME_RATE)
{
    //if post-processing
    if(readMedianImg && useMedians && i < bufferMemory + 5)
    {
        //read median image
        backgroundFrameMedian = imread("assets/froggerHighwayDrunkMedian.jpg");

        backgroundFrameMedian.copyTo(drawAnomalyCar);
        backgroundFrameMedian.copyTo(backgroundFrameMedianColor);

        //convert to grayscale
        cvtColor(backgroundFrameMedian, backgroundFrameMedian, CV_BGR2GRAY);

        displayFrame("backgroundFrameMedian", backgroundFrameMedian);

        backgroundFrameMedian.copyTo(finalTrackingFrame);
    }

    //if real-time calculation
    else
    {
        //after initial buffer read and using medians
        if(i == bufferMemory && useMedians)
        {
            grayScaleFrameMedian();

            while(medianImageCompletion != 1) {}
        }
        //every 3 minutes
        if (i % (FRAME_RATE * 180) == 0 && i > 0)
        {
            //calculate new medians
            grayScaleFrameMedian();

            while(medianImageCompletion != 1) {}

        }
    }


    medianImageCompletion = 0;
}

//method to draw canny contours
Mat cannyContourDetector(Mat srcFrame)
{
    //threshold for non-car objects or noise
    const int thresholdNoiseSize = 200;
    const int misDetectLargeSize = 600;

    //instantiating Mat and Canny objects
    Mat canny;
    Mat cannyFrame;
    vector<Vec4i> hierarchy;
    typedef vector<vector<Point> > TContours;
    TContours contours;

    //run canny edge detector
    Canny(srcFrame , cannyFrame, 300, 900, 3);
    findContours(cannyFrame, contours, hierarchy, CV_RETR_CCOMP, CV_CHAIN_APPROX_NONE);

    //creating blank frame to draw on
    Mat drawing = Mat::zeros( cannyFrame.size(), CV_8UC3 );

    //moments for center of mass
   vector<Moments> mu(contours.size() );
   for( int i = 0; i < contours.size(); i++ )
     { mu[i] = moments( contours[i], false ); }

    //get mass centers:
   vector<Point2f> mc( contours.size() );
   for( int i = 0; i < contours.size(); i++ )
     { mc[i] = Point2f( mu[i].m10/mu[i].m00 , mu[i].m01/mu[i].m00 ); }

    //for each detected contour
    for(int v = 0; v < contours.size(); v++)
    {
        //if large enough to be object
        if(arcLength(contours[v], true) > thresholdNoiseSize && arcLength(contours[v], true)  < misDetectLargeSize)
        {
            //draw object and circle center point
            drawContours( drawing, contours, v, Scalar(254,254,0), 2, 8, hierarchy, 0, Point() );
            circle( drawing, mc[v], 4, Scalar(254, 254, 0), -1, 8, 0 );
            fillCoordinates(mc);
        }
    }

    //return image with contours
    return drawing;
}

Mat slidingWindowNeighborPointDetector (Mat sourceFrame, int numRowSections, int numColumnSections, vector
<Point> coordinates)
{
    //if using default num rows
        if(numRowSections == -1 || numColumnSections == -1)
        {
            //split into standard size
            numRowSections = sourceFrame.rows / 10;
            numColumnSections = sourceFrame.cols / 20;
        }

        /*

        double numRowSectionsDouble = numRowSections;
        double numColumnSectionsDouble = numColumnSections;

        while(sourceFrame.rows % numRowSections != 0)
        {
            numRowSections++;
            numRowSectionsDouble = numRowSections;
        }

        while(sourceFrame.cols % numColumnSections != 0)
        {
            numColumnSections++;
            numColumnSectionsDouble = numColumnSections;
        }

        */

        //declaring percentage to calculate density
        double percentage = 0;

        //setting size of search area
        int windowWidth = sourceFrame.rows / numRowSections;
        int windowHeight = sourceFrame.cols / numColumnSections;

        //creating destination frame of correct size
        Mat destinationFrame = Mat(sourceFrame.rows, sourceFrame.cols, CV_8UC1);

        //cycling through pieces
        for(int v = windowWidth/2; v <= sourceFrame.rows - windowWidth/2; v++)
        {
            for(int j = windowHeight/2; j <= sourceFrame.cols - windowHeight/2; j++)
            {
```

```
/*
//variables to calculate density
double totalCounter = 0;
double detectCounter = 0;
*/

int pointsCounter = 0;

//cycling through neighbors
for(int x =  v - windowWidth/2; x < v + windowWidth/2; x++)
{
    for(int k = j - windowHeight/2; k < j + windowHeight/2; k++)
    {
        /*
        int z = 0;
        int pointsCounter = 0;
        while(pointsCounter <= 1 && z < coordinates.size())
        {
            for(int b = 0; b < coordinates.size())
        }
        */

        Point currentPoint(x,k);

        for(int z = 0; z< coordinates.size(); z++)
        {
            if(coordinates[z] == currentPoint)
            {
                pointsCounter++;
            }
        }

        /*
        //if object exists
        if(sourceFrame.at<uchar>(x,k) > 127)
        {
            //add to detect counter
            detectCounter++;
        }

        //count pixels searched
        totalCounter++;
        */
    }
}

/*
//prevent divide by 0 if glitch and calculate percentage
if(totalCounter != 0)
    percentage = detectCounter / totalCounter;
else
    cout << "Ted Cruz" << endl;
*/

//if object exists flag it
if(pointsCounter >  1)
```

```
                {
                    destinationFrame.at<uchar>(v,j) = 255;
                }

                //else set it to 0
                else
                {
                    //sourceFrame.at<uchar>(v,j) = 0;
                    destinationFrame.at<uchar>(v,j) = 0;
                }
            }
        }

        //return processed frame
        return destinationFrame;
}

//method to perform proximity density search to remove noise and identify noise
Mat slidingWindowNeighborDetector(Mat sourceFrame, int numRowSections, int numColumnSections)
{
    //if using default num rows
    if(numRowSections == -1 || numColumnSections == -1)
    {
        //split into standard size
        numRowSections = sourceFrame.rows / 10;
        numColumnSections = sourceFrame.cols / 20;
    }

    /*

    double numRowSectionsDouble = numRowSections;
    double numColumnSectionsDouble = numColumnSections;

    while(sourceFrame.rows % numRowSections != 0)
    {
        numRowSections++;
        numRowSectionsDouble = numRowSections;
    }

    while(sourceFrame.cols % numColumnSections != 0)
    {
        numColumnSections++;
        numColumnSectionsDouble = numColumnSections;
    }

    */

    //declaring percentage to calculate density
    double percentage = 0;

    //setting size of search area
    int windowWidth = sourceFrame.rows / numRowSections;
    int windowHeight = sourceFrame.cols / numColumnSections;

    //creating destination frame of correct size
    Mat destinationFrame = Mat(sourceFrame.rows, sourceFrame.cols, CV_8UC1);
    //cycling through pieces
    for(int v = windowWidth/2; v <= sourceFrame.rows - windowWidth/2; v++)
    {
        for(int j = windowHeight/2; j <= sourceFrame.cols - windowHeight/2; j++)
        {
            //variables to calculate density
            double totalCounter = 0;
            double detectCounter = 0;

            //cycling through neighbors
            for(int x =  v - windowWidth/2; x < v + windowWidth/2; x++)
            {
                for(int k = j - windowHeight/2; k < j + windowHeight/2; k++)
                {
                    //if object exists
                    if(sourceFrame.at<uchar>(x,k) > 127)
                    {
                        //add to detect counter
                        detectCounter++;
                    }

                    //count pixels searched
                    totalCounter++;
                }
            }

            //prevent divide by 0 if glitch and calculate percentage
            if(totalCounter != 0)
                percentage = detectCounter / totalCounter;
            else
                cout << "Ted Cruz" << endl;

            //if object exists flag it
            if(percentage >  .25)
            {
                destinationFrame.at<uchar>(v,j) = 255;
            }

            //else set it to 0
            else
            {
                //sourceFrame.at<uchar>(v,j) = 0;
                destinationFrame.at<uchar>(v,j) = 0;
            }
        }
    }

    //return processed frame
    return destinationFrame;
}


//method to handle median image subtraction
Mat medianImageSubtraction(int FRAME_RATE)
{
    //generate or read background image
    generateBackgroundImage(FRAME_RATE);

    //calculate image difference and return
    return imageSubtraction();
}

//method to perform vibe background subtraction
void *computeVibeBackgroundThread(void *threadarg)
{
    struct thread_data *data;
    data = (struct thread_data *) threadarg;

    //instantiating Mat frame object
    Mat sWNDVibeCanny;

    //if done buffering
    if(i == bufferMemory)
    {
        //instantiating Mat frame object
        Mat resizedFrame;

        //saving current frame
        globalFrames[i].copyTo(resizedFrame);

        //initializing model
        bgfg.init_model(resizedFrame);

        //return tmp frame
        vibeDetectionGlobalFrame = sWNDVibeCanny;

        vibeDetectionGlobalFrameCompletion = 1;
    }

    else
    {
        //instantiating Mat frame object
        Mat resizedFrame;

        //saving current frame
        globalFrames[i].copyTo(resizedFrame);

        //processing model
        vibeBckFrame = *bgfg.fg(resizedFrame);

        displayFrame("vibeBckFrame", vibeBckFrame);

        //performing sWND
        Mat sWNDVibe = slidingWindowNeighborDetector(vibeBckFrame, vibeBckFrame.rows/10, vibeBckFrame.cols/20);
        displayFrame("sWNDVibe1", sWNDVibe);

        //performing sWND
        sWNDVibe = slidingWindowNeighborDetector(vibeBckFrame, vibeBckFrame.rows/20, vibeBckFrame.cols/40);
        displayFrame("sWNDVibe2", sWNDVibe);

        Mat sWNDVibeCanny = sWNDVibe;
        if(i > bufferMemory * 3 -1)
        {
            //performing canny
            Mat sWNDVibeCanny = cannyContourDetector(sWNDVibe);
            displayFrame("sWNDVibeCannycanny2", sWNDVibeCanny);
        }

    //saving processed frame
    vibeDetectionGlobalFrame = sWNDVibeCanny;

    //signalling completion
    vibeDetectionGlobalFrameCompletion = 1;
    }
}

void vibeBackgroundSubtractionThreadHandler(bool buffer)
{
    //instantiating multithread object
    pthread_t vibeBackgroundSubtractionThread;

    //instantiating multithread Data object
    struct thread_data threadData;

    //saving data into data object
    threadData.data = i;

    //creating threads
    int vibeBackgroundThreadRC = pthread_create(&vibeBackgroundSubtractionThread, NULL,
computeVibeBackgroundThread, (void *)&threadData);
}

//method to do background subtraction with MOG 1
void *computeBgMog1(void *threadarg)
{
    struct thread_data *data;
    data = (struct thread_data *) threadarg;

    //instantiating Mat objects
    Mat fgmask;
    Mat bck;
    Mat fgMaskSWNDCanny;

    //performing background subtraction
   bckSubMOG.operator()(globalFrames.at(i), fgmask, .01); //1.0 / 200);

    displayFrame("MOG Fg MAsk", fgmask);
    displayFrame("RCFrame", globalFrames[i]);

    //performing sWND
    Mat fgmaskSWND = slidingWindowNeighborDetector(fgmask, fgmask.rows/10, fgmask.cols/20);
    displayFrame("fgmaskSWND", fgmaskSWND);

    fgmaskSWND = slidingWindowNeighborDetector(fgmaskSWND, fgmaskSWND.rows/20, fgmaskSWND.cols/40);
    displayFrame("fgmaskSWNDSWND2", fgmaskSWND);
    fgmaskSWND = slidingWindowNeighborDetector(fgmaskSWND, fgmaskSWND.rows/30, fgmaskSWND.cols/60);
    displayFrame("fgmaskSWNDSWND3", fgmaskSWND);

    //performing canny
    fgMaskSWNDCanny = cannyContourDetector(fgmaskSWND);
    displayFrame("fgMaskSWNDCanny2", fgMaskSWNDCanny);

    //return canny
    mogDetection1GlobalFrame = fgMaskSWNDCanny;

    //signal completion
    mogDetection1GlobalFrameCompletion = 1;
}


void mogDetectionThreadHandler(bool buffer)
{
    //instantiating multithread object
    pthread_t mogDetectionThread;

    //instantiating multithread Data object
    struct thread_data threadData;

    //saving data into data object
    threadData.data = i;

    //creating threads
    int mogDetectionThreadRC = pthread_create(&mogDetectionThread, NULL, computeBgMog1, (void *)&threadData);
}

//method to do background subtraction with MOG 1
void *computeBgMog2(void *threadarg)
{
    struct thread_data *data;
    data = (struct thread_data *) threadarg;

    //instantiating Mat objects
    Mat fgmaskShadow;
    Mat frameToResizeShadow;

    //copying into tmp variable
    globalFrames[i].copyTo(frameToResizeShadow);

    //performing background subtraction
    pMOG2Shadow->operator()(frameToResizeShadow , fgmaskShadow, .01);

    //performing sWND
    displayFrame("fgmaskShadow" , fgmaskShadow);
    Mat fgmaskShadowSWND = slidingWindowNeighborDetector(fgmaskShadow, fgmaskShadow.rows/10,
fgmaskShadow.cols/20);
    displayFrame("fgmaskShadowSWND", fgmaskShadowSWND);

    fgmaskShadowSWND = slidingWindowNeighborDetector(fgmaskShadowSWND, fgmaskShadowSWND.rows/20,
fgmaskShadowSWND.cols/40);
    displayFrame("fgmaskShadowSWND2", fgmaskShadowSWND);
    //performing canny
    Mat fgMaskShadowSWNDCanny = cannyContourDetector(fgmaskShadowSWND);
    displayFrame("fgMaskShadowSWNDCanny2", fgMaskShadowSWNDCanny);

    //return canny
    mogDetection2GlobalFrame = fgMaskShadowSWNDCanny;

    //signal completion
    mogDetection2GlobalFrameCompletion = 1;
}

void mogDetection2ThreadHandler(bool buffer)
{
    //instantiating multithread object
    pthread_t mogDetection2Thread;

    //instantiating multithread Data object
    struct thread_data threadData;

    //saving data into data object
    threadData.data = i;

    //creating threads
   int mogDetection2ThreadRC = pthread_create(&mogDetection2Thread, NULL, computeBgMog2, (void *)&threadData);
}


//method to handle median image subtraction
void *computeMedianDetection(void *threadarg)
{
    struct thread_data *data;
    data = (struct thread_data *) threadarg;
    int tmp = data->data;

    medianDetectionGlobalFrame = medianImageSubtraction(FRAME_RATE);

    /*
    //generate or read background image
    generateBackgroundImage(FRAME_RATE);

    //calculate image difference and save to global
    medianDetectionGlobalFrame = imageSubtraction();
    */

    medianDetectionGlobalFrameCompletion = 1;
 }

void medianDetectionThreadHandler(int FRAME_RATE)
{
    //instantiating multithread object
    pthread_t medianDetectionThread;

    //instantiating multithread Data object
    struct thread_data threadData;

    //saving data into data object
    threadData.data = FRAME_RATE;

    //creating threads
    int medianDetectionThreadRC = pthread_create(&medianDetectionThread, NULL, computeMedianDetection, (void
*)&threadData);
}


//method to handle all image processing object detection
void objectDetection(int FRAME_RATE)
{
    cout << " ENTER OBJdetect" << endl;
    //save all methods vehicle canny outputs
    //Mat gmmDetection = gaussianMixtureModel();
    Mat gmmDetection;

    //Mat tmpMedian = medianImageSubtraction(FRAME_RATE);

    Mat ofaDetection = opticalFlowFarneback();

    //Mat ofaDetection;
    //vibeDetectionGlobalFrame = vibeBackgroundSubtraction(false);
    //Mat tmpMOG1 =  bgMog(false);
    //Mat tmpMOG2 = bgMog2(false);

    //opticalFlowFarneback();
    vibeBackgroundSubtractionThreadHandler(false);
    mogDetectionThreadHandler(false);
    mogDetection2ThreadHandler(false);
    medianDetectionThreadHandler(FRAME_RATE);

    bool firstTimeMedianImage = true;
    bool firstTimeVibe = true;
    bool firstTimeMOG1 = true;
    bool firstTimeMOG2 = true;
    bool enterOnce = true;

    Mat tmpMedian;
    Mat tmpVibe;
    Mat tmpMOG1;
    Mat tmpMOG2;

    /*
    while(medianDetectionGlobalFrameCompletion != 1||
        vibeDetectionGlobalFrameCompletion != 1||
        mogDetection1GlobalFrameCompletion != 1||
        mogDetection2GlobalFrameCompletion != 1 ||
        enterOnce
        )
    */

    bool finishedMedian = false;
    bool finishedVibe = false;
    bool finishedMOG1 = false;
    bool finishedMOG2 = false;
```

```
cout << " Init OBJdetect" << endl;

while(!finishedMedian ||
        !finishedVibe ||
        !finishedMOG1||
        !finishedMOG2 ||
        enterOnce
        )
{
    enterOnce = false;

    if(firstTimeMedianImage && medianDetectionGlobalFrameCompletion == 1 )
    {
        tmpMedian = medianDetectionGlobalFrame;
        displayFrame("medianDetection", tmpMedian);
        firstTimeMedianImage = false;
        finishedMedian = true;
        cout << "MEDIAN DONE" << endl;
        welcome("Median -> Set");
    }
    if(firstTimeVibe && vibeDetectionGlobalFrameCompletion == 1 )
    {
        tmpVibe = vibeDetectionGlobalFrame;
        displayFrame("vibeDetection", tmpVibe);
        firstTimeVibe = false;
        finishedVibe = true;
        cout << "VIBE DONE" << endl;
        welcome("ViBe -> Set");

    }
    if(firstTimeMOG1 && mogDetection1GlobalFrameCompletion == 1 )
    {
        tmpMOG1= mogDetection1GlobalFrame;
        displayFrame("mogDetection1", tmpMOG1);
        firstTimeMOG1 = false;
        finishedMOG1 = true;
        cout << "MOG1 DONE" << endl;
        welcome("MOG1 -> Set");
    }
    if(firstTimeMOG2 && mogDetection2GlobalFrameCompletion == 1)
    {
        tmpMOG2 = mogDetection2GlobalFrame;
        displayFrame("mogDetection2", tmpMOG2);
        firstTimeMOG2 = false;
        finishedMOG2 = true;
        cout << "MOG2 DONE" << endl;
        welcome("MOG2 -> Set");
    }
}

vibeDetectionGlobalFrameCompletion = 0;
mogDetection1GlobalFrameCompletion = 0;
mogDetection2GlobalFrameCompletion = 0;
medianDetectionGlobalFrameCompletion = 0;

/*
```

```
    displayFrame("vibeDetection", tmpVibe, true);
    displayFrame("mogDetection1", tmpMOG1, true);
    displayFrame("mogDetection2", tmpMOG2, true);
    displayFrame("medianDetection", tmpMedian, true);
    displayFrame("gmmDetection", gmmDetection);
    displayFrame("ofaDetection", ofaDetection, true);
    */
    displayFrame("Raw Frame", globalFrames[i], true);

    if(i > (bufferMemory + mlBuffer  -1)  && tmpMOG1.channels() == 3 && tmpMOG2.channels() == 3 &&
ofaDetection.channels() == 3 && tmpMedian.channels() == 3)
    {
        if(i > bufferMemory * 3 + 2 && tmpMOG1.channels() == 3 && tmpMOG2.channels() == 3 &&
ofaDetection.channels() == 3 && tmpMedian.channels() == 3
                && tmpVibe.channels() == 3 && gmmDetection.channels() == 3 && 1 == 2)
        {
            Mat combined = tmpMOG1 +  tmpMOG2 + ofaDetection + tmpMedian + tmpVibe + gmmDetection;
            displayFrame("Combined Contours", combined);

            double beta = ( 1.0 - .5 );
            addWeighted( combined, .5, globalFrames[i], beta, 0.0, combined);
            displayFrame("Overlay", combined, true);
        }
        else
        {
            Mat combined = tmpMOG1 +  tmpMOG2 + ofaDetection + tmpMedian;
            displayFrame("Combined Contours", combined);

            double beta = ( 1.0 - .5 );
            addWeighted( combined, .5, globalFrames[i], beta, 0.0, combined);
            displayFrame("Overlay", combined, true);
        }
    }

    else
    {
        cout << "Sync Issue" << endl;
    }

}

void fillCoordinates(vector <Point2f> detectedCoordinatesMoments)
{
    for(int v = 0; v < detectedCoordinatesMoments.size(); v++)
    {
        Point tmpPoint ((int) detectedCoordinatesMoments[v].x,  (int) detectedCoordinatesMoments[v].y);

        if((tmpPoint.x > 30 && tmpPoint.x < globalGrayFrames[i].cols - 60) &&
                (tmpPoint.y > 30 && tmpPoint.y < globalGrayFrames[i].rows - 30))
        {
            detectedCoordinates.push_back(tmpPoint);
        }
    }
}

bool point_comparator(const cv::Point2f &a, const cv::Point2f &b) {
    return a.x*a.x + a.y*a.y < b.x*b.x + b.y*b.y; // (/* Your expression */);
}

void displayCoordinates(vector <Point> coordinatesToDisplay)
{
    for(int v = 0; v < coordinatesToDisplay.size(); v++)
    {
        cout << "(" << coordinatesToDisplay[v].x << "," << coordinatesToDisplay[v].y << ")" << endl;
    }

}

void displayCoordinate(Point coordinate)
{
    cout << "(" << coordinate.x << "," << coordinate.y << ")" << endl;
}

Mat checkBlobVotes(Mat srcFrame, vector <Point> coordinates)
{
    return srcFrame;
}

void drawCoordinates(vector <Point> coordinatesToDisplay, String initialName)
{
    //Mat tmpToDraw;

    //globalFrames[i].copyTo(tmpToDraw);

    Mat tmpToDraw; // = Mat::zeros(globalFrames[i].size(), CV_8UC3);

    backgroundFrameMedian.copyTo(tmpToDraw);

    /*
    for(int v = 0; v < tmpToDraw.rows * 1; v++)
    {
        for(int j = 0; j < tmpToDraw.cols; j++)
        {
            tmpToDraw.at<uchar>(v,j) = 0;
        }
    }
    */

    for(int v = 0; v < coordinatesToDisplay.size(); v++)
    {
        circle( tmpToDraw, coordinatesToDisplay[v], 4, Scalar(254, 254, 0), -1, 8, 0 );
    }

    //destroyWindow("Coordinates to Display");
    displayFrame(initialName ,tmpToDraw, true);
    //imwrite("trackingFrame" + initialName + ".TIFF", trackingFrame);

    //tmpToDraw = slidingWindowNeighborPointDetector(tmpToDraw, tmpToDraw.rows / 5, tmpToDraw.cols / 10 ,
coordinatesToDisplay);
    //**tmpToDraw = slidingWindowNeighborPointDetector(tmpToDraw, tmpToDraw.rows / 10, tmpToDraw.cols / 20 ,
coordinatesToDisplay);
```

```
    //tmpToDraw = slidingWindowNeighborPointDetector(tmpToDraw, tmpToDraw.rows / 20, tmpToDraw.cols / 40 ,
coordinatesToDisplay);
    //tmpToDraw = slidingWindowNeighborPointDetector(tmpToDraw, tmpToDraw.rows / 30, tmpToDraw.cols / 60 ,
coordinatesToDisplay);
    //tmpToDraw = slidingWindowNeighborPointDetector(tmpToDraw, tmpToDraw.rows / 40, tmpToDraw.cols / 80 ,
coordinatesToDisplay);

    //tmpToDraw = checkBlobVotes(tmpToDraw, coordinatesToDisplay);

    //displayFrame("sWNPD Frame", tmpToDraw, true);

}

vector <Point> sortCoordinates(vector <Point> coordinates)
{
    sort(coordinates.begin(), coordinates.end(), point_comparator);
    return coordinates;
}

Point averagePoints(vector <Point> coordinates)
{
    if(coordinates.size() != 0)
    {
        double xCoordinate = 0;
        double yCoordinate = 0;

        for(int v = 0; v < coordinates.size(); v++)
        {
            xCoordinate += coordinates[v].x;
            yCoordinate += coordinates[v].y;
        }

        Point tmpPoint(xCoordinate/coordinates.size(), yCoordinate/coordinates.size());

        return tmpPoint;
    }

    else
    {
        return coordinates[0];
    }
}

vector <Point> averageCoordinates(vector <Point> coordinates, int distanceThreshold)
{
    if(coordinates.size() > 1)
    {
        cout <<"BIGGER THAN 1" << endl;

        vector <Point> destinationCoordinates;
        vector <Point> pointsToAverage;
        coordinates = sortCoordinates(coordinates);
        Point tmpPoint = coordinates[0];

        bool enteredOnce = false;
        for(int v = 0; v < coordinates.size(); v++)
        {
            double tmp1 = abs( tmpPoint.y - coordinates[v].y);
            double tmp2 =  abs(tmpPoint.x - coordinates[v].x);
            double tmp = sqrt(tmp1 * tmp2);
            /*
            cout << tmp1 << " tmp1 " << endl;
            cout << tmp2 << " tmp2 " << endl;
            cout << tmp << " tmp " << endl;
            */
            if(sqrt((abs(tmpPoint.y - coordinates[v].y) * (abs(tmpPoint.x - coordinates[v].x)))) > distanceThreshold)
            {
                //cout << "Entered Refresh " << v << endl;
                destinationCoordinates.push_back(averagePoints(pointsToAverage));
                tmpPoint = coordinates[v];
                pointsToAverage.erase(pointsToAverage.begin(), pointsToAverage.end());
                bool enteredOnce =  true;
            }
            else
            {
                //cout << "Entered Old " << v << endl;
                pointsToAverage.push_back(coordinates[v]);
            }
        }

        if(!enteredOnce)
        {
            destinationCoordinates.push_back(averagePoints(pointsToAverage));
        }

        else if(pointsToAverage.size() == 1)
        {
            destinationCoordinates.push_back(pointsToAverage[0]);
        }

        else if(pointsToAverage.size() > 0)
        {
            destinationCoordinates.push_back(averagePoints(pointsToAverage));
        }

        return destinationCoordinates;
    }

    else
    {
        return coordinates;
    }
}

void drawAllTracking()
{
    for(int v = 0; v < detectedCoordinates.size(); v++)
    {
        globalDetectedCoordinates.push_back(detectedCoordinates[v]);
        circle( finalTrackingFrame, detectedCoordinates[v], 4, Scalar(254, 254, 0), -1, 8, 0 );
    }
    displayFrame("All Tracking Frame", finalTrackingFrame, true);
 }

void registerFirstCar()
{
    vectorOfDetectedCars.push_back(detectedCoordinates);

    for(int v= 0 ; v< detectedCoordinates.size(); v++)
    {
        if(detectedCoordinates[v].x < 75)
        {
            vector <Point> carCoordinate;
            carCoordinate.push_back(detectedCoordinates[v]);
            carCoordinates.push_back(carCoordinate);
        }
    }
}

void drawTmpTracking()
{
    const int thresholdPointMemory = 10;
    int counter = coordinateMemory.size() - thresholdPointMemory ;

    Mat tmpTrackingFrame;
    globalFrames[i].copyTo(tmpTrackingFrame);

    if(counter < 0)
    {
        //counter = coordinateMemory.size(); //globalDetectedCoordinates.size();
        counter = 0;
    }

    cout << "COUNTER " << counter << " DETECT GP " << globalDetectedCoordinates.size() << endl;

    for(int v = counter; v < coordinateMemory.size(); v++)
        for(int j = 0; j < coordinateMemory[v].size(); j++)
            circle(tmpTrackingFrame, coordinateMemory[v][j], 4, Scalar(254, 254, 0), -1, 8, 0);

    displayFrame("Tmp Tracking Frame", tmpTrackingFrame, true);
}

bool checkBasicXYAnomaly(int xMovement, int yMovement, Point carPoint, double currentAngle)
{
    const double maxThreshold = 5;
    const double minThreshold = -5;
    const int maxMovement = 30;

    const int angleThresholdMax = 10;
    const int angleThresholdMin = -10;

    cout << "X MOVEMENT " << (double) xMovement << endl;
    cout << "Y MOVEMENT " << (double) yMovement << endl;

    cout << "X LEARNED " << (double) xLearnedMovement << endl;
    cout << "Y LEARNED " << (double) yLearnedMovement << endl;
    cout <<"X SCALAR " << ((double) xMovement) / ((double) xLearnedMovement) << endl;
    cout <<"Y SCALAR " << ((double) yMovement) / ((double) yLearnedMovement) << endl;

    //cout << "ANGLE 1 " << currentAngle << endl; //to_string(((atan ((double) yMovement) / (double) xMovement)) * 180 /
PI) << endl;


    if(/*currentAngle > learnedAngle + angleThresholdMax || */currentAngle > 20)
    {
        displayCoordinate(carPoint);
        /*
        circle(drawAnomalyCar, carPoint, 5, Scalar(255, 255, 0), -1);
        String tmpToDisplay = "ANOMALY DETECTED (ANGLE) -> Frame Number: " + to_string(i);
        welcome(tmpToDisplay);
        */
        cout << " !!!!!!!!!!!ANOMALY DETECTED (ANGLE)!!!!!!!!!!!" << endl;
        //displayFrame("drawAnomalyCar", drawAnomalyCar, true);
    }

    /*
    if(yMovement < maxMovement) //&& xMovement < maxMovement)
    {
        /*if(((xMovement > xLearnedMovement + maxThreshold || xMovement < xLearnedMovement + minThreshold))
                || ((yMovement > yLearnedMovement + maxThreshold) || (yMovement < yLearnedMovement +
minThreshold)))
            */
    /*
        if(((yMovement > yLearnedMovement + maxThreshold) || (yMovement < yLearnedMovement + minThreshold)))
        {
            circle(drawAnomalyCar, carPoint, 5, Scalar(255, 255, 0), -1);
            String tmpToDisplay = "ANOMALY DETECTED (Amplitude) -> Frame Number: " + to_string(i);
            welcome(tmpToDisplay);
            cout << " !!!!!!!!!!!ANOMALY DETECTED (Amplitude)!!!!!!!!!!!" << endl;
            displayFrame("drawAnomalyCar", drawAnomalyCar, true);
        }

        else
        {
            if(yLearnedMovement != 0 && xLearnedMovement != 0)
            {
                //if(((double) xMovement) / ((double) xLearnedMovement) > 3 ||
                //   (((double) yMovement) / ((double) yLearnedMovement) > 5))


                if(((double) yMovement) / ((double) yLearnedMovement) > 5)
                {
                    String tmpToDisplay = "ANOMALY DETECTED (Scalar) -> Frame Number: " + to_string(i);
                    welcome(tmpToDisplay);
                    circle(drawAnomalyCar, carPoint, 5, Scalar(0, 0, 255), -1);
                    cout << " !!!!!!!!!!!ANOMALY DETECTED (Scalar)" << endl;
                }
            }
            displayFrame("drawAnomalyCar", drawAnomalyCar, true);
        }
    }
    */
```

```
}

int findMin(int num1, int num2)
{
    if(num1 < num2)
    {
        return num1;
    }
    else if(num2 < num1)
    {
        return num2;
    }
    else
    {
        return num1;
    }
}

void analyzeMovement()
{
    vector <Point> currentDetects =  vectorOfDetectedCars[vectorOfDetectedCars.size() - 1];
    vector <Point> prevDetects =  vectorOfDetectedCars[vectorOfDetectedCars.size() - 2];

    Mat drawCoordinatesOnFrame;
    backgroundFrameMedianColor.copyTo(drawCoordinatesOnFrame);

    Mat drawCoordinatesOnFrameXY;
    backgroundFrameMedianColor.copyTo(drawCoordinatesOnFrameXY);

    Mat drawCoordinatesOnFrameSpeed;
    backgroundFrameMedianColor.copyTo(drawCoordinatesOnFrameSpeed);

    const int distanceThreshold = 50;

    int least = findMin(currentDetects.size(), prevDetects.size());

    for(int v = 0; v < least; v++)
    {
        currentDetects = sortCoordinates(currentDetects);
        prevDetects = sortCoordinates(prevDetects);

        double lowestDistance = INT_MAX;
        double distance;

        Point tmpPoint;
        Point tmpDetectPoint;
        Point bestPoint;

        for(int j = 0; j < prevDetects.size(); j++ )
        {
            tmpDetectPoint = prevDetects[j];
            tmpPoint = currentDetects[v];

            distance = sqrt(abs(tmpDetectPoint.x - tmpPoint.x) * (abs(tmpDetectPoint.y - tmpPoint.y)));

            if(distance < lowestDistance)
            {
                lowestDistance = distance;
                bestPoint  = tmpDetectPoint;
            }
            //carCoordinates.push_back();
        }

        int xDisplacement = abs(bestPoint.x - tmpPoint.x);
        int yDisplacement = abs(bestPoint.y - tmpPoint.y);

        Mat distanceFrameAnomaly;
        backgroundFrameMedianColor.copyTo(distanceFrameAnomaly);

        cout << "DISTANCE FROM NORMAL SIZE " << distanceFromNormal.size() << endl;

        for(int d = 0; d < distanceFromNormal.size(); d++)
        {
                learnedDistanceCounter++;
                currentDistance = distanceFromNormal[d];
                learnedDistance += distanceFromNormal[d];

                cout << "CURRENT DISTANCE " << currentDistance << endl;

                String movementStr = to_string(currentDistance); //to_string(((atan ((double) yDisplacement) / (double)
xDisplacement)) * 180 / PI);

                putText(distanceFrameAnomaly, movementStr,
distanceFromNormalPoints[d],CV_FONT_HERSHEY_SIMPLEX,
                        1, cvScalar(254, 254,0), 1, CV_AA, false);

                putText(distanceFrameAnomaly, to_string(learnedDistanceAverage), Point(0,
30),CV_FONT_HERSHEY_SIMPLEX,
                        1, cvScalar(254, 254,0), 1, CV_AA, false);
        }

        distanceFromNormal.erase(distanceFromNormal.begin(), distanceFromNormal.end());
        distanceFromNormalPoints.erase(distanceFromNormalPoints.begin(), distanceFromNormalPoints.end());

        cout << "LEARNED DISTANCE " << learnedDistanceAverage << endl;;

        displayFrame("distanceFrameAnomaly", distanceFrameAnomaly, true);

        learnedDistanceAverage = learnedDistance / learnedDistanceCounter;

        if(lowestDistance < distanceThreshold && yDisplacement < 11)
        {
            cout << "xDisplacement " << xDisplacement << endl;
            cout << "yDisplacement " << yDisplacement << endl;

            xAverageMovement += xDisplacement;
            yAverageMovement += yDisplacement;
            xAverageCounter++;
            yAverageCounter++;
            xLearnedMovement = (xAverageMovement / xAverageCounter);
            yLearnedMovement = (yAverageMovement / yAverageCounter);
            currentSpeed = sqrt(xDisplacement * yDisplacement);
            learnedSpeed += currentSpeed;
            learnedSpeedAverage = learnedSpeed / xAverageCounter;

            cout << "LEARNED SPEED " << learnedSpeedAverage << " CURRENT SPEED " << currentSpeed << endl;

            double currentAngle =  ((atan ((double) yDisplacement) / (double) xDisplacement)) * 180 / PI;
            float currentAngleFloat = (float) currentAngle;

            cout <<"CURRENT ANGLE " << currentAngle << " ANGLE LEARNED " << learnedAngle << endl;

            if(currentAngle > 360)
            {
                currentAngle = 0;
                //cout << " INF " << endl;
            }

            if(currentAngleFloat != currentAngleFloat)
            {
                currentAngle = 0;
                //cout << " NaN " << endl;
            }

            learnedAggregate += currentAngle;
            learnedAngle = learnedAggregate / xAverageCounter;

            if( learnedSpeedAverage * 3 < currentSpeed )
            {
                String movementStr = to_string(currentSpeed); //to_string(((atan ((double) yDisplacement) / (double)
xDisplacement)) * 180 / PI);

                putText(drawCoordinatesOnFrameSpeed, movementStr, bestPoint,CV_FONT_HERSHEY_SIMPLEX,
                        1, cvScalar(0, 0,255), 1, CV_AA, false);

                putText(drawCoordinatesOnFrameSpeed, to_string(learnedSpeedAverage), Point (0,
30) ,CV_FONT_HERSHEY_SIMPLEX,
                                        1, cvScalar(255, 255, 0), 1, CV_AA, false);

                displayFrame("Speed Movement", drawCoordinatesOnFrameSpeed, true);
            }
            else
            {
                String movementStr = to_string(currentSpeed); //to_string(((atan ((double) yDisplacement) / (double)
xDisplacement)) * 180 / PI);

                putText(drawCoordinatesOnFrameSpeed, movementStr, bestPoint, CV_FONT_HERSHEY_SIMPLEX,
                        1, cvScalar(255, 255, 0), 1, CV_AA, false);

                putText(drawCoordinatesOnFrameSpeed, to_string(learnedSpeedAverage), Point (0,
30) ,CV_FONT_HERSHEY_SIMPLEX,
                                        1, cvScalar(255, 255, 0), 1, CV_AA, false);

                displayFrame("Speed Movement", drawCoordinatesOnFrame, true);
            }

            if(currentAngle > 15)
```

```
            {
                String movementStr = to_string(currentAngle); //to_string(((atan ((double) yDisplacement) / (double)
xDisplacement)) * 180 / PI);

                putText(drawCoordinatesOnFrame, movementStr, bestPoint,CV_FONT_HERSHEY_SIMPLEX,
                        1, cvScalar(0, 0,255), 1, CV_AA, false);

                putText(drawCoordinatesOnFrame, to_string(learnedAngle), Point (0, 30) ,CV_FONT_HERSHEY_SIMPLEX,
                                                        1, cvScalar(0, 0,255), 1, CV_AA, false);

                displayFrame("Angular Movement", drawCoordinatesOnFrame, true);
            }
            else{
                String movementStr = to_string(currentAngle); //to_string(((atan ((double) yDisplacement) / (double)
xDisplacement)) * 180 / PI);

                putText(drawCoordinatesOnFrame, movementStr, bestPoint,CV_FONT_HERSHEY_SIMPLEX,
                        1, cvScalar(255,255,0), 1, CV_AA, false);
                displayFrame("Angular Movement", drawCoordinatesOnFrame, true);

            }
            if(yDisplacement > yLearnedMovement * 3)
            {
                String movementStr = to_string(xDisplacement) +"|" + to_string(yDisplacement);
                putText(drawCoordinatesOnFrameXY, movementStr, bestPoint,CV_FONT_HERSHEY_SIMPLEX,
                        1, cvScalar(0, 0,255), 1, CV_AA, false);
                putText(drawCoordinatesOnFrameXY, (to_string(xLearnedMovement) + "|" + to_string(yLearnedMovement)),
Point (0, 30) ,CV_FONT_HERSHEY_SIMPLEX,
                                                                1, cvScalar(0, 0, 255), 1, CV_AA, false);
                displayFrame("XY Movement", drawCoordinatesOnFrameXY, true);

                circle(drawAnomalyCar, bestPoint, 5, Scalar(0, 0, 255), -1);
                String tmpToDisplay = "ANOMALY DETECTED (ANGLE) -> Frame Number: " + to_string(i);
                welcome(tmpToDisplay);
                cout << " !!!!!!!!!!!ANOMALY DETECTED (ANGLE)!!!!!!!!!!!" << endl;
                displayFrame("Anomaly Car Detect Frame", drawAnomalyCar, true);
                cout << "here " << endl;
            }

            else
            {
                String movementStr = to_string(xDisplacement) +"|" + to_string(yDisplacement);

                putText(drawCoordinatesOnFrameXY, movementStr, bestPoint,CV_FONT_HERSHEY_SIMPLEX,
                        1, cvScalar(255, 255, 0), 1, CV_AA, false);
                putText(drawCoordinatesOnFrameXY, (to_string(xLearnedMovement) + "|" + to_string(yLearnedMovement)),
Point (0, 30) ,CV_FONT_HERSHEY_SIMPLEX,
                                                                1, cvScalar(255, 255, 0), 1, CV_AA, false);
                displayFrame("XY Movement", drawCoordinatesOnFrameXY, true);
            }

            if(i > (bufferMemory + mlBuffer + 3) && tmpDetectPoint.x <  FRAME_WIDTH - 150)
            {
                checkBasicXYAnomaly(xDisplacement, yDisplacement, tmpDetectPoint, currentAngle);
            }
        }
```

```
    }
}

void learnedCoordinate()
{
    cout << " ENTERING LEARNED COORDINATES " << endl;
    Mat distanceFrame;
    backgroundFrameColorMedian.copyTo(distanceFrame);

    //cout <<" IN HERE " << endl;
    int initialCounter = 0;

    bool enterOnce = false;

    cout << "DETECTED COORDINATES " << detectedCoordinates.size() << endl;
    for(int v = 0; v < detectedCoordinates.size(); v++)
    {
        //   cout << " V " << v << " left " << (detectedCoordinates.size() - v) <<  endl;
        Point tmpPoint = detectedCoordinates[v];
        for(int x = 0; x < lanePositions.size(); x++)
        {
            cout << "LANE POSITIONS " << lanePositions.size() << endl;
            cout << lanePositions[v] <<  " LANE POSITION " << endl;
            cout << tmpPoint.y << " tmpPoint" << endl;

            bool breakNow = false;
            //cout << " X " << x << " left " << (lanePositions.size() - x) <<   endl;

            if((lanePositions[v] > tmpPoint.y || x == lanePositions.size()) && enterOnce)
            {
                enterOnce = true;
                initialCounter = x;
                breakNow = true;
            }
        }

        //cout << "ACCESS TIMES SIZE " << accessTimes.size() << endl;
        //cout << "ACCESS TIMES SIZE 2" << accessTimes[0].size() << endl;

        cout << "LEARNED COORDINATES " << learnedCoordinates[initialCounter].size() << endl;

        for(int j = 0; j < learnedCoordinates[initialCounter].size(); j++)
        {
            //cout << " J " << j << " left " <<  (learnedCoordinates[initialCounter].size() - j) << endl;

            Point tmpPoint2 = learnedCoordinates[initialCounter][tmpPoint.x * 7];
            Point tmpPoint3(((tmpPoint2.x + tmpPoint.x)/2),((tmpPoint2.y + tmpPoint.y) / 2));
            //accessTimes[initialCounter][tmpPoint.x * 7] = Point(     accessTimes[initialCounter][tmpPoint.x * 7].x + 1, 0);
            //tmpPoint3 = Point(tmpPoint2.x  + 5, tmpPoint2.y - 5);
            learnedCoordinates[initialCounter][tmpPoint.x * 7] = tmpPoint; // tmpPoint3;
            //cout << "ACCESS TIMES SIZE " << accessTimes.size() << endl;
            //accessTimes[initialCounter][tmpPoint.x * 7] = accessTimes[initialCounter][tmpPoint.x * 7] + 1;

            //learnedCoordinates[initialCounter][tmpPoint.x * 7] = tmpPoint;
        }
        bool breakNow = false;

        for(int j = 0; j < lanePositions.size(); j++)
        {
            double tmpLanePosition = 0;
            double normalLanePosition = 0;

            tmpLanePosition = lanePositions[j];
            vector <Point> tmpPointVector;

            if((lanePositions[j] >= tmpPoint.y) && !breakNow)
            {
                for(int v = 0; v < FRAME_WIDTH; v+= 7)
                {
                    if((v >= tmpPoint.x - 6 && v <= tmpPoint.x + 6) && !breakNow)
                    {
                        tmpPointVector = learnedCoordinates[j];

                        cout << tmpPointVector.size() << " SIZE OF TMP POINT VECTOR" << endl;

                        //tmpPointVector.push_back(Point(v, tmpPoint.y));

                        cout <<" tmpPoint.x " << tmpPoint.x << endl;

                        Point oldTmpPoint = tmpPointVector.at(tmpPoint.x / 7);
                        //Point newAveragePoint = Point(v,((oldTmpPoint.y + tmpPoint.y)/2));

                        //tmpPointVector.at(tmpPoint.x / 7) = newAveragePoint; //(Point(v, tmpPoint.y));
                        //accessTimes[j][tmpPoint.x/7] = accessTimes[j][tmpPoint.x/7]  + 1;

                        //Point averagePoint = Point(v, (tmpPoint.y * 1+ accessTimes[j][tmpPoint.x/7] * oldTmpPoint.y)
                        //        / (accessTimes[j][tmpPoint.x/7]));

                        //cout << accessTimes[initialCounter][tmpPoint.x * 7].x << " ACCESS TIMES SUM1" << endl;
                        int tmpATI =  accessTimesInt[initialCounter][tmpPoint.x * 7];
                        tmpATI++;

                        cout << " OLD TMP POINT ";
                        displayCoordinate(oldTmpPoint);

                        cout << " CURRENT TMP POINT ";
                        displayCoordinate(tmpPoint);

                        cout << "tmpATI " << tmpATI << endl;

                        int tmp = ((tmpPoint.y + ((tmpATI - 1) * oldTmpPoint.y)))
                                                        / tmpATI;

                        //tmp =  (tmpPoint.y + learnedCoordinates[j][tmpPoint.x/7].x) / 2;

                        //cout << accessTimesInt[initialCounter][tmpPoint.x * 7] << "ACCESS TIMES 1" << endl;
                        accessTimesInt[initialCounter][tmpPoint.x * 7] = tmpATI;
                        //cout << accessTimesInt[initialCounter][tmpPoint.x * 7] << "ACCESS TIMES 2" << endl;

                        //accessTimes[initialCounter][tmpPoint.x * 7] = Point(accessTimes[initialCounter][tmpPoint.x * 7].x + 1,
accessTimes[initialCounter][tmpPoint.x * 7].y);
```

```
                        //cout << accessTimes[initialCounter][tmpPoint.x * 7].x << " ACCESS TIMES SUM2" << endl;

                        cout << tmpPoint.y << " tmpPoint.y" << endl;
                        cout << tmp << " VALUE TMP" << endl;

                        //tmp = ((tmpPoint.y) * 1 + (tmpPointVector.at(tmpPoint.x / 7) .y*
accessTimes[initialCounter][tmpPoint.x * 7].x))
                        //        / (1 + accessTimes[initialCounter][tmpPoint.x * 7].x);

                        //tmp = 10

                        cout << " DISTANCE FROM NORMAL  = " << abs(tmpPoint.y -tmpPointVector.at(tmpPoint.x / 7) .y)
<< endl;

                        distanceFromNormal.push_back(abs(tmpPoint.y -tmpPointVector.at(tmpPoint.x / 7) .y));
                        distanceFromNormalPoints.push_back(tmpPoint);

                        putText(distanceFrame,to_string(abs(tmpPoint.y -tmpPointVector.at(tmpPoint.x / 7) .y)), tmpPoint,
3,1,Scalar(254, 254,0),2);
                        circle( distanceFrame, tmpPoint, 4, Scalar(254, 254, 0), -1, 8, 0 );

                        Point averagePoint(v, tmp);

                        //Point averagePoint = Point(v, (tmpPoint.y * 1+ accessTimes[j][tmpPoint.x/7] * oldTmpPoint.y)
                      //          (accessTimes[initialCounter][tmpPoint.x * 7].x)); // (accessTimes[j][tmpPoint.x/7]));

                        //Point averagePoint = tmpPoint;

                        displayCoordinate(averagePoint);

                        tmpPointVector.at(tmpPoint.x / 7) = averagePoint; //(Point(v, tmpPoint.y));

                        learnedCoordinates[j] = tmpPointVector;

                        breakNow = true;

                    }

                    if(breakNow)
                        v = FRAME_WIDTH;

                    /*
                    if(j == 0)
                        normalLanePosition = tmpLanePosition/2;
                    else
                        normalLanePosition = (tmpLanePosition + lanePositions[j-1]) / 2;
                    */

                }
            }
            if(breakNow)
                j = lanePositions.size();
        }
    }
    displayFrame("distanceFrame", distanceFrame, true);
    cout << " DONE " << endl;
}

void calculateDeviance()
{
    cout << "ENTERED CALCULATE DEVIANCE" << endl;

    learnedCoordinate();

    cout << "FINISHED LEARNED DEVIANCE" << endl;

    /*
    for(int j = 0; j < detectedCoordinates.size(); j++)
    {
        Point tmpPoint = detectedCoordinates[j];

        for(int v = 0; v < lanePositions.size(); v++)
        {
            if(tmpPoint.y > lanePositions[v] || v == lanePositions.size() -1)
            {
                break;
            }
        }
    }
    */

    Mat tmpToDisplay;
    backgroundFrameMedianColor.copyTo(tmpToDisplay);

    for(int v = 0; v < learnedCoordinates.size(); v++)
    {
        for(int j = 0; j < learnedCoordinates[v].size(); j++)
        {
            Point tmpPoint = learnedCoordinates[v][j];
            circle( tmpToDisplay, tmpPoint, 4, Scalar(254, 254, 0), -1, 8, 0 );
        }
    }

    cout << "PRINTING LEARNED PATH " << endl;

    displayFrame("Learned Path", tmpToDisplay, true);

    cout << "FINISHED DISPLAYING" << endl;
}

void individualTracking()
{
    const double distanceThreshold = 25;

    cout << detectedCoordinates.size() << " SIZE OF DETECT" << endl;
    cout << carCoordinates.size() << " SIZE OF CAR COORDINATES" << endl;

    bool registerdOnce = false;

    if(i == (bufferMemory + mlBuffer + 3) || ((carCoordinates.size() == 0) && i > (bufferMemory + mlBuffer)))
    {
        registerFirstCar();
        cout << "DETECT LESS THAN 1" << endl;
    }

    else if(detectedCoordinates.size() > 0)
    {
        cout << "DETECT GREATER THAN 1" << endl;

        vectorOfDetectedCars.push_back(detectedCoordinates);
        coordinateMemory.push_back(detectedCoordinates);
        calculateDeviance();

        cout << " ONE " << endl;
        analyzeMovement();

        cout << " THIS TEST" << endl;
        cout << " TWO " << endl;

        /*
        cout << "SECOND" << endl;

        for(int v = 0; v < carCoordinates.size(); v++)
        {
            cout << " DETECTED " << detectedCoordinates.size() << endl;
            cout << " CAR COORDINATES 2 " << carCoordinates.size() << endl;
            carCoordinates[v] = sortCoordinates(carCoordinates[v]);

            detectedCoordinates = sortCoordinates(detectedCoordinates);

            double lowestDistance = INT_MAX;
            double distance;

            Point tmpPoint;
            Point tmpDetectPoint;
            Point bestPoint;

            for(int j = 0; j < detectedCoordinates.size(); j++ )
            {
                tmpDetectPoint = detectedCoordinates[j];
                tmpPoint = carCoordinates[v][carCoordinates[v].size()-1];

                distance = sqrt(abs(tmpDetectPoint.x - tmpPoint.x) * (abs(tmpDetectPoint.y - tmpPoint.y)));

                cout << " ALL DISTANCES " << distance << endl;

                if(distance < lowestDistance)
                {
                    lowestDistance = distance;
                    bestPoint  = tmpDetectPoint;
                    cout << " CAR DISTANCE IS " << distance << endl;
                }
                //carCoordinates.push_back();
            }

            int xDisplacement = abs(bestPoint.x - tmpPoint.x);
            int yDisplacement = abs(bestPoint.y - tmpPoint.y);
            cout << " LOWEST CAR DISTANCE IS " << lowestDistance << endl;
            cout << " X DISPLACEMENT is " <<xDisplacement<< endl;

            if(lowestDistance < distanceThreshold)
            {
                xAverageMovement += xDisplacement;
                yAverageMovement += yDisplacement;
                xAverageCounter++;
                yAverageCounter++;
                xLearnedMovement = (xAverageMovement / xAverageCounter);
                yLearnedMovement = (yAverageMovement / yAverageCounter);

                cout << " Y DISPLACEMENT is " <<yDisplacement << endl;

                cout << " AVERAGE X DISPLACEMENT IS " << xLearnedMovement << endl;
                cout << " AVERAGE Y DISPLACEMENT IS " << yLearnedMovement << endl;

                if(i > bufferMemory + 15)
                    checkBasicXYAnomaly(xDisplacement, yDisplacement, tmpDetectPoint);
            }
        }
        */
    }
}

void processCoordinates()
{
    //displayCoordinates(detectedCoordinates);

    drawCoordinates(detectedCoordinates, "1st Pass");

    //displayCoordinates(detectedCoordinates);

    imwrite(fileTime + "finalTrackingFrame.TIFF", finalTrackingFrame);
    detectedCoordinates = averageCoordinates(detectedCoordinates, 60);

    //displayCoordinates(detectedCoordinates);

    drawCoordinates(detectedCoordinates, "2nd Pass");
}

void pollOFAData()
{
    for(int v = 0; v < detectedCoordinates.size(); v++)
    {
        Point tmpPoint = detectedCoordinates[v];

        cout << "OFA VALUE" << ((double) ofaGlobalHeatMap.at<uchar>(tmpPoint.x, tmpPoint.y)) << endl;
    }
}

void trackingML()
{
    cout << " ENTERING TRACKING ML " << endl;
    if(i <= bufferMemory + mlBuffer)
    {
       welcome("Final Initialization; Running ML Startup -> Frames Remaining: " + to_string((bufferMemory + mlBuffer + 1)
- i));
    }

    else if(i > bufferMemory + mlBuffer + 1)
    {
        cout << " INTO tML2" << endl;

        if(i == bufferMemory + mlBuffer + 1)
        {
            welcome("Initialization Complete -> Starting ML");
        }

        else if(i > bufferMemory +mlBuffer + 1)
        {
            String tmpToDisplay =  "Running ML Tracking -> Frame Number: " + to_string(i);
            welcome(tmpToDisplay);
        }

        processCoordinates();

        individualTracking();

        drawTmpTracking();

        drawAllTracking();

        //pollOFAData();
        cout << " EXIT tML2" << endl;
    }
    cout << " EXITING TRACKING ML " << endl;

    detectedCoordinates.erase(detectedCoordinates.begin(), detectedCoordinates.end());
}

//method to initalize Mats on startup
void initilizeMat()
{
    const int numberOfLanes = 6;

    //if first run
    if(i == 0)
    {
        //initialize background subtractor object
        backgroundSubtractorGMM->set("initializationFrames", bufferMemory);
        backgroundSubtractorGMM->set("decisionThreshold", 0.85);

        //save gray value to set Mat parameters
        globalGrayFrames[i].copyTo(backgroundFrameMedian);

        lanePositions.push_back(57);
        lanePositions.push_back(120);
        lanePositions.push_back(200);
        lanePositions.push_back(290);
        lanePositions.push_back(390);
        lanePositions.push_back(FRAME_HEIGHT - 1);

        for(int j = 0; j < lanePositions.size(); j++)
        {
            double tmpLanePosition = 0;
            double normalLanePosition = 0;

            tmpLanePosition = lanePositions[j];
            vector <Point> tmpPointVector;

            vector <Point> accessTimesFirstVect;
            vector <int> accessTimesFirstVectInt;

            for(int v = 0; v < FRAME_WIDTH; v+= 7)
            {
                if(j == 0)
                    normalLanePosition = tmpLanePosition/2;
                else
                    normalLanePosition = (tmpLanePosition + lanePositions[j-1]) / 2;

                tmpPointVector.push_back(Point(v, normalLanePosition));
                accessTimesFirstVect.push_back(Point(1,0));
                accessTimesFirstVectInt.push_back(0);

            }

            learnedCoordinates.push_back(tmpPointVector);
            accessTimes.push_back(accessTimesFirstVect);
            accessTimesInt.push_back(accessTimesFirstVectInt);
        }

        /*
        globalFrames[i].copyTo(trackingFrame);

        for(int v = 0; v < trackingFrame.rows; v++)
            for(int j = 0; j < trackingFrame.cols; j++)
                trackingFrame.at<uchar>(v,j) = 0;
        */
    }
}

//method to process exit of software
bool processExit(VideoCapture capture, clock_t t1, char keyboardClick)
{
    //if escape key is pressed
    if(keyboardClick==27)
    {
        //display exiting message
        cout << "Exiting" << endl;

        //compute total run time
        computeRunTime(t1, clock(), (int) capture.get(CV_CAP_PROP_POS_FRAMES));

        //delete entire vector
        globalFrames.erase(globalFrames.begin(), globalFrames.end());
        //report file finished writing
        cout << "Finished writing file, Goodbye." << endl;

        //exit program
        return true;
    }

    else
    {
        return false;
    }
}

//while buffering
void buffer()
{
}

//main method
int main() {

    //display welcome message if production code
    if(!debug)
        welcome();

    //creating initial and final clock objects
    //taking current time when run starts
    clock_t t1=clock();

    //random number generator
    RNG rng(12345);

    //defining VideoCapture object and filename to capture from
    VideoCapture capture(filename);

    //collecting statistics about the video
    //constants that will not change
    const int NUMBER_OF_FRAMES =(int) capture.get(CV_CAP_PROP_FRAME_COUNT);
    FRAME_RATE = (int) capture.get(CV_CAP_PROP_FPS);
    FRAME_WIDTH = capture.get(CV_CAP_PROP_FRAME_WIDTH);
    FRAME_HEIGHT = capture.get(CV_CAP_PROP_FRAME_HEIGHT);

    writeInitialStats(NUMBER_OF_FRAMES, FRAME_RATE, FRAME_WIDTH, FRAME_HEIGHT, filename);

    // declaring and initially setting variables that will be actively updated during runtime
    int framesRead = (int) capture.get(CV_CAP_PROP_POS_FRAMES);
    double framesTimeLeft = (capture.get(CV_CAP_PROP_POS_MSEC)) / 1000;

    //creating placeholder object
    Mat placeHolder = Mat::eye(1, 1, CV_64F);

    //vector to store execution times
    vector <string> FPS;

    //string to display execution time
```

```
string strActiveTimeDifference;

//actual run time, while video is not finished
while(framesRead < NUMBER_OF_FRAMES)
{
    clock_t tStart = clock();

    //read in current key press
    //char keyboardClick = cvWaitKey(33);

    //create pointer to new object
    Mat * frameToBeDisplayed = new Mat();

    //creating pointer to new object
    Mat * tmpGrayScale = new Mat();

    //reading in current frame
    capture.read(*frameToBeDisplayed);

    //for initial buffer read
    while(i < bufferMemory)
    {
        //create pointer to new object
        Mat * frameToBeDisplayed = new Mat();

        //creating pointer to new object
        Mat * tmpGrayScale = new Mat();

        //reading in current frame
        capture.read(*frameToBeDisplayed);

        //adding current frame to vector/array list of matricies
        globalFrames.push_back(*frameToBeDisplayed);

        //convert to gray scale frame
        cvtColor(globalFrames[i], *tmpGrayScale, CV_BGR2GRAY);

        //save grayscale frame
        globalGrayFrames.push_back(*tmpGrayScale);

        //initilize Mat objects
        initilizeMat();

        buffer();

        //display buffer progress
        if(!debug)
            cout << "Buffering frame " << i << ", " << (bufferMemory - i) << " frames remaining." << endl;

        //display splash screen
        welcome();

        //incrementing global counter
        i++;
    }
```

```
//adding current frame to vector/array list of matricies
globalFrames.push_back(*frameToBeDisplayed);
Mat dispFrame;
globalFrames[i].copyTo(dispFrame);
putText(dispFrame,to_string(i),Point(0,50),3,1,Scalar(0,255,0),2);

//display raw frame
displayFrame("RCFrame", globalFrames[i]);

//convert to gray scale
cvtColor(globalFrames[i], *tmpGrayScale, CV_BGR2GRAY);

//save gray scale frames
globalGrayFrames.push_back(*tmpGrayScale);

//gather real time statistics
framesRead = (int) capture.get(CV_CAP_PROP_POS_FRAMES);
framesTimeLeft = (capture.get(CV_CAP_PROP_POS_MSEC)) / 1000;

//clocking end of run time
clock_t tFinal = clock();

//calculate time
strActiveTimeDifference = (to_string(calculateFPS(tStart, tFinal))).substr(0, 4);

//display performance
if(debug)
    cout << "FPS is " << (to_string(1/(calculateFPS(tStart, tFinal)))).substr(0, 4) << endl;

//saving FPS values
FPS.push_back(strActiveTimeDifference);

if(i > bufferMemory + mlBuffer + 1)
{
    String tmpToDisplay = "Running Image Analysis -> Frame Number: " + to_string(i);
    welcome(tmpToDisplay);
}

cout << "BEGIN CV" << endl;

//running computer vision
objectDetection(FRAME_RATE);

cout << " END CV " << endl;

/*
if(i >= bufferMemory + 14)
{
    cout << " CLICK TO CONTINUE 1" << endl;
    //waitKey(0);
    cout << " CONTINUING 1" << endl;
}
*/

trackingML();
```

```
        /*
        if(i >= bufferMemory + 14)
        {
            cout << " CLICK TO CONTINUE 2" << endl;
            //waitKey(0);
            cout << " CONTINUING 2" << endl;
        }
         */
        //display frame number
        cout << "Currently processing frame number " << i << "." << endl;

        //method to process exit
        //if(processExit(capture,  t1, keyboardClick))
            //return 0;

        //deleting current frame from RAM
        delete frameToBeDisplayed;

        //incrementing global counter
        i++;

        waitKey(30);
    }

    //delete entire vector
    globalFrames.erase(globalFrames.begin(), globalFrames.end());

    //compute run time
    computeRunTime(t1, clock(),(int) capture.get(CV_CAP_PROP_POS_FRAMES));

    //display finished, promt to close program
    cout << "Execution finished, file written, click to close window. " << endl;

    //wait for button press to proceed
    waitKey(0);

    //return code is finished and ran successfully
    return 0;
}
```