Pedestrian Detection Using Camera and LiDAR data

Weekly meeting

Every Thursday 12pm with 2 critical stakeholders - Mr. Barth and Prof. Fromm.

Teammembers and Tasks

Name	Matriculation Number	Email	Development Task	Management Task
Sandeep Raju	762595	sandeep.raju@...	Developer	Requirements Manager
Sameera Bhavani Rao Chilmattur	762527	sameera.b.r.chilmattur@...	Developer	Project Manager
Nicolas Ojeda Leon	762582	nicolas.leon@...	Developer	Test Manager
Bharath Ramachandraiah	762596	bharath.ramachandraiah@...	Developer	Quality Manager

Meeting Minutes

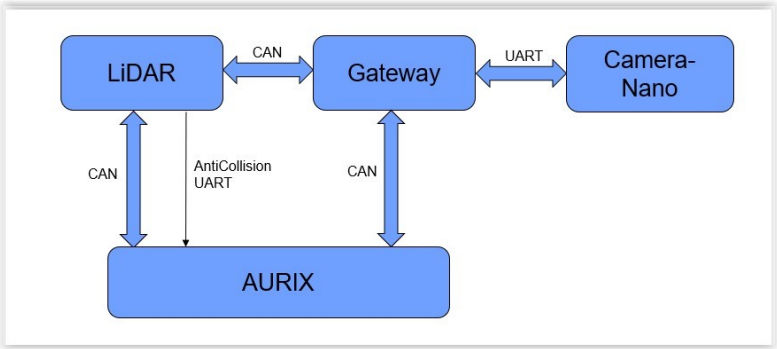
Will be tracked and updated under Scrum board. All information captured in Vivify Scrum board, Action Points updated. https://app.vivifyscrum.com/boards

Requirements

The requirements document is updated here:

http://project.eit.h-da.de/studentcar/studentcar/20_Projects/480_NVidia_Nano/20_Requirements/

System Architecture



Components Interface Design

Design Goals :

- Common interface between Camera-Nano and Maze team projects
- Modular and service-based implementation of features
- New hardware connected to car should be through CAN interface

Initial idea : LiDAR detecting obstacle and requests Nano to check if it is pedestrian

- Reduces overall power consumption
- Increases load on PSoC
- Not modular and not scalable

Final Design : Based on discussion within team and with stake-holders, the design was revised to accomodate addition of features / change in interfaces of devices in future. In final design, as shown in the system architecture diagram above, AURIX independently queries LiDAR and Camera-Nano for Pedestrain data. Based on these data, AURIX can decide on next steps. Below are salient features of our design -

- Common interface for both Camera-Nano and MAZE camera's through introduction of Gateway
- Single code-base on Gateway with option to select either camera variant
- Camera data and LiDAR data can be seperately queried by AURIX application via CAN
- No design impact for camera change and multiple cameras addition
- Extendibility : More features can be added in AURIX app / Nano

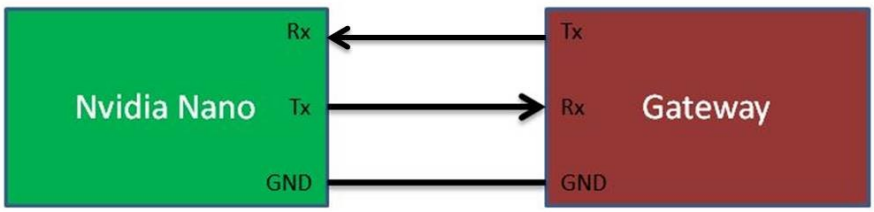
The responsibilities of each component in our design is as follows -

- **LiDAR** : Scans for obstacles in the range of 250cm to 800cm and same is passed on to gateway in cyclic manner.
- **Camera-Nano** : Continuously scans the environment for pedestrian detection. It continuously sends detection data to Gateway.
- **Gateway** : Aggregates data from both LiDAR and Camera-Nano modules. The latest aggregated data can be queried from Gateway via CAN
- **AURIX** : Application that makes use of pedestrian detection data shall run on AURIX

High Level Design: Camera Nano Application Software

High level interfaces to outside world

The NVidia board is connected to Gateway to transmit the detected information via UART at 115200 baudrate.



Protocol information is depicted below:

The byte information is as follows:

- SOP - 1 byte - 0xFA - Start of Protocol
- DLC - 1 byte - 0x04 - Data length code
- Theta - 1 byte - 0xXX - Angle at which subject is present
- Delta Theta - 1 byte - 0xXX - delta angle of the subject position
- Reserved - 2 byte - 0xXXXX - Reserved bytes
- EOP - 1 byte - 0xFD - End of Protocol

Class Diagram

CApplication is the main class of the Pedestrian Detection Application.

There is a single process - 5 threads in the Nvidia nano which are implemented as classes:

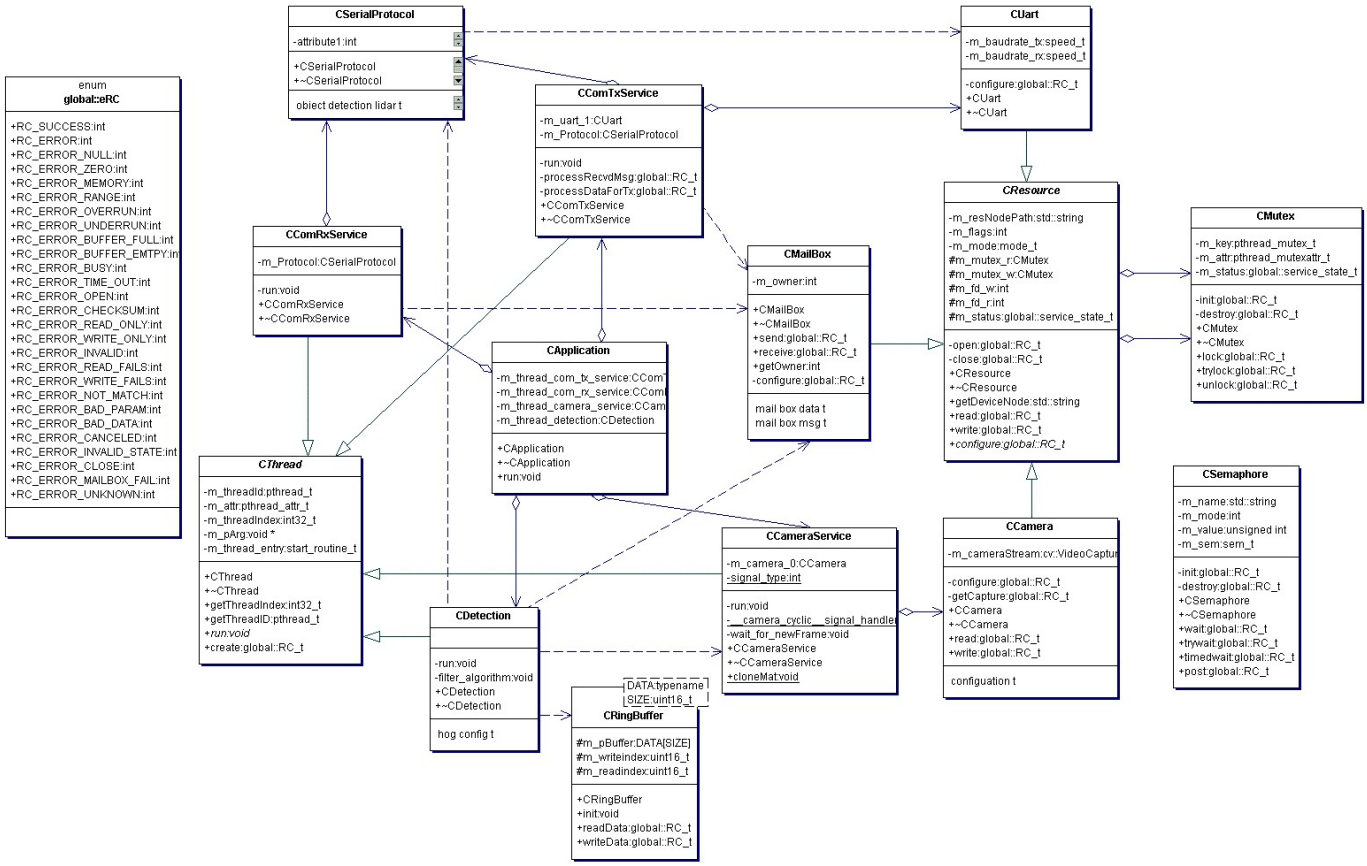
- **CDetection** : Used to detect the pedestrian using the detection algorithms, filters false positives, extract the information such as Angle and Delta Angle of the subject and sends the information for transmission.
- **CCameraService** : Used to capture the image frame, using gstreamer, and store it into a ring buffer for further processing .
- **CComRxService** : Used to receive information from the external world. A single thread for the reception.
- **CComTxService** : Used to Transmit the information via different peripherals such as UART, SPI, Ethernet, Wifi. A single thread for the transmission.
- **Background** : Background thread.

All the threads inherit the base class CThread.

Other important classes

- **CThread** : Thread creation class which employes pthread library.
- **CMailbox** : Inter-thread communication - Mailbox for each thread to transmit and receive information from other threads.
- **CCamera** : Initialises and configures the Camera module for video capturing - hardware specific.
- **CUart** : Initialises the uart port (used in tty) for the Transmission and Reception at Baudrate 115200.
- **CResource** : The class which interacts with the hardware resources for reading and writing from the hardware. Eg. UART, Camera, etc,.
- **CMutex** : The class to protect the resource from concurrent access by multiple threads.

More about the class can be found in Doxygen report http://project.eit.h-da.de/studentcar/studentcar/20_Projects/480_NVidia_Nano/40_Implementation/Release/NvidiaNano/Docs/Doxygen/html/index.html



Class diagram explanation

The class design was based on the principle of Services. Following this project wide philosophy, the idea behind this principle is to design in such a way that the components (for example the camera) could be used by several modules so it should not be tightened to any module but instead provide an independent interface which allows extendibility aiming to add more modules using the same component.

The first part of the system comprises the system related components, this is, all the hardware and OS components that provide the backbone of the application.

Considering the API provided by Linux to access such components, a standard class CResource was created to deal with this "file like" access to drivers as well as to the OS artefacts. This base class provides the opening, closing and IO functionality of the components with the critical section's protection implemented with the use of mutexes. The writing and reading capabilities were separated (implying 2 file descriptors and 2 mutexes) to support a wider variety or components, for instance the mailbox, which was implemented using Pipes, requires 2 streams in order to enable the simultaneous operations.

In a similar way, the UART and Camera components (handling the hardware drivers) are also system resources and thus inherit from the standard CResource which already provides the basic functionality and just a specific configuration is to be done for each individual case. To handle the specific configuration, a pure virtual method is declared in CResource to make sure the derived classes implement their own configuration.

On top of the hardware and OS components, several services are defined: ComTxService?, ComRxService? and CCameraService. These 3 services are intended to offer the hardware components as software for the application to consume. Services imply they are not synchronous with the application but instead run separately and provide information whenever required.

The Com Service relies on the class CSerialProtocol which defines the structure of the communications (the protocol).

The class CDetection is the actual application specific class implementing the pedestrian detection algorithm (HOG). Associated to this class, the CRingBuffer class provides a ring buffer to store frames in a sequential way and retrieves the most recent frame on every read operation.

All those classes that are intended to operate as threads, either because they provide a service or any other reason, extend the CThread class which contains the basic functionality of a Linux thread. Similar to what was done with CResource, CThread is intended to handle all the thread's common details. As explained in the Tasks architecture, the classes that run as threads are CDetection, CCameraService, CComTxService, CComRxService and CApplicaiton.

For inter-thread communication, the class CMailbox was developed. This class provides a Mailbox based service in which a thread that needs to receive messages from other thread, instantiates a mailbox and receives messages on it.

Finally CApplication is the background thread and basic class which spawns all the application's threads and then keeps on waiting to maintain the process active.

Nano application

As explained in the class diagram section, the services and the application itself are to run in parallel, for this reason 3 threads were defined, plus the background thread.

The services offered by the application, the ones encapsulating hardware drivers and OS artefacts are running as threads as they do not depend on any other component but are always either active performing a task or awaiting the request from other thread. On the other hand, the application thread searching for detected pedestrians is to run continuously and as fast as possible to achieve the highest frame rate possible.

The design is as follows:

http://project.eit.h-da.de/studentcar/studentcar/20_Projects/480_NVidia_Nano/10_Projectmanagement/50_Documentation/TasksDiagram.png

The Vision algorithms (pedestrian detection in this case) are running on the Detection thread (T2). This thread is always running searching for results (detections) as fast as possible.

At the same time, the Camera Service is running on thread T3 acquiring frames from the camera hardware at the configured frame rate. Each time a new frame is available it is added to the Frame Buffer so that T2 can retrieve it.

Finally, on T1, the Com Service is running. This thread, on the contrary to the camera service, is not cyclically running or continuously running as T2 but is sleeping. The associated mailbox is pending and as soon as a message is received, it is executed and a protocol built and sent through the UART.

Thread description:

Now, after giving the big picture, a deeper description will be provided:

http://project.eit.h-da.de/studentcar/studentcar/20_Projects/480_NVidia_Nano/10_Projectmanagement/50_Documentation/Sequence.gif

The sequence diagram shows, in a simplified way, the actions and interactions that each thread has as a function of time.

T4 (Application): This thread is the background thread which keeps the process alive and, at an initial point, spawns all the other threads.

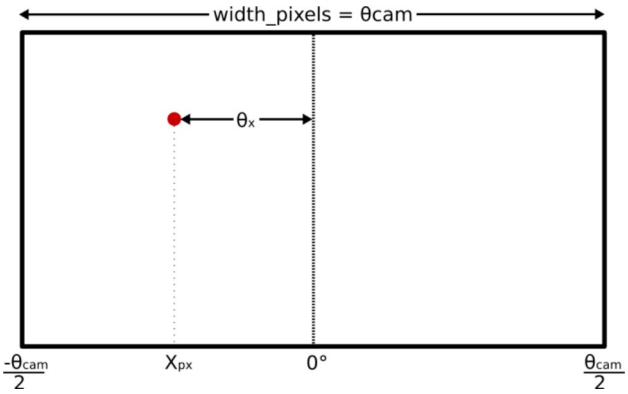
T3 (Camera Service): The camera service is running cyclically at the same frame rate the hardware is working. Every time a new frame is available (given by the sleep timer), the thread is awakened, and a frame is read from the Camera driver (gstreamer) and stored in the global Ring Buffer that holds the most recent frames. This buffer holds several frames but it always retrieves the most recent frame as it is the only important frame. Once the buffer is full, the oldest frame is overwritten by the newest.

T2 (Detection): The main application thread which runs the pedestrian detection algorithm. This thread runs continuously (non-stop) trying to achieve the highest frame rate possible. Since the detection algorithm is long running software, this thread needs to be run all the time. The thread first retrieves the most recent frame from the global buffer, then runs the Pedestrian Detection algorithm (based on HOG) and after having the detected elements, sends a message to the ComTxService?'s mailbox.

T1 (Com Tx service): This thread is normally blocked waiting for a message to arrive at its mailbox. Using the receive method of the Mailbox, a blocking function is called, which is released when a message is available at the mailbox. When a message arrives and the thread is executed, the detected pedestrians' information is encapsulated according to the Serial Protocol and then, after it is parsed to a raw array, sent over the UART port.

Angle of detected pedestrian

For determining the angle and the range spanned by the detected pedestrian, the 62° angle of the camera is to be considered in the following calculations:

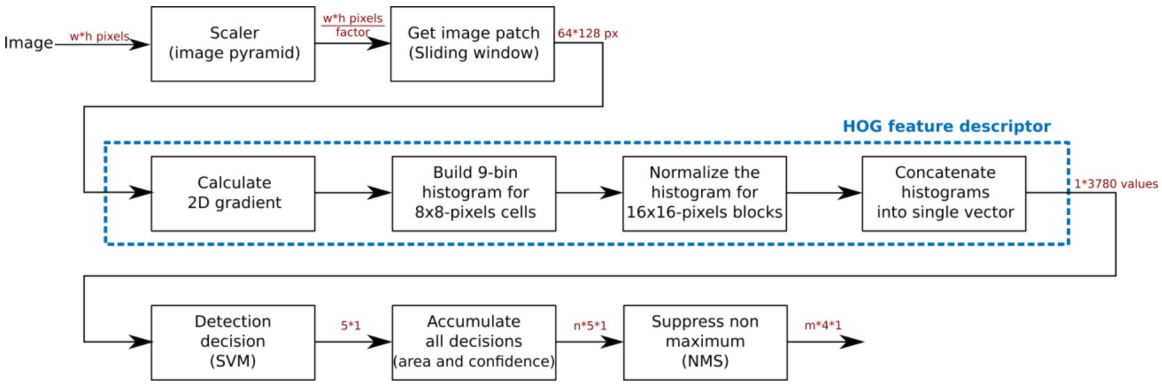


Using the previous diagram, the angle of a given point inside the frame can be calculated as:

$$\theta_x = -\frac{\theta_{cam}}{2} + \frac{\theta_{cam}}{width_{pixels}} * X_{px}$$

Pedestrian detection algorithm (HOG)

Focusing on the detection of pedestrian, the Histogram of Oriented Gradients (HOG), was used for the image processing. HOG is the most commonly used method for pedestrian detection, purpose for which Dalal and Triggs optimized the parameters as exposed in their technical paper "Histograms of Oriented Gradients for Human Detection". The basic idea behind the algorithm is to use the gradient (vectorial derivate) to make the borders stand out. As other Feature Descriptors, HOG's main objective is to make shapes clearer, reduce the impact of illumination differences and deliver a compact representation of the data for further analysis and decision.



OpenCV provides an implementation of the algorithm including the SVM. At the end, by using the provided API, a vector of detections is generated. Nevertheless, it is necessary to understand the internal algorithm in order to tune its parameters.

Block description:

Scaler: The first step in the process is to scale the image (starting with the original size and then shrinking). This is accomplished by an Image pyramid by using the given factor.

Image patch: The HOG works on a predefined image patch of 64x128 pixels, so an sliding window is to be used moving the analysis area all over the image, for each of the steps in the pyramid.

Calculate 2D gradient: Being the heart of the algorithm, in this step the vertical (y) and horizontal (x) derivate, both magnitude and phase, is calculated for each pixel in the window.

Build histogram: Taking the pixels' gradients and segmenting the window in 8x8 pixels cells, build up a histogram of 9 bins. Each bin represents 20° phase. Only 20° because an "unsigned phase" is used; this means phases from 0 to 180 and the remaining 180 degrees are aliased back. The idea behind the unsigned phase is that the direction of the change makes no differences; if the transition is from dark to light or light to dark, the important thing is just the change. For each pixel's gradient, the phase is used to determine a percentage of contribution to 2 bins (the closest bins to the pixel's phase) with the idea of having the biggest contribution to the closest bin. Using the percentage, the magnitude is divided in the 2 bins and after performing the calculation for all the beans in the cell, the histogram is obtained.

Normalize the histogram: For each block of 16x16 pixels, that is 2x2 cells, the 36 bins (9 for each block) are concatenated and the L2 norm is applied to get a normalized histogram. This is done aiming to reduce the effect of illumination differences.

Concatenate histograms: The normalized histograms from each block (for all the combinations of cells) are concatenated into a one dimension array. Since the window has 64x128 pixels and each cell is 8x8 pixels wide, there are 8x16 cells in a window. Each block comprises 2x2 contiguous cells, so there are 7x15 blocks (considering that each block overlaps by 1 with the previous block). That yields 7x15x36 = 3780 values.

Detection decision (SVM): The concatenated histograms are fed to a Support Vector Machine to determine if the analyzed windows contains or not a person. A SVM is a classification mechanism, which divides categories (usually 2) by a hyperplane. It is a supervised training method described by a set of coefficients (hyperplane) and a simple classification is achievable by determining in which side of the hyperplane the point lays.

Accumulate decisions: After running the HOG on all the possible windows for all the scales of the image, the positive detected windows are accumulated (buffered).

Suppress non maximum (NMS): The algorithm generates several hypothesis regions and usually, for a positive detection, many results arise which, naturally, overlap each other. The NMS merges all those overlapping areas leaving the biggest one, which contains all the overlapping regions.

Open cv provides the following API for detecting multiple pedestrians in a frame:

```
hog.detectMultiScale(   image,                  /* Source image */
                        detections,             /* foundLocations, vector of Rect objects with the boxes where a person was detected */
                        detection_weights,      /* Weights of each detection. Vector of same dimension as previous parameter */
                        hitThreshold,           /* hitThreshold: SVM threshold to filter final results */
                        winStride,              /* Windows stride: Horizontal and vertical step in pixels for the template matching process (object of type opencv::Size) */
                        padding.value,          /* Padding: Outer padding in the 4 edges of the image (object of type opencv::Size) */
                        scale,                  /* Scale: Scale stride for the image pyramid. Factor in the form of 1.xx */
                        finalThreshold          /* FinalThreshold */
                );
```

Before using the detectMultiScale method, a classification model for the SCM has to be set. OpenCV provides 2 pretrained models: the default model, trained with the INRIA dataset, and the Daimler model, trained with 48x96 pixels patches and using a bigger dataset. For setting the model the following method need to be called:

```
hog.setSVMDetector(HOGDescriptor::getDefaultPeopleDetector());
```

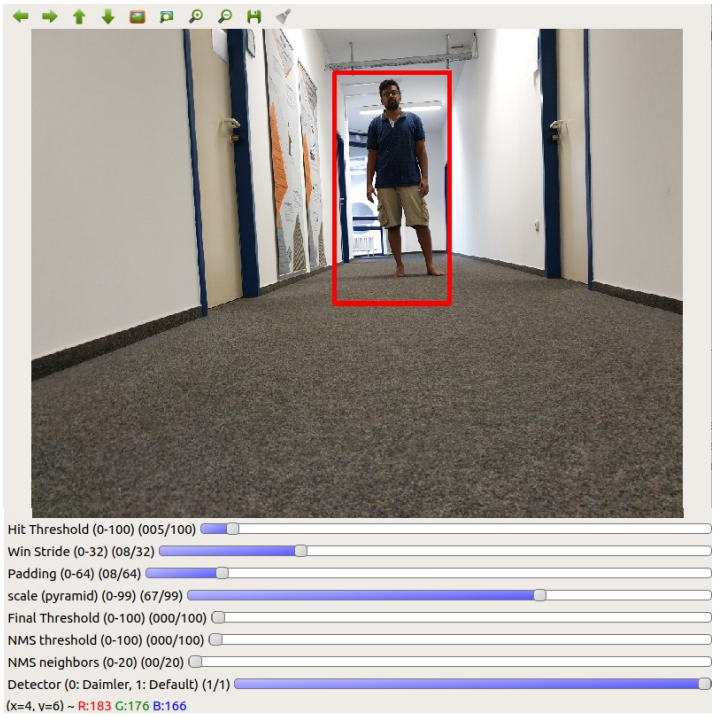
Or

```
hog.winSize = Size(48, 96);
hog.setSVMDetector(HOGDescriptor::getDaimlerPeopleDetector());
```

For this last case, a change in the window size needs to be performed, because Daimler's model uses a non defaut window size.

After several tests with both of the models, the conclusion drawn was that even though the Daimler model has a better positive detection, in that it can find even partially visible people, it produces so many false psotiives that the subsequent filtering becomes very hard. For this reason, the default model was choosen and the parameters tuned to get the best result out of it.

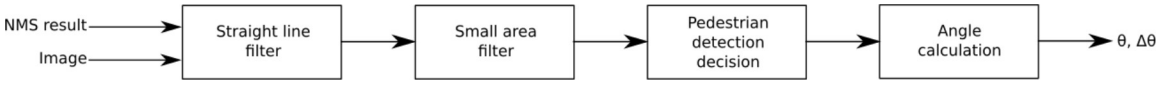
Looking to have a fast and easy to use tool for parameter tuning, an application was developed providing a GUI with trackbars that allow to modify each parameter's value individually and assess the result in a graphic representation. An example of such a process is given in the following image:



Using this application, multiple testing and tuning sessions were conducted to find a configuration that provided the best possible results. Furthermore, the same principle was used (similar GUI) on the Nano to provide a way of modifying the parameters directly using the final harware and, in realtime, achieve a suitable configuration for the detection.

Result filtering

Finally, after getting all the possible detections from the NMS, a filtering is performed to further decide if a pedestrian is present or not.



Due to the fact that HOG gets easily tricked by straight lines, behavior observed during testing, a Straight line filter was added to discard all those areas with a certain number (6) of straight lines present. To achieve this, another image processing algorithm, available in OpenCV, was used, the Hough Lines algorithm. This method, identifies straight lines in the given image. Since it operates on gray images, a conversion is performed previously and also the Canny algorithm is applied in order to leave only the borders so that the line detection is more efficient.

Those areas with a positive detection, even after filtering out the areas containing many straight lines, an area filter is applied to rule out those detections of very small size which very likely correspond to false positives. In case of a pedestrian getting discarded by the area filtering, since it is very small area, it would be a pedestrian standing far away of the car.

Following the 2 filters, the final decision is made on the remaining hypothesis by taking the biggest area, which would be the pedestrian standing closer to the car. Having the final decision, the angle calculation (as explained before) is done. After this process, the result consists of an angle (θ) together with the spanning angle of the detection (Δθ).

Software Package: Camera Nano Platform

Linux Distribution

Getting started with the NVidia Nano - set up

https://developer.nvidia.com/embedded/learn/get-started-jetson-nano-devkit#intro

Download the package from

https://developer.nvidia.com/jetson-nano-sd-card-image-r322

Custom build linux (only for the expert users)

http://nv-tegra.nvidia.com/gitweb/?p=linux-4.9.git;a=summary

Packages

- OpenCV - 3.4
- cmake
- pthread
- man pages

The packages can be downloaded to board using the linux commands such as:

```
sudo apt-get install cmake
```

High Level Design: LiDAR Application Software

LiDAR is used in this project to act as additional check and minimize false positives in final result of pedestrian detection. The application running on AURIX can intelligently use LiDAR data for sensor fusion.

Implementation:

- Implemented as separate service and not along with anti-collision
- The detection service is called cyclically by RTOS
- The service provides obstacle distance and angle in relevant ROI
- The detection info is sent to Gateway as response to SDO request
- The design is kept modular and independent from other services so that additional processing, if required, can be added in future.

Testing:

- Simulating AURIX query through PCAN viewer to Gateway
- Gateway sends SDO requesting data to LiDAR
- LiDAR responds to request with latest scan-based detection info
- Gateway forwards results to AURIX (simulated via PCAN viewer)

Future scope of work:

- More robust algorithms to detect human can be run on LiDAR data
- Multiple objects detected and their widths can be sent by expanding the existing protocol

Software Package: LiDAR

http://project.eit.h-da.de/studentcar/studentcar/20_Projects/480_NVidia_Nano/40_Implementation/Release/Lidar

High Level Design: Gateway Application Software

Gateway software was designed so that we achieve below goals -

- Act as Primary interface to AURIX for Camera / LIDAR data retrieval
- Act as both CANOpen Client and Server
- Periodically recieve and maintain latest camera and LiDAR data
- Support separate CANOpen indices for Camera and LiDAR data query
- Common code framework across student car projects

Testing:

Simulating AURIX query for LiDAR data and Camera data through PCAN and verify through manual inspection and in PCAN. Please check tutorial video for better understanding.

Challenges faced:

- CANOpen SDO request for LiDAR data fails after few attempts
- root-cause: SDO dispatcher full due to non-deletion of entries
- solution: periodic call of CANopen_Client_Tick() needs to be present

Software Package: Gateway

http://project.eit.h-da.de/studentcar/studentcar/20_Projects/480_NVidia_Nano/40_Implementation/Release/Gateway

Pedestrian Detection System Testing

http://project.eit.h-da.de/studentcar/studentcar/20_Projects/480_NVidia_Nano/50_Quality/20_Tests/

Links to latest Software release packages

- Nvidia Nano : **http://project.eit.h-da.de/studentcar/studentcar/20_Projects/480_NVidia_Nano/40_Implementation/Release/NvidiaNano**
- Gateway : **http://project.eit.h-da.de/studentcar/studentcar/20_Projects/480_NVidia_Nano/40_Implementation/Release/Gateway**
- LiDAR : **http://project.eit.h-da.de/studentcar/studentcar/20_Projects/480_NVidia_Nano/40_Implementation/Release/Lidar**

NOTE: Please check the software compatibility versions

Tutorial Video and documents

Pedestrian detection: https://youtu.be/yvu5OiZNAi8

Unit tests for LiDAR / Camera data via CAN: https://youtu.be/dXqr_r2ndgk

Hardware assembly and demo - http://project.eit.h-da.de/studentcar/studentcar/20_Projects/480_NVidia_Nano/40_Implementation/Release/Assembly%20tutorial.pdf

Attachments (16)

Last modified on Oct 30, 2019, 10:05:50 PM