Tester

För att se till att koden håller den kvalité som krävs, och att olika funktioner och delar av prototypen faktiskt gör det som är eftersökt har vi arbetat en del med tester. I vissa fall har detta inneburit att skriva regelrätta testklasser till koden. I andra fall har det inneburit att kontinuerligt testa det vi utvecklat med hjälp av en mockup i simulatorn tills det gick att testa koden med MOPEDen. Bland annat är ACCn utvecklad genom att kontinuerligt testa lösningarna, vilket är varför det inte finns några skrivna tester kring denna koden.

Anledningen till att vi valde att använda en mockup i simulatorn var så att vi kunde få respons tidigt på om det vi utvecklade faktiskt fungerade, eftersom det var mycket problem med MOPEDen i början av projektet.

Ett annat sätt vi har testat vår kod är med verktyget FindBugs. Med hjälp av FindBugs har vi letat reda på buggar och svaga punkter i vår prototyp och korrigerat dem. Viktigt att notera är att även delar av den kod som vi tagit från andra källor och alltså inte skrivit själva också är analyserad med hjälp av FindBugs.

Kommentarer på FindBugs för canJava:

- Bad practice
    - Detta uppstår i en klass som inte vi skrivit. Om den tas bort fungerar inte koden som tänkt.
- Experimental
    - Detta uppstår i en klass som vi inte skrivit.
- Dodgy code
    - Detta uppstår i en klass som vi inte skrivit. Står “Useless object” men om kodstycket tas bort fungerar inte koden som tänkt.

Kommentarer på FindBugs för OveControl:

- Bad practice (x25)
    - Detta uppstår i en klass som vi inte skrivit. Beror på att klassen *R* innehåller flera klasser och dessa klassnamn börjar inte med stor bokstav. Klassen är låst och vi kan inte ändra den.
- Dodgy code
    - Variablerna behöver vara *static* för att koden ska fungera.

Det finns även element av prototypen som har testats genom experiment, däribland kameran. Hur detta gick till behandlas nedan.

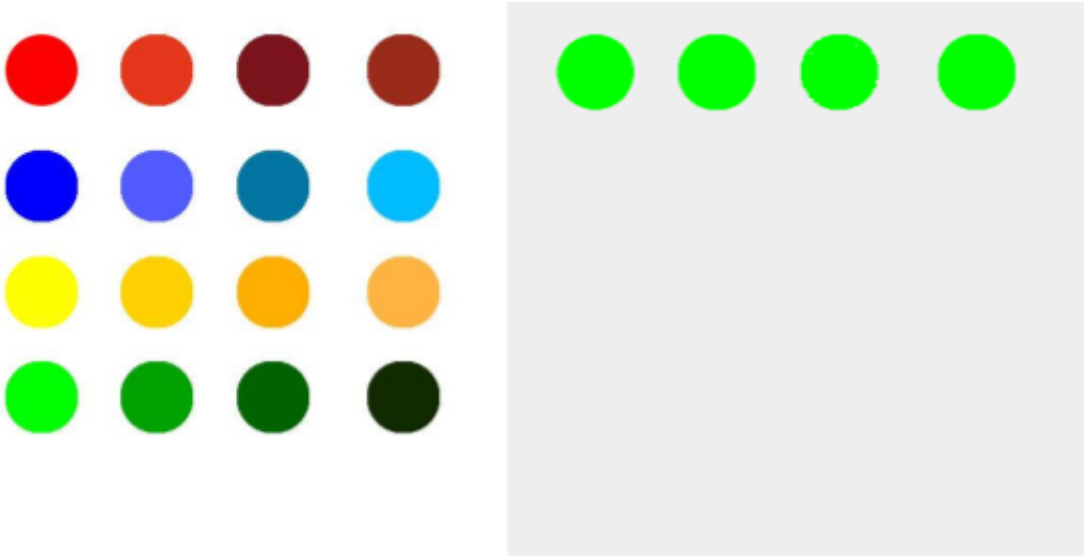
Experiment med kameran

De slutgiltiga villkoren är resultatet av tester där till slut ett tillräckligt högt antal bilder av cirklar kunde upptäckas. Det var en balansgång mellan prestanda och noggrannhet där resultatet är det vi bedömde fungerade bäst. För prestandan satte hårdvaran gränser då det begränsade antalet bilder per sekund och hur snabbt dessa kunna processas och sedan hur snabbt MOPEDen kunde reagera på den bearbetade informationen från kamerabilderna.

Det fanns två utmaningar, där den ena var att lösa ett färgintervall som vår röda cirkel (utskrivet på ett A4-papper hållandes framför MOPEDen) kunde uppfattas som genom MOPEDens kamera. Andra delen var att avgöra om formen av de figurer i angiven färg som upptäcktes faktiskt var cirkeln vi var ute efter. Ibland kunde det finnas röda saker i bakgrund som upptäcktes, men som inte MOPEDen skulle reagera på då det var fel figur. Sedan skulle även situationen om det råkade upptäckas två cirklar av rätt färg, då fick fordonet agera på den största av dem.

Experiment 1

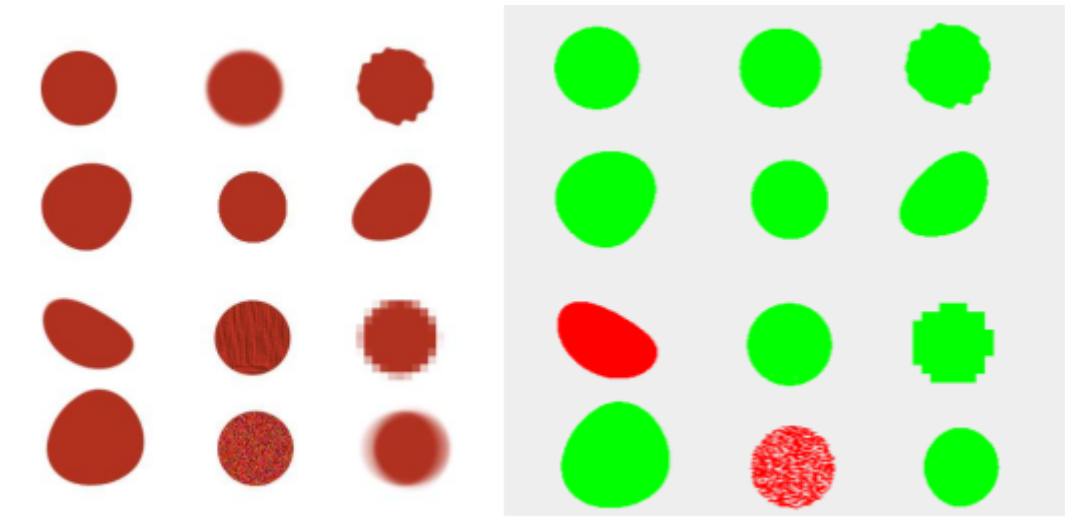
Ett första test gick ut på att se att färgen röd upptäcktes av *ImageDetection*. I och med att många olika ljusförhållanden kan uppstå så blir det även lite olika röda färger som måste kunna hanteras. Allt från mörkrött till ljusrött, och lite andra nyanser när det inte finns så mycket ljus för MOPEDens kamera så det bildas skuggor på vår röda cirkel. I *Figur 1* ett visas ett antal cirklar i olika färger. Den översta raden är lite olika röda färger och det är dessa som markeras som röda i resultatet med grön färg (*Figur 2*). Detta test visade att våra villkor verkade vara i rätt intervall för att kunna upptäcka mer än bara “en perfekt röd färg”. De resterande blåa, gula och gröna cirklarna visades inte i resultatet, vilket är korrekt.



Figur 1 - 2

Experiment 2

Ett annat test gick ut på att testa cirkelformen. Det hade inte varit realistiskt att endast perfekta cirklar skulle passera testet då det inte kommer vara sådana MOPEDen kommer hantera, speciellt inte när fordonet framför svänger och en bild av en ellips kommer visas. Kamerans skärpa och upplösning är inte heller perfekt vilket gör att vi måste ha med lite feltolerans i bild-bedömningen för att kunna använda denna teknik för MOPEDens styrning. Nedan i *Figur 3* visas först testbilden i form av olika “nästan cirkel-formade” figurer. *Figur 4* visar resultatet av *ImageDetection* där man kan se att de flesta av figurerna passerade cirkel-testet (grönmarkerade). De rödmarkerade figurerna passerade färgtestet, men inte cirkelform-testet.



Figur 3 - 4

Experiment 3

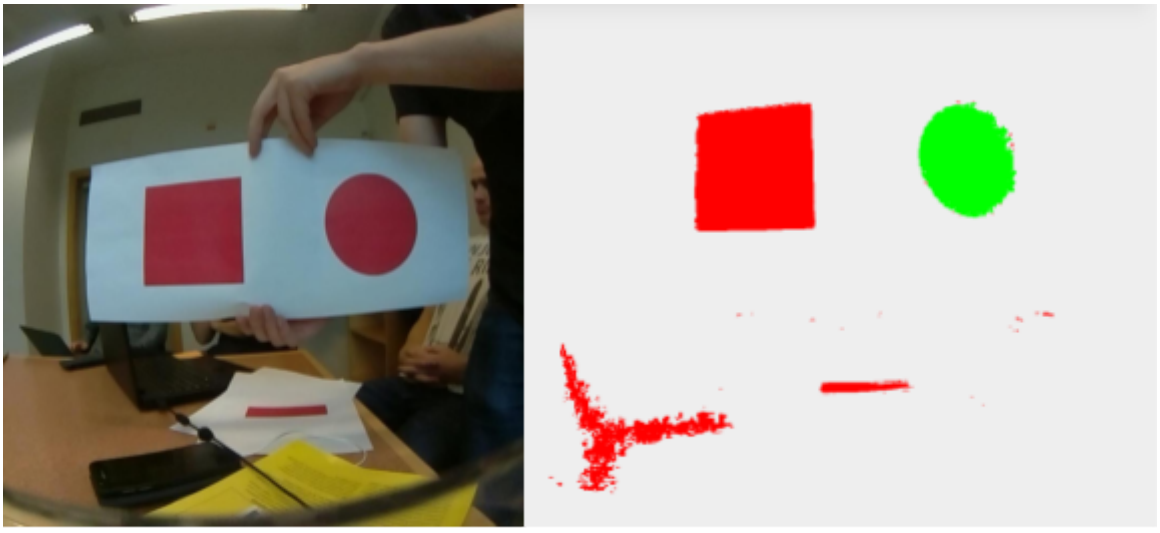
Nästa test gick ut på att testa om *ImageDetection* kunde urskilja cirklar från andra geometriska figurer. I detta experiment jobbade vi med perfekta cirklar eller liknande cirkelformade figurer och testet gick ut på att andra former, trots t.ex. samma area, inte skulle markeras som en cirkel av *ImageDetection*. *Figur 5* visar bilden vi testade med olika geometriska figurer. *Figur 6* visar det tänkta resultatet där de cirklar (eller liknande nog cirklar), som borde upptäckas av programmet, är utmärkta med pilar. *Figur 7* visar resultatet av programmet efter den bearbetat alla figurer. Om en figur upptäcks med både rätt färg och form (en röd cirkel) så märks de upp med grönt. Resterande av figurerna har markerats röda vilket betyder att de har rätt färg men fel form!

Figur 5 -7

Experiment 4

Nästa test var med bilder ifrån den faktiska kameran för att kunna se om det gick att applicera *ImageDetection* på de bilder som MOPEDen producerade. Figur 8 visar bilden från kameran, där det finns 3 röda figurer, men bara en av dem är en cirkel. *Figur 9* visar resultatet ifrån *ImageDetection* där den röda cirkeln har märkts ut med grönt, vilket är korrekt! Det går även att se hur de andra figurerna har upptäckts, men att de inte klarade “cirkel-testet” och bara markerades med rött. Det går också att se att en viss del av bordet har markerats eftersom dess färg verkar falla inom de villkor vi satt upp. Här kan man se

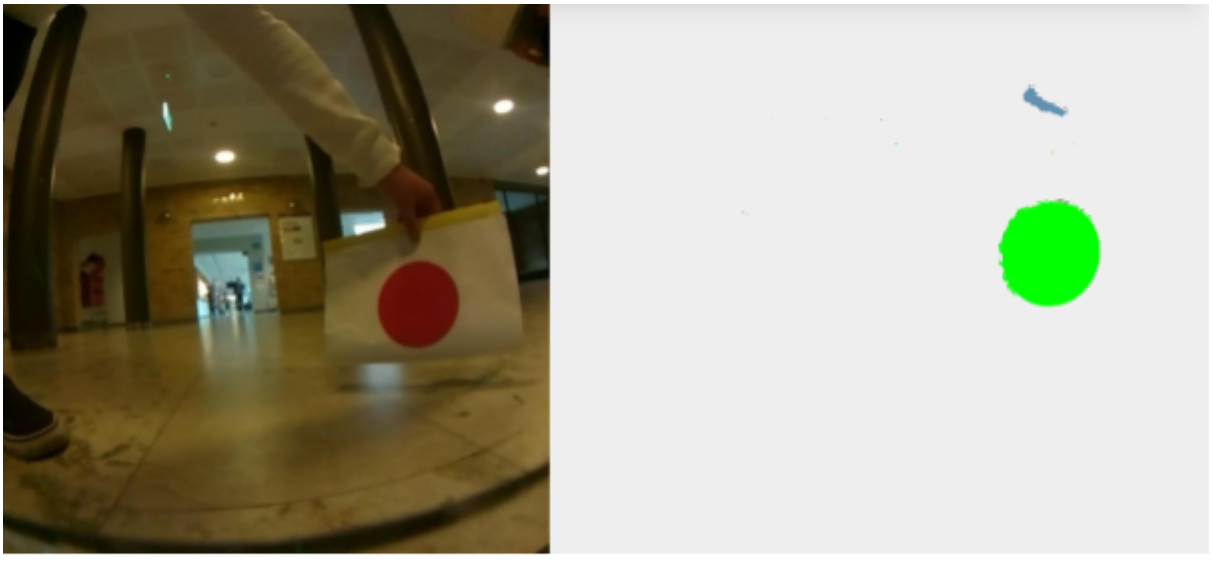
varför man behöver cirkel-testet också för att kunna använda *ImageDetection* för styrning av MOPEDen.



Figur 8-9

Experiment 5

I nästa test var MOPEDen på golvet med inomhusbelysning. I *Figur 10* kan man se utmaningen i “färg-testet” eftersom ljusförhållanden påverkar den röda färgen. *Figur 11* visar resultatet av *ImageDetection,* vilket verkar vara korrekt! Den röda cirkeln har markerats med grönt.

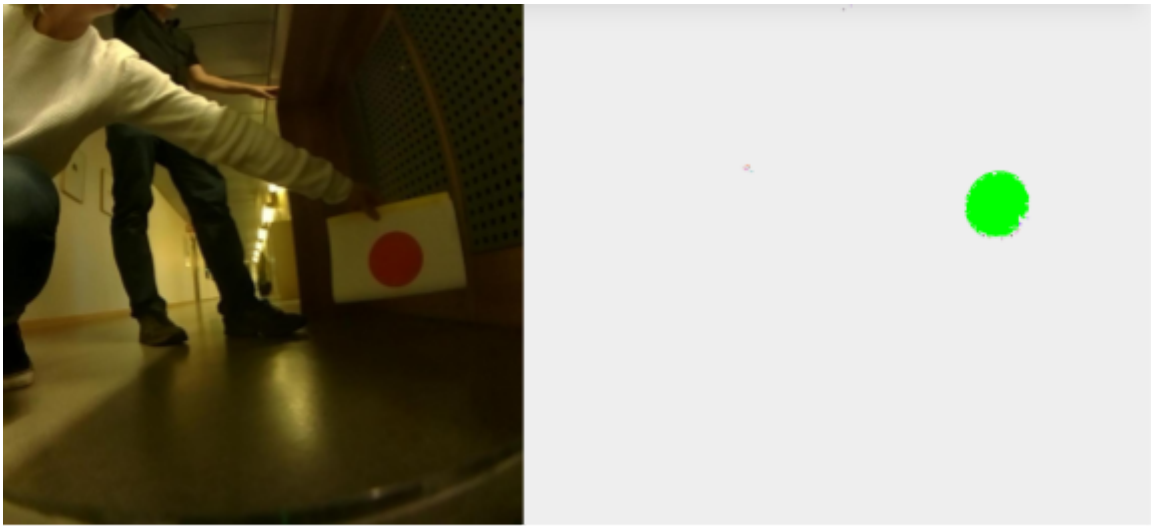


Figur 10 - 11

Experiment 6

I nästa test fick *ImageDetection* försöka hitta den röda cirkeln i sämre ljusförhållanden. För att lyckas med detta krävdes noggrannare justeringar av tillåtna röda färger i *ImageDetection* Dock går det inte att sätta villkoret för det röda spektrat för brett då det leder till att för många röda figurer i en bild upptäcks och gör det helt omöjligt att urskilja specifika former, t.ex. cirkeln som vi är ute efter.

I *Figur 13* kan man se hur inte hela kanten av cirkeln kommer med eftersom det blev för mörkt, men att *ImageDetection* fortfarande kan få med den eftersom den trots lite deformation kan klara “cirkel-testet”.



Figur 12 - 13

Experiment 7
I nästa test testades kameran med dagsljus. Det var lättare eftersom det inte bildas så mycket skuggor på vår röda cirkel utan att det är en klar röd färg som lätt passerar testet.

Design Rationale

Varför man har valt att strukturera koden som man har gjort.

API:er

Prototypen är implementerad med hjälp av en del API:er, främst PiCamera, MOPEDens API, och androids API. PiCamera API:et används eftersom det gav oss en färdig lösning för att hämta ut bilderna från kameran på MOPEDen. MOPEDens API används för att kunna arbeta mot det gränssnitt som finns på MOPEDen.

Den app som har utvecklats för att manuellt kunna styra MOPEDen. Denna app är utvecklad med Android SDK version 25. Detta val har gjorts med bakgrunden att det är många mobiler som stödjer denna version.

Språk

Denna prototyp är utvecklad med hjälp av både Python och Java. Detta har vi gjort eftersom gruppen är mest kunnig inom Java, vilket är varför detta blev vårt huvudspråk. Givet den något tidspressade situationen valde vi sedan att implementera inläsningen av bilder från kameran med hjälp av ett api som vi hittat istället för att ta fram en egen lösning, och eftersom språket det APIet använde var Python används även detta språk i prototypen.

Sensorer (Kamera och avståndsmätare)

Kameran används till den del av *platooning* som går ut på att ligga direkt bakom fordonet framför. För att lösa problemet med att hålla kursen i sidled efter framförvarande fordon i platoon-läge valde vi att utnyttja att det fanns en kamera monterad på MOPEDen. Denna var från början riktad uppåt, vilket vi justerade så att den var riktad framåt. Detta för att vi med hjälp av bilder skulle kunna bedöma om MOPEDen framför ändrar sin kurs i sidled, varpå vår MOPED skulle kunna justera sin kurs så att den återigen följer rakt efter MOPEDen framför. För att lösa detta använde vi oss av en grafisk form (närmare bestämt en röd cirkel) fastsatt på fordonet framför för avgöra om den svänger. Valet av formen cirkel och färgen röd berodde på att vi trodde att denna figur skulle sticka ut från bakgrunden så MOPEDens kamera lätt kunde avgöra om den hade denna figur framför sig. Vi valde att utveckla en kameralösning på grund av att det var det enda alternativ med den tillgängliga indatan, som inte innebar mycket avancerade algoritmer.

Ultraljudssensorn som finns på MOPEDen kan bara ge avståndsdata, därför används den till att avgöra vad avståndet är till framförvarande MOPED, och till ACCen, som bedömer vilken hastighet MOPEDen ska ha.

Servers

Prototypen använder sig av två javaservers:

- En på MOPEDen, för enkel dataöverföring
- En extern, för bildöverföring och -analys

Den externa servern används till beräkningar och bildanalysering. MOPEDen, som kör Python, hämtar och skickar bilderna till den externa servern, som kör Java, för analyseringen av dem. När det sker på en extern server belastar det inte MOPEDen, så att den kan fortsätta få och skicka data om hur den ska åka.

När beräkningarna sedan är klara skickas de till mopedservern. Det är enklare att skicka datan till Python egentligen, men eftersom vi senare behöver datan i javamiljön på MOPEDen och vi inte hittade någon annan lösning på att skicka data från Python till java än att spara det till en fil. Denna lösningen bedömde vi skulle ta för lång tid, vilket är varför vi valde lösningen med servers istället. Anledningen till att vi behöver komma åt data i java på MOPEDen är att vi använder java för att få åtkomst till can-bussen på MOPEDen.

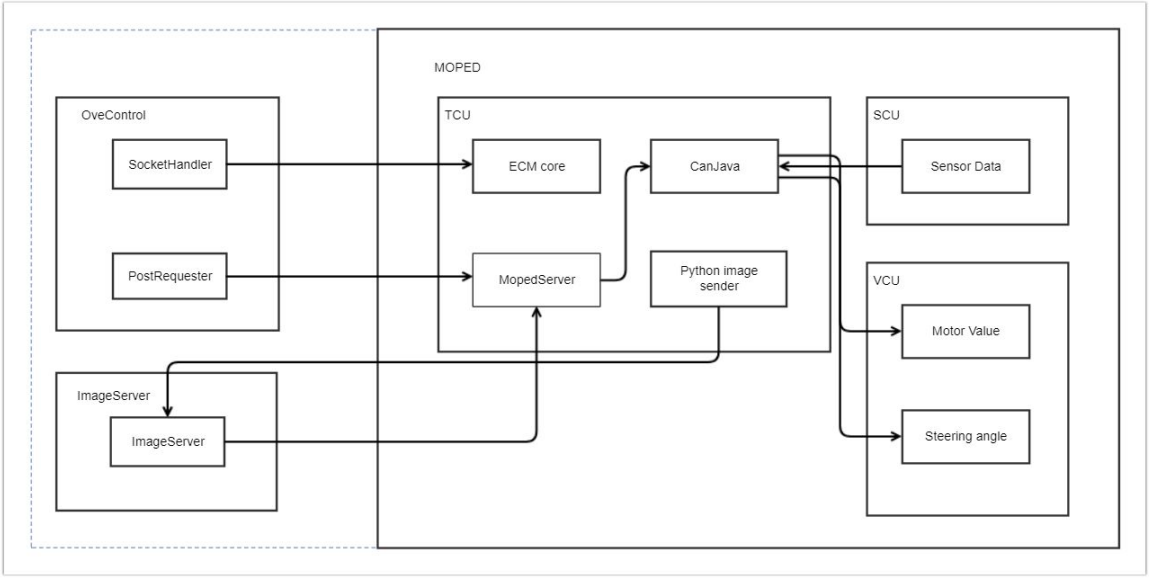
En annan fördel med denna lösningen är att vi får bra direktrespons, eftersom vi får en notifikation om att det har ändrats något och behöver t.ex. inte slumpvis läsa av en fil för att se om något har ändrats.

CanJava

Stora delar av prototypen, som ACC och platooning, hänger på att kunna skicka styrsignaler baserade på den data vi får in från sensorerna och kameran. Detta var en av de större utmaningarna vi stötte på i utvecklingen, och vi prövade flera olika lösningar innan vi valde denna. Dels så försökte vi utveckla en lösning som använde sig av plug-ins. Detta fick vi aldrig till att fungera eftersom det inte gick att ladda ner plug-ins på MOPEDen från servern. Detta var dels på grund av att servern och MOPEDen hade olika squawk versioner. Istället bytte gruppen strategi för att kunna köra kod på MOPEDen, till att köra Pythonkod direkt i MOPEDens terminal. MOPEDens TCU hade redan massa python filer som skötte kommunikationen med VCUn, vilket gjorde att vi direkt kunde skriva kod för ACC. Denna lösningen var dock inte optimal, eftersom den gav alldeles för långsam respons, och det gjorde att vi var tvungna att implementera merparten i Python, som är ett språk som gruppens medlemmar inte är lika erfarna i.

Den lösning som vi i slutändan använde oss av var att skriva Javakod direkt till MOPEDens Can-buss. Det var en lösning som vi fick genom att samarbeta med en annan grupp. Vi valde att använda deras lösning eftersom den fungerade och vi då kunde lägga mer energi på annat.

Overview



Denna bild visar i stora drag hur prototypens system är organiserad.

Funktioner

ACC

Prototypen har en ACC, vilket står för Adaptive Cruise Control. Den kan känna av om det kommer ett hinder framför med hjälp av sensorer, och undvika att köra in i detta hinder genom att stanna. Prototypen kan även anpassa farten till ett föremål framför som rör sig framåt och hålla samma fart.

ALC

Prototypen har även en ALC, vilket står för Adaptive Latitudinal Control. Den använder sig av en kamera som vi riktat framåt på MOPEDen. Kameran tar bilder som skickas till en server för analys (ImageDetection). Om en bild innehåller en röd cirkel eller ellips kommer ImageDetection skicka data om positionen

Platooning

Prototypen har även ett platooning-läge, som gör att MOPEDen kan hålla samma kurs och fart som framförvarande MOPED. Detta förutsatt att MOPEDen framför har en färgad cirkel bakpå, eftersom det är detta som prototypen använder för att urskilja vilket föremål som ska följas efter. Platooning-läget använder sig av både ACCn och ALCn.

Översikt

Prototypen har flera delar som gör det möjligt för den att genomföra platooning, använda ACC och styra MOPEDen manuellt. Med hjälp av den input prototypen får från sensorer och kamera kan den bedömma om det finns ett hinder framför den, avgöra när den behöver svänga vid platooning, och ta emot styrsignaler från en app.

Image Detection

Den här modulen har hand om logiken för att analysera de bilder som MOPEDen skickar. Detta gör den genom att analysera alla pixlar i bilden och skapa horisontella linjer av pixlar med den eftersökta färgen (ett intervall inom det röda spektrat som togs fram genom tester). Sedan skapas objekt av dessa linjer, som bildar solida former. Sedan undersökts om det objekt som skapas är en ellips. Detta togs också fram genom tester, först med bild av en perfekt cirkel och andra figurer och senare med faktiska bilder från MOPEDens kamera. Kamerans bilder hade såklart bakgrund som kunde störa eller ljusförhållanden som gjorde färgen eller formen av cirkeln annorlunda än vad de tänkta villkor kunde hantera.

MOPEDen kan sedan få information om fordonet skulle svänga eller inte berodde på hur mycket cirkeln skiljde sig ifrån den tänkta “mittlinjen” i kamerans bild. Beroende på cirkelns position i bilden så ändras hjulens riktning. Ju snabbare bilden flyttades desto snabbare justerades även bilens riktning för att kunna följa med i skarpare svängar.

CanJava

CanJava är den kod som sköter kommunikationen från den ECU där vi kör vår egen kod, TCUn, och de två andra ECUerna. Den fungerar som ett gränssnitt som ser till att vi i vår java kod kan både skicka och ta emot data från MOPEDens CAN-buss, utan att behöva göra alla konverteringar på egen hand då datan från CAN-bussen är på ett annat format än det vi kan läsa av direkt utan att behöva omvandla till ett format som är vettigt att använda. CanJava ser även till att våra signaler till bland annat motorn följer de konventioner som CAN kräver.

OveControl

Prototypen kan köras med hjälp av appen OveControl. I den här appen kan man kontrollera framåt- och bakåtdrift, svänga, sätta av och på ACC och platooning, och hantera uppkopplingen till MOPEDen. Det går även att byta till platoon-läge, och då är appens input avaktiverad. Appen använder sig av sockets för att kommunicera med MOPEDen, genom att skicka ett postrequest till mopedservern.

Servers

Det finns två servers till prototypen, ImageDetection-servern och moped-servern, och de funkar på liknande sätt. De är skrivna i java och tar emot post-requests, där de tar emot meddelandet.

Servrarna är kopplade till sin enhets IP-adress där de öppnar förbestämda portar. För MOPED servern öppnas flera adresser där varje adress hanterar sin egen typ av meddelanden, till exempel hanterar “http://192.168.1.1:9090/app” meddelanden från appen medan “http://192.168.1.1:9090/response“ tar emot svar från ImageDetection servern.

ImageDetection server tar emot en ström som den skriver om till en bild som den senare analyserar och skickar iväg resultatet till MOPED-servern. För enkelt hitta den intressanta delen av strömmen markeras meddelanden till MOPED-servern med startM i början och endM i slutet, meddelandet false skulle alltså se ut som “startMfalseendM”.