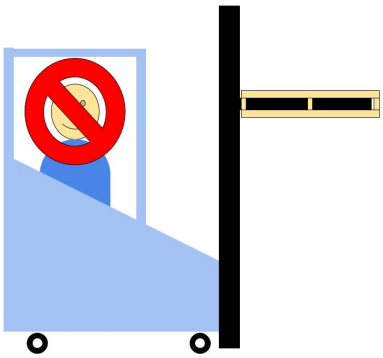
Designspecifikation
Autonom truck

Version 1.0

Redaktör: Joar Manhed
Datum: 12 oktober 2018



Status

Granskad	Kim Byström	2018-09-26
Godkänd	Andreas Bergström	2018-10-12

Kursnamn:	Reglerteknisk projektkurs	E-post:	kimby803@student.liu.se
Projektgrupp:	TRUCK-HT18	Dokumentansvarig:	Joar Manhed
Kurskod:	TSRT10	Dokumentansvarigs e-post:	joama350@student.liu.se
Projekt:	Autonom truck	Dokumentnamn:	Designspecifikation.pdf

Projektidentitet

Grupp E-post:	kimby803@student.liu.se
Hemsida:	
Beställare:	Andreas Bergström, ISY, Linköping University **Telefon:** +46 10-711 54 54, **E-post:** andreas.bergstrom@liu.se
Kund:	Magnus Persson, Toyota Material Handling, Mjölby **Telefon:** +46 771-220 220 , **E-post:** magnus.persson@toyota-industries.eu
Kursansvarig:	Daniel Axehill, ISY, Linköping University **Telefon:** +46 13-28 40 42, **E-post:** daniel.axehill@liu.se
Projektledare:	Kim Byström
Handledare:	Erik Hedberg, ISY, Linköping University **Telefon:** +46 13-28 13 38, **E-post:** erik.hedberg@liu.se

Gruppmedlemmar

Namn	**Ansvarsområde**	**Telefon**	**E-post (@student.liu.se)**
Kim Byström	Projektledare	072-7432190	kimby803
Lovisa Jansson	Designansvarig	076-3906525	lovja529
Anton Johansson	Komponentansvarig rörelseplanering	076-1962818	antjo244
Joar Manhed	Dokumentansvarig	076-5865400	joama350
David Sandmark	Komponentansvarig modellering & simulering	073-7613213	davsa696
Niklas Stenberg	Komponentansvarig reglering	076-8018632	nikst888
Pär Sörliden	Mjukvaruansvarig	076-5955950	parso619
Gustaf Westerholm	Testansvarig	070-8257421	guswe541

Dokumenthistorik

Version	Datum	Utförda förändringar	Utförda av	Granskad
0.1	2018-09-21	Första utkast.	Alla	Kim Byström
0.2	2018-09-26	Första inlämning	Alla	Lovisa Jansson
0.3	2018-10-09	Andra utkast	Alla	Kim Byström
1.0	2018-10-12	Utvecklat delområden	Alla	Kim Byström

Innehåll

1 Introduktion

Toyota Material Handling i Mjölby producerar cirka 80000 truckar per år där andelen autonoma truckar i nuläget utgör ungefär 1% av dessa. Andelen autonoma truckar förväntas öka samt behovet av lagerhanteringssystem som utnyttjar en truckflotta optimalt.

I denna designspecifikation beskrivs övergripande hur en autonom truck kan modelleras och simuleras i spelmotorn Unity. Dessutom beskrivs hur planering och reglering av trucken ska göras i projektet. För att skydda Toyotas immateriella rättigheter kan viss information ha utelämnats från detta dokument.

2 Översikt

Det tänkta systemet består av en eller flera autonoma truckar inom ett lagerområde. I nuläget finns dock bara en truck i skala 1:3 att använda för utveckling. Trucken är en reach-truck med tre hjul, två stödhjul fram samt ett styrande bakhjul. Gafflarna är ej elektroniskt inkopplade och kan därav ej aktueras.

Trucken är utrustad med två LIDAR-sensorer, en framåt och en bakåt. Dessa används för att läsa av omgivningen och skapa en karta samt för att undvika kollisioner. Dessutom finns en RGB-kamera samt en positionssensor för gafflarna. Kameran används för kunna avgöra truckens position i förhållande till pallar och är placerad överst på gaffeltornet. Trucken styrs av kommandon via en annan dator som är ansluten via ssh alternativt ett tangentbord kopplat direkt till trucken. Trucken styrs med hjälp av hastighet och hjulvinkel på bakhjulet.

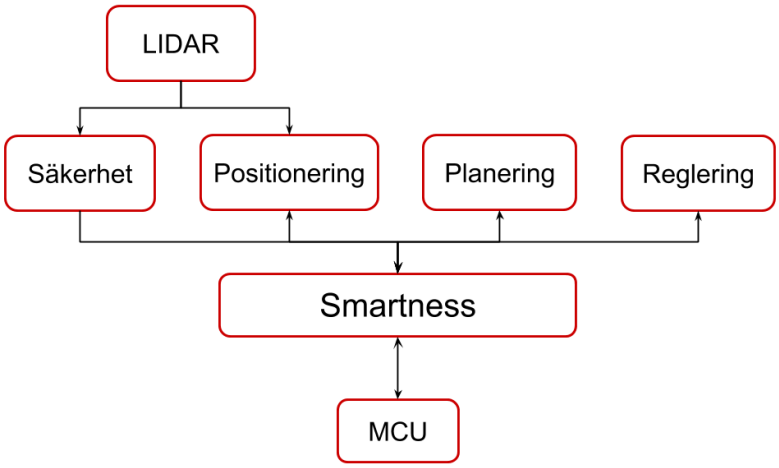
Trucken styrs av tre moduler som alla är kopplade till trucken via Smartness. Dessa är positionerings-, planerings- och regleringsmodulen. Positioneringsmodulen använder information från LIDAR-sensorena samt biblioteket Google Cartographer och en SLAM-algoritm för att, vid manuell styrning, kunna rita upp en karta över rummet. Både den karta samt truckens position som genereras av positioneringsmodulen skickas sedan till planeringsmodulen. Planeringsmodulen använder sig sedan av denna information för att beräkna den rutt som trucken ska följa för att utföra ett givet uppdrag. Den rutt (ett antal punkter) som genereras skickas sedan till regleringsmodulen som använder dessa för att reglera trucken. Regleringsmodulen är uppdelad i två delar där den ena sköter reglering när en rutt följs, och en för precisionskörning när trucken närmar sig en pall och ska köra in under den. I detta projekt ska planerings- samt regleringsmodulen utvecklas och inget fokus kommer läggas på positioneringsmodulen.

Tidigare använde Toyota ett ROS-baserat system för att styra trucken, men det har nyligen bytts ut till en Nuvo-5095GC industridator som kör Linux. Detta leder till att den simuleringsmiljö som tidigare användes, Gazebo, inte längre kan användas. Det som behöver utvecklas är därmed en ny simuleringsmiljö, där Toyota har valt att göra detta i spelmotorn Unity. Andra utvecklingsområden för trucken är att bygga ut den statiska planeraren till att vara dynamisk, samt att utveckla en reglering som kan följa den planerade rutten. När trucken närmar sig en pall behöver även en mer noggrann reglering utvecklas för att kunna köra in och plocka upp pallen.

2.1 Gränssnitt

De olika delsystemen ska kommunicera med varandra via Toyotas egenutvecklade gränssnitt Smartness. En översikt på kommunikationen finns i figur 1. Positionering och säkerhet finns redan implementerat på trucken och även en början av planering och reglering för att följa en trajektoria. Reglering till precisionskörning finns inget implementerat vid projektets start.

Vid simulering kommer själva trucken (MCU:n), LIDAR-sensorn, samt kommunikationen via Smartness (se figur 1) att simuleras i spelmotorn Unity. I Unity kommer trucken samt truckens miljö att visualseras i 3D. De övriga modulerna för reglering och planering kommer att placeras utanför Unity i separata program. Dessa program kan köras direkt på datorn alternativt på virtuella maskiner. Dessutom ska de kunna köras på varsin Raspberry Pi för att möta de hårdvarukrav som finns på projektet. Dessa moduler ska sedan kommunicera med den simulerade trucken i Unity via TCP/IP. Denna kommunikation ska följa Toyotas Smartness-protokoll. Om protokollet följs ska det i teorin gå att implementera verifierad kod från Unity i den verkliga trucken. Förutom simuleringsmodulen



Figur 1: Översikt på hur de olika delsystemen på trucken kommunicerar.

som skrivs i C# ska de olika modulerna skrivas i språket C++ och kommunicera med Unity via TCP/IP. Dessa delsystem kommer att agera klient och Unity agerar server vid kommunikationen.

Eftersom truckens positioneringsmodul inte ska utvecklas i detta projekt kommer en enkel variant av den att användas. Eventuellt kommer positioneringsmodulen som använder Google Cartographer för SLAM inte att köras på en Raspberry Pi på grund av prestandabegränsning. Alternativt används modulen inte alls och i stället simuleras kartdata direkt från Unity och skickas till navigeringsmodulen.

2.2 Hårdvara

De tre Raspbery Pi:s som används saknar föristallerat operativsystem. Det finns flertalet operativsystem tillgängliga på Raspberry Pi:s hemsida gratis. Där ibland: Raspbian, Ubuntu Mate, Windows 10 IOT Core. Här väljes Ubuntu Mate eftersom det använder ett gränssnitt vi känner igen och liknar det som finns på truckens industridator. Övrig utrustning som behövs är:

- datorskärm med HDMI-adapter
- mikro USB-laddare
- datormus och tangentbord med USB-sladd eller bluetooth
- samt mikro SD-minneskort (8GB eller mer).

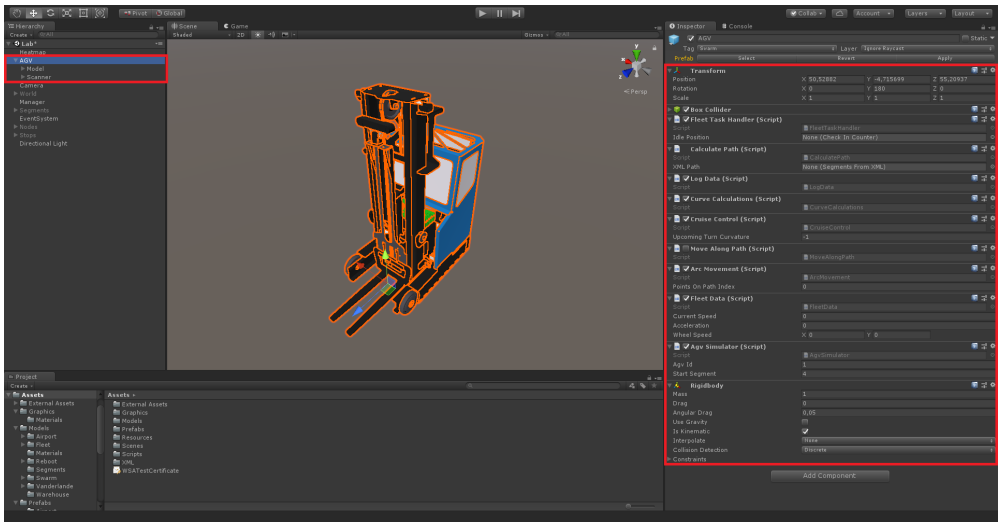
3 Delsystem - Simuleringsmodul

Simuleringsmodulen ska simulera en truck i spelmotorn Unity. Den ska kunna ersätta den verkliga trucken och kommunicera med de andra modulerna på samma sätt som den verkliga trucken gör.

3.1 Inledande beskrivning

Trucken ska modelleras och simuleras i spelmotorn Unity och vara i 3D. Unity-programmet kommer att skrivas i språket C# och simuleras och redigeras i Unitys grafiska miljö. De objekt som finns beskrivna nedan i fonten `UnityThing` är redan existerande funktionalitet som finns i Unity, och ytterligare information om dessa kan hittas i Unitys dokumentation.

För närvarande finns en väldigt enkel modell av trucken i Unity. Rent fysikaliskt modelleras den som ett rätblock. Nuvarande modell syns i figur 2. Denna ska utökas så att varje hjul modelleras som ett eget objekt och sedan kopplas samman med den nuvarande trucken. Sedan ska simuleringen styras genom att ge det bakre hjulet en styrvinkel och hastighet, i stället för som tidigare, bara ge hela trucken en riktning och hastighet. Tidigare har trucken modellerats i Gazebo bestående av olika sammankopplade volymer med givna massor och tröghetsmoment. Där har modellen varit mycket mer fullständig än den som idag finns i Unity. Målet är att helt eller delvis återanvända tidigare modeller och migrera dessa till Unity. Tidigare CDIO-projekt från 2017 tog även fram en mer fullständig matematisk modell för trucken. Denna modell ska testas att implementeras i Unity.[1]



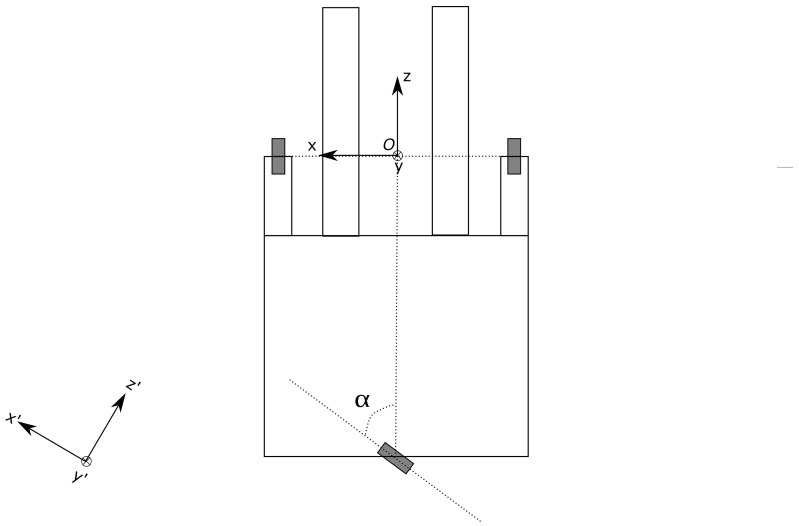
Figur 2: Nuvarande modell av trucken i Unity.

3.2 Truckmodell

Trucken ska kunna modelleras som ett trehjuligt fordon med ett styrande och drivande bakhjul. De storheter som ska modelleras är truckens hastighet, position och vinkel. Trucken kommer att bestå av ett övergripande `GameObject` som beskriver trucken. Den kommer sedan bestå av en kaross samt tre hjul. För att modellera karossen kommer ett visuellt objekt som ser ut som en truck användas i kombination med ett `Collider`-objekt som beskriver hur trucken beter sig vid kollisioner. Det enklaste är att låta truckens

kollisionsegenskaper beskrivas av en eller flera `BoxCollider`, det vill säga rätblock, som inkapslar den visuella trucken. För att modellera hjulen används `WheelCollider`, vilket är en inbyggd standardkomponent som finns i Unity. Den beskriver ett hjuls kollisionsegenskaper och kan styras med hjulets styrvinkel samt vridmoment. Komponenten innehåller även inställningar för flera andra hjulegenskaper, såsom friktion, massa och fjädring som kan justeras. Till hjulmodellen ska även en visuell representation av hjulet användas. För att modellera truckens gafflar används också en visuell representation (i enklaste fallet som ett rätblock) samt ett kollisionsobjekt som beskriver hur gafflarna interagerar med omvärlden (i enklaste fallet med objekt av typen `BoxCollider`).

En illustration av simuleringens koordinatsystem kan ses i figur 3. Det globala koordinatsystemet är markerat med x', y' och z' och truckens lokala koordinatsystem med x, y och z. Truckens styrvinkel är markerad med α i figuren.



Figur 3: Lokalt och globalt koordinatsystem samt styrvinkeln α.

Trucken är även utrustad med LIDAR-sensorer, vilka också ska modelleras i Unity. Dessa har en vidd på 270°. Höger definieras som 0°, och sensorn har därmed ett synfält från $-45°$ till 225°. LIDAR-sensorerna kan modelleras med hjälp av `Physics.Raycast` vilket är en funktion som skickar en stråle i en given riktning och returnerar om den träffade något `Collider`-objekt och i så fall på vilket avstånd. För att visualisera skannern kan till exempel objekt av typen `LineRenderer` användas för att rita ut linjer i 3D-världen. Datan som fås från denna skanner skickas sedan via TCP-kommunikationen till de andra modulerna.

För att göra simuleringen lätt att bygga på med nya truckmodeller i framtiden ska truckens egenskaper läsas in från en konfigurationsfil, så att egenskaperna lätt kan ändras utan att gå in i koden. Några exempel på truckegenskaper som kommer att återfinnas i konfigurationsfilen är olika typer av massor, längd på gafflar, antal hjul - styrbara och fasta - samt deras positioner, truckens utsträckning i olika dimensioner, masscentrum och maxhastighet.

3.3 Karta

Planeringsmodulen behöver en karta för att kunna planera rutter. På den verkliga trucken genereras denna karta av positioneringsmodulen genom att använda LIDAR-data och

använda en SLAM-algoritm. Dock ingår inte positioneringsmodulen i detta projekt. En lösning på detta är att ändå skicka LIDAR-data och göra en simpel version av positioneringsalgoritmen som genererar en karta, alternativt genererar en statisk, förbestämd karta. Ett annat alternativ är att generera kartan i simuleringsmiljön och skicka direkt till planeringsmodulen.

3.4 Kommunikation

Det program som tas fram i Unity ska kunna kommunicera med de andra modulerna via TCP/IP. Vid denna kommunikation används Toyotas Smartness-protokoll för att skicka meddelanden och Unity-progammet ska agera server i TCP-kommunikationen. Smartness är kopplat till de tre modulerna reglering-, planering- samt positioneringsmodulen. Eftersom utveckling av positioneringsmodulen inte ingår i detta projekt ska denna modul antingen utelämnas eller göras en simpel version av den.

Kommunikationen sköts av ett skript i Unity (kopplat till ett tomt `GameObject`). Detta skript kan skicka meddelanden till klienterna samt starta en separat tråd för att lyssna efter inkommande klient-anrop. För varje klient som vill ansluta startas en ny tråd som tar hand om kommunikationen med just den klienten. För TCP-kommunikationen kan till exempel klasserna `TcpClient` och `TcpListener` användas.

För LIDAR-datan kommer en separat kommunikationskanal sättas upp mellan positioneringsmodulen och simuleringsmiljön, eftersom detta inte skickas genom Smartness i verkligheten.

3.5 Smartness

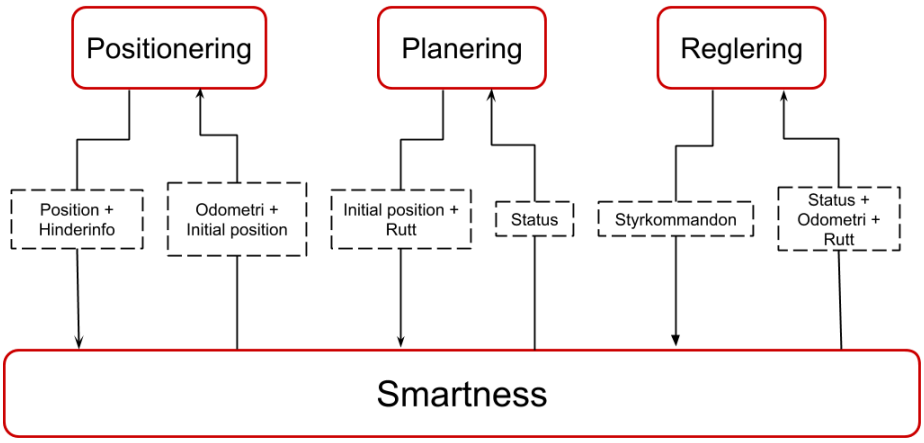
Smartness agerar som kommunikationslager mellan de olika modulerna samt truckens MCU. Det är utvecklat av Toyota och finns beskrivet i Toyotas interna dokument. Smartness-protokollet går ut på att skicka meddelanden med en header, ett fält med datainformation samt fält med parametervärden beroende på vilken typ av meddelande det är. Headern innehåller information om vilken typ och storlek meddelandet har samt ett unikt ID för varje meddelande. Sedan följer ett fält med meddelandeinformation som innehåller parametrar kopplade till den aktuella meddelandetypen. Slutligen har vissa meddelanden ett eller flera ytterligare fält med fler parametrar. Vad som skickas till och från de olika modulerna finns sammanfattat i figur 4 och beskrivs i mer detalj nedan.

3.5.1 Regleringsmodulen

Från regleringsmodulen får Smartness meddelanden om hur hjulen ska styras. Dessa innehåller hastighet (mm/s) och hjulvinkel (rad) för varje hjul samt en eventuell felkod, enligt kommunikationsprotokollet Smartness. Denna information ska sedan skickas till truckens aktuatorer eller, i simuleringsmiljön, till truckens hjulkomponenter. Dessa meddelanden ska skickas med 100 ms intervall.

Smartness skickar meddelanden till regleringsmodulen med odometri-information, det vill säga nuvarande hjulvinkeln och hjulhastighet för varje hjul samt eventuell felkod. I simuleringsmiljön tas denna information från hjulkomponenterna och skickas till regleringsmodulen. Dessa meddelanden ska skickas inom 50ms efter att ett kommando har kommit från modulen.

Regleringsmodulen får meddelanden om truckens status från Smartness. Dessa meddelanden innehåller information om truckens position, riktning, hastighet, vikt på gafflarna



Figur 4: Översikt på kommunikationen med Smartness

samt batteristatus. Dessa meddelanden ska skickas inom 50ms efter att Smartness mottagit ett kommando.

Regleringsmodulen får även meddelanden om den planerade rutten från planeringsmodulen, via Smartness. Dessa innehåller en rad koordinater som beskriver rutten samt hastighet och riktning för varje koordinat. Dessa meddelanden skickas en gång i sekunden från planeringsmodulen.

Regleringsmodulen kan även skicka kommandon till Smartness, för att till exempel avbryta körningen.

3.5.2 Planeringsmodulen

Från planeringsmodulen får Smartness kommandomeddelanden för en planerad rutt. Dessa innehåller en rad koordinater som beskriver rutten med hastighet och riktning för varje koordinat. Dessa meddelanden ska skickas en gång i sekunden och Smartness skickar dem sedan vidare till regleringsmodulen.

Trajektorian skickas styckvis till regleringsmodulen där varje del visar hur trucken ska köra de närmaste tre (3) sekunderna. Detta gör att sträckan som skickas inte är en fast längd, utan behöver integreras fram från den önskade medelhastigheten med avseende på tiden.

Från planeringsmodulen får Smartness även ett meddelande om en initial position. Detta skickas sedan till positioneringsmodulen.

Planeringsmodulen får meddelanden om truckens status från Smartness. Dessa meddelanden innehåller information om truckens position, riktning, hastighet, vikt på gafflarna samt batteristatus. Dessa meddelanden ska skickas inom 100ms efter att Smartness mottagit ett kommando.

Planeringsmodulen kan även skicka kommandon till Smartness, för att till exempel avbryta körningen.

3.5.3 Positioneringsmodulen

Från positioneringsmodulen får Smartness meddelanden med truckens position (mm) och vinkel (rad) samt meddelanden med information om det närmaste upptäckta hindret.

Smartness skickar meddelanden till positioneringsmodulen med odometri-information, d.v.s. nuvarande hjulvinkeln och hjulhastighet för varje hjul samt eventuell felkod. I simuleringsmiljön tas denna information från hjulkomponenterna och dessa meddelanden ska skickas inom 50ms efter att ett kommando har kommit från positioneringsmodulen.

Positioneringsmodulen tar emot meddelanden om initial position för trucken. Detta skickas via Smartness, från planeringsmodulen.

4 Delsystem - Planeringsmodul

4.1 Inledande beskrivning

Planeringsmodulen genererar en rörelsebana för trucken att följa givet en start- och slutposition. För närvarande är denna rörelsebana statisk, det vill säga att den inte ändras om hinder dyker upp. För att beräkna en väg mellan start- och slutposition används en algoritm baserad på A*.

För att planeringsmodulen ska kunna skapa rutter för trucken att följa behövs information om miljön den befinner sig i, bland annat positioner för ställage, hämt- och avlastningszoner samt en karta över omgivningen. Eftersom planeringsmodulen kommer att testas/utvecklas mot simuleringsmiljön i Unity kommer det initialt i utvecklingsarbetet antas att planeringsmodulen får direkt tillgång till hur miljön ser ut i Unity och det alltså inte behöver köras någon mappning av miljön. Vidare kommer en användare få explicit ange start- och slutposition till planeringsmodulen.

Algoritmen som ska användas för ruttplanering skall köras vid initiering av uppdrag. Efter detta uppdateras rutten endast då LIDAR-sensorerna upptäcker ett hinder som blockerar beräknad rutt. LIDAR-sensorernas frekvens är 15 Hz. Eventuell oanvändbar mätdata gör att det är rimligt att stämma av minst 10 gånger per sekund om den har hittat något oförutsett hinder.

4.2 Nuvarande algoritm

Nedan följer en översiktlig beskrivning av A*-algoritmen baserad på beskrivningen i [3] och [2]. A*-algoritmen beräknar vägen med lägst kostnad i en viktad graf mellan en start- och en slutnod.

Huvudformeln som används i A*-algoritmen ges i (1). Den beräknar ett värde för en nod n, där $g(n)$ är kostnaden för vägen från startnoden fram till nod n, och $h(n)$ är ett heuristiskt estimat av kostnaden för vägen mellan n och slutnoden. Exempelvis kan $h(n)$ vara det euklidiska avståndet mellan nod n och slutnoden.

$$f(n) = g(n) + h(n) \tag{1}$$

Algoritmen använder sig av två listor: O (open) och C (closed). Listan O innehåller de noder som undersöks för att hitta vägen med lägst kostnad. Den andra listan C innehåller noder som inte ska undersökas igen.

A*-algoritmen fungerar i stora drag enligt följande.

1. Initiera O och C till tomma listor. Lägg startnoden till O och sätt dess kostnad till 0.
2. Välj från O den nod med lägst kostnad (om flera välj till exempel den senast använda). Beteckna denna nod med N.
3. Ta bort N från O och lägg den till C.
4. Om N är slutnoden, gå till steg 7.
5. Gör något av följande för varje nod M som är närliggande till N samt passerbar:
   (a) Om M ligger i C, gör inget.

(b) Om M inte ligger i O, lägg den till O samt beräkna dess kostnad enligt (1) och sätt dess föregångare till N.

(c) Om M ligger i O, beräkna en ny kostnad för M givet noden N. Om den nya kostnaden är lägre än den gamla kostanden, uppdatera M:s kostnad och sätt dess föregångare till att vara N. Låt M vara kvar i O.

6. Om O inte är tom gå till steg 2, annars om O är tom och målnoden inte nåtts finns ingen passerbar väg mellan start- och slutnod.

7. Vägen med lägst kostande mellan slut- och startnod har hittats. Den fås genom att hitta föregångaren till slutnoden, och sedan dess föregångare och så vidare tills startnoden nås.

4.3 Förbättring

Planeringsmodulen kommer även att efterbehandla data från A*-algoritmen för att bestämma önskad hastighet för samtliga punkter i rörelsebanan. Genom att sätta lägre hastigheter i skarpa kurvor kan trucken få en bättre kurvtagningsförmåga. Rörelsebanan som genereras kommer att bestå av punkter med koordinater och hastighetsangivelser i varje punkt, som sedan regleringsmodulen kan reglera efter. Hastighetsändring mellan två punkter ska ske genom linjärinterpolering för att få mjuka övergångar.

Projektgruppen önskar ta fram en dynamisk rörelseplanerare som kan ändra rutt om hinder dyker upp. Det kommer att antas att denna modul får information från andra system (som ännu inte är utvecklade) som detekterar, klassificerar och spårar/följer hinder (andra truckar, människor etc.). Planeringsmodulen får alltså en uppdaterad karta då nya hinder dykt upp eller flyttat på sig.

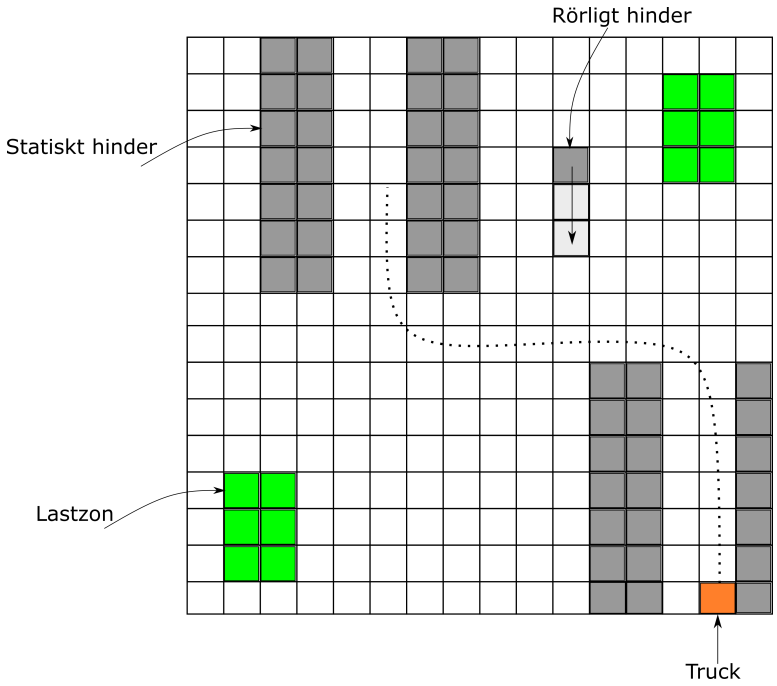
4.3.1 Karta

Planeringsmodulen kommer att använda en statisk karta över miljön. Den innehåller huvudsakligen väggar, ställage, samt av- och pålastningszoner. Den kan till att börja med till exempel genereras manuellt från Unity-miljön. För att sedan få en dynamisk planerare kommer information om nya statiska hinder och/eller rörliga hinder att läggas ovanpå den statiska huvudkartan över miljön. Information om sådana hinder skickas till planeringsmodulen från en extern modul. Där ingår hindrets position, utsträckning och eventuellt riktning den rör sig i. Denna information läggs ovanpå huvudkartan.

Figur 5 visar principiellt hur kartan ska utformas. Hela lagret delas upp i ett rutnät, där egenskaperna för varje ruta specificeras. Statiska hinder, såsom ställage (mörkgrå) och kommer ansättas som opasserbart. Även lastzoner (grön) och rörliga hinder kommer ansättas som opasserbart. Rörliga hinder kommer definieras som en ruta i rutnätet (mörkgrå) och en förlängd kollisionszon (ljusgrå) i rörelseriktningen. Båda ansätts som opasserbara. Kollisionszonens längd är beroende på hindrets hastighet. Den streckade linjen i figuren visar en planerad trajektoria för trucken mellan ställagen.

4.3.2 Algoritm

Den nuvarande A*-algoritmen behöver utökas eller ersättas av en ny för att planeringsmodulen ska klara av en dynamisk miljö. Ett första försök skulle kunna vara att köra A*-algoritmen med jämna mellanrum då en ny ändrad karta fås. Detta kan kanske visa sig vara för beräkningskrävande. En alternativ algoritm att undersöka är D* (Lite) vilket kan ses som en dynamisk A*-algoritm.



Figur 5: Principiellt hur kartan för planeringsmodulen ser ut.

Det finns en hel uppsjö med olika alternativ för ruttplanering. I detta projekt läggs fokus på A*-algoritmen som det redan finns grund för sedan tidigare projekt. Detta beror på att denna algoritm kan utvecklas till att bli en mer dynamisk A* som beskrivits ovan, alternativt att vidareutveckla befintlig A* till D* (Lite). Det går alltså att utnyttja att grundarbetet redan är gjort för dessa utvecklingar, vilket gör arbetet mer tidseffektivt och därmed ger ett mer genomarbetat slutresultat. Detta jämfört med att utveckla en helt ny ruttplaneringsalgoritm från grunden.

4.3.3 Sekvensplanering

En överordnande planering av jobbprioritering väljer det jobb som har närmast upphämtning. Efter detta initialiseras ruttplanering för upphämtning av denna pall. Ruttplaneringen skickas till regleringsmodulen iterativt tills trucken är framme vid upphämtningsplatsen. När trucken anlänt, påbörjas ruttplanering på nytt. Denna gång till pallens avlastningsplats. Samma iterativa process upprepas igen med regleringsmodulen, tills trucken är framme vid avlastningsplats. Efter detta börjar den överordnade jobbplaneringen om igen.

5 Delsystem - Regleringsmodul

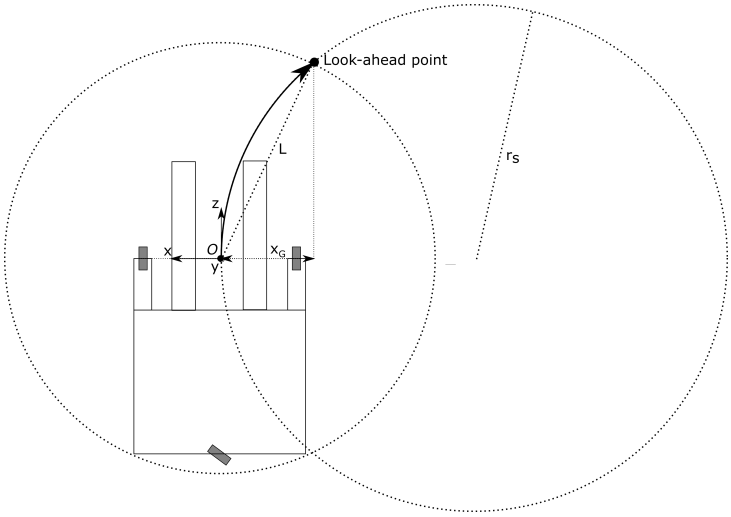
5.1 Inledande beskrivning

Regleringsmodulen har till uppgift att styra trucken så att den följer den rörelsebana som genereras av planeringsmodulen på ett önskvärt sätt. Rörelsebanan som följs beskrivs av punkter med koordinater samt hastigheter.

5.2 Nuvarande algoritm

Den reglering som finns implementerad vid projektstart är av typen Pure Pursuit. Till denna används en cirkel med radie (L), vilket är den vänstra cirkeln i figur 6. L är en designparameter. Den punkten där cirkel korsar den planerade trajektorian används som målpunkt, detta är "look-ahead point" i figur 6. Utifrån designparametern L och målpunktens x-koordinat (x_G) kan krökningen (γ) beräknas. Krökning är inversen av svängradien (r_s), radien av högra cirkeln i figur 6. Den heldragna linjen i samma figur är den bana som trucken ska ta för att komma tillbaka till den önskade trajektorian. Denna beräknas enligt ekvation 2.

$$\gamma = \frac{1}{r_s} = -\frac{2x_G}{L^2} \tag{2}$$



Figur 6: Illustration av Pure Pursuit.

Det koordinatsystem som används kan också ses i figur 6. Det har sitt origo i truckens rotationscentrum vilket ligger mittemellan framhjulen, där z-led är positivt i truckens gaffel-riktning och x-led positiv åt vänter. Krökningen är också en funktion av truckens vinkelhastighet (ω) och hastighet i z-led (v), se ekvation (3).

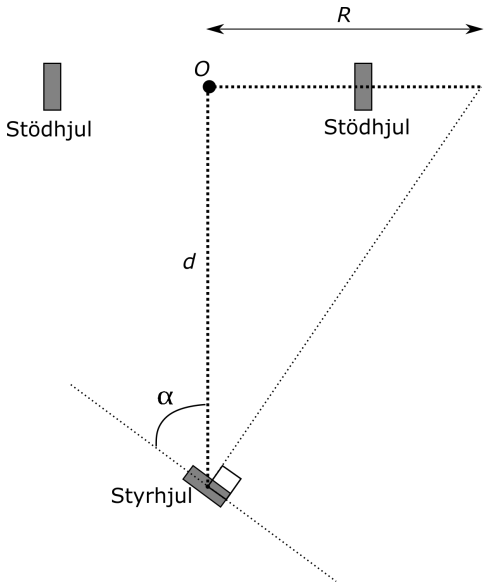
$$\gamma = \frac{\omega}{v} \tag{3}$$

I dagsläget är dessa hastigheter modellerade utifrån att trucken är en trehjuling. Vinkelhastigheten tas fram med hjälp av figur 7 och ekvation 4 där ω_h är styrhjulets vinkelhastihet, v_h är styrhjulets hastighet och r_h är styrhjulets radie.

$$\omega_h = \frac{v_h}{r_h} \tag{4}$$

Dessa ger ett uttryck för truckens vinkelhastighet enligt ekvation 5.

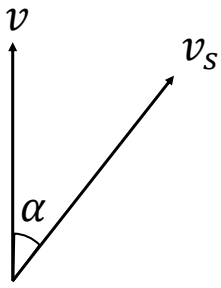
$$\omega = \frac{\omega_h r_h}{\sqrt{d^2 + R^2}} = \frac{v_h \sin(\alpha)}{d} \tag{5}$$



Figur 7: Avstånd mellan truckens stödhjul, styrhjul och rotationscentrum.

Hastigheten i z-riktningen tas fram med hjälp av figur 8 vilket ger ekvation 6.

$$v = v_h \cos(\alpha) \tag{6}$$



Figur 8: Samband mellan styrhjulets hastighet och truckens hastighet.

Det som ska bestämmas av regleringen är önskad styrvinkel (α). Med lite omskrivningar med hjälp av ekvationerna 3, 5 och 6 fås styrvinkeln från

$$\gamma = \frac{v_h \sin(\alpha)/d}{v_h \cos(\alpha)} \Leftrightarrow \alpha = \arctan(\gamma \cdot d) \tag{7}$$

5.3 Förbättrad algoritm

Den reglering som tas fram under projektet ska kunna följa den angivna trajektorian och hastighetsangivelserna. Dessutom ska den klara av skarpa kurvor bättre än den gör vid projektstarten då den gör mjuka kurvor, det vill säga genar i hörnen. Det finns lite olika vägar att gå för att göra det. Ett sätt är att vidareutveckla den algoritmen som finns idag eller att helt byta algoritm.

5.3.1 Vidareutveckling av algoritm

Sträckan (L) är i dagsläget statisk men algoritmen skulle fungera bättre om denna var hastighetsberoende. Då trucken kör med högre hastighet är det bättre att kolla lite längre fram och vid långsammare hastighet närmare trucken. Om planeringsmodulen har möjlighet att skicka en önskad hastighet för varje punkt kan denna hastighet användas för att bestämma L.

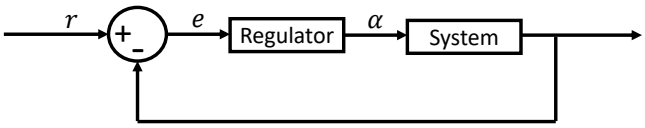
Ett annat sätt att bygga vidare på den nuvarande algoritmen är att lägga till en vinkelreglering (baserat på truckens vridningsvinkel). Det borde förbättra kurvtagningen och även göra att trucken har den önskade riktningen när den har nått sin slutposition, vilket är viktigt om den till exempel ska kunna ta en pall. Vinkelregleringen går att implementera som en PID-regulator där referenssignalen är den önskade vinkeln från planeringsmodulen. Reglerfelet är skillnaden mellan referenssignalen och den faktiska vinkeln som trucken har.

5.3.2 Alternativa reglerstrukturer

Ett alternativ är att implementera algoritmen Vector Pursuit. Denna har samma grundtanke som Pure Pursuit men tar även hänsyn till den önskade vinkeln i varje punkt. Vid närmare efterforskning konstaterades att denna algoritm bygger på många ekvationer och är inte helt lätt att förstå. Därför görs inget mer på denna algoritm för tillfället. Om de tidigare föreslagna metoderna inte fungerar som önskat kan denna metod undersökas vidare.

Ett andra alternativ är MPC-reglering. Fördelen med denna typ av reglering är att den kan ta hänsyn till begränsade styrsignaler och eventuella säkerhetsgränser. En annan fördel med MPC är att den kan ta hänsyn till flera in- och utsignaler. Med MPC skulle alltså både truckens styrvinkel och hjulhastighet kunna regleras. MPC kan även ta hänsyn till referenssignalen (banan att följa) framåt i tid. Metoden är dock väldigt beräkningskrävande. Tillståndsmodellen för trucken är även olinjär vilket innebär att antigen krävs olinjär MPC vilket är ännu mer beräkningskrävande eller kan modellen till exempel linjäriseras. Implementeringen av en MPC-regulator kommer därför kräva mycket mer arbete än vad simplare reglerstrategier gör, vilket kan vara onödigt om de enklare regleralgoritmerna fungerar väl.

Ett tredje alternativ är den vanligaste reglerstrukturen PID. Det finns ingen tidigare PID-regleralgorithm i trucken men den är välanvänd för andra autonoma fordon som till exempel självkörande bilar och därav borde det vara möjligt att applicera även på truckar. Då PID en allmänt vedertagen reglerlösning och kommer därför utforskas vidare. Styrsignalen för PID-regulator är densamma som för Pure Pursuit dvs styrhjulvilnkeln. För PID-regulatorn beror styrsignalen av tre delar; en proportionerlig del, en deriverande del och en integrerande del. Reglerfelet e kommer att vara $e = \Delta x$, där Δx är distansen mellan verklig och önskad position i x-led, i det lokala koordinatsystemet i figur 6. Den proportionerliga delen kommer då att vara reglerfelet multiplicerat med en reglerparameter K_P. För integraldelen används parametern K_I och för derivatadelen K_D.



Figur 9: En blockbeskrivning av PID regleringen. Regulatorblocket defineras av ekvation 8 och r är referensignalen dvs önskad avstånd mellan truckposition och trajektorian i x-led. Oftast kommer det önskade värdet på r vara 0. Det estimerade felet e går in regulatorn som ger en styrvinkel α. Styrvinkeln går in i systemet (trucken) och en ny x-position i förhållande till trajectorian är utsignal.

$$\alpha(t) = K_P e(t) + K_I \int_0^\infty e(t)\,dt + K_D \frac{d}{dt} e(t) \tag{8}$$

Eftersom systemet inte är linjärt kommer reglerparametrarna K_P, K_I och K_D troligtvis inte fungera bra för alla körscenarier, till exempel olika hastigheter. De kan även bero på trajektorian, till exempel om det är en rak sträcka eller en skarp kurva. För skarpa kurvor behöver styrvinkeln kunna vara stor, medan för att åka rakt bör den inte vara stor för att undvika att trucken slingrar sig fram. En lösning på detta är att använda sig av parameterstyrning. Reglerparametrarna kan bestämmas för olika fall och sparas i en tabell.

6 Delsystem - Precisionskörning med pallhantering

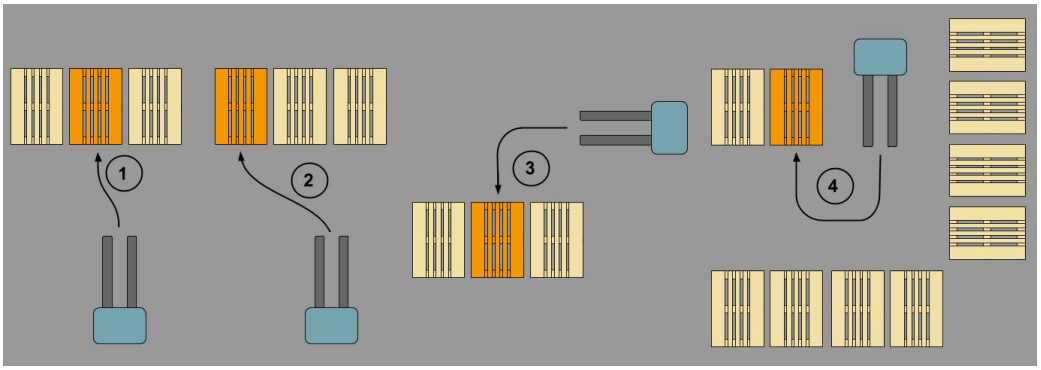
6.1 Inledande beskrivning

När trucken närmar sig en pall och ska köra in under den behövs en noggrannare reglering än den reglering som används under den planerade rutten. Därmed behöver trucken ha en modul för precisionskörning. Detta delsystem ska då även kunna lyfta pallen efter att trucken kört in under den. Detta delsystem kommer endast att utvecklas i mån av tid och intresse.

Precisionsregleringen kommer att använda information från LIDAR-sensorer för att utföra denna reglering. Denna sensordata anses vara given som insignal till modulen.

För att kunna reglera in sig på mitten till pallen behövs även någon form av pall-detektion implementeras. Detta kan eventuellt lösas med hjälp av maskininlärning eller liknande.

Precisionsregleringen ska testas på fyra olika scenarier där trucken och pallen är placerade olika relativt varandra. De fyra fallen illustreras i figur 10. Dessa fall användes även vid tidigare års arbete med trucken.



Figur 10: De fyra olika fallen av precisionskörning.

6.1.1 Alternativ: MPC

Precisionskörning skulle kunna använda sig av en MPC-regulator. En sådan skulle då använda samma utsignaler som den vanliga regleringsmodulen. För att försäkra sig om att trucken hamnar i rätt läge för att plocka en pall går det även att använda sig av krav på sluttillstånden.

Referenser

[1] S. Dam. Teknisk rapport- planering och sensorfusion för autonom truck. http://www.isy.liu.se/edu/projekt/tsrt10/2017/autonom_truck/Teknisk_rapport.pdf, 2017. hämtad 2018-10-09.

[2] J. Fradj. Introduction to a* pathfinding. https://www.raywenderlich.com/3016-introduction-to-a-pathfinding, 2011. Hämtad: 2018-10-08.

[3] Wikipedia. A* search algorithm. https://en.wikipedia.org/wiki/A*_search_algorithm, 2018. Hämtad: 2018-10-08.