TITTEL:

Autonom bil følger ball

KANDIDATNUMMER(E):

LARS-HARALD BÅTNES

EIVIND FUGLEDAL

JØRGEN TRYGSTAD

MAGNUS GRIBBESTAD

DATO:	EMNEKODE:	EMNE:	DOKUMENT TILGANG:
29.11.16	IE303812	Sanntids datateknikk	

STUDIUM:	ANT SIDER/VEDLEGG:	BIBL. NR:
Automasjonsteknikk	52/0	

VEILEDER(E) :

IVAR BLINDHEIM

GIRTS STRAZDINS

SAMMENDRAG:

Dette prosjektet omhandler utvikling og testing av en bil som skal kunne kjøres manuelt eller detektere og følge etter et objekt. Hovedfokuset i prosjektet har vært utvikling av en god sanntidsløsning, sånn at applikasjonene og trådene kan samarbeide på en slik måte at en unngår vranglås, sult og inferens. Ved å utforske samtidighet og sanntidssystemer, har vi fått god innsikt i bruken av trådbaserte løsninger. Løsningen benytter seg av objekt gjenkjennings metoder via et webkamera som er realisert i Java og applikasjonen kjører på en Odroid montert på bilen. PC'en kommuniserer serielt til en Romeo mikrokontroller som igjen styrer all IO på bilen. Kommunikasjon via Ethernet/UDP gjennom WiFi er brukt for å kommunisere til et GUI realisert på en klient PC. For sanntidsapplikasjonen er det benyttet Semaphore og et delt objekt som løsning på synkroniserings problemer mellom trådene.

Denne oppgaven er en eksamensbesvarelse utført av studenter ved NTNU Ålesund.

FORORD

Denne rapporten er skrevet på grunnlag av vårt prosjekt i faget «Sanntids Datateknikk». Faget har hovedfokus rundt «samtidighet og sanntidssystemer» og oppgavens tema vil ha hovedfokus på dette. Vi har i utgangspunktet av kravene satt av fagplan, gruppens kunnskaper og interesser, valgt en oppgave som vi selv mener kan være både interessant og utfordrende.

På grunnlag av informasjonen over har vi valgt prosjektet «Autonom bil følger ball».

Vi mente dette kunne være ett prosjekt som tilfredsstilte alle kravene, samtidig som vi selv måtte utfordre oss selv på flere nivåer.

Vi har også gjennom prosjektet tilegnet oss nye kunnskaper om trådbasert programmering, som vi selv vil tro vil komme godt til nytte i senere anlegninger.

Vi vil også rette en takk til Girts Strazdins og Ivar Blindheim for veiledning og gode råd underveis i utviklingen av prosjektet.

INNHOLD

SAMMENDRAG

Dette prosjektet omhandler utvikling og testing av en bil som skal kunne kjøres manuelt eller detektere og følge etter et objekt. Hovedfokuset i prosjektet har vært utvikling av en god sanntidsløsning, sånn at applikasjonene og trådene kan samarbeide på en slik måte at en unngår vranglås, sult og inferens. Ved å utforske samtidighet og sanntidssystemer, har vi fått god innsikt i bruken av trådbaserte løsninger. Løsningen benytter seg av objekt gjenkjennings metoder via et webkamera som er realisert i Java og applikasjonen kjører på en Odroid montert på bilen. PC'en kommuniserer serielt til en Romeo mikrokontroller som igjen styrer all IO på bilen. Kommunikasjon via Ethernet/UDP gjennom WiFi er brukt for å kommunisere til et GUI realisert på en klient PC. For sanntidsapplikasjonen er det benyttet Semaphore og et delt objekt som løsning på synkroniserings problemer mellom trådene.

TERMINOLOGI

Forkortelser

UML	Unified Modeling Language
UP	Unified Process
JVM	Java Virtual Machine
OOP	Objekt orientert Programmering
UDP	User Datagram Protocol
GUI	Brukergrensesnitt (graphical user interface)

1 INNLEDNING

1.1 Valg av oppgave

Vår oppgave omhandler utviklingen av en autonom dreven bil, som skal selv kunne detektere og følge definerte objekter. Bilen blir satt sammen av en robotplattform med 4 hjul, en robotarm og kamera montert i front. Roboten skal kunne følge en ensfarget ball ved hjelp av kamera og avstandsmåling. Når ballen er kommet nært nok, skal en arm på roboten bli aktivert slik at ballen skyves lengre unna roboten og deretter skal prosessen begynne på nytt.

1.2 Krav til oppgave

I tabell 1.1 har vi gjengitt hvilke funksjoner studieleder har satt som krav at prosjektet må inneholde, og hvilke prioriteter det innehar.

Funksjon	**Prioritet**	**Hardware**
Gjenkjenning av objekt med bildebehandling.	Skal ha.	Web kamera.
Trådstyrt bevegelser av robot.	Skal ha.	Robot Platform med integrerte motorer og hjul.
Trådstyrt bevegelser av servomotorer.	Skal ha.	Servomotor og robot arm.
Plukking av objekter ved hjelp av robot arm og bildebehandling.	Bør ha.	Robot arm, web kamera.
Kaste objekter ved hjelp av robot arm.	Kan ha.	Robot arm.
Identifisere fjes eller andre objekter med bildebehandling.	Kan ha.	Web kamera.

Tabell 1 - Funksjonskrav til prosjektet

1.3 Målsetting

For å gi en skisse over hva vi ønsker å oppnå med denne oppgaven har vi definert noen mål. Disse har vi videre definert i tre kategorier: Produkt-, prosjekt- og prosessmål. Med å definere disse målene, gjør vi det enklere for oss selv å se om ønsket resultat er oppnådd i enden av prosjektet.

1.3.1 Produktmål

Målsettingen med produktet (bilen) er at den skal ha følgende egenskaper:

- Bilen skal kunne utføre oppgitte oppgaver autonomt uten inngrep fra oss.
- Den skal både ha manuell og selvdreven funksjon, og skal være fullt funksjonell i begge modusene.
- Den skal være programmert slik at den kan utføre en rekke arbeidsoppgaver samtidig, uten at de forstyrrer hverandre

1.3.2 Prosjektmål

Siden vi i dette prosjektet har en avgrenset tidsperiode å arbeide på, samt koordinering av tid med andre prosjekter, er det viktig for oss at vi arbeider strukturert. Vi har da utviklet en detaljert plan med milepæler, som vi håper kan være til hjelp for å lykkes med dette. Denne blir også benyttet underveis med å se status til prosjektet. Dette har også blitt nøye oppfulgt av studieveileder da vi også har hatt i oppgave å sende inn oppdatert prosjektstatus hver 14 dag.

Nr	Milepæl	Dato
1	**Samle komponenter** *Få en oversikt over alle komponentene som skal brukes, og bestill de som mangler*	**uke 37**
2	**Starte på GUI** *Sette opp en enkel frame på GUI*	**uke 37**
3	**Kommunikasjon mellom GUI og Ordoid** *Få kommunikasjonen mellom GUI og Odroid til å virke. UDP kommunikasjon*	**uke 39**
4	**Kommunikasjon mellom Odroid og Mikrokontroller** *Seriell kommunikasjon klar mellom Odroid og kontroller*	**uke 40**
5	**Kontrollere bilen fullt fra GUI** *Test å sende kommandoer fra GUI til kontroller*	**uke 41**
6	**Kommunikasjon fra Mikrokontroller til GUI** *sending av sensorverdier til GUI*	**uke 43**

7	**Web kamera stream integrert i GUI** *Få integrert webkamera bilde i GUI bildet.*	**uke 45**
8	**Bilen klar til første "ferdige" test** *Alt satt sammen, og klar til første test*	**uke 46**
9	**Bilen ferdig** *Rettet opp i alle feil funnet under test*	**uke 47**
10	**Rapport innlevering** *Rapporten ferdigskrevet og levert*	**uke 48**

Tabell 2 - Prosessmål

1.3.3 Prosessmål

Prosessmålet i vår gruppe dreier seg i hovedsak om læring. Tilegne oss gode kunnskaper og nye erfaringer innen de forskjellige fagområdene vi vil berøre underveis i prosjektet.

Gjennom prosjektet vil vi komme godt innpå trådbasert programmering og benyttelse av samtidighet og sanntidssystemer i programmer, noe vi anser som veldig lærerikt og viktig i videre arbeid.

1.4 Problemstilling

«Hvordan kan vi programmere en robot på best mulig måte til å automatisk detektere og følge et spesifikt objekt ved hjelp av et kamera og en sensor».

En slik robotløsning har flere oppgaver som skal skje på samme tid; kameraet skal følge objektet, avstandssensor skal bedømme avstand mellom robot og objektet, en kontroller må bestemme hvilke kommandoer som skal utløses ut ifra verdier inn og kommunikasjons tråder skal skrive og lese verdier til og fra robot og kontrollsystem/GUI.

Programmeringsspråket Java har støtte for trådbasert programmering i klassene «Thread» og «Runnable». Med disse klassene så kan hver enkelt oppgave deles opp i tråder som går «parallelt» med hverandre, dvs. de deler på Odroiden's CPU-tid og kjører virtuelt samtidig. Et problem med slik trådbasert programmering er tråder som deler på en ressurs. Hvis en tråd skriver til en metode som oppdaterer et felt i en delt ressurs, «samtidig» som en annen tråd leser fra samme feltet i ressursen, kan det oppstå korrupte/ufullstendige data som leses ut.

Gruppens hovedoppgave vil bli å utvikle en best mulig trådbasert løsning for de gitte krav roboten skal kunne utføre.

1.5 Gruppens bakgrunn

Gruppen består av fire studenter med ulik bakgrunn:

Eivind Fugledal er fra Longva. Han gikk TAF-EL (Tekniske Allmenne fag innen programområde for elektrofag) ved Haram Videregående Skule. Han har fagbrev som automatiker, og har jobbet i Rolls-Royce i 7 år. Var først ansatt som lærling, før han jobbet i testavdelingen som tester av alarmsystem. Har også vært innom teknisk- og utviklingsavdelingen og jobbet med tegning av GUI og test av Power Management System, i tillegg til programmering av C++ i Qt. Har nå studiepermisjon under studiene.

Magnus Gribbestad er fra Brattvåg. Han gikk TAF-EL (Tekniske Allmenne fag innen programområde for elektrofag) ved Haram Videregående Skule. Han har fagbrev som Elektriker, og har jobbet i VARD Electro i 6 år. Først som lærling, deretter som montør. Har erfaring fra mange ulike områder og systemer på skip, og har vært bas i maskinrom, thrusterrom, og på IAS(alarmsystem på skip). Har studiepermisjon fra Vard Electro mens han studerer.

Jørgen Berge Trygstad er fra Ålesund. Har tidligere utdannelse fra BI (Økonomi og IT-Ledelse), med påfølgende 7 års arbeidserfaring som blant annet controller i Norges Bank, og IT konsulent ved ØkonomiBistand AS.

Lars-Harald Båtnes er fra Langevåg og har to fagbrev; Automasjonsmekaniker og Elektriker fra Borgund VGS. Han fullførte lærlingtid og arbeidet i daværende ODIM ASA (nå Rolls Royce) fra 2008 til 2011. Arbeidsoppgavene ved ODIM var innen montering/komplettering av vinsjesystemer/hydraulikk aggregat inkl. oppkobling og testing. Reisevirksomheter og bas ved utemonteringer. Fra april 2011 skiftet han til Cflow Fishhandling i Langevåg og har fremdeles stilling der som supervisor automasjon. Denne stillingen innebærer alt fra salg av reservedeler, prosjektering, utvikling og support. Han har nå en midlertidig redusert stilling der som er tilpasset skolegang.

2 TEORETISK GRUNNLAG

I dette kapittelet tar vi for oss det teoretiske grunnlaget for oppgaven. Teoriene som blir presentert er knyttet opp mot vårt pensum, og er en viktig del av arbeidet for å komme til en løsning opp mot vår problemstilling.

2.1 Nødvendige egenskaper med løsningen

Vi har tidligere utredet hvilke egenskaper vi ønsker å implementere i vår løsning. Vi vil her gå videre inn på hvilke teorier som er grunnleggende for oss, underliggende hver av de egenskapene. Vi har da spesielt fokus på:

- Sanntidssystemer (Utførelse av oppgaver i sanntid)
- Samtidighet (Kjøring av flere prosesser samtidig)

2.1.1 Sanntidssystemer (Real-time systems)

Sanntidssystemer (eng: Real-time system) er en beskrivelse på datasystemer som er avhengig av at systemets beregninger kommer innen en spesifisert tidsramme. Tidskravet kommer av at et sanntidssystem skal samhandle med et system i den virkelige verden som ofte er avhengig av tiden. Et sanntidssystem er ofte samtidige da de kan være en del av et større system som skal observere og kontrollere modeller som er avhengig av parallelle handlinger.

Sanntidssystemer kan klassifiseres i forhold til hvilke tidskrav de skal følge.

- Hard: Hvis en beregning ankommer etter tidsfristen er ute, så har beregningen ingen verdi. I slike systemer er en avhengig av at responsen kommer innen gitt tidsfrist. Det er ikke akseptert å ikke rekke en tidsfrist.
- Myk: I et slikt system er responstiden viktig, men systemet vil virke selv om tidsfristene ikke blir nådd en gang iblant.

Mange systemer kan ha både harde og myke sanntidsundersystemer. Videre kan slike systemer ha flere egenskaper:

- Store og komplekse: Kan være et enkelt system med svært få kodelinjer til store og komplekse system over flere plattformer og mange ulike programmeringsspråk.

- Pålitelig og sikker: Kan ofte være systemer som styrer viktige deler av samfunnet vårt. Det er derfor viktig at slike systemer kan fortsette å kjøre, selv om noen feil oppstår. Om systemet ikke kan fortsette å kjøre så er det viktig at det går over i trygg tilstand før det stenger ned.
- Sanntidskontroll: Gir mulighet til å synkronisere metoder og handlinger med tiden. Det kan være viktig å kunne spesifisere når noen handlinger skal begynne, når de skal avslutte og hvordan systemet skal reagere om en bommer på tiden.
- Interaksjon mot hardware: Systemet skal ofte lese og skrive til utganger på et virkelig system. Derfor kan det være viktig at et signal inn fra en enhet kan avbryte en pågående prosess. Dette kan for eksempel være om en trykker på en stoppknapp, da burde systemet behandle denne interaksjonen med en gang. (Lindsey,Javatech)(Concurrent and Real Time programming in Java)

I senere kapitler vil vi gå nærmere inn på hvilke teknikker og metoder vi har valgt for at systemet vi skal lage er i likhet med et system beskrevet over.

2.1.2 Samtidighet (Concurrency)

Samtidighetsprogrammering (eng: concurrency) er et kjent begrep som er brukt for å uttrykke potensiell parallellitet i en applikasjon. Samtidighet omfatter teknikker som behandler kommunikasjon og synkronisering mellom parallelle enheter.

Samtidighet er viktig for å kunne utnytte kraften i en prosessor. Moderne prosessorer kjører i veldig stor hastighet i forhold til kravene til lesing og skriving av innganger og utganger. Ved å bruke samtidighet kan en foreta beregninger, presentasjon og lesing av innganger samtidig. Alternativet er å bruke sekvensielle program, da kan en risikere at presentasjonen av dataen låses, mens en regner ut de neste dataene som skal presenteres.

Hvis en ønsker å bruke flere prosessorer for å løse et problem så er en avhengig av å bruke samtidighet. Sekvensielle program kan bare utføres på en prosessor, et samtidighetsprogram kan utføres på flere prosessorer slik at en oppnår ekte parallellisme og raskere utføringstid.

Samtidighetsapplikasjoner skal ofte kunne styre og være grensesnitt mot virkelige enheter som roboter, biler, samleband eller lignende som har behov for parallelle prosesser.

Emnet samtidighet introduserer problemer som er ukjent for sekvensielle program. Aktiviteter og hendelser som skal jobbe sammen må være synkronisert og koordinert.

Synkronisering av slike hendelser er svært viktig. Hvis en mislykkes med dette kan en få mange feil og problemer. Noen kjente problemer er:

- Vranglås (eng: deadlock): Kan forekomme om flere samtidige aktiviteter venter på at andre skal utføre noe.
- Interferens (eng: interference): Kan forekomme om to eller flere tråder prøver å oppdatere det samme objektet.
- Sult (eng: starvation): Kan forekomme om én eller flere tråder blir nektet tilgang til en ressurs på grunn av handlingen til andre tråder.

Det er viktig å nevne begrep som sikkerhet og «liveness» når en snakker om kvalitetene til en samtidighetsapplikasjon. Sikkerheten tar for seg at ingenting galt skal skje, som for eksempel at en får interferens slik at data kan bli korrupt. «Liveness» betyr at en skal sørge for at noe bra skjer, at systemet ikke går i vranglås eller lignende. (Concurrent and Real Time programming in Java) (Lindsey,Javatech)

Tråder

I et operativsystem så er en prosess en selvstendig programenhet. En tråd (eng: thread) er en prosess i et objekt orientert programmeringsspråk. Det vil si at i Java er tråder prosesser som kjører parallelt i Java Virtual Machine (JVM). Visst JVM kjører på én prosessor så vil trådene bare se ut til å kjøre parallelt, fordi det skiftes veldig fort mellom trådene. Selv om trådene i realiteten ikke kjører parallelt kan det gi mange fordeler i utvikling av samtidighets- og sanntidsprogrammer. I et multiprosessorsystem kan tråder virkelig kjøre parallelt siden JVM kan tildele Java tråder til operativsystemtråder.

En tråd brukes til å lage et samtidighetsprogram, slik at operasjoner kan skje tilsynelatende samtidig. Bruken av slike tråder kan skape utfordringer og problemer som vranglås, interferens og andre problemer nevnt i kapittel 2.6.1. (Concurrent and Real Time programming in Java) (Lindsey,Javatech)

Timer og Timertask

Timer og TimerTask er to klasser underliggende java.util. Hensikten med timer-funksjonene er å knytte trådfunksjoner relativt til systemklokken og ikke på timer avhengig av andre tråder (delay(), sleep()). Dette gjør da at en tråd kan starte avhengig av den reelle tiden, og ikke bli utsatt pga. andre tråder har brukt tid på å kjøre sin kode.

TimerTask er en hjelpeklasse for Timer der et Timer objekt kan holde på mange TimerTask objekter.

Semaphore

En semaphore er en abstrakt datatype eller en variabel som har i oppgave å kontrollere tilganger til en felles ressurs fra flere prosesser i et samtidighetsprogram. En semaphore er en verdi i en spesifikk plass i minnet i operativsystemet, som hver enkelt prosess kan sjekke og endre. Avhengig av verdien en får, så kan en prosess enten bruke fellesresursen eller finne ut at den er i bruk, og derfor måtte vente til den er tilgjengelig.

En binær semaphore har verdien 0 eller 1. Denne er gjensidig utelukkende. Dette vil si at om en prosess har anskaffet (eng: aquire) semaphoren så har ingen andre tilgang til denne ressursen før prosessen har sluppet (eng: release) semaphoren. En telle-semaphore er en tellevariabel som alltid kan telle oppover, men kun kan telle ned dersom verdien er større enn null. Dette betyr at en setter opp semaphoren med et visst antall tilganger, om en prosess anskaffer en semaphore så telles det nedover, når en prosess slipper så inkrementerer verdien. På denne måten så kan kun et definert antall prosesser få tilgang til en fellesressurs på samme tid.

En semaphoreløsning i Java kan implementeres ved å lage en egen semaphoreklasse, eller implementere de allerede eksisterende semaphoreløsningene fra «java.util.concurrent.Semaphore».

Buffer

En databuffer, eller bare buffer, er et definert område i minnet til en datamaskin som benyttes for å mellomlagre data mens dataene flyttes fra en destinasjon til en annen.

Normale bruksområder er:

- Lagre data i en buffer så snart det kommer inn fra en komponent
- Lagre data før det skal sendes ut til en komponent

En buffer har som regel en kø, der gitte algoritmer bestemmer hvilke data som skal prosesseres først og sist. (Wikipedia, 2016). Det blir som oftest benyttet FIFO (First in First out). Når bufferen er full, vil data som er ankommet i et senere tidspunkt stille seg i kø til det er gjort tilgjengelig ny plass i bufferen (flush metode)

(Wikipedia, 2016)

One-to-one, one-to-many, Many-to-many

Et kombinert system bestående av både brukernivåtråder og kernel-nivå tråder har fått beskrivelsen «multitråd modell». Her kan flere tråder innenfor samme applikasjon kjøre

parallelt på flere prosessorer uten at prosessen blir blokkert når en tråd kaller på «system block».

«One-to-One», «One-to-many» og «many-to-many» er tre typer av «multitråd modeller» vi har tilgjengelig.

- «One-to-one» beskriver situasjoner der hver enkelt bruker-nivå tråd er linket til en egen kernel tråd. Har bedre samtidighet enn one-to-many med tanke på at om en tråd kaller system blokk, vil det ikke påvirke andre tråder siden de kjører mot egen kernel tråd.
- «one–to-many» beskriver en situasjon der det er flere bruker-nivå tråder som er linket til en kernel-tråd. Problemet her er at om en tråd kaller på system blokk, vil hele prosessen bli blokkert. Det er bare én tråd som har adgang til kernel samtidig. (Silberschatz)
- «Many-to-many» er en modell der det blir benyttet mange bruker-tråder kombinert med likt, eller mindre antall kernel tråder. Er den modellen som gir best «samtidighet», med tanke på at om en tråd kaller et system blokk, kan kernelen gi beskjed til andre tråder å starte.

2.2 Viktigheter rundt programmeringsbruk

Når en benytter objekt-orientert programmering er det viktig å opprettholde godt design ved å følge gitte prinsipper som «cohesian» og «coupling». Når en møter utfordringer ved å opprettholde slike prinsipper, så kan en se på ulike «design patterns». Design patterns er problemer som har blitt løst før og kan hjelpe til med å få god design på en kode.

2.2.1 Cohesian

Begrepet «cohesian» er kjent innenfor objekt-orientert programmering, og snakker om sammenhengs-kraften til en modul i et program. Dette handler om hvilke oppgaver hver klasse utfører. I et program så ønsker en høy sammenhengs-kraft (eng: high cohesion), dette betyr at en klasse er fokusert på hvilke oppgaver den har. En klasse skal bare ha funksjoner relatert til det som er intensjonen for klassen. Dette vil si at om en har en klasse som skal sende data over UDP, så burde denne klassen bare ha funksjoner som er relevant til dette. Hvis denne klassen i tillegg har funksjoner for å motta data og regne ut noen verdier så har den lav sammenhengs-kraft (eng: low cohesian). (Michael Kolling,BlueJ, 2012)

2.2.2 Coupling

«Coupling» er også et kjent begrep innen OOP. Dette legger vekt på hvor tett knyttet to klasser eller moduler er, altså hvor avhengige de er av hverandre. Når en programmerer så ønsker en lav kobling (eng: loose coupling), dette vil i praksis si at om du endrer noe stort i en klasse, så skal det påvirke den andre klassen i minst mulig grad. Om et program har høy kobling (eng: high coupling) vil det være utfordrende å gjøre endringer, siden klassene er tett knyttet til hverandre. Da kan en liten endring i en klasse gjøre at en må endre hele systemet. (Michael Kolling,BlueJ, 2012)

2.2.3 Design patterns

Et designmønster (eng: design pattern) er en generell løsning til et vanlig problem innen programvareutvikling. Dette blir ofte brukt under utviklingsfasen av et datasystem siden mange designmønster tar for seg elementære programvarearkitekturer. Slike mønster er utviklet for å kunne optimalisere et datasystem slik en opprettholder gode design-prinsipper, samtidig som en tilegner utvalgte egenskaper. Slike ønskede egenskaper kan variere mye, men noen eksempler er: ytelse, forutsigbarhet, pålitelighet, sikkerhet, gjenbrukbarhet og skalering.

Utfordringen ved å bruke designmønster er ofte å lete frem til det som passer til den aktuelle problemstillingen. Et designmønster har som regel en overordnet beskrivelse av problemet som hjelper å finne det rette. Videre så inneholder de som regel implementasjonsstrategier, oversikt over struktur, UML-diagrammer, konsekvenser av implementering og relaterte mønstre. (Eric Freeman, Head Frist Design Patterns, 2004)

2.3 Kommunikasjon

Prosjektet vårt bygger på tre ulike komponenter (GUI, Mikrokontroller og Odroid) som behøver å kommunisere og sende data mellom hverandre. Måten vi vil løse dette er å ta i bruk ulike typer av kommunikasjonsprotokoller, samt sette opp ulike protokoller for dataoverføring mellom komponentene. Vi har valgt å benytte seriell kommunikasjon mellom mikrokontroller og Odroid og nettverksomkommunikasjon med UDP protokoll til kommunikasjon mellom Odroid og GUI.

2.3.1 Seriell kommunikasjon

Forskjellen mellom seriell- og parallellkommunikasjon er at en parallell kobling har flere ledere som går parallelt for hverandre, der data kan sendes over alle lederne samtidig, mens ved seriell kommunikasjon blir bare en leder brukt til å sende data. Seriell

kommunikasjon er prosessen som går på å sende data sekvensielt over kabel eller andre kommunikasjonskanaler en «bit» av gangen. Seriell kommunikasjon krever derfor færre ledninger, og kan som regel brukes over større avstander enn parallell kommunikasjon. Dette er på grunn av at ved parallell kommunikasjon kan det oppstå interferens mellom bits fra lederne, samtidig som en må bruke lavere båndbredde som betyr lavere bitrate.

En seriell port er det fysiske grensesnittet for seriell kommunikasjon. Ethernet, USB og FireWire sender data som en serie, men selve begrepet seriell port blir som regel brukt om utstyr som er kompatibelt med RS-232 standarden. Siden de fleste moderne datamaskiner ikke har en RS-232 plugg, kan det kreve at en har seriell til USB konvertere for å være kompatibel med RS-232 utstyr (ARC Electronics u.d.).

2.3.2 Protokoll

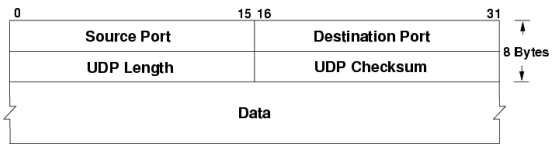
En protokoll kan sies å være en regel for hvordan sammenhengen mellom to endepunkter skal være definert. Dette kan innebære regler for:

- Hvilke kommunikasjonstyper som skal benyttes
- Hvilke datatyper som skal bli sendt
- Hvilken tilkobling som skal benyttes.

En protokoll kan implementeres i både programvare og maskinvare (eller kombinasjon av begge). (Wikipedia, Protokoll)

2.3.3 UDP – Server og Client

UDP (User Datagram Protocol) er en nettverksprotokoll for forbindelsesløs overføring av informasjon og opererer i transportlaget i OSI modellen. At UDP er forbindelsesløs vil si at det ikke opprettes noe fast forbindelse mellom sender og mottaker. Dette gjør at det ikke gis noen garantier for at en pakke blir levert og heller ingen tilstand om at pakken lagres hos avsender etter at pakken er sendt ut av nettverket. (wikipedia, u.d.) I figuren til høyre vises feltene som er tilknyttet UDP-Headeren. Som fremvist er det bare 4 felt, derav to er sender- og mottakerport. Grunnet at UDP protokollen er tilstandsløs, er det ikke nødvendig å fylle ut senderporten. Siden det ikke forventes noe svar tilbake, trenger heller ikke mottaker å vite hvor pakken kommer fra.



Figur 1 - Datagram packet

2.3.4 Port

For at en pakke sendt over nettverket skal ankomme ønsket destinasjon, trenger den en mottakeradresse. Denne må inneholde to ting:

- Server/Datamaskin navn
- Portnummer

Med tanke på at en datamaskin kan kjøre flere programmer samtidig, og som kommuniserer via samme kommunikasjonsmedier, må pakken vite hvor den skal gå når den har ankommet mottakermaskinen. Med andre ord, når den har ankommet datamaskinen, må den vite hvilket program den skal koble seg opp mot. Det er dette portnummeret har i oppgave å definere.
Normalt er portnummerene 1-1023 reservert til system applikasjoner, men videre er det fritt frem for hvilke portnummer som skal benyttes. (Notes, Ports)

2.3.5 Socket

En data socket er et endepunkt mellom en toveis kommunikasjonslink mellom to programmer kjørende på nettverket. Data som skal sendes fra et program blir puttet i en socket, og transportert videre over nettverket til en socket hos mottaker. Her blir socketen pakket ut og sendt videre oppover i OSI lagene. (notes, Ports)

3 MATERIALER OG METODER

I dette kapittelet vil vi gå tett inn på hvilke komponenter, programmer og metoder som er tatt i bruk for å løse de ulike utfordringene med vårt prosjekt.

3.1 Komponenter

Vi har gått nøye gjennom hvilke komponenter som trengs for å løse oppgaven på det vi mener er best mulig måte. I dette kapittelet vil vi gå gjennom alle komponentene vi har benyttet hver for seg, før vi i kapittel 4. Resultat vil vise det endelige resultatet hvor alle komponentene er satt sammen.

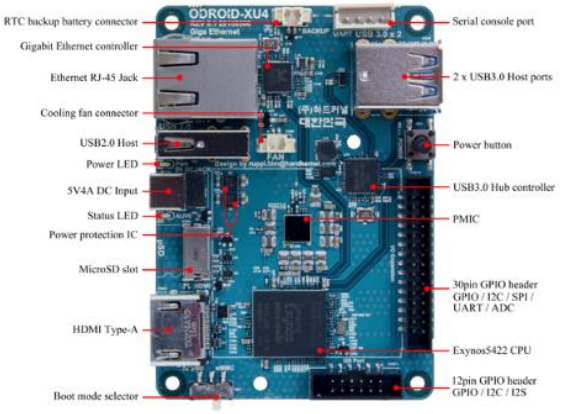
3.1.1 Odroid XU4

Navnet Odroid betyr Open + Droid. Det er en utviklingsplattform for hardware- og software-prosjekter. (Wiki, 2016).

En Odroid er en kraftig databehandligsenhet i en relativ liten størrelse som gjør det svært gunstig å benytte i prosjekter med krav på høy ytelse og begrenset plass.

Odroid er basert på «åpen kildekode», som gjør det mulig å benytte operativsystem etter egne ønsker som blant annet:

- Linux
- Android (lollipop og KitKat)



Figur 2 - Brettdetaljer Odroid XU4

I tabell 3 er det fremstilt full spesifikasjonsliste over Odroid XU4, og figur 2.2.1 viser brettdetaljene med oversikt over alle I/O.

Name	**Odroid XU4**
Released Year	2015
CPU	xynos 5422 Octa big.LITTLE ARM Cortex-A15 @ 2.0 GHz quad-core and Cortex-A7 quad-core CPUs
GPU	Mali-T628 MP6(OpenGL ES 3.0/2.0/1.1 and OpenCL 1.1 Full profile)
RAM	2 GB LPDDR3 RAM at 933 MHz (14.9 GB/s memory bandwidth) PoP stacked
Storage	microSD card slot, eMMC5.0 HS400 Flash Storage
USB	1 × USB 2.0 A Host, 2 x USB 3.0 Host
Video out	HDMIconnector 1.4a output Type-A
Audio in	
Audio out	HDMI
Network	10/100/1000Ethernet(8P8C)
Peripherals	expansion ports forGPIO, UART, I^2C,I^2S, SPI bus, PWM, ADC
Power source	5 volt @ 4 A

Tabell 3 - Spesifikasjkon Odroid XU4

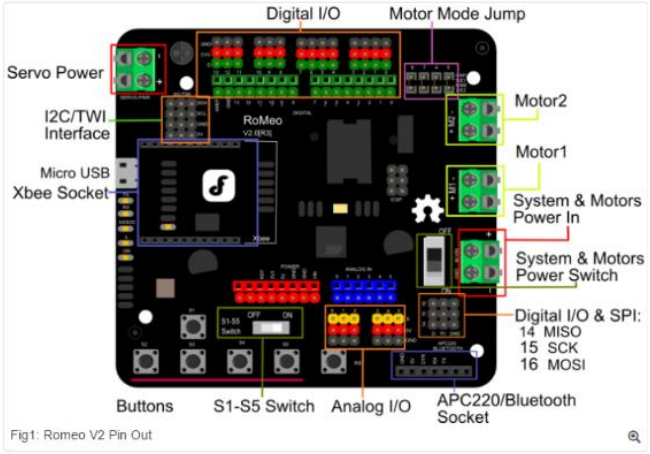
3.1.2 Arduino

«Arduino er en plattform for prototyping av elektronikk basert på program- og maskinvare med åpen kildekode» (Wiki, 2016)

Programmeringsspråket som benyttes i Arduino er C/C++, selv om selve Arduinos IDE er skrevet i Java.

3.1.3 Romeo V2.0 (R3)

Mikrokontrolleren vi har benyttet i prosjektet er et Romeo V2 brett. Dette er en «Alt i ett» Arduino kompatibel mikrokontroller. Fordelen med dette brettet er at det benytter seg av Arduino sin åpne plattform, kan lett kombineres med ett eller flere av Arduino's utbyggings kretskort (shield's) og har flere standard tilkoblinger enn en normal mikrokontroller.



Figur 3 - Romeo V2 PinOut

Romeo brettet er tilnærmet lik «Arduino Leonardo» mikrokontrolleren, noe som da også er viktig å spesifisere i Arduino programvaren under «tilkoblede kort». Som vist i figur 2.3.1 har Romeo kontrolleren også to integrerte DC-motor drivere, med egne Power in til motorene, noe som gjør det mulig for oss å koble våre motorer direkte inn på «Motor 1» og «Motor 2» uten noen ekstra komponenter.

3.1.4 Pixy

PixyCam, eller CMUcam5 ser ut som et normalt webkamera (som vises i figur 2.4), men fungerer i stedet som en bildesensor. Sensoren er programmert til å benytte objekt gjenkjennelse på både farge og fasong. Sensoren kan gjenkjenne 7 forskjellige «unike» farger samtidig. Det vil si at den kan programmeres til å samtidig gjenkjenne «uendelig» med objekter som innehar opptil 7 forskjellige farger. Pixy behandler bilder i 50fps (bilder i sekundet), det vil si at det vil bli behandlet en bilderamme pr 20ms. Dette gjør at vi får en utrolig rask respons tilbake til vårt system om hvor objektene befinner seg. Pixy har også en innebygd funksjon for å lære seg nye objekter. Her kan du holde det ønskede objektet foran sensoren, og når du trykker på knappen, vil det lyse et lys som markerer fargen på objektet du holder foran. På denne måten trenger du ikke noe program for å se hvilket objekt du har lært sensoren. Dette indikeres med fargen på led-lyset. Pixy kommer også med en egen software, PixyMoon. Denne er til oppsett av sensoren om du ønsker dette.

Figur 4 - Pixy sensor

Sensoren kan kobles direkte til Arduino /Romeo kortet via USB, men har også andre kommunikasjonsporter som I2C, UART og SPI.

3.1.5 HD Webcam c270

Et webcamera (på norsk: nettkamera) er et videokamera uten mulighet for lokal lagring, men istedenfor sender bildet direkte til en PC for videre prosessering. Bildet kan lagres direkte på mottakers PC, men kan også sendes som en direkte overføring, der ingen data blir lokalt lagret (live stream). Overføring av bildestrømmen kan skje både via kabel/internett. Bruksområder er blant annet:

- Kommunikasjon mellom to/flere parter via internett
- Overvåkning av omgivelser i et bestemt område
- Overføring av en bildestream til en GUI.
- Opptak av bilder/video til direkte lagring på lokal PC

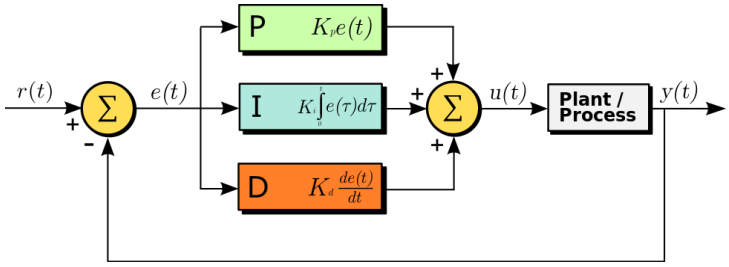
HD Webcam c270 overfører bilder med en maksimal oppløsning på 1280 * 720 piksler. Kameraet har innebygd mikrofon med støyreduksjon, som gjør det godt egnet til kommunikasjon mellom andre parter uten ekstra bakgrunnsstøy.

Det er dette kameraet vi har valgt til bildegjengivelse og objekt gjenkjenning i vårt system.

3.1.6 PID regulator

PID står for «Proporsjonal Integrasjon Deviasjon» (Wikipedia, 2016). En PID regulator benyttes ofte i industrien med å regulere elektriske og mekaniske apparater for å stabilisere ønskede verdier. Verdiene som skal stabiliseres kan være:

- Temperatur
- Vekt
- Volum
- Tykkelse



Figur 5 - PID

Det er tre verdier som blir benyttet i en PID regulator. Målerverdien (reell verdi), ønsket verdi og reguleringsavvik. (målt verdi minus reell verdi).

En PID regulator regner ut et pådrag som må tilføres systemet for å sette reell verdi lik ønsket verdi. Dette gjøres med å først finne reguleringsavviket for så å multiplisere dette med en gitt faktor (en faktor som er tilknyttet ønsket måleenhet/type). Dette utgjør et pådrag som for eksempel kan være «avvik * faktor (W) = pådrag (W)».

Vi har i vårt prosjekt benyttet PID regulator for å kontrollere hastigheten på motorene til roboten. Regulatoren vår vil prøve å minimere avviket mellom senter på objektet vi følger, og den vertikale senterlinjen i bilderammen til webkameraet.

3.1.7 Servo

Servomotorer blir brukt i mange forskjellige applikasjoner, de kan være små i størrelse, men ha et relativt kraftig vrimoment. Bruksområder er som regel radiostyrte leker, slik som modellfly, biler etc. Man har også industrielle versjoner av servomotorer som brukes til roboter og lignende.

Måten servomotoren fungerer på er at den har en posisjonsstyrt aksling, som vanligvis er koblet mot et internt gir i servoen. Motoren er kontrollert av et elektrisk signal som bestemmer posisjonen eller hastigheten til den ugående akslingen. Servomotoren består av en DC-motor, enkoder (potensiometer) og en kontrollkrets. Motoren er koblet med gir til utgående aksling. Når motoren roterer, endres verdien i enkoderen slik at kontrollkretsen kan presist regulere posisjon eller hastighet på akslingen, og i hvilken retning. Servomotorer for hobbybruk er som regel styrt av et pulsmodulert signal som bestemmer vinkel 0 til 180 grader, eller hastighet til servomotoren. (Jameco, 2016)

3.1.8 TP-LINK TL WR841N

TP-Link TL-WR841N er en trådløs nettverksrouter som er laget spesifikt for små nettverk. Hastigheten på 300Mbps gjør den ideell til blant annet video-streaming. Routeren gir også mulighet til å kunne sette statiske IP-adresser knyttet til MAC-adresser. (TP-LINK, 2016).

3.2 Programmer og bibliotek

Vi har gjennom hele prosjektperioden vært borti flere forskjellige programmer/systemer. Dette er alt fra program benyttet til selve programmeringen, til program benyttet til 3D-tegninger. Vi vil her gi et kort innblikk i alle programmene og bibliotekene vi har benyttet på veien for å kunne fullføre løsningen vår.

3.2.1 Netbeans

Netbeans er en IDE for utvikling av programmer i mange ulike programmeringsspråk som f.eks. Java, JavaScript, PHP, Python, Groovy, C, C++, Scala, CSS og HTML. Netbeans tilbyr gode muligheter for feilsøking og utvikling av Java-kode, som gjør det ideelt for vårt bruk (Netbeans).

3.2.2 GitHub

GitHub er et gratis og «open-source» system som brukes til å holde kontroll på versjoner, og oppdateringer av versjoner i prosjekt. Tanken bak er at flere kan jobbe på samme prosjekt, og få filer oppdatert automatisk, samt enkelt flette kode sammen i et felles prosjekt (GitHub,2015.).

3.2.3 Dropbox

Dropbox er en fildelingstjeneste som tilbyr muligheter for lagring av filer i nettsky, filsynkronisering mellom ulike enheter, personlig nettsky og delte mapper. Programmet kan brukes til å dele dokumenter, bilder og andre filer mellom flere personer, samtidig som filene på nettskyen automatisk blir en sikkerhetskopi (DropBox, 2015).

3.2.4 OpenCV

OpenCV står for «Open Source Computer Vision Library» og er et bildebehandling- og maskinlæringsbibliotek som er gratis å bruke for akademisk og kommersielt bruk. Det har brukergrensesnitt mot C++, C, Python, Java og MATLAB, i tillegg så støtter det Windows, Linux, Mac OS, iOS og Android. Biblioteket er laget slik det skal være mulig å bruke bildebehandling og maskinlæring på en enkel måte. OpenCV har mer enn 2500 optimaliserte algoritmer som kan brukes til å blant annet detektere og gjenkjenne ansikter, identifisere objekter, klassifisere menneskelige bevegelser i video, følge objekter i bevegelse, følge øyebevegelse og mye mer.

Det er mange kjente bedrifter som har tatt i bruk biblioteket, deriblant Google, Yahoo, Microsoft og Toyota. (opencv.org, 2016)

3.2.5 Swing

Swing er en GUI (Graphical User Interface, på norsk «grafisk brukergrensesnitt») modul for Java. Denne modulen gjør det enklere for brukeren å sette opp sitt eget brukergrensesnitt uten unødvendig mye kode. Swing-modulen inneholder flere komponenter som:

- Knapper
- Avkryssingsbokser
- Tekstbokser
- Lister
- Navigerbare ruter

- Tabeller

Hensikten med Swing er å være et bindeledd mellom kode strukturen og det visuelle grensesnittet brukeren ønsker. (Wikipedia, Swing(java))

3.2.6 PixyMoon

PixyMoon er en Software knyttet til Pixy sensoren. Dette er et program hvor du får opp et live bilde av hva sensoren detekterer. Vi kan med andre ord si at PixyMoon omgjør sensoren til et kamera der vi kan få frem bilder samt data sensoren registrerer. Her kan vi lære sensoren forskjellige typer objekter og se live hvordan sensoren detekterer objektene. Vi kan også sette opp verdier tilknyttet objektene (istedenfor at systemet auto-genererer titler som «block1» og «block2» kan det her settes opp «rød» og «blå» for å gjøre det mer oversiktlig.

3.2.7 RXTXcomm

RXTX er et ferdig kompilert bibliotek som gir tilgang til seriell og parallell kommunikasjon for Java utviklingsverktøy (JDK). Biblioteket er en videreført og oppdatert versjon av CommAPI'et produsert av Sun Microsystems. (Kean)

3.2.8 MiniPID

MiniPID er et Java prosjekt som er designet for å gi tilgang til en PID regulator som har enkel og effektiv tuning, i tillegg til enkel integrasjon uansett hvilket ledd av PID kontroll som er nødvendig. Prosjekt ligger fritt tilgjengelig på GitHub for nedlastning. (Sheadel, u.d.)

3.2.9 Autodesk Inventor

Autodesk Inventor er et 3D CAD modelleringsprogram som brukes til å designe, visualisere og teste produkt-idéer. Inventor gir mulighet til å teste et produkt i simulerte 3D-omgivelser, der man blant annet kan påføre produktet krefter og utføre en stressanalyse. Dette gir gode muligheter til å utvikle og teste et produkt uten å måtte lage fysiske prototyper. (Edulearn, 2016)

3.2.10 Cura

Cura er et program tilpasset til Ultimaker sine 3D-printere. Programmet har som oppgave å forberede 3D-modeller til printing. Det har 200 forskjellige innstillinger man kan endre på for best mulig resultat av 3D-printingen, i tillegg til noen ferdige profiler man kan bruke direkte. Cura gir altså mulighet til å ta 3D-modellen direkte fra for eksempel

Autodesk Inventor til 3D-print, så lenge man bruker filformatene STL, 3MF eller OBJ. (Ultimaker, 2016)

3.3 Metoder

I dette kapittelet vil vi se nærmere på hvilke metoder vi har tatt i bruk for å løse utfordringene i emnene:

- Objekt gjenkjennelse
- Prototyping
- Kommunikasjon
- Samtidighet

3.3.1 Objekt gjenkjennelse

En svært viktig funksjon i prosjektet er å finne en måte å detektere spesifikke objekter og få frem verdiene til lokasjonen de befinner seg på. Vi ble først tipset om en sensor (Pixy) som inneholder kode for objekt gjenkjenning, og gir ut avstandsverdier i x- og y-retning. Denne sensoren kan settes opp til å følge et objekt av en valgt farge. Når sensoren oppdager objektet som imøtekommer de innstilte parameterne, så vil den returnere x- og y-koordinatene til objektet, relativt til bilderammen som er gitt av kameraet.

Pixy sensoren har gitt oss en del utfordringer og problemer underveis i prosjektet, noe som har endt med at vi har bestemt å legge denne til side. Et webkamera skal brukes isteden. Et og et bilde hentes fra kameraet å går igjennom objekt gjenkjenningsprosessen.

Bildebehandling er prosesskrevende. For å redusere tiden det tar å behandle et bilde reduseres oppløsningen på bildene til 480x640. Objekt gjenkjenningen teknikken tar utgangspunkt i å følge en spesifisert farge. Den spesifiserte fargen defineres med maksimum- og minimumsverdier for «hue», «saturation» og «value/brightness» i henhold til objektet en ønsker å følge. Objekt gjenkjennings metoden kjører følgende punkt:

1. Filtrerer og glatter ut bildet ved bruk av et «Blur Normalize box filter» for glatting.
2. Konvertere bildet til HSV
3. Konverterer til et binært bilde der verdiene innenfor rekkevidden av fargeparameterne blir hvite, mens resten blir svarte.

4. Deretter fjernes salt og pepper støy. Dette glatter ut objektene for så å øke antall hvite piksler igjen.
5. Deretter hentes ut konturene i bildet. På denne måten får en ut grensene mellom det svarte og det hvite. På denne måten vet en hvor objektet en ønsker å søke ligger.
6. Finner senterpunktet på objektet og regner ut avstanden til senter.

Utregninger for tilbakesendte verdier

For å regne ut vinkelen mellom objektet og senterpunktet brukte vi parameterne fra kameraspesifikasjonene. Kameraet har et synsfelt på 60 grader horisontalt og 43,3 grader vertikalt. I tillegg bruker vi senterpunktet i bildet, samt senterpunktet av objektet.

1. Først regnes pikselavviket i x- og y-retning.

$$pixel\ Error\ X = x_{object} - x_{center}$$

$$pixel\ Error\ Y = -y_{object} - y_{center}$$

2. Deretter regner vi ut vinkelavviket.

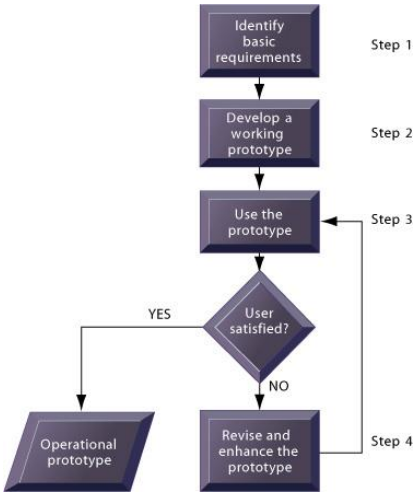
$$angle\ Error\ X = \frac{pixel\ Error\ X}{x_{center}} * CameraAngleX\ (60)$$

$$angle\ Error\ Y = \frac{pixel\ Error\ Y}{y_{center}} * CameraAngleY\ (43{,}3)$$

Metoden vi har brukt gir ikke helt nøyaktige vinkler, men er rikelig god nok til å få et forhold til hvor objektet er. Vinkelavviket i x-retning går mot 0 når objektet er rett foran kameraet. (opencv.org, 2016)

3.3.2 Prototyping

Med tanke på at prosjektet har den tidsrammen som vi har, vil vi utvikle produktet vårt med samme fremgangsmåte som en prototype. Vi har da valgt å benytte oss av «firestegsmodellen» til prototyping. (Laudon). Denne blir illustrert i figur 2.2.1.3
Vi vil utifra kravene fra oss og studieveileder (steg 1) utvikle en prototype, som vi ønsker å teste selv for å sammenligne med ønskede mål. Vi vil her være både utvikler og tester, siden produktet ikke er noe vi har fått på bestilling av eksterne oppdragsgivere.



Figur 6 - Prototyping

3.3.3 Samtidighet

Et av hovedmålene med prosjektet var at det skulle kunne gjøres flere arbeidsoppgaver samtidig uten forstyrrelser mellom hver prosess. Metoden vi har valgt å implementere i prosjektet for å løse dette på en god måte er tråd-programmering. Vi har da mulighet til å tilegne flere tråder forskjellige arbeidsoppgaver, og selv velge hvilke av trådene som skal ha noe med hverandre å gjøre.

4 RESULTAT

I underliggende kapittel vil vi først gjennomgå emne for emne resultatene vi har endt opp med, før vi til slutt viser den endelige løsningen for vårt prosjekt.

4.1 Programvareløsning

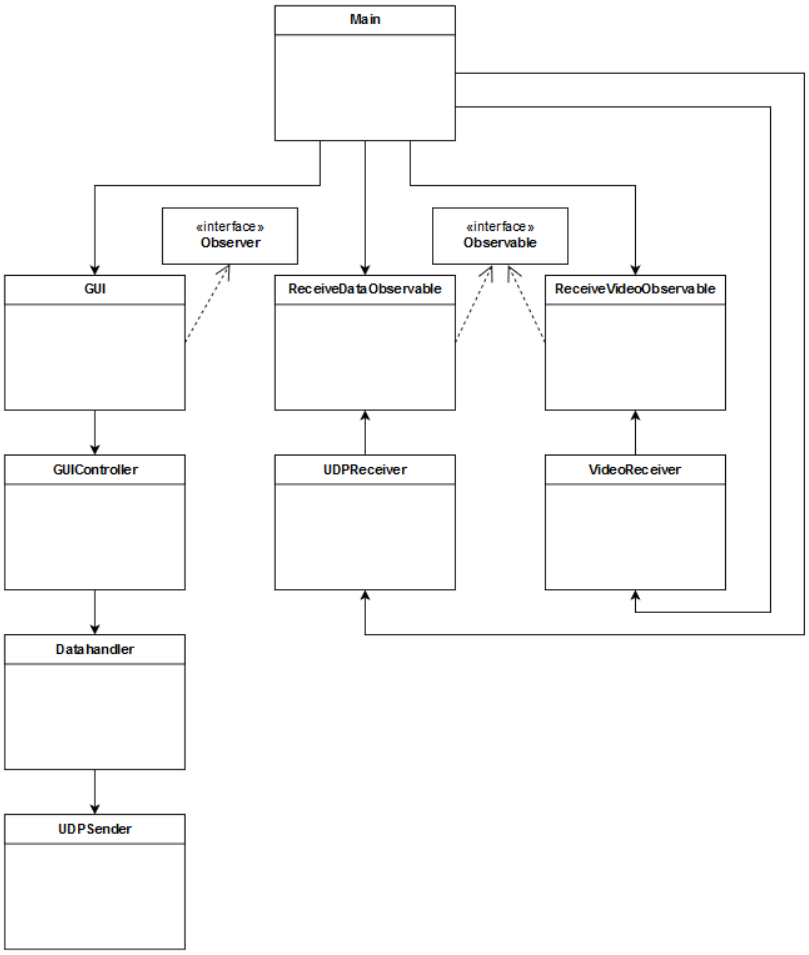
En stor utfordring en møter i programvareutvikling er å få en oppbygging av applikasjonen som holder seg innen gode designprinsipper beskrevet i kapittel 2.2. Måten vi har løst dette på blir beskrevet under. Her tar vi for oss systemet delt i to. Den ene delen er GUI siden. (Remote Controlleren). Den andre er selve bilen (Odroiden).

4.1.1 PC – Remote Controller

Utviklingen av programmet som kjører på PC' en og inneholder GUI var noe vi ønsket å utforme på en god måte. Boken «Head First: Design Patterns» ble flittig brukt til å søke etter et godt design for å kunne opprettholde løs kobling og høg sammenhengs-kraft. En av utfordringen her er å holde selve det grafiske brukergrensesnittet løst koblet mot resten av koden. I dette tilfelle var det viktige å tenke over at GUI' en ikke bare skulle sende kommandoer, men motta oppdateringer som skulle visualiseres i form av tekst eller bilde. Vi valgte her å bruke «Observer Pattern» for å få en god designløsning.

Under klassen som representerer selve presentasjonslaget så har vi en klasse: «GuiController» som knytter sammen underliggende program og presentasjonen. Dette ble gjort for at GUI' en skal vite minst mulig, men ha ansvaret med å være et grafisk brukergrensesnitt.

Den ferdige system oppbyggingen vi endte opp med ser slik ut:



Figur 7 - UML diagram for "Remote Controller"

«Remote Controller»-applikasjonen har både UDP servere og klienter. Denne kommunikasjonsløsningen er beskrevet i kapittel 4.1.7.

Klassen «UDPReceiver» ligger og venter på oppdatering av avstand og vinkeldifferanse fra objektet bilen følger. Når den mottar noe, settes verdiene i «ReceiveDataObservable»-klassen som arver metoder fra «Observable» grensesnittet. GUI 'en observerer den observerbare klassen og blir derfor varslet når oppdateringer ankommer. På denne måten kan visualiseringen oppdateres fortløpende ved oppdateringer, samtidig som gode design prinsipper blir fulgt. GUI 'en observerer også «ReceiveVideoObservable»-klassen slik at videostrømmen kan vises til operatøren.

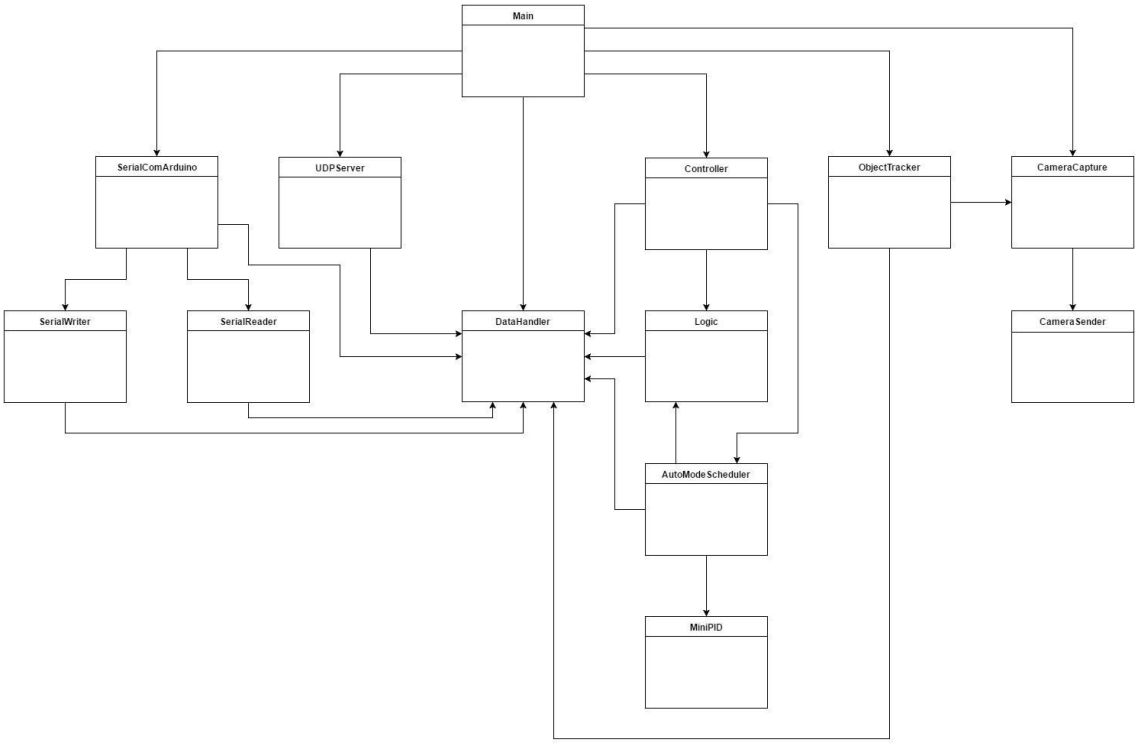
Tanken bak å ha både en «GUIController» og en «Datahandler» er at en får et klarere ansvarsområde for hver klasse. Den sist nevnte klassen har i ansvar å manipulere bytene som skal sendes, på denne måten vet verken visualiseringen eller «GUIController»-klassen noe om bit og bytes i henhold til protokollen. Da har vi klart å holde det grafiske brukergrensesnittet uvitende, mens den underliggende klassen kan binde sammen protokoll, byte og bits før det blir sendt.

4.1.2 Odroid

Når det kom til utviklingen av programmet som skulle ligge på Odroiden, altså bilen, så lå litt av utfordringen i at vi skulle både sende og motta data til/fra GUI, i tillegg til å sende og motta data til/fra Arduino. På grunn av dette ble det tidlig klart at vi måtte ha en klasse som inneholdt felles ressurser for å klare å få en løs kobling. Dermed har vi flere tråd-klasser som deler på fellesressurser ved hjelp av semaphore, slik at alle trådene får kjøre.

Fellesdataen vår ligger i DataHandler-klassen, mens UDPServer, Controller, SerialReader, SerialWriter og ObjectTracker er tråd-klassene som deler på disse dataene. Inne i DataHandler har vi da laget alle metodene som trengs for å kunne oppdatere og hente ut hele array av data, i tillegg til å kunne sette spesifikke bits og bytes. På denne måten er DataHandler den eneste klassen vi trenger å gjøre endringer i dersom vi skulle gjøre endringer i protokollen vår.

I og med vi har valgt å bruke semaphore til å organisere tråd-kjøringen vår er vi sikret at alle trådene får kjøre ved et eller annet tidspunkt ettersom denne har mulighet til å aktivere ''fairness''. ''Fairness'' fungerer på den måten at etter en tråd har sluppet semaphoren vil den tråden som har ventet lengst være den neste som tar den. Dersom en ikke bruker ''fairness'' risikerer en at en tråd slipper semaphoren for deretter å ta den igjen direkte og dermed hindre de andre trådene fra å noen gang få kjøre. Med andre ord tillater semaphore i sammenheng med ''fairness'' at en tråd får ta semaphoren, gjennomføre det den skal, for så at neste tråd får ta semaphoren.



Figur 8 - UML diagram for Odroid applikasjon

4.1.3 Objekt gjenkjenning

Vi endte som sagt opp med å legge bort arbeidet med Pixy sensoren og selv legge inn den nødvendige koden for å få til objekt gjenkjenningen. Noen av utfordringene med pixy har vært:

- Vi trodde først dette var et fullt funksjonelt kamera som i tillegg inneholdt den koden vi trengte for å utføre den nødvendige bildebehandlingen. Vi fant i senere tid ut at det ikke kunne benyttes som et kamera (overføring av bilde), men bare overføring av tekstinformasjon som da er knyttet til objektet det har funnet (sensor).
- I koden oppstod det problemer så snart pixy biblioteket ble initialisert i programmet. Det vil si at allerede i setup() funksjonen i Arduino softwaren, oppstod det problemer når pixy.init() ble kalt. Det virket som om denne koden forstyrret de andre bibliotekene, som da gjorde at det ikke var noe av programmet som fungerte som det skulle. Romeo-kortet frøs, og måtte resettes for å virke igjen. Med litt undersøking på nett, fant vi ut at vi heller ikke var de første som har opplevd dette, men også at det ikke var noe spesifikk løsning på det.
- En har mindre muligheter for å justere parameterne til objekt gjenkjenningen ved å bruke pixy kameraet. Dette resulterte i at enkelte ganger detekterte sensoren flere objekter enn det vi ønsket å følge. Dette resulterte i at det ble returnert x- og y-koordinater til flere objekter, uten mulighet for å vite hvilke objekt som var det ønskede.

Med tanke på punkt én over, var det allerede bestemt at vi måtte integrere et webkamera på bilen for å få live bildeoverføring til GUI'en vår. Dette mener vi er et krav med tanke på at bilen også skal kunne kjøre i manuell modus.

Etter å ha benyttet mye tid på å få Pixy sensoren til å fungere, bestemte gruppen seg for at det var mer tidsbesparende å bare benytte oss av det webkameraet som vi allerede har integrert til bilen, og også benytte dette til bildebehandlingen.

Webkameraet skal erstatte pixy sensoren og brukes til både objekt gjenkjennelse og videooverføring. Kameraet har 720p oppløsning og returnerer bilder i formatet 1280 x 960 (1.2 MP). Den maksimale bilderaten er på 30 bilder per sekund (fps). Bildene kameraet returnerer er av god kvalitet, men fargene i bildet endrer seg mye avhengig av lysforhold. Det har derfor vært en utfordring å finne et objekt som egner seg godt til gjenkjenning.

Kameraet kobles direkte til Odroiden med USB og objekt gjenkjenningen skal skje der. Siden prosjektet utvikles i Java ble det bestemt at vi skulle bruke biblioteket OpenCV, da

dette er mye brukt til bildebehandling. Metoden for objekt gjenkjenning er beskrevet i kapittel 3.3.1, mens en beskrivelse av implementasjonen er beskrevet under.

Verdiene som returneres fra kameraet er på en rekkevidde fra: «-60 til 60», som forteller oss hvilken retning fra senter objektet befinner seg.

Om objektet befinner seg i senter, returneres verdien 0.

For å unngå at programmet går i lås om det ikke detekteres noe objekt har vi også satt opp to faste variabel verdier på -100 og 100. Dette er verdier som blir levert til systemet om det ikke er noe objekt som blir detektert i bilderammen. Begrunnelsen for at vi har levert to verdier (både positiv og negativ 100) er at dette forteller oss i hvilken retning objektet ble sist detektert. Om vi har en retur verdi på -100, forteller vi systemet at det siste observerte objekt var på venstre side, som da gjør at bilden kan svinge denne veien å se om det fortsatt befinner seg noe objekt der.

Prosessering av data

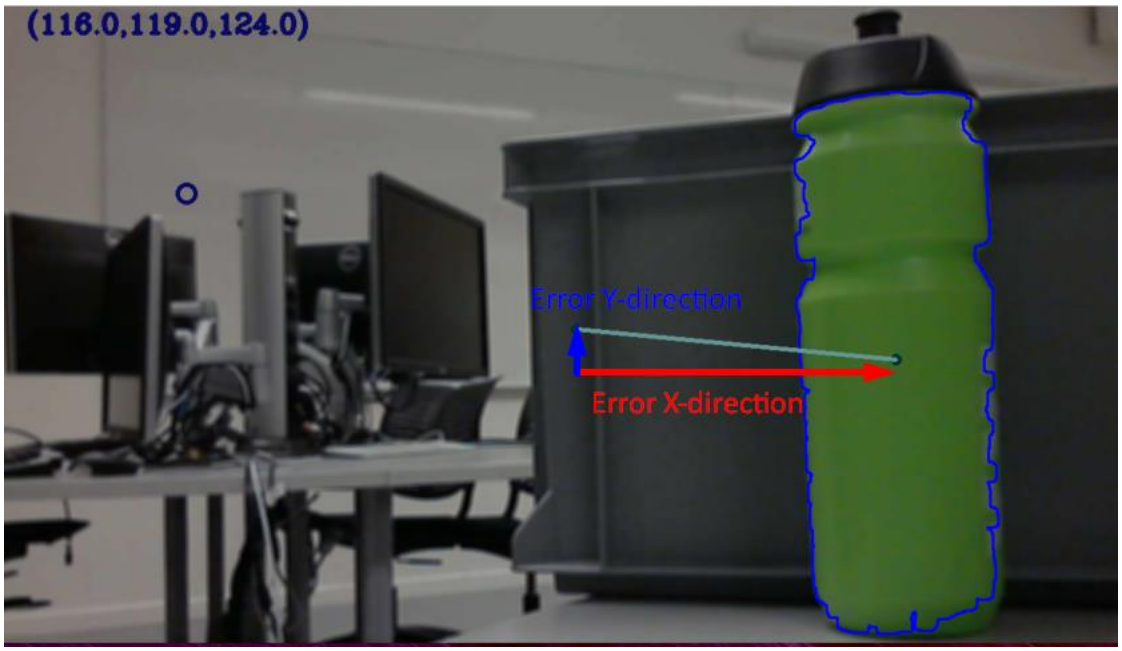
All dataprosessering vedrørende objekt gjenkjenning har vi lagt i klassen «ObjektTracker» som ligger på Odroid siden. Denne inneholder fem viktige funksjoner:

- setColorTrackingValues()
- trackColors()
- getTargetError()
- getX()
- getY()

Den første funksjonen setter fargeparameterne som en ønsker å søke etter. Funksjonen tar inn maksimum og minimum verdier for «hue», «saturation» og «value». Disse parameterne blir brukt i «trackColors()». Denne gjør punkt 1-5 i kapittel 3.3.1. Det blir brukt funksjoner som «cvtColor», «inRange» og «blur» fra OpenCv biblioteket for å utføre disse oppgavene. Funksjonen «getTargetError» regner ut differansevinklene fra konturene funnet i «trackColors». Metoden som er brukt for å hente ut vinklene er beskrevet i kapittel 3.3.1.

Funksjonen som følger objektet tar for seg et bilde av gangen, og søker etter farger innenfor de gitte minimum og maksimum verdiene av farge og lysforhold. Når et objekt er innenfor kravene, settes det en ring rundt objektet og tyngdepunktet regnes ut og blir illustrert med en prikk på objektet. Deretter trekkes en linje i fra senter av skjermen og ut til tyngdepunktet på objektet. Da kan en enkelt bruke metodene beskrevet i kapittel 3.3.1 for å regne ut differansevinkel i x- og y-retning. Denne metoden, og bildet vist

under viser bare hva som skjer i bakgrunnen av bildebehandlingen. For å redusere både prosessorkraft og forsinkelse for bildevisualisering i GUI har vi valgt å ikke vise eller sende de behandlede bildene videre, men bare hente ut nyttig informasjon.



Figur 9 - Prinsipp for objekt gjenkjenning

Videovisualisering i GUI

Selve prosesseringen og utregningen av objekt gjenkjennelsen utføres som sagt på Odroid siden. Verdiene av den prosesserte dataen blir så sendt videre oppover, og visualisert i GUI applikasjonen. For å få til dette endte vi opp med å definere to nye javaklasser:

- VideoReceiver
- CameraThread

VideoReceiver

Denne klassen tar for seg selve mottaket av bildestrømmen over nettverket. Bildet blir sendt som en byteArray i en DatagramPacket inn mot port «8765». Figur 10 viser koden vi har lagt inn i VideoReceiver klassen som tar seg av mottaket av bildestrømmen. Vi har også valgt å skalere bildet som kommer inn til en fast verdi på 640px * 480px.

```
public void run()
{
    try {
        serverSocket = new DatagramSocket(this.videoPort);
        BufferedImage buff;

        byte[] receiveData = new byte[57654];

        while(true)
        {
            DatagramPacket receivePacket = new DatagramPacket(receiveData, receiveData.length);
            serverSocket.receive(receivePacket);
            this.scale(receiveData, 640, 480);

        }
    } catch (SocketException ex) {
        Logger.getLogger(UDPreceiver.class.getName()).log(Level.SEVERE, null, ex);
    } catch (IOException ex) {
        Logger.getLogger(UDPreceiver.class.getName()).log(Level.SEVERE, null, ex);
    }
}
```

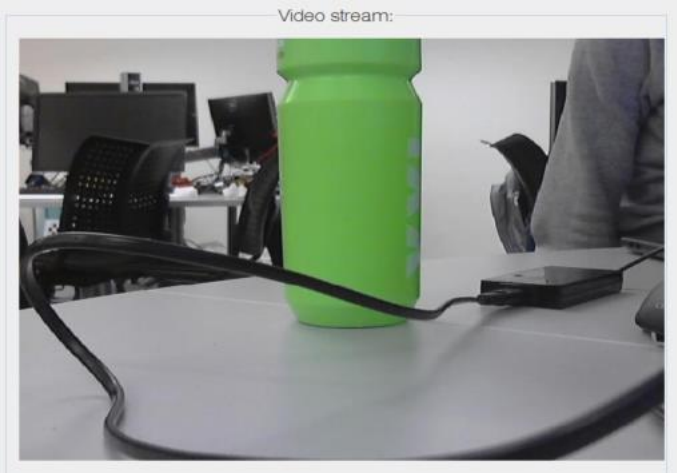
Figur 10 - Utsnitt av UDP for videooverføring

CameraThread

Oppgaven med denne klassen er å gjenskape bildet som er mottatt i VideoReceiver klassen. Vi henter bildet ut med receiver.getImage(), og oppretter det på ny med de to linjene :

```
Graphics g=videoPanel.getGraphics();
g.drawImage(buff, 0, 0, videoPanel.getWidth(), videoPanel.getHeight() -150 , 0, 0, buff.getWidth(), buff.getHeight(), null);
```

Det endelige resultatet av hvordan dette ble seende ut hos oss vises i figur 4.2.2 under.



Figur 11 - Utsnitt av Video stream fra GUI

4.1.4 Fremdrift

I oppgaven er det gitt at bilen skal være både selvdreven og manual. Dette har vi løst med å ha en knapp i GUI'en, som vi kan veksle mellom auto og manual modus.

Autonom modus

Når bilen er i Auto-mode er det informasjonen fra kameraet som bestemmer hvordan bilen skal oppføre seg. Kameraet sender x-verdier (verdier fra -60 til 60 som tilsvarer vinkelen fra senter i kamerarammen) til bilen. Verdiene blir videre prosessert i Odroid'en som da kaller på de nødvendige metodene for å fange objektet. Vi har benyttet en PID regulator som skal gjøre at bevegelsen til bilen skal gå glattest mulig. Dette gjør at når kameraet detekterer et objekt utenfor senter (har en vinkel ulik 0), vil det sendes et ønsket settpunkt til PID regulatoren, som da justerer hastigheten til motorene slik «reel vinkel == 0».

PID regulatoren kommer fra et ferdig bibliotek tilgjengelig på GitHub (MiniPID). (GitHub, 2016)

Vi valgte denne regulatoren i stedet for å lage vår egen på grunn av at dette ikke er hovedfokuset for oppgaven. Biblioteket var godt dokumentert med ferdige eksempler som kunne tas i bruk øyeblikkelig.

PID regulatoren får tilgang til den delte resursen datahandler (dh) via Semaphore objektet. Vinkelen gitt fra kamera leses ut og lagres i feltet xAngle. Etter dette så slippes tilgangen fri til dh for neste tråd. Videre så foretas en beregning av prosessverdi ut fra PID regulator og det sjekkes at ny verdi er ulik forrige verdi. Hvis dette er oppfylt så hentes tilgang til dh på ny og det skrives deretter nye verdier til motorene på roboten. Er objektet til venstre for roboten så vil venstre hjulsett slakke ned hastigheten mens høyre hjulsett vil øke hastigheten. På denne måten ønsker vi å oppnå en jamn hastighet rett frem og når roboten svinger.

```
/**
 * PID tracking object, object in sight of  the camera
 */
private void advance() {

    output = pid.getOutput(xAngle, setpoint); // pid regulator
    output = limit(output, -pidOutputLimit, pidOutputLimit); // begrens output

    float leftSpeed = (255f / 2f) + (float) output;
    float rightSpeed = (255f / 2f) - (float) output;

    acquire();
    logic.runFWD(leftSpeed, rightSpeed);
    logic.decideToHitBallOrNot(dh.getDistanceSensor());
    dh.incrementRequestCode();
    release();
}
```

Figur 12 - Utsnitt av run metode i autonom modus

Når objektet er nært nok roboten, så skal en arm svinge ut fra roboten og skyve det videre vekk i fra roboten. Avstanden til objektet måles med en IR sensor som har område fra 4-30 cm. Sensoren er koblet til Romeo-brikken som sender den skalerte verdien via seriell til Odroid. Måten robotarmen er kontrollert på er at når avstanden er nær nok mellom robot og objektet så trigges en metode for å svinge armen ut. Denne metoden starter en ny tråd som svinger armen ut og deretter benytter seg av Thread.sleep() metoden for å lage et 2 sekunders delay før den svinger armen inn igjen og stenger tråden. På denne måten vil ikke metoden blokkere for andre tråder.

Manuell modus

I manuell modus kan roboten kontrolleres fra piltastene på tastaturet til en klient-PC (GUI) koblet til Odroid via Wifi. Måten dette blir gjort på er at GUI sender kommandoer til Odroid via UDP gjennom en felles protokoll. Videre så spør UDP server etter tilgang til semaphore objektet og et byte-array blir oppdatert i datahandler.

```
public void setDataToDatahandler(byte[] data) {
    this.acquire();

    dh.setDataFromGUI(data);

    this.release();
}
```

Figur 13 - Utsnitt fra Datahandler

For å prosessere data fra GUI, hvilken knapper er påvirket etc, kaller Controller klassen på en hjelpe klasse som inneholder all logikk for kontrollering av roboten. Denne klassen er nærmere beskrevet i avsnitt 3.4. Logikk klassen krever en del spørringer og skrivinger til datahandler for å utføre kommandoene satt fra GUI, desse operasjonene mot datahandler er forholdsvis små, med lite eller ingen aritmiske operasjoner, som fører til at metodene vil utføres forholdsvis raskt. Derfor har vi valgt å gi tilgang til semaphoree objektet helt i starten og slutten av metoden fra Controller. På denne måten får vi en ryddig kode som blir enklere å følge.

```
/**
 * Logic while running in manual mode
 */
private void runManual() {
    if (dh.getDataFromGuiAvailable()) {
        acquire();

        if (dh.isDataFromArduinoAvailable()) {
            System.out.println("Camera x value: " + dh.getPixyXvalue());
            System.out.println("Camera y value: " + dh.getPixyYvalue());
            System.out.println("Distance: " + dh.getDistanceSensor());
        }
        logic.prossesButtonCommandsFromGui();

        dh.setDataFromGuiAvailable(false);
        release();
    }
}
```

Figur 14 - Utsnitt fra logikk for manuell kjøring

```
protected void prossesButtonCommandsFromGui() {
    this.handleButtonStates();
    this.handleServoStatesFromGui();
    this.switchCaseButtonStates();
    this.switchCaseMotorSpeeds();

}
```

Figur 15 - Utsnitt fra prosessering av GUi kommandoer

Vist ovenfor er metoden for manuell kjøring, her testes en bit om nye data er tilgjengelig fra GUI før semaphore blir spurt etter. Testing av en bit har vi vurdert sammen med faglærer at det er en trygg operasjon som kan utføres utenfor synkronisering, siden det har bare to mulige tilstander og har ikke mulighet til å bli korrupt. På denne måten

unngår vi at semaphore spørres etter hvis ikke det er nye data tilgjengelig for prosessering.

4.1.5 Logikk

Logikk for kontrollering av roboten er lagt til i en hjelpeklasse som Controller benytter seg av. Denne klassen kalles Logic. Her er alle funksjonene for kjøring og kontrollering av servo-bevegelser samlet.

```
protected void handleButtonStates() {
    // setter først buttonstate til null
    int buttonState = 0;
    // sjekker at controlbyte er ulik null
    if (0 != dh.getFromGuiByte((byte) 0)) {
        // sjekker at ingen kommandoer er ulovlige (frem/bak samtidig)
        if (!(((1 == dh.getFwd()) && (1 == dh.getRev())) || ((1 == dh.getLeft()) && (1 == dh.getRight())))) {
            // gå frem
            if (1 == dh.getFwd()) {

                buttonState = STATES.GOFWD.getValue();
                // gå bakover
            } else if (1 == dh.getRev()) {
                buttonState = STATES.GOREV.getValue();
            }
            if (1 == dh.getLeft()) {
                buttonState += STATES.GOLEFT.getValue();
            } else if (1 == dh.getRight()) {
                buttonState += STATES.GORIGHT.getValue();
            }
        }
    }
    this.setStateByValue(buttonState);
}
```

Figur 16 - Utsnitt for å håndtere tilstander

Button state regnes ut med en algoritme som setter verdi 1 hvis frem er påvirket fra GUI, eller -1 hvis bakover er påvirket. Videre så vil frem + venstre gi verdien 1 + 10 = 11, eller frem + høyre 1 + 20 = 21 (9 og 19 for bakover). På denne måten får vi unike koder for hver av kommandoene som kan videre brukes for å kontrollere retningen til roboten.

Retningen er gitt ved at roboten må ha kommandoer for å vite hvilken vei den skal kjøre; frem, bak, venstre eller høyre. Videre så må hastigheten til hjulsettene settes riktig i forhold til gitt kommando. Disse funksjonene utføres i to forskjellige switch case statements.

```java
protected void switchCaseButtonStates() {
    dh.resetToArduinoByte(0);
    switch (this.getState()) {
        case STOP:
            dh.stopAUV();
            break;
        case GOFWD:
            dh.goFwd();
            break;
        case GOREV:
            dh.goRev();
            break;
        case GOLEFT:
            dh.goLeft();
            break;
        case GORIGHT:
            dh.goRight();
            break;
        case GOFWDANDLEFT:
            dh.goFwd();
            break;
        case GOFWDANDRIGHT:
            dh.goFwd();
            break;
        case GOREVANDRIGHT:
            dh.goRev();
            break;
        case GOREVANDLEFT:
            dh.goRev();
            break;
        // unknown command
        case DEFAULT:
            break;
        // just to be safe
        default:
            break;
    }

}
```

```java
protected void switchCaseMotorSpeeds() {
    switch (this.getState()) {
        case STOP:
            dh.setLeftMotorSpeed(minSpeed);
            dh.setRightMotorSpeed(minSpeed);
            break;
        case GOFWD:
            dh.setLeftMotorSpeed(maxSpeed);
            dh.setRightMotorSpeed(maxSpeed);
            break;
        case GOREV:
            dh.setLeftMotorSpeed(maxSpeed);
            dh.setRightMotorSpeed(maxSpeed);
            break;
        case GOLEFT:
            dh.setLeftMotorSpeed(maxSpeed);
            dh.setRightMotorSpeed(maxSpeed);
            break;
        case GORIGHT:
            dh.setLeftMotorSpeed(maxSpeed);
            dh.setRightMotorSpeed(maxSpeed);
            break;
        case GOFWDANDLEFT:
            dh.setLeftMotorSpeed(maxSpeed / 4);
            dh.setRightMotorSpeed(maxSpeed);
            break;
        case GOFWDANDRIGHT:
            dh.setLeftMotorSpeed(maxSpeed);
            dh.setRightMotorSpeed(maxSpeed / 4);
            break;
        case GOREVANDRIGHT:
            dh.setLeftMotorSpeed(maxSpeed);
            dh.setRightMotorSpeed(maxSpeed / 4);
            break;
        case GOREVANDLEFT:
            dh.setLeftMotorSpeed(maxSpeed / 4);
            dh.setRightMotorSpeed(maxSpeed);
            break;
```

Figur 17: A - Kjøretilstander , B - Hastighetstilstander

4.1.6 Seriell kommunikasjon mot Romeo mikrokontroller

På roboten endte vi opp med å benytte seriell kommunikasjon via USB for å kommunisere mellom Odroid og Romeo mikrokontrolleren. Dette har vi løst med å benytte et bibliotek for seriell kommunikasjon i Java, RXTXcomm. Ved hjelp av dette biblioteket har vi laget klassen SerialComArduino, som setter opp innstillinger til seriell port og input/output streams. Med disse innstillingene så opprettes to tråder, SerialReader og SerialWriter i metoden connect.

```java
SerialComArduino sca = new SerialComArduino(dh);
try {
    sca.connect(comport[1], semaphore);
} catch (Exception ex) {
    Logger.getLogger(Main.class.getName()).log(Level.SEVERE, null, ex);
}
```

```java
reader = new Thread(new SerialReader(in,datahandler, semaphore));
writer = new Thread(new SerialWriter(out,datahandler, semaphore));

writer.start();
reader.start();
```

Figur 18 - Utsnitt av tråder i metoden connect

Disse trådene har som oppgave å skrive data eller lese data til og fra Romeo'en. SerialWriter klassen har en løkke som sjekker om nye data i datahandler er tilgjengelig for skriving til Romeo. Hvis nye data er tilgjengelig så hentes tilgang til datahandleren fra semaphore objektet og dataene hentes ut, deretter løses tilgangen fra semaphore og data blir skrevet til Romeo via seriell. På denne måten vil tilgangen til semaphore bli holdt så kort som mulig før den gir tilgang til neste tråd i køen.

```
public void run() {
    try {
        while (datahandler.shouldThreadRun()) {

            if (datahandler.checkSendDataAvailable()) {
                acquire();
                byte[] sendByte = datahandler.getDataFromController();
                release();
                System.out.println(Arrays.toString(sendByte) + "SERIAL");
                this.out.write(sendByte);
                this.out.flush();
            }
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

Figur 19 - Utsnitt av tråder i SerialWriter

SerialReader fungerer mye på samme måten, den lytter til en seriell port og venter til nye data kan leses ut. Når dette er gjort så hentes tilgang til semaphore og de nye dataene skrives til datahandler og tilslutt sleppes tilgang fra semaphore fri.

```
@Override
public void run() {
    while (dh.shouldThreadRun()) {
        try {
            byte[] data = new byte[6];
            in.read(data,0,data.length);
            System.out.println(Arrays.toString(data));

            semaphore.acquire();

            dh.handleDataFromArduino(data);
        } catch (InterruptedException ex) {
            Logger.getLogger(SerialWriter.class.getName()).log(Level.SEVERE, null, ex);
        } catch (IOException ex) {
            Logger.getLogger(SerialReader.class.getName()).log(Level.SEVERE, null, ex);

        } finally {
            semaphore.release();
        }
    }
}
```

Figur 20 - Utsnitt av tråder i SerialReader

4.1.7 UDP kommunikasjon

Når det kommer til UDP kommunikasjon har vi valgt å sette opp én server og to klienter på Odroiden (bilen), i tillegg til én klient og to servere på PC'en (GUI). Grunnen til at vi har valgt å ha to klienter på bilen er fordi vi har en egen klient for videostreaming, og en som sender de resterende dataene.

For å teste kommunikasjonen mellom komponentene våre, var det mest praktisk at vi satte opp et privat nettverk. Dette var på grunn av begrensninger på nettverket i labbygget ved NTNU i Ålesund. På denne måten fikk vi tilgang til kommunikasjon mellom PC (GUI) og robot (Odroid) uten å måtte skaffe spesielle tillatelser fra IT-ansvarlige ved NTNU.

I og med at vi har satt opp vårt eget lokale nettverk har vi gitt en statisk IP-adresse til Odroiden. På denne måten kan GUI'en koble seg til uten at vi må tenke på IP-adressen for hver gang. Når GUI'en da kobler seg til Odroiden kan vi hente ut IP-adressen i datagrampakken vi mottar i Odroiden, slik at Odroiden sine klienter kan koble seg til GUI'en igjen.

```
try {

    DatagramPacket packet = new DatagramPacket(data,
                                data.length,
                                InetAddress.getByName(ipAddress),
                                port);
    clientSocket.send(packet);
```

Figur 21 - Utsnitt fra DatagramPacket

Figuren over er hentet fra GUI'en sin UDP klient, der vi ser at den lager en datagrampakke som inneholder IP-adressen og port-nummer til UDP-serveren på Odroiden.

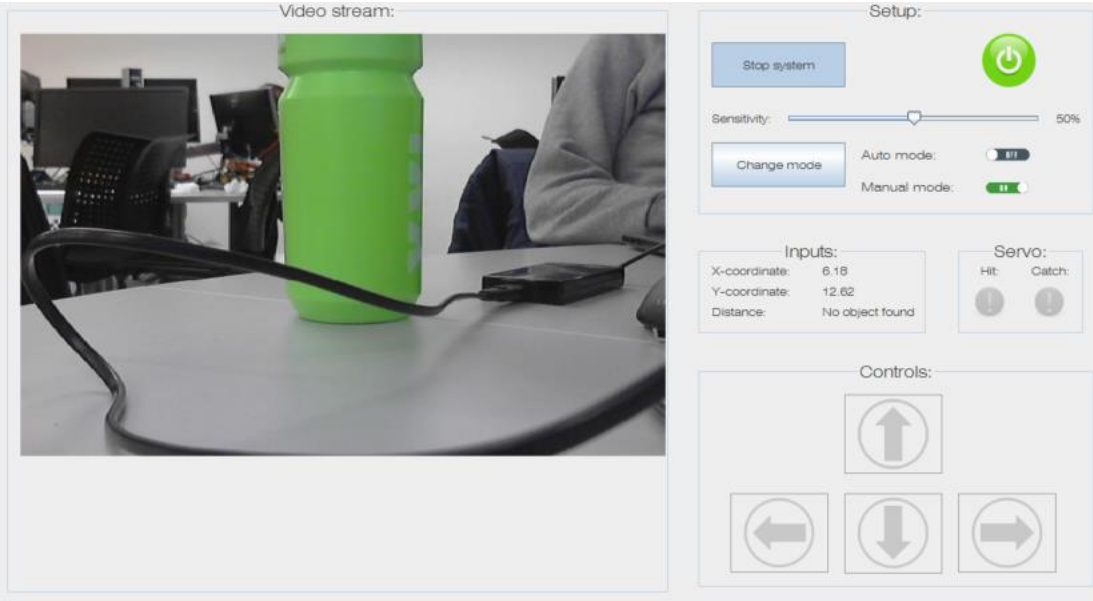
```
try {
    serverSocket = new DatagramSocket(serverPort);

    byte[] receiveData = new byte[6];

    while (dh.shouldThreadRun()) {
        DatagramPacket receivePacket = new DatagramPacket(receiveData, receiveData.length);
        serverSocket.receive(receivePacket);

        Main.ipAdress = receivePacket.getAddress().getHostAddress();
```

Figur 22 - Utsnitt fra UDP Server på Odroid

Figuren over er fra Odroiden sin UDP server. Der kan man se at den henter ut IP-adressen til PC'en som GUI'en kjører på og oppdaterer et felles felt. På den måten kan klientene koble seg til riktig IP-adresse igjen.

4.1.8 GUI

Figur under viser det endelige resultatet av hvordan vi har valgt å ha grensesnittet på vår GUI. Her har vi valgt å dele skjermen i to, der venstre delen er bildestrømmen fra bilen, og kontrollsystemet samlet på høgre side. Vi har valgt at styringen av bilen skal gjøres via tastaturet på PC, ikke via museklikk på GUI. Dette har vi gjort pga. muligheten for å aktivere flere kommandoer samtidig (om vi vil aktivere fwd og left på samme tid) og muligheten for raskere kontroll.



Figur 23 - Bilde av GUI

Inputs

Her vises verdiene tilknyttet distansesensoren vi har plassert på bilen, og X-/Y-koordinatene til objektet vi søker. Verdiene her blir kontinuerlig oppdatert så snart sensorene registrerer en endring i verdien. Om objektet er utenfor rekkevidde for kameraet, vil feltene bli oppdatert med «No object found».

Setup

Her har vi satt inn hovedkontrollen til systemet. Det er her vi

- Aktiverer/deaktiverer systemet
- Velger om det skal kjøres i Auto eller manuell modus
- Sensitivitetsnivået på systemet, som bestemmer hvor stor prosent andel av maksimal fart motorene skal kjøre på

Controls

Her har vi simulert 4 knapper som er tilknyttet kommandoene sendt fra tastaturet. Om vi trykker «*pil fremover*» på tastaturet, vil knappen «*Fwd*» bli aktivert. På denne måten kan vi følge med på GUI 'en hvilke kommandoer som blir sendt videre.

Figur 3.5.3 viser ett eksempel til hvordan vi har satt opp koden. Dette er koden som er tilknyttet kommandoen «Rev». Vi har også gjort tilsvarende med «keyReleased». Det vil si at så snart vi slepper knappen vil det bli sendt ny kommando om at knapp er deaktivert. På denne måten slepper vi å toggle knappene (trykke en gang for å aktivere, og en gang til for å deaktivere).

```
revButton.setText("Rev");
revButton.addKeyListener(new java.awt.event.KeyAdapter() {
    public void keyPressed(java.awt.event.KeyEvent evt) {
        revButtonKeyPressed(evt);
    }
    public void keyReleased(java.awt.event.KeyEvent evt) {
        revButtonKeyReleased(evt);
    }
});
```

```
private void revButtonKeyPressed(java.awt.event.KeyEvent evt) {
    this.revButton.setBackground(Color.green);
    if(!this.rev){
    this.controller.setRev(true);
    this.rev = true;
    }
}
```

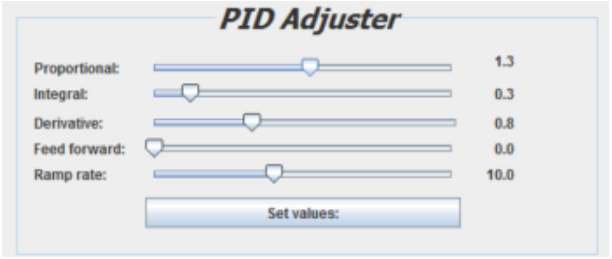
Figur 24 – KeyPressed og KeyReleased

Servo

Servo panelet inneholder indikasjon på om servo er aktivert eller ikke. Denne handlingen kan aktiveres ved å trykke på «A» eller «D» på tastaturet. I nåværende tilstand har bilen bare slå handlingen tilgjengelig, da dette var målet vårt for prosjektet. Tanken var å lage mulighet for å utvide bilen med en fange funksjon, slik at objektet kunne både skubbes videre og fanges.

PID justerer

Bilen bruker en PID regulator for å sikte seg inn på det ønskede objektet. Siden det kan være mye arbeid å justere en PID regulator har vi laget en skjult funksjon i applikasjonen vår som gir mulighet for å justere ulike PID parametere. Ved å trykke på «S» på tastaturet åpner det seg et lite panel, som vist i figuren under.



Figur 25 - Utsnitt av PID Adjuster

Dette panelet gir mulighet til å endre proporsjonal-, integral- og derivatleddet av regulatoren. En kan i tillegg endre «Feed forward» som er faktor brukt ved hastighetsregulering og «Ramp rate» som bestemmer hvor mye ut-verdien av regulatoren kan endre seg for hver gjennomkjøring. Etter en har justert verdiene trykker en på knappen, da blir verdiene sendt til og oppdatert i regulatoren.

4.1.9 Protokoller

Den endelige protokollen vi endte opp med fungerte med ønsket resultat. Vi satt opp to protokoller for sending. Der den ene tok for seg sending fra GUI til mikrokontroller, og den andre for sending av data fra Mikrokontroller til GUI. De blir begge beskrevet under:

Overføring fra GUI til mikrokontroller

Denne protokollen tar for seg hvilke funksjoner på mikrokontrolleren de forskjellige verdiene sendt fra GUI skal kobles mot. Ut ifra funksjonene vi ønsker utført i mikrokontrolleren, og kommandoene vi sender fra GUI, definerte vi en protokoll som skulle bygges på en fast størrelse av et byte Array. Størrelsen satt vi til 6 bytes, og tabellen under beskrives hvilke funksjoner som er tildelt de forskjellige Byte/bit i protokollen.

	Byte 0	**Byte 1**	**Byte 2**	**Byte 3**	**byte 4**	**byte 5**
bit 0	*stopp*					
bit 1	*fwd*			*servo*		
bit 2	*rev*			*auto/manuell kjøring*		
bit 3	*left*			*start program*		
bit 4	*right*					
bit 5						
bit 6						
bit 7						

Kommando:		*left motor speed*	*right motor speed*		*sensitivity*	*request feedback*

Figur 26 - Protokoll GUI til MikroKontroller

Som vist i tabell 3.2.1 har vi i det første byten tatt for oss selve styringen av bilen. Her har hver enkelt kommando fått tildelt ett spesifikt bit. Når vi for eksempel trykker på «fwd» i GUI, vil byte 0, bit 1 settes høy. (Kode for dette er vist i figur 3.2.1 som ett eksempel.)

Byte 1 og Byte 2 er tildelt hastigheten/verdiene til motorene på bilen. Hastigheten har en verdi fra 0 til 255, noe som da gjør at vi har tilegnet en hel Byte til hver av motorene. Det samme gjelder byte 4, som er tilknyttet «sensitivity».

```
public void goFwd() {
    dataFromGui[0] = this.setBit(dataFromGui[0], 1);
    this.fireStateChanged();
}

private byte setBit(byte b, int bit) {
    return b |= 1 << bit;
}
```

Figur 27 - Set bit

I byte 3 har vi lagt inn 3 funksjoner. Aktiver servo (bit 1), om det skal være manuell eller auto modus (boolsk verdi der true er auto og false er manual), og bit 3 for å aktivere selve programmet. Denne må være true for at bilen skal kunne utføre kommandoer.

Overføring fra mikrokontroller til GUI

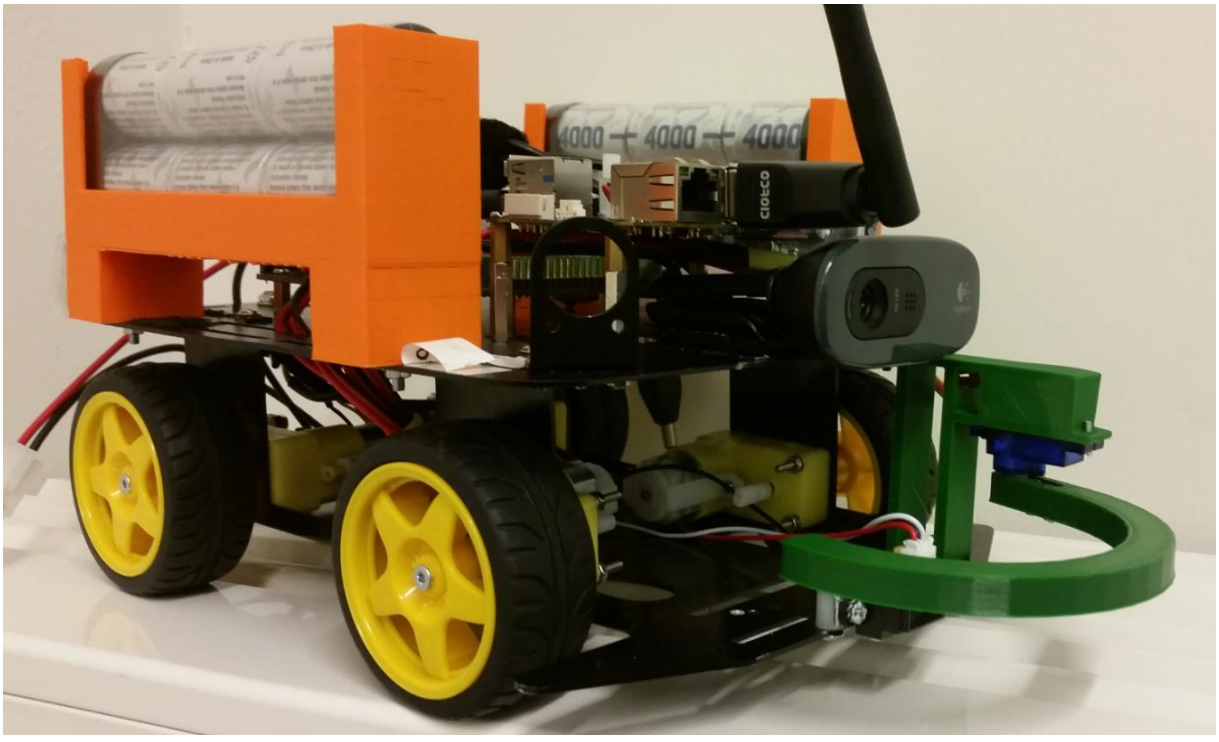
I motsetning til «GUI til Mikrokontroller» der vi hadde flere kommandoer tilknyttet ett spesifikt bit i en byte, har vi i denne protokollen bare satt opp tilknytninger på verdier mot en spesifikk byte. Dette er pga. vi fra mikrokontrolleren ikke sender kommandoer, bare verdier gitt fra sensorer. Det vil si at alle verdiene som blir sendt fra Mikrokontroller vil ha størrelse på over 1 bit. Tabell 3.2.2 viser hvilke verdier som er tilknyttet de spesifikke bytene.

Byte 0	**Byte 1**	**Byte 2**	**Byte 3**	**Byte 4**	**Byte 5**
pixy x value lowbyte	pixy X value highbyte	pixy y value lowbyte	pixy y value highbyte	distance sensor 4-30cm	request feedback

Figur 28 - Protokoll GUI til Mikrokontroller

4.2 Prototype

Figur 29 viser hvordan bilen endte opp tilslutt. Produktet er satt sammen av en to etasjers metallramme. I nedre del av rammen er det montert 4 DC motorer med hver sitt hjul, 2 på hver langside. I fronten av nedre del av rammen er det montert en distanse sensor (4-30 cm rekkevidde) og en robotarm som skal brukes til å slå eller fange en ball.



Figur 29 - Bilen

Robotarmen drives av en servo, og både robotarm og servo-brakett er tegnet og printet ut av gruppen.

I øvre del av rammen er det montert en egenutviklet batteriholder på hver langside. Mellom batteriholderne er det montert et «Arduino Romeo» kort og en «Odroid XU4». I fremre del av øvre etasje er det montert et webkamera, dette ser rett frem i kjøreretningen til bilen. Kameraet er koblet til en USB port i Odroiden, det er også en trådløs nettverksmottaker.

4.2.1 Bilen

Som nevnt ovenfor er selve bilen satt sammen av en øvre og en nedre metallramme med to DC-motorer på hver side. Fordelen med måten bilen er satt sammen på er alle mulighetene den gir for påmontering av ekstra utstyr, og dette er også grunnen til at vi valgte å bruke denne.

4.2.2 Kontrollsystem

Kontrollsystemet vårt kjører på en Odroid XU4. Som nevnt i punkt 3.1.1 er en Odroid en kraftig databehandlingsenhet som er liten av størrelse, noe som gjorde den ideell til vårt formål. I tillegg til dette så har den mulighet til å bruke Netbeans, noe som gjorde at vi enkelt kunne bruke den innebygde Git-utvidelsen i Netbeans til å enkelt overføre kode mellom PC og Odroid.

I og med Odroid XU4 har veldig begrenset med muligheter for I/O valgte vi å bruke Arduino Romeo til styring av I/O. Denne er koblet til Odroiden via USB.

4.2.3 Mikrokontrollere og I/O

Som nevnt over har vi valgt å bruke en Arduino Romeo til styring av I/O. Her har vi lagt inn en enkel kode for kontrollering av alle DC-motorene, servo-armen og distansesensoren. Dette blir igjen styrt av Odroiden ved å sette de forskjellige bytes 'ene ut fra den forhåndsdefinerte protokollen vår.

4.2.4 Batteri

Arduino-kortet blir supplert direkte fra batteriene, mens Odroiden blir supplert fra batteriene gjennom en DC-omformer. Grunnen til at vi måtte ha to relativt store batterier på bilen er fordi DC-omformeren som forsyner Odroiden måtte ha en spenning på minimum 8V, i tillegg til at Odroiden trekker en god del strøm slik at vi måtte ha en del kapasitet. Derfor har vi seriekoblet to batterier på 7,2V slik at vi får en spenning på til sammen 14,4V. Hvert av disse batteriene består av 6 battericeller på 3/4000mAh hver.

4.3 Resultat fra testing

Prosjektet ble testet mye underveis for å få fungerende og gode trådløsninger. Komplette bilen og dens funksjoner ble teste mer mot slutten av prosjektet når alt kom på plass.

4.3.1 Trådløsning

Remote Controller

I utgangspunktet brukte vi en klasse med fellesresurser og synkroniserte metoder i denne applikasjonen. Underveis i utviklingen ble det klart at en slik løsning ikke var nødvendig her.

Vi endte isteden opp med en god løsning der de to klassene for å motta data og video kjører i tråder, da disse lytter etter innkommende data. I tillegg blir det brukt en «Timer» i GUI' en som har en «TimerTask» på kontrolleren. Denne er brukt slik at GUI' en kan bestemme oppdateringshastigheten på input verdiene fra Odroid.

Når en utvikler et grafisk brukergrensesnitt er samtidighet svært viktig, slik at visualiseringen aldri fryser og stopper opp. Programmet skal alltid respondere mot brukeren sine operasjoner. Swing har gode løsninger for dette ved bruk av 3 typer tråder:

- Initialiseringstråd
- Hendelsestråder: Tråder som håndterer hendelser, som lytter til tastatur og overvåker musepeker aktiviteter.
- Bakgrunns tråder som tar seg av tidskrevende bakgrunns oppgaver.

For å kunne lage en visualisering som gir god responstid for kontrollering er spesielt hendelsestråder brukt. Det er da brukt hendelseslytter (eng: ActionListener) for å overvåke museoperasjoner og tastaturtrykk. Dette fungerte utmerket.

Odroid

Tidlig i prosjektet prøvde vi en trådløsning på Odroid applikasjonen der vi brukte synkroniserte metoder på fellesresursene, der trådene gikk i en vente tilstand etter å ha utført ønsket metode. Dette ble ikke en god løsning da det ble vanskelig å kontrollere at alle trådene fikk tilgang til fellesresursene. Det endte med andre ord med at noen tråder gikk i sult, et kjent begrep beskrevet i kapittel 2.1.2.

Utover i semesteret fikk vi bedre oversikt og kunnskaper om samtidighetsløsninger, og oppdaget at semaphore var en god løsning for vår applikasjon. Vi brukte dette og møtte et problem ved at en tråd anskaffet semaphoren, utførte den aktuelle koden, slapp den,

for så å anskaffe semaphoren igjen med en gang. Dette resulterte i samme problem som i forrige avsnitt, nemlig sult.

Ved oppretting av et semaphore objekt kan en velge en parameter som heter rettferdighet (eng: fairness) til sant eller ikke. Ved å sette denne til sant løste problemet seg og alle trådene slapp til. Denne løsningen gjorde dermed at trådkommunikasjonen fungerte på en svært god måte. Applikasjonen oppfyller kravene om å beskytte seg mot farene i kapitel 2.1.2: sult, inferens og vranglås.

Remote Controller og Odroid

Trådløsningene i begge applikasjonen ble testet mot hverandre underveis i prosjektet. De siste løsningene beskrevet over, fungerte godt.

En kan ta styre bilen fra tastaturet, visualiseringen responderer uten å fryse seg, og verdiene blir sendt over UDP til Odroid applikasjonen. Disse blir mottatt, semaphoren blir anskaffet og verdiene skrevet inn i fellesresursen. Kontrolleren henter ut dataene, og motorhastighet blir bestemt og satt i fellesresursen, som deretter blir sendt videre over seriellkommunikasjon til mikrokontrolleren.

På samme måte fungerer også den autonome metoden godt. Da går objekt gjenkjenningen i en tråd som beregner vinkelavvikene, og deretter anskaffer semaphoren og setter dataen i fellesresursen. Da henter kontrolleren disse vinklene, og bruker PID regulatoren som kjører i en egen tråd til å sette motorhastighetene. semaphoren blir anskaffet og verdiene blir sendt til fellesresursen. Verdiene blir deretter videresendt over seriellkommunikasjon til mikrokontrolleren.

4.3.2 Helhetlig

I forrige kapittel kunne en se at resultatet av trådløsningen fungerte svært godt. Prosjektet i seg selv er basert på mer enn bare trådløsninger.

Manuell modus:

Bilen styres fra piltastene på tastaturet og hastigheten reguleres ved å velge sensitivitet mellom 0 og 100. Responsen på kommandoer fra piltastene, til bilen responderer på kommandoen er relativt rask, og det gjør at kontrollerbarheten av bilen ved direkte visuell kontakt fungerer utmerket. Styres bilen ved indirekte visuell tilbakemelding, gjennom kamera, oppstår det en liten forsinkelse fra kommando blir gitt, til bilen responderer og tilbakemelding blir gitt gjennom kamera tilbake til GUI. Denne forsinkelsen gjør at kontrollerbarheten blir mere utfordrende og man kan ikke gi kommandoer til bilen i lik frekvens som ved direkte visuell tilbakemelding.

Autonom modus:

Som forklart i punkt 4.1.3 og 4.1.4 så styres bilen gjennom bildebehandling og objekt gjenkjenning, det vil si at bilens oppførsel styres her direkte fra hva kamera ser. Objekt gjenkjenningen er avhengig av at en bare ser et objekt innen de bestemte parameterne. Det var en utfordring og finne et godt objekt å følge som ikke ble påvirket av lysforholdene. Vi brukte hovedsakelig en grønn drikkeflaske under testing av prosjektet.

Posisjonen, relativt til kamerarammen fra objektet som er gjenkjent, sendes via datahandler til PID reguleringen som bestemmer retningen bilen skal gå i. Hvis objektet forsvinner ut av kameraet sitt synsfelt vil bilen gå inn i en søkemodus i den horisontale retningen som objektet var sist sett. Etter vi hadde testet en del med innjusteringer av PID parametere så fulgte bilen objektet ganske stabilt. Hvis objektet ble flyttet til sides for bilen så endret den retning slik at objektet igjen var rett foran bilen. Når objektet kom nært nok så ble robot armen utløst slik at ballen ble skyvet vekk.

Oppdateringshastighet på PID regulator ble først satt til en frekvens på 10 Hz. Under testing så kom det frem at dette ikke var raskt nok, bilen ble ustabil i høyere hastigheter på grunn av regulator ikke kunne respondere raskt nok. Frekvensen ble så endret til det dobbelte, 20 Hz og en ny runde med parameter justeringer ble gjort. Etter innjusteringer så responderte bilen mye bedre og virket mye mer stabil i større hastigheter.

5 DRØFTING

Etter endt prosjektfase, sitter gruppen igjen med flere tanker. Vi mener selv at produktet tilfredsstiller alle kravene/målene vi satt i starten av oppgaven. For å svare på resultatmålene, kan vi videre dele opp i flere underkategorier.

5.1 Resultat fra testing

Prosjektet har gått som forventet og vi har hatt en del utfordringer, noe vi har lært mye av. Til tross for at vi er fornøyd med resultatet er det deler som kunne vært bedre.

5.1.1 Bilens oppførsel

Manuell modus:

I manuell modus fungerte bilen forholdsvis bra med visuell kontakt med bilen, som beskrevet i kapittel 4.3.2. Utfordringen var som nevnt når en kjørte bilen kun ved bruk av video, da blir det en forsinkelse stor nok til at kontrollen blir unaturlig. Vi prøvde å gjøre så lite med bildet som ble sendt over UDP, slik det skulle gå så fort som mulig. Det

ble likevel stor nok forsinkelse til at det ble utfordrende å styre bilen. Dette kunne vært forbedret ved å få til raskere videooverføring.

Autonom modus:

Den autonome modusen fungerer bra, vi klarer å følge et objekt i lav hastighet og slå til objektet når bilen er nær nok. En av de store utfordringene i den autonome biten av prosjektet er å klare å detektere objektet på en god måte, samtidig som en ikke oppdager andre objekt i tilsvarende farge. Her kunne nok objekt gjenkjenningen vært forbedret ved å bruke andre teknikker, eventuelt hatt bedre og jevnere lysforhold.

I vårt prosjekt har vi ikke tatt hensyn til hindringer, dette gjør at om en ikke passer på bilen så kjører den inn i stoler, vegger eller andre hindringer. Vi valgte å ikke ta hensyn til dette da, dette ville blitt for mye til å fullføre innenfor prosjektfristen. Her er forbedringsmuligheter.

Likevel er vi fornøyde med resultatet av den autonome funksjonen siden det viktigste i dette prosjektet er sanntids datateknikker.

5.1.2 Programvareløsning

Programvareløsningen vår fungerer på en god måte. Vi klarer å kommunisere mellom tråder, beskytte fellesresurser og unngå vranglås, sult og inferens. Vi har også prøvd å utvikle programvaren ved å ha fokus på gode design prinsipper. Disse punktene har vi løst på gode måter som resulterte i applikasjoner som fungerer.

Det finnes alltid forbedringspotensial når en utvikler programvareløsninger.

5.2 Erfaringer fra prosjektet

Prosjektet i sanntids datateknikk har vært interessant og svært lærerikt. Vi har tilegnet oss kunnskaper innen sanntids datateknikk og fått testet dette i praksis. I tillegg har vi lært om objekt gjenkjenning, programvareutvikling, Linux, kommunikasjon, prosjektplanlegging og koordinering.

5.2.1 Bruk av tråder

Før prosjektet hadde vi lite kunnskap om tråder og sanntids datateknikk. Underveis i prosjektet og undervisningen har vi lært samtidighets- og sanntidsteknikker. Utfordringene har gitt mulighet for å utforske og teste ulike metoder for å løse oppgaven på en god måte.

Vi ser tydelig at bruk av trådløsninger er svært viktig i mange applikasjoner, og det er god erfaring å ha med seg videre.

5.2.2 Arbeidsfordeling/gruppearbeid

Gruppen består av 4 medlemmer, noe som var et bra antall med tanke på oppgaven vi valgte. Arbeidsfordelingen var lett da prosjektet var stort og inneholdt mange ulike deler. I utgangspunktet jobbet vi slik at en begynte på PC applikasjonen eller kontroll applikasjonen, 2 jobbet med Odroid applikasjonen mens en jobbet med arduino-koden. Utover i prosjektet ble arbeidsoppgavene fordelt alt ettersom behovet.

Kommunikasjonen mellom gruppemedlemmene har gått svært fint. Vi har stort sett jobbet i samme rom, noe som har forenklet kommunikasjonsprosessen. For å ha versjonskontroll og mulighet til å jobbe samtidig på ulike deler av prosjektet har vi brukt «GitHub».

5.2.3 Prosessene bak Software-utvikling

Utvikling av programvare er en utfordrende prosess der mye uventet kan skje. Det er da viktig å ha «backup» av applikasjonene og mulighet for å jobbe med ulike deler av programvaren på samme tid. Vi har brukt «GitHub», som beskrevet i kapittel 3.2.2. Dette har gitt trygghet og gjort det enklere å utvikle programvaren sammen.

Før prosjektet hadde vi lite erfaring med versjonskontroll, noe som ga noen utfordringer i starten. Etter hvert ble vi veldig glad i dette, og setter pris på erfaringen vi har fått av bruken med versjonskontroll.

5.3 Mulige forbedringer

En utfordring vi hadde underveis var at under utvikling av software og testing så gikk batteriene relativt raskt tom for strøm. Vi vekslet ofte mellom å bruke batteriforsyning eller strøm fra nettet ved en 5V strøm adapter og dette resulterte i at vi måtte starte Odroid på nytt mange ganger. Vi kunne hatt en mye mere sømløs testing hvis vi hadde hatt en velger mellom batteri og strømadapter som forsynte Odroiden med strøm kontinuerlig, selv om en av strømkildene falt bort. En mulig løsning på dette er å benytte dioder koblet slik at den høyeste spenningsforsyningen til Odroid blir alltid valgt ut i fra batteri eller adapter.

En annen utfordring vi hadde var Romeo kortet, vi benyttet USB porten til Romeo kortet for kommunikasjon. Dette er en USB port av typen micro-USB og det viste seg at denne var meget ømfintlig for mekaniske påkjenninger. Det resulterte i at porten løsnet tilslutt og vi kunne ikke lengre kommunisere med kortet. Et nytt kort ble bestilt, men det hadde

relativt lang leveringstid. Et alternativ til å bruke dette kortet var å benytte en vanlig Arduino UNO med et motorshield montert på toppen av UNO'en. UNO'en har en mye mer robust tilkoblings-port en Romeo kortet og har dermed ikke så stor sjans for at den blir defekt som følge av mekaniske påkjenninger fra kabeltilkobling.

Objekt gjenkjenningen var også en utfordring som kunne vært forbedret om en hadde bedre tid til å finne ut av ulike metoder. Vi kunne trolig brukt mer avanserte metoder som ville gitt bedre gjenkjenning, dette kunne derimot gått utover kjøretiden av objekt gjenkjenningen. En forbedring kunne vært og montert en lyskilde på bilen slik lysforholdene ble jevnere.

I autonom modus så har bilen lett for å kjøre inn i andre objekter som nevnt i kapittel 5.1.1. Dette kunne vært forbedret ved å montere flere avstandssensorer, slik en kunne overvåket forholdene der en kjører.

5.4 Prosjektmål

Med tilbakeblikk på målet vi satt internt i gruppen, mener vi absolutt vi har klart oss bra med dette målet. Vi har som gruppe jobbet strukturert, og kommunisert oss imellom på en veldig bra måte.

Prosjektplanen har blitt fulgt opp, med noen unntak med tanke på en periode hvor vi hadde defekt Romeo kort, og ikke fikk testet produktet.

5.5 Prosessmål

Da vi satt oss dette målet var vi ukjente med trådbasert programmering og alt hva dette innebærer. Vi kan absolutt si at dette prosjektet har gitt oss mye ny læring innenfor emnet, og gitt oss kunnskap som vi har mulighet til å ta med oss videre.

5.6 Produktmål

Det endelige produktet utfører alle oppgavene vi og studieveileder har satt som krav. Grunnet en periode på ett par uker med defekt Romeo-kort hemmet dette kanskje oss i å få lagt til noen ekstra funksjoner som vi hadde i baktanken at hadde vært kjekt å få med.

6 KONKLUSJON

Alt i alt er vi veldig fornøyd med utførelsen i dette prosjektet. Med bruk av trådbasert programmering løste vi godt utfordringen med samtidighet og prosess forstyrrelser. Bildebehandlingen sender også verdier så snart den detekterer objektet, og sender verdier videre til regulator som kontrollerer retningen til roboten. Dette utgjør da også at vi har oppfylt kravet til at det skal være ett sanntids system.

7 REFERANSER

Concurrent and Real Time programming in Java. (u.d.). I *Concurrent and Real Time programming in Java.*

DropBox. (2015). Hentet fra DropBox: https://www.dropbox.com/about

Eric Freeman, E. R. (2004). *Head First Design Patterns.* O'Reilly.

GitHub. (u.d.). Hentet fra GitHub: https://github.com/

GitHub. (2016). *GitHub*. Hentet fra tekdemo/MiniPID-Java: https://github.com/tekdemo/MiniPID-Java

Jameco. (2016). *How do servo motors work.*

Javatec. (u.d.). Hentet fra Javatec: http://javatec-machinery.com/

Kean, J. (u.d.). *RXTX.*

Laudon, L. &. (u.d.). *Information system development*. Hentet fra Prototyping process: http://www4.comp.polyu.edu.hk/~csajaykr/ISD.pdf

Lindsey, T. L. (u.d.). *JavaTech.*

Michael Kolling, D. J. (2012). Objects First with Java. I *A Practical Introduction Using BlueJ .* BlueJ.

Netbeans. (u.d.). Hentet fra Netbeans: https://netbeans.org

notes, L. (u.d.). Porst¨. s. Lecture notes.

notes, L. (u.d.). Ports. s. Serial+Communication+and+Networking.pdf.

opencv.org. (2016, 10 4). Hentet fra http://opencv.org

Sheadel, D. (u.d.). *Github*. Hentet fra https://github.com/tekdemo/MiniPID-Java

Silberschatz. (u.d.). *Operating System Concepts .*

Wiki, O. (2016, 10 5). *Odroid Wiki*. Hentet fra http://odroid.com/dokuwiki/doku.php

Wikipedia. (2016, 04 29). *PID-regulator*. Hentet fra PID-regulator: https://no.wikipedia.org/wiki/PID-regulator

Wikipedia. (2016, August 27). *Wikipedia*. Hentet fra Data buffer: https://en.wikipedia.org/wiki/Data_buffer

Wikipedia. (u.d.). *Protokoll*. Hentet fra wikipedia: https://no.wikipedia.org/wiki/Protokoll_(datamaskiner)

Wikipedia. (u.d.). *Swing(java)*. Hentet fra Swing: https://en.wikipedia.org/wiki/Swing_(Java)

wikipedia. (u.d.). *wikipedia*. Hentet fra UDP: https://no.wikipedia.org/wiki/UDP

8 FIGURER OG TABELLER

9 VEDLEGG

Kildekoden til begge applikasjonene er lastet opp på innleveringsmappe for kildekode i fronter:

A) RemoteController – Java kildekode for PC applikasjon

B) UDPServer – Java kildekode for Odroid applikasjon