

Navigering i LEGO vej elementer

Rasmus Winther Lauritsen - 20021859

Christian Jonigkeit - 20021232

Mehdi Abyar - 20023623

Rasmus Nygaard Andersen - 20020543

1 Introduktion

Efter det snak vi havde med Ole besluttede vi os i gruppen at lave det projekt der hedder "Navigation through LEGO road elements". Vores projekt er inspireret af [?]. I den artikel har robotten fire hovedopgaver, først at navigere rundt i byen uden at stå i noget, andet at detektere landmarks, tredje skal den konstruere og vedligeholde et kort over byen, og til sidst skal den kunne planlægge en korteste vej mellem to valgte punkter. Vores robot navigerer rundt i byen og detektere vejens forløb. Når bilen er færdig med at køre står stille, det vil sige, at den har detekteret alle byens feature, derefter er den i stand til at finde korteste vej mellem to punkter. Vi droppede detektering af hindringer på kørebanen, fordi der ikke er nogen hindringer på byens baner.

Vi blev delt op i to grupper, den ene gruppe tog sig af kort bygning og planlægning af den kortestevej, og den anden gruppe tog sig af kørsel af bilen. Vi gjorde en masse overvejelser omkring hvordan bilen skal holdes på banen, hvordan man holder styre på hvorlagt bilen er kørt på banen, hvornår er en vejstykket færdig, hvor bilen befinder sig i byen og til sidst men ikke mindst hvordan vi detekterer en kryds, T-kryds, højre/venstre swing.

Fig. 1: Første robot

2 Første design

2.1 Første design af robotten

I det første design af robotten havde vi nogle krav vil ville have opfyldt: robotten skulle køre meget præcis og tage højde for den resterende upræcise bevægelse ved at justere for den.

2.1.1 Sensorer

Omgivelserne skulle detekteres ved brug af to lyssensorer og en rotationssensor. Ideen var at lyssensoren skulle sidde uden for gadeelementerne for at detektere features, se figur ??, mens rotationssensoren skulle bruges til at beregne positionen indenfor et enkelt gadeelement.

2.1.2 Akuatorer

Robotten skulle fremdrives af to motorer, et per hjul. Da vi kun havde en rotationssensor til rådighed, samtidigt med at vi ville måle, hvor langt motorerne havde bevæget robotten blev nødt til at finde en løsning. Et differentiale, hvor de to motorer sættes på hver sin akse, tilsluttes rotationssensoren. Drejer enten den højre motor, den venstre motor eller begge sammen vil rotationssensoren ligeledes dreje, se figur ?? . Med præcision som et af de store krav har vi

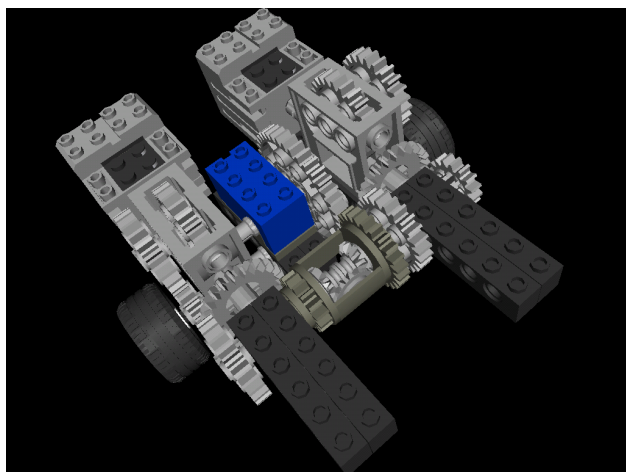


Fig. 2: Differentiale og rotationssensor

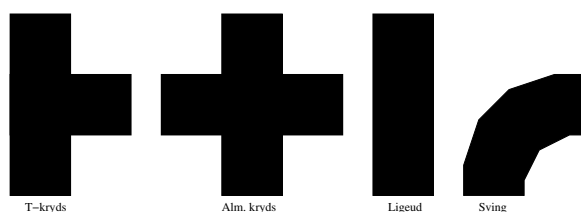


Fig. 3: Initielle gadeelementer

valgt at gearne hjulene meget ned i forhold til rotationssensoren, for at få en højre opløsning.

2.2 Første design af byen / gaderne

Vi ville prøve at bruge de Legos egne gadeelementer, men de viste sig at detektion af farverne på disse plader var alt for usikker, idet råværdierne for grøn og grå lå for tæt op ad hinanden. Vi valgte derfor at designe vores egne gadeelementer, sort på hvid baggrund, og bruge de originale lego plader som forbillede. De fire elementer er et sving, et T-kryds, et almindeligt kryds og et lige stykke vej, se figur ??.

2.3 Detektion af features

Med de fysiske modeller på plads var det nu muligt at diskutere, hvorledes de enkelte features skulle detekteres. De informationer vi havde til rådighed var dis-

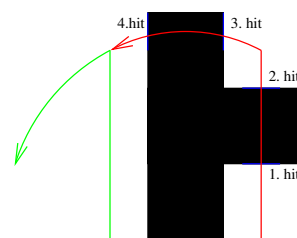


Fig. 4: Hit record eksempel

tance vha. rotationssensoren og information af over hvilken farve lyssensorer lå. Ideen var at bruge hit records for at detektere gadeelementtyper.

2.3.1 Hit records

Vores hit record er en datastruktur som forbinder distancer med farveskift på sensorer. Ud fra disse records vil det så være muligt at konkludere type og orientering af et gadeelement. For at anskueliggøre metoden vises et eksempel for et T-kryds.

Hit record for T-kryds I dette eksempel kører robotten ind på et T-kryds fra den ene hovedgren, sådan at bigrenen er på højre side, se figur ???. Robotten vil nu køre lige ud. Når den højre sensor (rød) bevæges fra hvidt til sort, et farveskift, registreres et hit på højre side med den aktuelle distance. Det andet hit bliver registret på samme måde. Har robotten kørt en tilpas distance, udføres et venstre sving, i hvilket hit 3 og 4 optages. Ud fra de givne hit records kan nu typen og orientering bestemmes entydigt.

2.3.2 Invariant og justering

Vores detektionsmetode er meget afhængig af den præcise position af robotten samt den orientering når et hit record set skal optages. For at det lykkedes antages følgende invariant. Hver gang robot skal til at udføre detektionsrutinen befinder sig robotten præcis i midten af et gadeelementes grene og orienteret parallelt med denne gren. Dette kræver at robotten skal rejstures efter at den har kørt i et stykke tid. En sådan justering vil kunne blive foretaget på et T-kryds- og almindeligt krydsgadeelement. Ideen er

Algorithm 1 Followline programstump

```

38     while(Running)
39     {
40         // Reading sensors
41         right = LightSenseRight();
42         left = LightSenseLeft();
43
44         // Too much right?
45         if(right == WhiteColor) PortA(
46             Float);
47         else PortA(OnPos);
48
49         // Too much left?
50         if(left == WhiteColor) PortC(
51             Float);
52         else PortC(OnPos);
53     }

```

at justere robotten indtil begge sensor ligger på en sorthvid kant.

3 Followline test

Den første test, vi lavede for at teste robotens design, var en followline test. Princippet i followline er meget simpelt: robotten skal følge en streg ved brug af lyssensorer. Koden er ligeså simpelt som selve followlineopgaven, når robotten kommer får langt til venstre bremses højre motor og vice versa. Algoritme ?? viser hovedloopet af followlineprogrammet. I linje 41 og 42 opdateres værdierne for right og left, som er af typen Color. Opdatering sker via kald til Color modulet, som aflæser råværdier af sensoren og konverterer til en farve type på grundlag af et threshold, se algoritme ?? linje 102 - 108. I linjerne 44 - 50 tændes eller slukkes output for port A og B, dvs. motorerne drejer eller hviler, at efter de opdaterede værdier for right og left.

Koden blev uploadet til RCX'en og robotten blev testkørt på en række af ligeud og sving gadeelementer. Det viste sig at followline koden som sådan virkede fint nok, men at robotten ikke kørte 100% ligeud, selvom den befandt sig indenfor vejen.

4 Nyt design

Problemet med at køre lige ud, som followlinetesten viste, som robotten havde krævede et nyt design. Hovedændringerne bestod i at vi ville tilføje en mekanisme som kunne lås og åbne differentialet for de

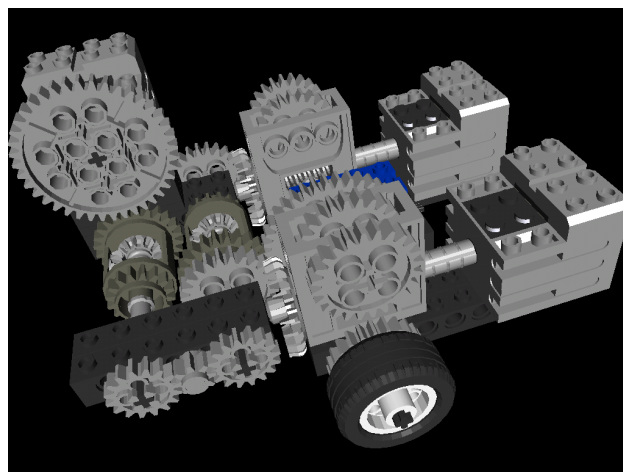


Fig. 5: Mekanisk Differentialelås

to hjul og at vi ville lave lidt om på gadeelementerne.

4.1 Differentiale lås

Robotten kunne ikke køre 100% ligeud, hvilket skyldes at motorerne havde forskellig drejmoment, samt at friktionen i tandhjulene ikke var den samme for højre som for venstre hjul. Løsningen krævede en avanceret mekanisk differentialelås.

På figur ?? ses et 3d model af differentialelåsen. De to motorhovedakser forbindes med modsat drejningsretning i differentialet, hvilket har den effekt at selve differentialet nu kun vil dreje, når hovedakserne ikke kører med samme hastighed, eller sagt mere præcist: differentialets rotation giver udtryk for forskellen af hastigheden af det højre og venstre hjul. Sættes nu motoren, som er direkte forbundet til differentialet, til at blokere, tvinges det højre og venstre hjul til at dreje med eksakt samme hastighed. Derved kan robotten nu gå 100% lige, mens mulighed for at dreje stadig opretholdes. I starten var vi lidt bekymret om robotten havde kraft nok til at drive alle de tandhjul som vi satte på, men da vi havde gearet hjulene meget ned viste det sig ikke at være et problem. Faktisk var vores robot så kraftigt at den uden videre kunne skubbe en kontorstol!

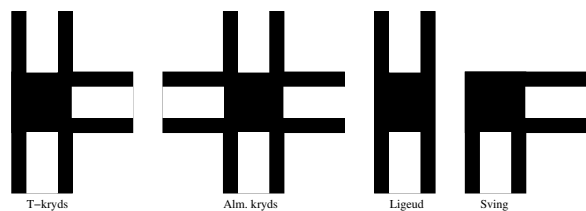


Fig. 6: Nye gadeelementer med followline striber

4.2 Nye gader

Efter nærmere omtanke blev vi enige om, at det ville være bedre at have en form for skinner, som robotten kunne køre på i hver enkelt gadeelement. Skinner blev så realiseret ved at tilføje en hvid stribe på hvert enkelt gadeelement, som i midten blev brudt. Det ville nemlig skabe mulighed for at robotten kunne holde sig stabilt, dvs parallelt med gadeforløbet ved brug af followline programmet, mens brudet af striben kunne bruges i den nye detektionsmetode se sektion ???. Så denne metode var mere inspireret af princippet “sensing rather than planning”. De nye gadeelementer er afbildet i figur ??.

4.2.1 Ny detektionsmetode

Som følge af de nye gadeelementer og indførelsen af et followline program blev vi nødt til at genoverveje, hvordan robotten skulle detektere de enkelte gadeelementer. Vi nåede frem til at tre dedikerede lysensensorer, som kun tog sig af denne opgave, ville ikke blot være en mulig løsning, men også den som ville kræve mindst ændring af den eksisterende kode og samtidigt sørge for at vi fik en logisk adskildelse af de to opgaver / opførelser robotten havde, nemlig følg en streg og detekter et feature.

De tre sensorer skulle anbringes således, at de ville befinde sig over de mulige followlinestriber, når robotten befandt sig i midten af et gadeelement, dvs når de to sensorer som blev brugt af followlineopførelsen lige netop havde bevæget sig væk fra followlinestriben. Figur ?? viser en sådan situation for et T-kryds. Alt som så behøves for at afgøre gadeelementtypen, er at måle hvilke farver, sort eller hvid, de enkelte detektionslysensensorer befinder sig over.

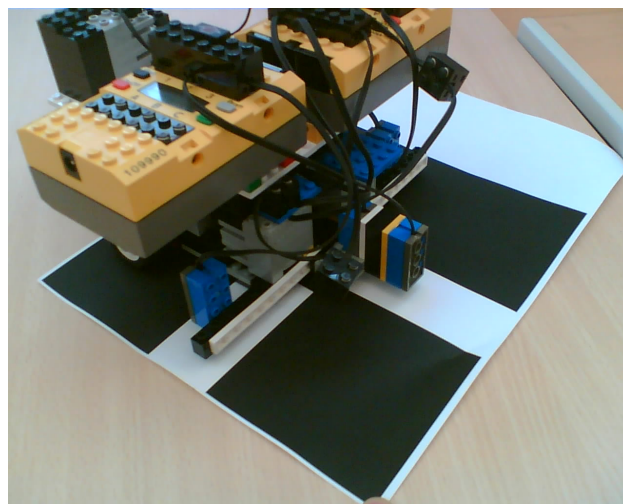


Fig. 7: Eksempel på forbedret detektionsmetode - T-kryds

4.2.2 Ekstra RCX

Beslutningen om at bruge tre dedikerede detektionslysensensorer betød at vi skulle bruge tre ekstra inputporte. Eftersom det ikke var os muligt at lave en multiplexor, sådan at man kan tilslutte flere sensorer til en og samme port, for så at vælge en sensor ud vha. nogle bestemte strømsignaler og vi ligesom havde opfattelsen at kurset i større dimension drejede sig om at forstå hvordan man bør programmere og styre digitale enheder frem for at udvikle små elektroniske dimser, hvilket har en mere ingeniørmæssig udartning, endes vi om at bruge en ekstra RCX-enhed, som udelukkende skulle stå for detektion af gadeelementerne og opbygning af den digitale repræsentation af kortet. Denne fremgangsmåde havde en yderligere fordel, da responsibilities ikke blot blev delt op i separate komponenter med også fysik kom til at ligge adskilt. Endvidere skulle der laves kommunikation mellem de to enheder, hvilket omtales i sektion ??.

5 Navigationstest

Med den nye detektionsmetode lagt på plads, fortsatte vi med vores næste testområde: navigation på

et gadeelement. Den ene RCX, i fremtiden refereret til som køreenheden, havde til opgave at holde bilen indenfor vejstriberne og udføre diverse køremønstre i området mellem vejstriberne, for at navigere rundt i gaderne. Mønstrene er at køre ligeud, dreje 90 grader til venstre eller højre samt at lave en 180 graders drejning (U-turn). At køre ligeud viste sig ikke at være noget problem, da vi havde en rotationssensor som kunne bruges til at måle distance. Derimod lagde selve konstruktionen af det første differentiale og den derpå anbragte rotationssensor grund til, at det ikke var muligt at bruge rotationssensoren til at bestemme udsvinget af en drejning, hvor begge motorer drejer modsat med samme fart, da det får differentialet til at stå stille.

5.1 Ekstra rotations sensor

Af mangel på information omkring udsving af drejninger tilføjede vi endnu en rotationssensor. Den blev anbragt imellem differentialet for differentialelåsen og motoren som låser og åbner for differentialet, se figur ???. Da differentialets rotation giver udtryk for forskellen af hastigheden af det højre og venstre hjul, nævnt i sektion ??, vil det netop kunne bruges til at måle udsving. Rotationssensoren blev tilsluttet parallelt med den anden rotationssensor, hvilket ikke har nogen forstyrrende effekt i og med vi kun måler på rotationen når enten den ene eller den anden står stille, e.g. robotten kører ligeud eller drejer.

5.2 Ekstra lyssensor og tidsdrevne rutiner

I en de næste test fandt vi ud af at vores ide at udelukkende bruge rotationssensor til at navigere rundt i gaderne gjorde, at en upræcis drejning ville i værste fald blive mere upræcis som robotten kørte rundt i gadenettet. Vores oprindelige håb, om at followline programmet ville udligne de upræcise drej, var blevet knust. Igen skulle vi have brugt "sensing rather than planning" paradigmet. Valget faldt så på at droppe rotationssensorene til fordel for en ekstra lyssensor, som vi anbragte lige i midt foran followline sensorerne. Ideen var nu den, at vi i et sving

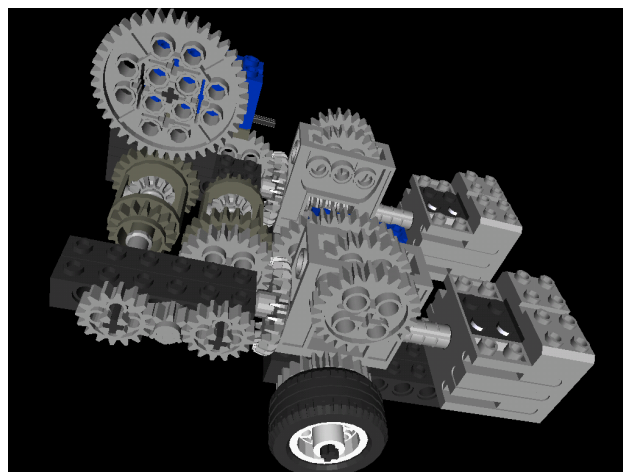


Fig. 8: En ekstra rotationssensor integreres i differentialelåsen

ville dreje indtil denne sensor havde målt råværdier for farverne hvid sort hvid og så et lille stykke længere. Sagt med andre ord ønskede vi robotten at dreje væk fra den aktuelle followlinestribe indtil den var drejet ind på en ny followlinestribe og så lige et stykke længere så den atter befandt sig midt over striben. Da denne process vil være uafhængigt af foregående drejninger vil fænomenet at robotens orientering bliver værre med tiden forsvinde, hvilket også viste sig i efterfølgende tests. Da begge rotationssensorene blev fjernet benyttede vi i stedet for tid som mål for hvor langt robotten skulle dreje, køre ligeud etc. Det havde så den ulempe at roboten blev afhængig af batteriniveauet, men en tilpasning af rotationstider tog typisk ikke længere en 1 -2 minutter, og holdt så omkring 3 - 4 timer.

Den endelige robot kan ses på figur ??.

6 Programmelt

I denne sektion beskrives det program som udføres af køreenheden.

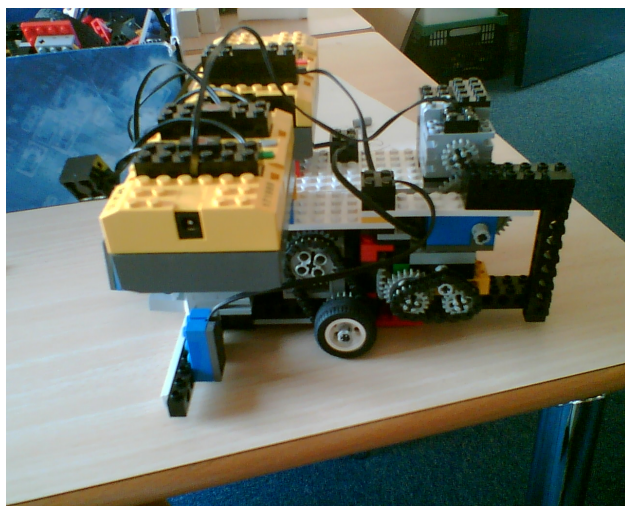


Fig. 9: Endelig robot

6.1 Programstruktur

Programmet består af en række moduler, som hver har deres ansvarsområder, og der eksekveres flere processer på RCXenheden for at opnå en bedre struktur og fleksibel kode. De moduler som ligger i køreenheden er

RobotRTS i en modificeret udgave, som har til opgave at switche mellem processer samt at kalde opdatering i sensor modulet, SensorControl.

SensorControl er ansvarlig for at opdatere interne variabler som distance og farverne målt af højre, venstre og den centrale lyssensor.

Color modulet tager sig af kalibrering af lyssensorerne, samt at fortolke råværdi på lyssensorer som farver.

Navigation er hovedkomponenten af robotten. Det er her information fra de enkelte moduler samles og bruges til at udføre followline såvel som at navigere på features.

På figur ?? ses en skematisk illustration af hvorledes moduler er sat sammen.

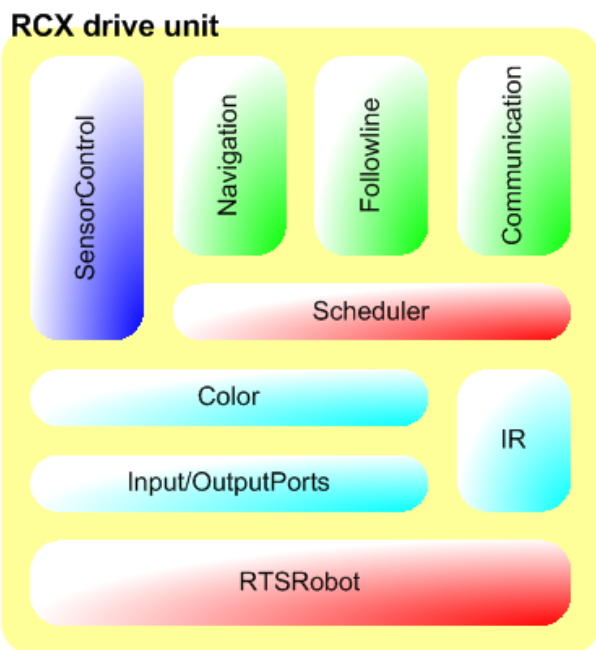


Fig. 10: Programstruktur for køreenheden

6.1.1 SensorControl modul

Som nævnt forinden er det i SensorControl modulet variablerne fra sensing af omgivelserne bliver opdateret. I algoritme ?? ses opdateringsmetoden af SensorControl-modulet. I linjerne 27 - 30 aflæses de fire sensor ved kald til Color modulet, men de ny værdier bliver ikke sat med det samme, som de iagttages i linjerne 34 - 62. Derimod bliver en præcisionstæller talt ned med en hver kan en af aktuelle værdierne er forskellig fra de globale værdier. Først når tælleren har en værdi på 0 assignes den aktuelle værdi. Hermed opnår vi en mere stabil aflæsning i områder hvor værdierne fluktuerer, dvs. i kantområder. Desuden holdes to andre variabler opdateret, EndOffRoad og OnRoad, som hhv. angiver om robotten befinder sig med begge sensorer på followlinestriben eller modsat, se linje 64 og 65.

Algorithm 2 SensorControl modul kode

```
25 void SensorControlUpdate( void )
26 {
27
28     currentLeft = LightSenseColor(1);
29     currentMid = LightSenseColor(2);
30     currentRight = LightSenseColor(3);
31     // not using rotationsensors any more
32     //currentRotation = Port2Raw();
33
34     if(currentLeft != left)
35     {
36         leftprecision --;
37         if (!leftprecision)
38         {
39             left = currentLeft;
40             leftprecision = precision;
41         }
42     }
43
44     if(currentMid != mid)
45     {
46         midprecision --;
47         if (!midprecision)
48         {
49             mid = currentMid;
50             midprecision = precision;
51         }
52     }
53
54     if(currentRight != right)
55     {
56         rightprecision --;
57         if (!rightprecision)
58         {
59             right = currentRight;
60             rightprecision = precision;
61         }
62     }
63
64     EndOffRoad = right == BlackColor && left == BlackColor;
65     OnRoad = right == WhiteColor && left == WhiteColor;
66
67     /* not using rotationsensors any more
68
69     if(currentRotation != rotation)
70     {
71         rotationprecision --;
72         if (!rotationprecision)
73         {
74             rotation = currentRotation;
75             distance++;
76             rotationprecision = precision;
77         }
78     }
79     */
80 }
```

6.1.2 Color modul

Color modulet er ansvarlig for mapping af råværdier til farveværdier, 0 for sort og 1 for hvid. I kalibrationsmetoden måles en serie af værdier per farve per sensor. Middelværdien af råværdierne for hvid og sort trækkes så fra hinanden deles med to og adderes med den mindste af de to middelværdier, se algoritme ?? linje 88 - 93. Denne ny værdi bliver så brugt som skillepunkt mellem sort og hvid, dvs. en råværdi som er større interpreteres som sort og eller som hvid, se linje 102 - 104.

6.1.3 Navigation modul

Navigationsmodulet er hovedmodulet, hvor al information samles og fortolkes. Det består af to processer, drive processen, som kører robotten, og en lcdprocess, der opdaterer lcddisplayet løbende med debug og anden nyttig information. Driveprocessen kan yderligere opdeles i tre, followline, kommunikation og selve navigationen mellem gade elementer.

Followlinekoden er stort set den samme, som i første test, men med nogle få ændringer, som skyldes at differentiallet blive lås når robotten befinder sig indenfor followlinestriben, se linje 93 - 99 i algoritme ???. Når robotten har forladt followlinestriben, hvilket af geometriske årsager kun kan ske parallelt, bremses begge motorer og kommunikationsdelen overtager, se linje 112 - 115.

I kommunikationsdelen initieres kommunikationen af kørerobotten ved irtransmittering af et request. Der forventes så at den næste besked som modtages er en ny navigationsdirektiv. Koden er meget simpel og kan ses i algoritme ??.

Navigationskode består af en større switch, hvor der switches på den modtagne navigationskommando, direction, se algoritme ???. I tilfældet af et ligeud kommando, tændes begge motorer og differentiallet låses, linje 130 - 133, for et tidsrum, som svarer til at robotten har nået centeret af followlinestriberne på et gadeelement, plus et tidsrum, der svarer til at køre ind på den foranliggende followlinestribe, linje 170.

Når direction er LEFT, dvs. robotten skal køre til venstre, tændes begge motoret og differentiallet låses

i ligeså lang tid som i en ligeud kommando, men når robotten har nået midten, dvs. det først tidsrum er ovre, skiftes strømmen på den venstre motor sammen med at differentiallet åbnes. I denne tilstand forbliver robotten indtil der er gået en fast tid minturntime og den centrale farveværdi er hvid, linje 146, derpå ventes der et ekstra tidsrum leftturntime, linje 147. Den første tidværdi bevirker at vi mindst dreje 60 grader, dvs. den centrale lyssensor med høj sandsynlighed befinder sig på sort baggrund, mens tidsrummet leftturntime søger for at rette robotten op, således at den er parallelt med den venstre followlinestribe. Tilsidst låses differentiallet og robotten kører ligesom under ligeud kommando ind på den venstre gade. Et højre sving er lige modsat og i et uturn drejes robotten mindst 120 grader før den retter ind.

Efter dreje kommandoen er udført overtager followlinedelen igen.

7 Kort-datastrukturen

Kortet af byen behøver en intern repræsentation i RCXen, hvori de detekterede veje gemmes. Datastrukturen skal muligvis gøre at bilen kan finde rundt i byen og kan finde ud af hvornår hele byen er kortlagt.

Byen er bygget op af vejelementer som har struktur der ligner almindelige Lego vejplader, dvs. alle vejelementer er lige store kvadrater og alle sving er 90 grader. Dette forsimples datastrukturen som vejen kan gemmes i, da man kan forestille sig at vejelementerne sidder i et grid hvor alle elementer har fire naboer (faktisk er det ikke kun datastrukturen der bliver enklere, men også detekteringen, navigeringen og kørslen). Pga. af forsimpningen af udformningen er det muligt at bruge et 2 dimensionelt array som base for datastrukturen, hver vejplade bliver gemt i en indgang i arrayet og dens nabo plader bliver gemt i de omkring liggende indgange, helt i henhold til den intuitive forståelse. Der er 4 retninger defineret: nord, syd, øst og vest, hvor nord og syd henholdsvis er faldende og stigende første koordinat i arrayet; øst og vest er ligeledes defineret som henholdsvis stigende og faldende anden koordinat i arrayet.

Der er forskellige typer af vejplader (sving, lige,

Algorithm 3 Color modul kode

```

72 void LightSenseCalibrate( void ){
73
74
75     Port1SetActive();
76     Port2SetActive();
77     Port3SetActive();
78
79     EstimateMinMaxValue( 1, & Color1Mid, 1);
80     EstimateMinMaxValue( 3, & Color3Mid, 2);
81     EstimateMinMaxValue( 5, & Color5Mid, 3);
82     EstimateMinMaxValue( 2, & Color2Mid, 1);
83     EstimateMinMaxValue( 4, & Color4Mid, 2);
84     EstimateMinMaxValue( 6, & Color6Mid, 3);
85
86     while( ! View ); while( View );
87
88     Color1Threshold =
89         Color1Mid < Color2Mid ? Color1Mid + (Color2Mid - Color1Mid) / 2 : Color2Mid + (Color1Mid - Color2Mid) / 2;
90     Color2Threshold =
91         Color3Mid < Color4Mid ? Color3Mid + (Color4Mid - Color3Mid) / 2 : Color4Mid + (Color3Mid - Color4Mid) / 2;
92     Color3Threshold =
93         Color5Mid < Color6Mid ? Color5Mid + (Color6Mid - Color5Mid) / 2 : Color6Mid + (Color5Mid - Color6Mid) / 2;
94
95     lcd_show_int16( Color1Threshold ); lcd_show_digit(1); BusyPauseMS(1000);
96     lcd_show_int16( Color2Threshold ); lcd_show_digit(2); BusyPauseMS(1000);
97     lcd_show_int16( Color3Threshold ); lcd_show_digit(2); BusyPauseMS(1000);
98
99 }
100
101
102 Color LightSenseColor( int port )
103 {
104     if( port == 1 && Port1Raw() > Color1Threshold ) return BlackColor;
105     if( port == 2 && Port2Raw() > Color2Threshold ) return BlackColor;
106     if( port == 3 && Port3Raw() > Color3Threshold ) return BlackColor;
107     return WhiteColor;
108 }

```

Algorithm 4 Navigation modul - followlinekode

```

82 // Forward, Differential Free
83 PortA( OnPos );
84 PortB( Float );
85 PortC( OnPos );
86
87 // Drive until some road is discovered
88 while( EndOffRoad );
89
90 // FollowLine
91 while( ! EndOffRoad )
92 {
93     if( OnRoad )
94     {
95         // Forward, Defferential Locked
96         PortA( OnPos );
97         PortB( Brake );
98         PortC( OnPos );
99     }
100     else
101     {
102         PortB( Float );
103
104         if( left == BlackColor ) PortC( Brake );
105         else PortC( OnPos );
106
107         if( right == BlackColor ) PortA( Brake );
108         else PortA( OnPos );
109     }
110 }
111
112 // All Stop
113 PortA( Float );
114 PortB( Float );
115 PortC( Float );

```

Algorithm 5 Navigation modul - kommunikationskode

```

117 // Request Command
118 IRTransmit(0,ACKNOWLEDGED);
119
120 recieve_remote_command:
121
122 while ((!IRReceive(&m1, &direction, &t)))
123 {
124     // debugging purposes
125     MSCount = -1;
126 }

```

Algorithm 6 Navigation modul - navigationskode

```

128 //Navigation algorithm
129
130 // Forward, Differential Locked
131 PortA(OnPos);
132 PortB(Brake);
133 PortC(OnPos);
134
135
136 MSCount = distance = 0;
137
138 switch(direction)
139 {
140     case LEFT:
141         while(MSCount < timetocenter);
142         MSCount = distance = 0;
143         PortB(Float);
144         PortA(OnNeg);
145         if(mid == WhiteColor) while(mid == WhiteColor);
146         while(mid == BlackColor || MSCount < minturtime);
147         MSCount = distance = 0;
148         while(MSCount < leftturtime);
149         MSCount = distance = 0;
150         PortB(Brake);
151         PortA(OnPos);
152         while(MSCount < timefromcenter toroad);
153         break;
154     case RIGHT:
155         while(MSCount < timetocenter);
156         MSCount = distance = 0;
157         PortB(Float);
158         PortC(OnNeg);
159         if(mid == WhiteColor) while(mid == WhiteColor);
160         while(mid == BlackColor || MSCount < minturtime);
161         MSCount = distance = 0;
162         while(MSCount < rightturtime);
163         MSCount = distance = 0;
164         PortB(Brake);
165         PortC(OnPos);
166         while(MSCount < timefromcenter toroad);
167         break;
168     case STRAIGHT:
169         while(MSCount < timetocenter + timefromcenter toroad);
170         break;
171     case UTURN:
172         PortB(Float);
173         PortA(OnNeg);
174         while(mid == BlackColor || MSCount < minturtime * 2);
175         MSCount = distance = 0;
176         while(MSCount < leftturtime);
177         PortA(OnPos);
178         break;
179     case NOTHING:
180         PortA(Brake);
181         PortB(Brake);
182         PortC(Brake);
183         while(!View);
184         goto recieve_remote_command;
185         break;
186 }

```

kryds og t-kryds), som hver kan vendes på 4 forskellige måder i kortet, dog kan nogle af disse muligheder være sammenfaldende, f.eks. kan et sving gå fra nord og dreje mod vest, fra syd og dreje mod øst osv., hvilket i alt bliver 4 måder som et sving kan vendes på. Derimod kan et kryds kun ligge på en måde, da ligegyldig hvor mange gange man drejer den 90 grader, så kan ikke se forskel. Hver af disse ikke sammenfaldende muligheder har fået tildelt et unikt nummer som identificerer den. Dette id gemmes i arrayet og repræsenterer at der på dette sted i kortet er et vejelement af en given type og retning. Dette gør det muligt at bygge hvilket som helst kort, af disse simple vejelementer.

Som bilen kører frem i byen og detekterer vejplader, så kan disse detekteringer konverteres til givne steder i kortet. Dette bliver gjort ved at holde styr på bilens nuværende position og retning, ud fra disse sammenholdt med en detektering af vejplade type, så kan man opdatere kortet med det rigtige vejelement på det rigtige sted, f.eks. hvis bilens nuværende position er (5,6), har retning mod syd og der bliver detekteret et sving til højre, så må position (5,7) være et sving fra nord til vest, derudover er bilens nye position (5,7) og har retning mod vest.

Retninger, vej typerne, vej elementerne og kort arrayet er defineret i Map.h. Funktionerne til opdatering af kortet, givet en position, en retning og en vej type er implementeret i Map.c.

8 Planlægningen

Når bilen kører rundt i byen er der steder, hvor den har flere forskellige valgmuligheder for retninger den kan køre hen, f.eks. i et kryds er der 3 forskellige steder hvor den kan køre hen - lige ud, til højre eller venstre - hvad skal den vælge? Det bedste ville være hvis den kørte hen et sted hvor den ikke har været før, sådan at den bruger mindst mulig tid på steder hvor den allerede har kortlagt. Derudover hvis bilen f.eks. kører lige ud, hvordan kommer den tilbage og kortlægger vejene der var højre og venstre i krydset.

Det er her kørsels planlægningen kommer ind i billedet, givet et kort, nuværende retning og position så skal den bestemme hvorhen bilen skal kører

næste gang. Der bliver brugt forskellige datastrukturer og algoritmer for at opnå dette: en stak, en kø, en prioritets kø og Dijkstra's algoritme (single-source shortest path).

Der ikke er implementeret nogen memory management på RCX og vi følte det var lidt et overkill at begynde at implementerer malloc og free. Alle vores datastrukturer bliver i stedet for allokeret globale og statiske, dermed bliver størrelsen af disse datastrukturer vigtig, da hukommelsen er stærkt begrænset i RCXen?? .

8.1 Stack

En stak fungerer som bekendt som LIFO (last-in first-out)[?, s. 57-60] og er implementeret som et array af positioner. Stakken indeholder positioner, da den bruges til at indeholde positioner på vejplader som man har kørt forbi, men som bilen ikke kørte udad. Som nævnt før når bilen f.eks. kommer til et kryds, der har man 3 forskellige valgmuligheder for vejplader man kan køre udad. Hvis man vælger at køre lige ud, så ligger man positionerne for højre og venstre vejplader på stakken, sådan man senere kan finde tilbage til dem igen og undersøge dem.

Ideen med at bruge netop en stak til at holde de positioner man mangler at besøge, er der større sandsynlighed for at der er kortere vej tilbage til de øverste elementer på stakken, for det er dem man senest har været ved. Det gælder selvfølgelig ikke altid at der er kortest vej til de øverste elementer og en bedre løsning ville være at holde en liste af alle manglende vejplader, og når man har brug for at køre til en manglende vejplade, så laver man single-source shortest path og finder den korteste vej fra nuværende position til alle andre positioner, derefter scanner man listen igennem og finde den manglende plade som reelt er tætteste på.

Arrayet - som stakken ligger i - har en fast størrelse på lige så mange som antallet af elementer i kortet. Dette giver ingen problemer med overflow, da der højst kan ligges antallet af elementer i kortet på stakken.

Implementeret af stakken kan findes i Stack.h/Stack.c.

8.2 Queue

En kø fungerer som bekendt efter princippet FIFO (first-in first-out) og er implementeret som et cyklisk array, som beskrevet i [?, s. 61-64]. Køen skal indeholde kørsels kommandoer (forward, left, right, u-turn), som bilen skal udføre. Dvs. efter f.eks. man har fundet en rute (vha. Dijkstra algoritme) mellem to forskellige positioner, så kan lave kommandoer - som man push på stakken - der navigere bilen fra udgangspunktet til målet.

Arrayet som bruges til implementation af køen, har ligesom i stakken en fast størrelse. Hvis array mindst har størrelse som antallet af mulige elementer i kortet plus en konstant, så er der ikke problemer med overflow, hvis køen kun bliver brugt til navigering af en korteste rute mellem 2 punkter ad gangen.

Implementeret af stakken kan findes i Queue.h/Queue.c.

8.3 Priority queue

Der er brug for en prioritets kø i Dijkstra algoritme's for single-source shortest path, dog skal denne prioritets kø være lidt speciel, da den skal være opdaterbar, dvs. der skal være mulighed at opdatere prioriteterne for elementer i køen sådan at invariansen for prioritet køen stadig gælder.

Prioritetskøen er implementeret vha. et array baseret min-Heap [?, s. 94-106]. Opdaterbare prioriteter er lavet ved en scanning igennem alle elementer i køen og updater prioriteten for det givne element. For at invariansen for Heap/Prioritets køen overholdes laves der en Heap-up på det opdaterede elementet. Det er nok kun at lave et Heap-up, da Dijkstra's algoritme altid kun opdaterer prioriteten til en bedre værdi.

Arrayet som prioritets køen ligger i, har samme størrelse som antallet af elementer i kortet. Dette giver ikke problemer med overflow, da prioritets køen kun bliver brugt i Dijkstra's algoritme, hvilket højst ligger alle elementer i kortet ind i køen.

Implementationen af prioritets køen kan findes i PQ.h/PQ.c.

8.4 Single-source shortest path

For at finde den korteste vej mellem vejplader i byen, bruges Dijkstra's algoritme[?, s. 342-348]. Efter den korteste rute er fundet så omsættes den til en række af kommandoer, som navigerer bilen fra nuværende position til bestemmelsesstedet. Disse kommandoer bliver lagt på køen ??, hvorefter bilen udfører dem. Når kommandoer er blevet udført og bilen er kommet til bestemmelses stedet, så genoptages den normale måde at bestemme hvor den så skal køre hen.

Alle veje mellem to nabo plader har fået vægten 1, lige meget om det er lige ud eller et sving. Da vores bil nok ikke verdens hurtigste, så talte vi om at det var bedre at måle hvor lang tid det tog at lave et "sving"/"kører lige ud" og så bruge dette som vægtene i grafen, da et sving tager meget længere tid. Dermed kunne man minimere tiden det tager at komme fra et sted til et andet, i stedet for at minimere afstanden bilen kører.

Implementationen af Dijkstra's algoritme kan findes i Planning.h/Planning.c i funktionen planRoute.

8.5 Kørsels muligheder

Som nævnt overfor kan man have flere forskellige valgmuligheder for kørselsretninger, når man kommer til man til et kryds eller t-kryds. Vi prøver altid at køre et sted hen vi ikke har været før, hvis bilen f.eks. står i et kryds og den allerede har været til plader til højre og lige ud, så kører den til venstre.

Vi havde en ide om at det var lettere at køre lige ud, så når bilen har valgmulighed om at køre lige ud eller dreje, så vælger den at køre lige ud. Senere eksperimenter har vist at det faktisk er en god ide at dreje en gang imellem, da måden vi drejer på kan rette bilen op hvis den kører skævt op ad en vej.

Hvis bilen kommer til et sted og den allerede har været på alle nabo pladerne, så tages der en position af stakken ?? og ser om den har været der siden den lagde den på stakken, hvis dette er tilfældet så tages den næste af stakken og der prøves igen, ellers planlægges der en rute til denne position ??.

9 Memory

Ifølge [?] er der 28671 bytes (hukommelses område 0x8000 - 0xEFFF) fri til program og data. Dette er relativt lidt plads og kan let give problemer, da der ikke skal særlig store mængder kode/datastrukturer til at fylde dette op. Specielt kortlægningen og planlægningen fylder meget pga. de mange speciale tilfælde som koden indeholder og de mange "store" datastrukturer der bliver brugt.

Faktisk fyldte hele den kompilerede kode for kortlægningen og planlægningen 32 kilo bytes, dette resulterede i at der blev sat arbejde ind for at nedbringe denne størrelse. Der blev først prøvet en compiler parameter (-Os), som skulle optimere kompileringen efter størrelse, men dette virkede kun minimalt. Dernæst begyndte vi at strikke alle unødvendige variabler, funktions kald osv. fra vores egen kode, specielt blev alle fejlbeskeder fjernet. Dette hjalp en hel del, men der manglende stadig nogle kilo byte før vi var under grænsen. Derefter blev de udleverede source filer - som bliver brugt til at styre RCXen med - gennemgået, for at fjerne alt unødvendig. Her i blandt blev alt hvad der har med lyd at gøre fjernede, vi behøver ikke at knapperne siger bip-lyde når der trykkes på dem. Derudover blev vores Light-sense modul også gjort en del simplere, kalibreringen blev fjernet og der blev brugt hardcodede thresholds i stedet for. I alt blev koden 5 KB mindre, hvilket var nok til at den kom under grænsen på 28 KB.

Senere tilføjede vi dog mere kode, for upload af kort til en computer. Hvilket gjorde at størrelse på den kompilerede oversteg grænsen på 28 KB, dog virkede programmet stadig. Dette skyldes at vi havde kigget på størrelsen af den kompilerede koden på PCen som filen blev kompileret på - hvilket er en 32-bit maskine - men når den uploades til RCX så konverteres disse til 16-bit, hvilket halvere størrelsen på koden ("Doh!" [?]).

Vi spildte nok en hel del tid på at gøre koden mindre, men det var nok ikke helt spildt for vi fik gennemgået vores egen kode rimelig grundig og fandt faktisk også et par fejl. Derudover fik vi også gennemgået noget af den udleverede kode, det hjalp os ikke direkte, men var lærerigt :-).

10 Program strukturen

Pseudo koden for kortlægnings/planlægnings programmet er vist i algoritme ???. Der startes med at initialiserer alle datastrukturerne og aktiverer sensorerne, derefter køres et loop indtil alle vejelementerne i byen er detekteret.

En kørsels af loopen indeholder planlægning, kørsels henad en vejplade, detektering og opdatering af kortet. Dette forsætter som sagt indtil alle vejplader er detekteret, hvorefter kortet uploades til en computer og der downloades en koordinat i byen hvor bilen skal køre hen. Til slut deaktiveres alle sensorerne igen.

Beregning og planlægningen af hvad bilen skal gøre - `nextCommand(currentPosition, currentDirection)` - er forklaret i ???. Efter at have beregnet/planlagt den næste kommando, sendes denne til RCXen som har ansvar for at køre bilen, hvorefter der ventes på at kørsels-RCXen melder tilbage at den har udført kommandoen. Der beregnes hvilken retning og position bilen nu står, ud fra den antagelse at kommandoen blev udført rigtig. Bilen detekterer hvilken type af vejelement den nu står på og opdaterer kortet med denne information.

Artiklen [?, s. 192-193] har 4 typer af basale måder hvorpå man kan kontrollere en robot: reactive-control, deliberative control, hybrid-control og behaviour-based control. Det er helt klart at kontrollen af vores bil hverken udelukkende sker via. reactive-control eller behaviour-based. Der sker simplehen for alt for meget central planlægning for at det kan være nogle af disse. Deliberative control er det heller ikke helt, da bilen ikke planlægger hvordan den skal køre, kun i hvilken retning den skal køre i og derudover reagerer vores bil reaktivt i selve kørslen. Hybrid modellen passer derimod perfekt, hvor den deliberative og reactive control er mixet sammen. Præcis som beskrevet i artiklen har bilen en del der planlægger og en anden der søger for kørslen, dette er også understreget af at de to dele faktisk ligger på hver deres separate RCX og snakker sammen over IR.

Algorithm 7 Pseudo kode for programmet der planlægger, detekterer og kortlægger byen

```
// init the map, stack, queue etc. and init and activate the sensors
initialize();
while(!done) {
    // calculate and plan the next command
    command = nextCommand(currentPosition, currentDirection);
    // send the command to the other RCX
    send(command);
    // Wait for the other RCX to signal that it has driven to the detect position
    waitReadyDetectSignal();
    // calculate the new position and direction
    currentPosition = calcNextPosition(command, currentPosition, currentDirection);
    currentDirection = calcNextDirection(command, currentDirection);
    // Detect what road type the car is standing on
    detectedRoadType = detect();
    // Update the map with the new information
    updateMap(detectedRoadType, currentPosition, currentDirection)
}
// upload the map to a computer
uploadMap();
// download the coordinates of a road element from a computer
downloadCoordinates();
// deactivate the sensors
finalize();
```

11 IR kommunikation

Vi bruger to RCX'er, den ene til planlægning (planlægningsenhed), og den anden til kørsel (kørselsenhed). Vi valgt at bruge to RCX'er var, at vi ikke havde tilstrækkelig hukommelse på en RCX, og vi havde flere inputsensor end der er inputport på en RCX. Vores programmer og de filer, der skulle inkluderes fylder lidt mere end den mængde hukommelse der er til rådighed på én RCX.

Brugen af to RCX'er kræver at de skal kommunikere sammen. Ifølge [?] skelnes mellem situated kommunikation og abstrakt kommunikation. Situated kommunikation er en kommunikationsmåde, hvor både de fysiske egenskaber af signal og indholdet af det, får beskeden til at give mening. Et eksempel på en situated kommunikation kan være, når et menneske siger "kom til mig". Fra de fysiske egenskaber af lyden, kan man lokalisere personen, og indholdet af beskeden fortæller, hvad der skal gøres. Den slags kommunikation er ikke særlig relevant for roboternes verden. I abstrakt kommunikation har det fysiske signal, der transporterer beskeden, slet ikke nogen mening. Til gengæld er det kun indholdet af beskeden der har mening. Kommunikation mellem robotter er som regel en abstrakt kommunikation [?]. I vores projekt indeholder RCX nummer et (planlægningsenheden) hoved programmet, den styrer bilen, dens hoved opgave er, at detektere vejefeatures (kryds, T-kryds, ligeud) og give kommando til RCX nummer to (kørselsenhed). Kørselsenheden sørger for, at den modtagne kommando bliver korrekt udført, desuden er den ansvarlig for at bilen kører ligeud på byens baner.

Bilen kører indtil den når enden af en vejstribе, derefter sender den signal til planlægningsenhed og venter på en kommando. Og så sender planlægningsenheden kommando om hvad bilen skal fortage sig. Kommunikationen mellem RCX'erne er ikke altid stabil dvs. nogle af beskeder går tab, dette skyldes to ting ifølge [?] stærkt lys kilder i omverden og overførelse af stor pakker. Vi løser dette problem ved at sende en besked mere end en gang.

12 Start på GUI'en

Formål:

Vi stiller følgende krav til vores gui, det skal være muligt at :

- 1.Se det kort som bilen bygger op
- 2.Give bilen ordre til at køre til en bestemt lokation
- 3.Lave og download'e et kort til lego bilen
- 4.Vise dele af bilens tilstand mens den køre.

En forløbige analyse af kravene til vores gui resulterede i at vi har brug for fire funktioner i guien:

- 1.Visning af et kort over lego byen, som starter med at være tomt. Vores repræsentation af byen på RCX'en er et to dimensionelt array af 10x10 integers. Et tal mellem 0 og 14 tilkendegiver et bestemt legovej element, og -1 betyder ukendt. Derfor virker det naturligt at kortets repræsentation bliver 10x10 felter som kan vise et tegnet-billede af et af legovej-elementerne.
- 2.En tilstand hvor kursoren indsætter et legovej element på kortet valgt af brugeren
- 3.En tilstand hvor kursoren (ved klik på kortet) udpeger et sted som bilen skal køre til og downloader dette steds koordinater til RCX'en.
- 4.En knap som hedder noget i retning af download kortet til RCX

Fremgangsmåde:

Det synes naturligt at

- 1.Finde en referance manual til GTK
- 2.Få overblik over strukturen i GTK
- 3.Få indblik hvordan GTK gui-builderen Glade-2 fungerer
- 4.Undersøge hvilke gui elementer som kan bruges til at repræsentere kortet

- 5. Undersøge hvordan vores kode opdaterer gui elementerne.

Resultater:

At finde en manual til C- GTK var ikke særlig svært, første gæt (<http://www.gtk.org>) virkede. Her var også et godt overblik over hvordan widget's (den mest generelle gui komponent hedder i GTK-jargon et widget) relaterer sig til hinanden (<http://developer.gnome.org/doc/API/2.0/gtk/ch01.html>) samt en tutorial (<http://www.gtk.org/tutorial/>) til hvordan man kommer i gang med at lave en GTK-applikation. Punkt et og to gik altså ganske smerte frit.

At starte et GTK projekt op i Glade-2 var lige ud af landevejen, jeg valgte at lave et nyt TOP_LEVEL vindue, med en layout manager kaldet GtkFixed som tillader absolut placering af elementerne.

Den første ide til at repræsentere kortet var at sætte 10x10 GtkImage's op ved siden af hinanden og så tegne legovej elementerne i Gimp. En funktion skulle have til opgave at opdatere billederne hvergang modellen opdateres. Desværre var der ikke en event handler for mouse-clicked tilrådighed for GtkImages. En GtkButton har derimod mouse-clicked eventet og man kan have et billede i stedet for en label. Det har dog den ulempe at knappen bliver 3D når musen føres hen over den.

Da sessions formål er at teste om konceptet er muligt er kortet i vores initiale gui repræsenteret vha. 4x10 GtkButtons, da det er utrolig tidskrævende at sætte GtkButtons ind ved siden af hinanden så de står pænt.

Det skal også bemærkes at det tog lidt over en time at at tegne bane elementerne i Gimp.

Ved et tryk på knappen Build Code, i glade-2 laves koden. I mappen legocargui skrives `./autogen.sh && ./configure && ./make && src/legocargui` og guien compileres, dog uden nogen funktionalitet i knapperne. Billedet herunder kommer frem på skærmen :

Dette afslutter denne første session som om handler guien. Konklusionen af denne session er at det indtil nu virker muligt at imødekomme vores krav til gui'en på PC siden (specielt indenfor en rimelig tidsramme). To alt afgørende ting mangler dog at blive undersøgt før konceptet har vist sit værd :

- 1. Hvordan hænger vi vores kode på gui
- 2. Det skal undersøges om IR-linket mellem PC og RCX tillader at gennemføre planen.

vi nåede derfor ikke helt at vise proof of concept i denne session.

13 Gui arbejdet forsætter

I forrige session producerede Glade-2 en mappe kaldt legocargui (efter projektets navn). Mappen indeholder tre under mapper. Mappen Po som tilsyneladende bruges af Glade selv, mappen pixmap's som indeholder billederne applikationen bruger og mappen src som indeholder koden. I src har Glade lavet en række filer, `interface.c/h`, `callbacks.c/h`, `support.c/h` og `main.c` samt `Makefile`. `Callbacks.c` er controler delen af en Model View Controler ligende struktur. `Interface.c/h` er View delen og modellen skal vi selvfølgelig selv tilføje.

Formål: I denne session vil vi lave en model og teste den på pc vha. et demo program som simulere input fra IR-tårnet.

Fremgangsmåde: Vi har overvejet tre modeller for hvordan bilen sender informationer til Computeren. I den første model sender bilen hele kortet til computeren hvergang et legovej element opdages (ie. hvergang RCX'ens indre repræsentation opdateres).

I den anden model sender RCX'en koordinatet og vej-typen hvergang bilen opdager en plade.

I den tredje model sender bilen først hele kortet til computeren når hele byen er opdaget. Da vi fik brug for to RCX'er valgte vi den sidste model. Vi vil forsøge med at bygge guien i fire dele, nemlig Model View Controler og ModelUpdater. Det sidste modul har ansvar for at lytte til IR-tårnet og underrette

modellen når et kort modtages.

Vi har identificeret følgende ansvar for modellen :

- 1.Vedligeholde en repræsentation af kortet.
- 3.Opdaterer Guien når et nyt kort modtages
- 4.Opdatere repræsentationen af kortet når brugeren sætter et vej element ind (på gui'en)
- 5.Holde styr på kursorens tilstand (sætte et vej element ind eller udvælge en destination for RCX)
- 6.Sende en kommando til bilen om at køre et bestemt sted hen når brugeren beder om det
- 7.Sende et kort til RCX'en når en bestemt knap aktiveres.

Model.h får interfacet:

- `init` - denne funktion sætter kursorens tilstand til 0 jf. diskussionen ovenfor (funktioner for guien) svare det jo til redigering af kortet.
- `setMap(int map[10][10])` - sætter gui'en interne repræsentation af kortet til `map`, og kalder `updateDisplay()`
- `setCarPosition(int x, int y)` - sætter en rød prik på kortet på den angivne position
- `int ** getMap()` - Returnere en pointer til et 10x10 int array som er en kopi af den interne repræsentation for gui'en.
- `updateDisplay()` - Funktion opdaterer displayet, ie. sætter alle `GtkImages` i knapperne til at vise billeder svarende til gui'ens interne repræsentation.

Efter at have implementeret ovenstående funktion i `Model.c`, samt diverse ekstra funktion hvis funktionalitet er forklaret som kommentare i koden, lavede vi `Demo.c` som starter en tråd der lige venter et sekundt hvorefter `main`-tråden forsætter med at starte gui'en. Denne ekstra tråd kalder metoderne implementeret i `Model.c` og det testes (manuelt ved at åbne øjnene

og kigge på skærmen) om funktionerne har den rigtige effekt.

Resultater: Denne session resulterede i at vi har en fungerende model. Alting er indtil nu gået godt, men det har taget lang tid at finde udaf hvordan et bestemt widget findes og opdateres. Vi har selv lavet funktion som kan søge op og ned i widget træet, for at illustrer problemstillingen betragt da følgende typiske problem vi har været stillet overfor.

Givet position `x,y` og et vej element (ie. `map[5][5]`) skal vi finde den knap som svarer til position 5,5 på kortet. Her er det knap-widget's navngivning som bliver brugt til at identificerer hvor på kortet et widget sidder. Knappen `map55` er en underkomponent af `Fixed1` som igen er en underkomponent af `Legocargui` (eller `Logocargui` i koden pga. en fejl i starten). Da vi tit leder efter widgets på deres navn gav dette anledning til to funktion `search[Up|Down]ForWidget(char *name, widget * start)`, hvor `start` er et widget som indeholder knappen (ie. vi søger ned) eller hvor `start` er et widget som er indeholdt i det widget vi søger efter (ie. vi søger op).

14 IR-mellem RCX og PC

Formål: I denne session vil vi lave `ModelUpdateren` og skrive et program som skal køre på RCX'en. Det sender et kort til RCX-tårnet sådan, som vi har tænkt os den endelige legobil skal gøre det.

Fremgangsmåde: Først kiggede vi på koden fra (http://www.daimi.au.dk/~jalp/rcx_ir.tar.gz) for at få en ide om hvor svært/let det er at modtage og sende bytes til og fra RCX'en. Efter at have kigget på koden i `ir_recv.c` har vi fundet ud af hvad der kan bruges (se kode fragmentet 1). Nu skal vi lave en funktion som lytter til IR-tårnet og kalder `void opdaterKort(int map[10][10])` når et kort er modtaget. Koden i `??` viser loopet som lytter til IR-tårnet, og køre i en tråd for sig selv.

Det meste af `ir_send.c` er parsning af kommando linje argumenter, det vi har brug for er vist i `??`

som er pakket ind i funktionen `sendPosition`

Algorithm 8 Den tilpasset udgave af `ir_recv ...`

```

131
132
133     else {
134         printf("m0:_%u_m1:_%u\n", m0, m1);
135         for(i=0; i < MAP_SIZE; i+=2) {
136             for(j=0; j < MAP_SIZE; j+=2) {
137                 if((recv_count = ir_receive(&m1, &m0, &t)) > 7) {
138                     map[i][j] = m1;
139                     map[i][j+1] = m0;
140                 } else {
141                     printf("error_recv_map\n");
142                 }
143                 if((recv_count = ir_receive(&m1, &m0, &t)) > 7) {
144                     map[i+1][j] = m1;
145                     map[i+1][j+1] = m0;
146                 } else {
147                     printf("error_recv_map_2\n");
148                 }
149             }
150         }
151         printf("Opdaterer_kort_");
152         opdaterKort(map);
153         if((recv_count = ir_receive(&m1, &m0, &t)) > 7) {
154             x = m1;
155             y = m0;
156         }
157         setCarPosition(x,y);
158     }
159 }

```

Algorithm 9 Koden til at sende en kommando til RCX'en

```

912 void sendPosition(int x, int y) {
913     word_t m0, m1, t;
914     void *shared_memory;
915     struct rcx_send_buffer *send_buffer;
916     int shmid;
917
918     if ((shmid =
919         shmget((key_t)1234, sizeof(struct rcx_send_buffer),
920             0666 | IPC_CREAT)) == -1){
921         perror("shmget_failure");
922         return -1;
923     }
924     shared_memory = shmat(shmid, (void*)0, 0);
925     send_buffer = (struct rcx_send_buffer*)shared_memory;
926
927     if (send_buffer == (void*)-1) {
928         perror("shmat_failure");
929         return -1;
930     }
931
932     /** ***** **/
933     /** SEND THAT POSITION **/
934     /** ***** **/
935     while (send_buffer->ready == 1);
936     send_buffer->m1 = x;
937     send_buffer->m0 = y;
938     send_buffer->ready = 1;
939
940     if (shmdt(shared_memory) == -1) {
941         perror("shmdt_error_..._what_ever_that_is_???.return(-1)_\n");
942         return -1;
943     }
944 } // end function sendPosition

```

(callbacks.c 912 - 946) som kaldes når der trykkes på kortet samtidig med at kursorens tilstand er 1. Er kursorens tilstand 0 opdateres kortet i stedet med currentBrick der hvor der trykkes. Jf. Model.h og callbacks.c. Koden til en test RCX'en som simulere afsending af et kort, altså et burst på 100 vej-element er skrevet. Den ses i ??

Resultater: Holdes RCX'en tæt på IR-Tårnet og afskærmes forbindelsen virker kommunikationen helt fint, men kommer der interferens (ie. nogle andre downloader kode til deres RCX) eller bliver forbindelsen (ie. en hånd føres ind i mellem RCX og IR-Tårn) afbrudt kort varrigt har vi oplevet at kortet der bliver tegnet overhovedet ikke ligner det som burde være kommet frem.

15 Future Work

Vores bil kører stabil og finder den korteste vej, det vil sige at, den udfører det ønskede funktionalitet, men der er lige nogle ting vi kunne ændre på, hvis vi havde lidt mere tid. Vi kunne eventuelt byg bilen om således at der blev brugt lidt færre sensor på spil så vi kunne forsøge at pakke koden på een RCX; bilens hastighed er også lavt, det kunne vi også godt gøre noget ved.

16 Konklusion

Vi har lavet en bil der kan køre i en by og lave et kort over den. Derefter kan vi upload kortet til computeren og se det på skærmen ved hjælp af den GUI vi har lavet. På GUI 'en kan man pege på et sted i byen hvor bilen skal køre hen til via den korteste vej. Vi tager udgangspunkt i [Maja] men implementere vores egen arkitekturer. Der blev diskuteret forskellige løsninger som gav anledning til forskellige kode-strukturer og bil design. Der er to applikationer som kører parallel hver for sig, er decentral og interagere via IR. Den ene ansvarlig for kørsel og den anden for kortbygning og planlægning af den korteste vej.

Algorithm 10 TestCar.c koden som sender kortet.

```
89  while (!Prgm) {
90      lcd_show_int16(12);
91      while (!View);
92      {
93          int i, j;
94          lcd_show_int16(13);
95          IRTransmit(0,0);
96          for (i=0; i < MAP_SIZE; i+=2) {
97              for (j=0; j < MAP_SIZE; j+=2) {
98                  IRTransmit(map[i][j], map[i][j+1]);
99                  IRTransmit(map[i+1][j], map[i+1][j+1]);
100             }
101         }
102         IRTransmit(currentPosition.x, currentPosition.y);
103     } // end of while !View
104 } // end of while !Prgm
```
