UTVECKLING OCH IMPLEMENTATION AV KOLLISIONSKONTROLL I REALTID PÅ TUNNELDRIVNINGSMASKINER.

Examensarbete utfört i Reglerteknik
vid Tekniska Högskolan i Linköping

av

FREDRIK GRAHN

Reg nr LiTH-ISY-EX-2050

UTVECKLING OCH IMPLEMENTATION AV KOLLISIONSKONTROLL I REALTID PÅ TUNNELDRIVNINGSMASKINER.

Examensarbete utfört i Reglerteknik
vid Tekniska Högskolan i Linköping

av

FREDRIK GRAHN

Reg nr LiTH-ISY-EX-2050

Handledare: JÖRGEN APPELGREN
MIKAEL NORRLÖF

Examinator: SVANTE GUNNARSSON

Linköping, 28 januari 1999

Sammanfattning

Atlas Copco Rock Drills AB utvecklar och tillverkar utrustning för bergsbrytning. I företagets sortiment finns bl.a. en maskinfamilj kallad *Boomer* som har två till tre hydrauliska robotarmar för att borra spränghål för tunneldrivning. Armarna kan förflyttas automatiskt av styrprogramvaran eller manuellt av en operatör.

De nya borrmaskinerna på *Boomer* borrar snabbare, vilket kräver en automatisk reglering av borrprocessen för att utnyttja maskinens möjliga kapacitet. Detta examensarbete omfattar utveckling av en strategi för kollisionskontroll mellan robotarmarna på *Boomer*. Kontrollen detekterar och varnar för kollisionsrisk mellan maskinens armar.

Den strategi som presenteras i rapporten bygger på teorier och algoritmer inom simulering och datorgrafik, där intensiv forskning bedrivs för att hitta effektiva algoritmer för att detektera kollision mellan objekt i tre dimensioner. Testkörning av kollisionskontrollen på befintlig hårdvara, där en polygonmodell modellerar borrarmarna, visar att den framtagna strategin har realtidsegenskaper som gör att den kan användas både vid manuell och automatisk styrning.

Innehållsförteckning

1 Inledning

Atlas Copco Rock Drills AB utvecklar och tillverkar utrustning för bergsbrytning. Utrustningen säljs och används i stort sett i hela världen. Norge, Kanada, Ryssland och Australien är några exempel utanför Sverige där Atlas Copcos kunder finns. Tillverkningen av maskiner som används under jorden delas i huvudsak upp efter tre typer av borriggar: *Boomer*—en maskin som borrar 5–7 meter djupa spränghål i tunnlar för vägtunnlar, orter i gruvor och liknande. Maskinen är utrustad med en borrmaskin vardera på de en till tre bommarna. *Boltec*—används för att förstärka berg genom att spänna ut berget med bultar i borrade hål. *Simba*—bryter malm i gruvor genom att borra en solfjäder av 30–50 meter djupa hål i en malmkropp som sedan sprängs.

Detta examensarbete inriktar sig på förbättringar inom maskintypen *Boomer* och utvecklar en strategi för kollisionskontroll mellan bommarna på *Boomer*. I arbetet ingår även att skriva en rapport och dokumentation till egenutvecklad programvara. Företaget tillhandahåller arbetsstation, nödvändig programvara samt möjlighet till testning av kod i hårdvara för målmiljön.

1.1 Bakgrund

De borriggar som tidigare tillverkades utrustades med ett centraliserat styrsystem med endast en enhet för alla beräkningar, vilket innebär att enheten fick ta hand om alltifrån insamling av mätsignaler från givare, till att ställa utstyrningar på ventildon och göra alla nödvändiga beräkningar däremellan. Den nuvarande strategin för kollisionskontroll är därför anpassad för att den tillgängliga beräkningskapaciteten inte ska överskridas hos den enda enheten. Den nya generationen *Boomer*-riggar däremot, är utrustade med ett distribuerat styrsystem där flera s.k. ”intelligenta enheter” är sammankopplade i ett nätverk på borriggen, vilket då innebär att regleringen av bommarna sker på respektive bom från meddelanden hos en överordnad enhet. Det distribuerade styrsystemet ger mer beräkningskapacitet i borriggen, vilket gör det möjligt att använda en mer naturtrogen men mer komplex modell av bommarna. Den komplexa modellen gör kollisionskontrollen mer beräkningskrävande, men bommarnas rörelser kan då utnyttjas bättre än tidigare.

I vissa fall har automatiken för positionering i onödan larmat för kollisionsrisk mellan bommarna. Orsakerna till falska larm behöver inte nödvändigtvis bero på kollisionskontrollen utan kan även ha andra orsaker, exempelvis undermåligt kalibrerade givare. I andra fall har den automatiska positioneringen varit riskabel vid förflyttning av bommar på riggen; ur operatörens synvinkel har bommar passerat varandra med ett ”för kort” avstånd. Larmen om kollisionsrisk yttrar sig så att operatören får en varning om att bommens position inte kan ändras enligt de regler som finns i programvaran, varefter operatören tvingas flytta den aktuella bommen manuellt. Detta upplevs som irriterande för operatörerna, som istället väljer att alltid positionera bommarna manuellt.

Den nya generationen borriggar borrar snabbare, men det får till följd att *en* operatör inte hinner positionera de tre bommarna på riggen. Efter att ha positionerat den första bommen och startat dess borrmaskin, hinner han inte ens positionera färdigt den andra bommen förrän den första har borrat färdigt sitt hål och väntar på att bli förflyttad till nästa hålposition. Vilken bom operatören än väljer att positionera därefter, kommer en bom att förbli stillastående. Operatören blir stressad och borriggen utnyttjas dåligt. Med en väl fungerande *automatisk* positionering och borrning får operatören en mindre stressande arbetssituation och kan ha en mer övervakande arbetsroll.

1.2 Problemformulering

En modernisering av den nuvarande strategin för kollisionskontroll av bommarna måste ske för att bättre avspegla bommarnas verkliga utseende, i hopp om att kunna utnyttja riggens kapacitet bättre.

Problemformuleringen för examensarbetet sammanfattas i följande punkter:

- Förbättra och förnya strategin för kollisionskontroll.
- Implementera, simulera och verifiera hela eller delar av kollisionskontrollen.
- Utföra tester av kollisionskontrollens prestanda i målmiljön.

1.3 Dokumentöversikt

I avsnitt 2, *Systembeskrivning*, ges en beskrivning av de fysiska förutsättningar som gäller för arbetet. Här ges en inblick i hur det går till att driva en tunnel genom ett berg, vilka maskiner som används samt hur de fungerar. Därefter, i avsnitt 3, *Teoretisk bakgrund*, förklaras de teorier som ligger till grund för arbetet.

En presentation av olika befintliga mjukvaror för kollisionskontrollgörs i avsnitt 4, *Mjukvara för kollisionskontroll*. Sedan presenteras en lösning på problemet med kollisionskontroll på borriggar i avsnitt 5, *Lösning*, följt av en sammanfattning och utvärdering av resultatet i avsnitt 6, *Sammanfattning och slutsatser*.

Slutligen i avsnitt 7 redovisas källor till den information som använts i rapporten och allra sist i rapporten, i avsnitt 8, återfinns dokumentation av den programvara som använts för testning och verifiering av den nya strategin för kollisionskontroll.

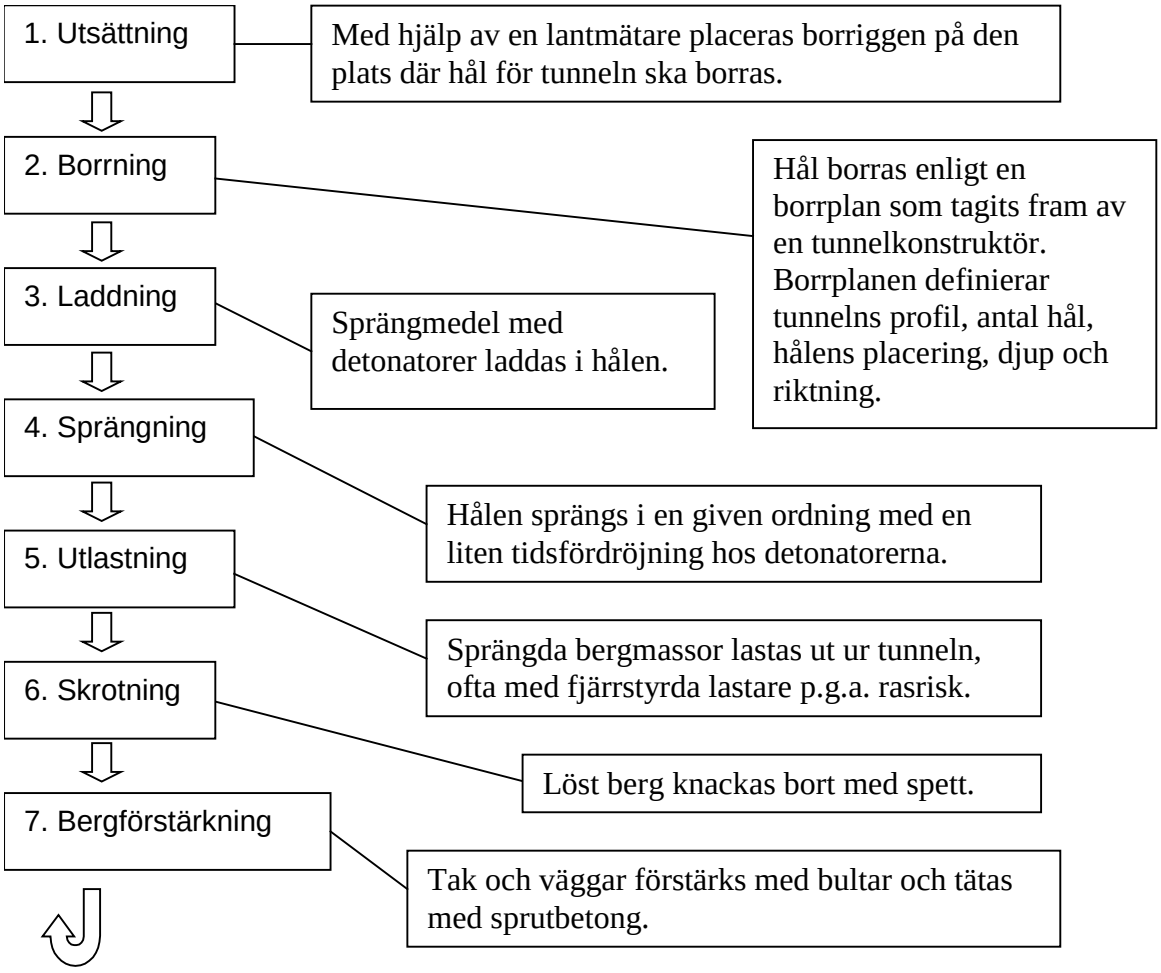
2 Systembeskrivning

I detta avsnitt ges en genomgång av de fysiska förutsättningarna för arbetet. Först kommer en beskrivning av de arbetsmoment som är involverade då en tunnel ska drivas genom ett berg, följt av borriggars konstruktion och deras styrsystem och sist den befintliga kollisionskontrollen på borriggarna.

2.1 Tunneldrivning

Innan själva tunneldrivningsprocessen startar måste undersökningar göras för att utreda bergets egenskaper. Beroende på dessa egenskaper och den framtida tunnelns användning skapas en borrplan som sedan framställs grafiskt i ett program för konstruktion av olika typer av tunnlar. Där anges vilken profil tunneln ska ha, hur många hål som ska borras för varje sprängsalva, samt den position, riktning och djup som varje hål ska ha. Resultatet sparas i en datafil och överförs sedan av operatören till en borrigg med hjälp av ett minneskort. Filen laddas i borriggen och borrplanen presenteras på en skärm som stöd för operatören under pågående borrning eller används som indata till styrprogramvaran vid automatisk borrning.

Processen att driva en tunnel genom ett berg kan beskrivas med hjälp av flödesschemat i Figur 2.1 nedan, som illustrerar de aktiviteter som utförs under jord.



Figur 2.1: Flödesschema för tunneldrivning.

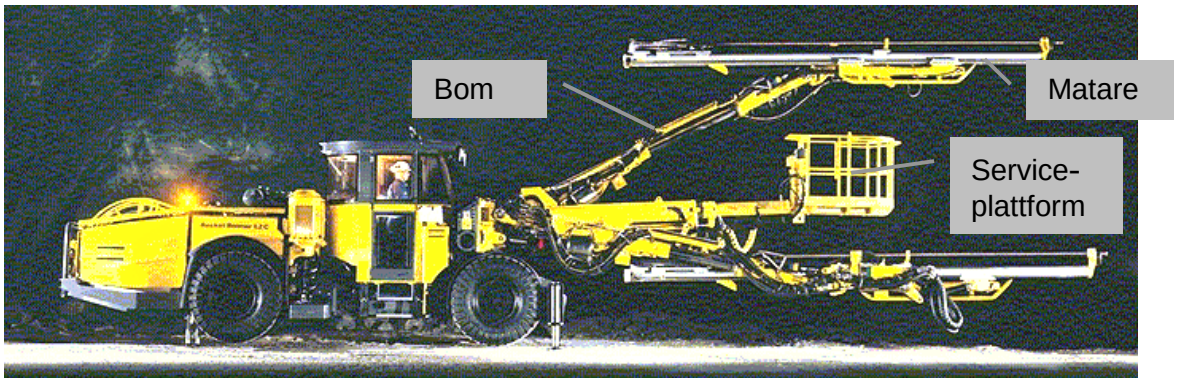
Tunneldrivningen inleds med att en lantmätare *sätter ut* platsen för tunnelns hål och riktar in borriggen med hjälp av en laser. Därefter borras ett hundratal hål som laddas med sprängmedel. Vid *borrningen* kan operatören välja att positionera och borra varje hål manuellt med stöd av den aktuella borrplanen eller låta ett program styra en eller flera bommar automatiskt. Programmet tar då hand om både positionering och borrning av hål samt ser till att bommarna inte kolliderar under förflyttning mellan de olika hålpositionerna.

Efter att alla hål är borrade och *laddade* med sprängmedel, aktiveras detonatorerna i en given ordning med ett visst tidsintervall. Sedan *lastas* de bortsprängda bergmassorna ut ur tunneln. På grund av den kraftiga rasrisken används ofta fjärrstyrda lastare. Därefter *skrotas* berget, vilket innebär att löst berg knackas bort med spett och forslas ut ur tunneln.

Det sista momentet i processen innefattar *förstärkning* och *tätning* av tunnelns tak och väggar. Förstärkningen kan göras med långa bultar i berget eller med speciella rör som sticks in i berget och fylls med vatten under mycket högt tryck. Därmed elimineras risken för bergras i tunneln. Tätningen används för att undvika att berget dräneras på grundvatten och genom att spruta betong under tryck i hål utanför tunnelns kontur tätas eventuella sprickor kring tunneln.

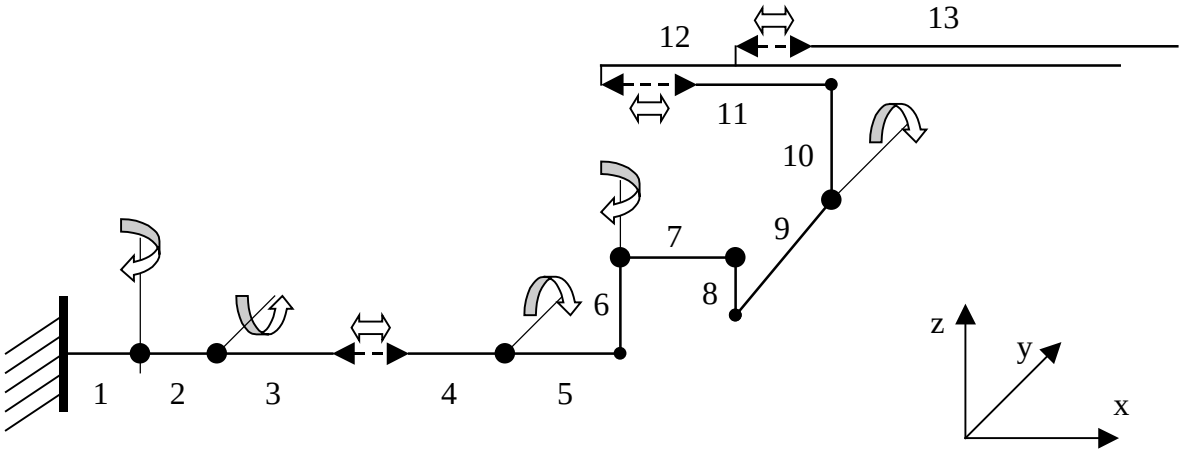
2.2 Borriggar

Vid tunneldrivningen används borriggar av varierande storlek, beroende på den täckarea som krävs för tunnelns kontur. För tunnlar med liten tvärsnittsarea används enboms-borriggar med en täckarea på 6–70 m^2. För tunnlar med stor area används treboms-riggar med en täckarea på upp till 165 m^2.



Figur 2.2: Den nya generationens borriggar. Här utrustad med två bommar och en serviceplattform.

Figur 2.2 ovan visar en medelstor borrigg utrustad med två bommar och en serviceplattform. Täckarean för en sådan borrigg är maximalt 90 m^2. Borriggen är utrustad med en serviceplattform för att operatören ska kunna inspektera de hål som borrats och att ladda hålen med sprängmedel. Då serviceplattformen är tänkt att manövreras direkt av operatören från plattformen, saknar den givare för positionsbestämning. Operatören måste därför se till att plattformen är flyttad ur vägen för bommarna innan borrningen börjar.



Figur 2.3: Trådmodell som beskriver en boms geometri och dess länkar.

Varje led på bommarna har varsin givare för mätning av vinkel eller längd. Vinklarna i de vridbara lederna kan bestämmas till en noggrannhet av en hundradels grad och längder i teleskoplederna mäts i millimeter. För att åskådliggöra en boms olika frihetsgrader finns en trådmodell framtagen, som illustreras i Figur 2.3. Modellen används i riggens styrprogramvara och kan

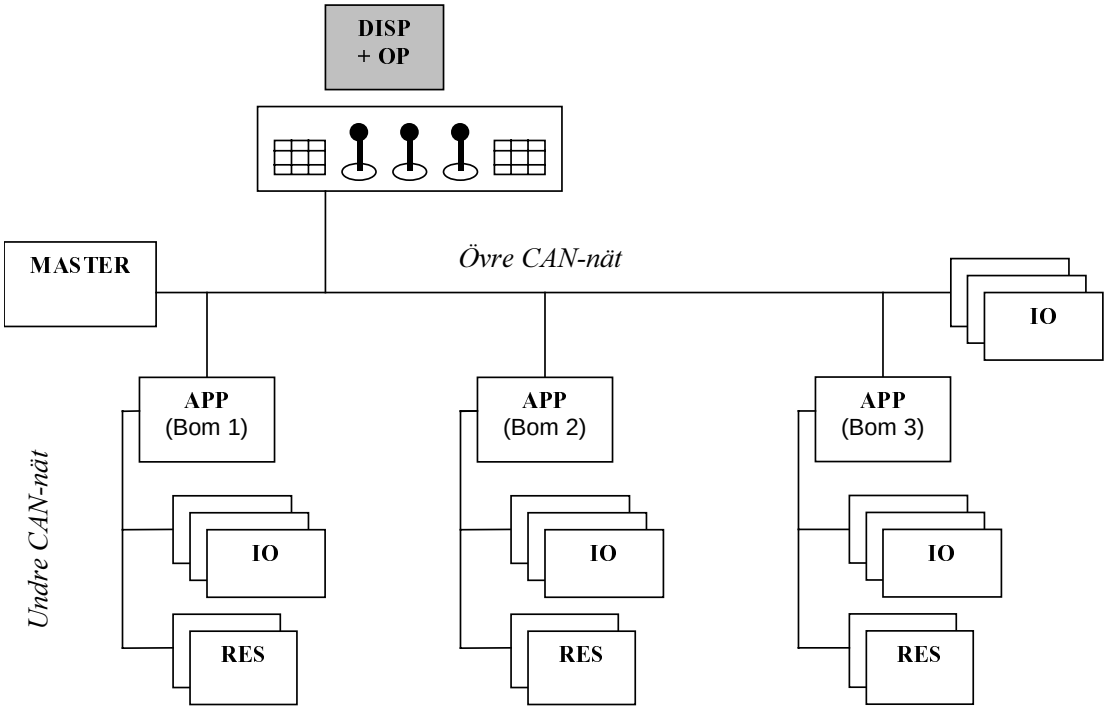
konfigureras vid systemstart med de längder som gäller för bommarnas länkar hos den aktuella borriggen.

2.3 Styrsystem

I den nya generationen borriggar utgörs styrsystemet av ett distribuerat system uppbyggt av ett antal ”intelligenta enheter” sammankopplade i ett CAN-nät[1], som illustreras i blockschemat i Figur 2.4. Systemet består av ett övre nät och ett undre nät för varje bom på borriggen.

Det övre nätet har till uppgift att vidareförmedla operatörens önskningar via spakar och knappar på operatörspanelen (**OP**), samt samla ihop data från de undre näten och presentera data på en display (**DISP**). Det finns även en överordnad modul (**MASTER**) som samordnar rörelser mellan bommarna. Detta sker genom att kommunicera med applikationsmodulerna (**APP**) för respektive bom. Till det övre nätet finns även ett antal enheter för in- och utmatning av data till och från riggen (**IO**).

Det undre nätet har till uppgift att sköta den kommunikation som endast har med en bom att göra. **APP** är ansluten till både det övre och det undre nätet för att kunna vidarebefordra längd- och vinkeldata från resolvermoduler[2] (**RES**) till **DISP** och **MASTER**. **APP** har även till uppgift att beordra utstyrningssignaler till **IO** och sköta reglering av bommens rörelser.

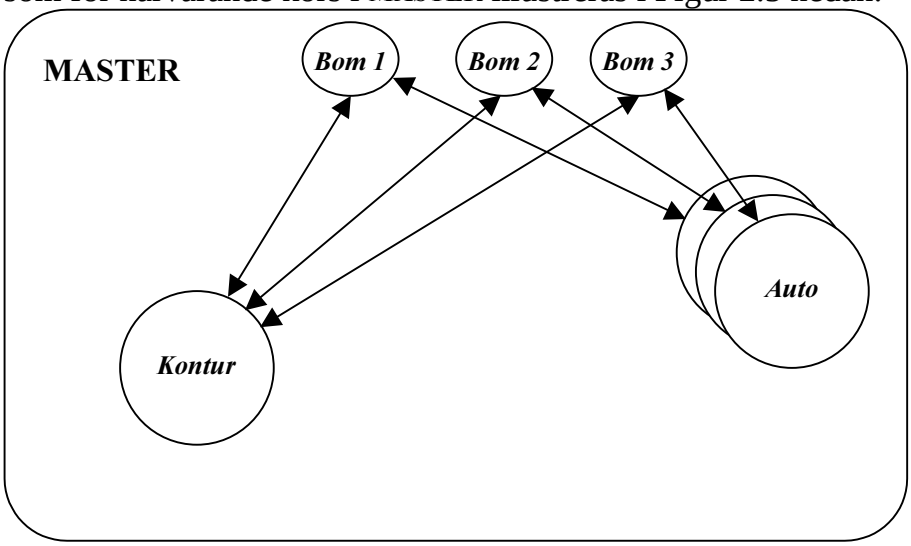


Figur 2.4: Blockschema över styrsystem till Boomer XL3C.

[1] CAN–Controller Area Network–är en standard för att skicka meddelanden mellan intelligenta enheter.

[2] Resolvermodulen läser av en fysisk givare och skickar givarvärdet på CAN-nätet.

I **MASTER** sitter en processor som kör RTKernel, ett operativsystem för realtidsapplikationer. Applikationerna i **MASTER**-modulen körs i form av olika processer som fördelar processorns beräkningskapacitet mellan sig. De processer som för närvarande körs i **MASTER** illustreras i Figur 2.5 nedan.



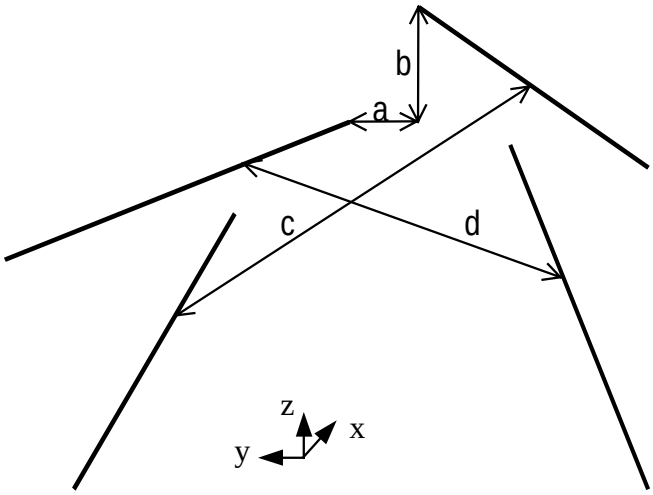
Figur 2.5: Befintliga processer i mastermodulen.

I mastermodulen körs *Kontur*, en process som hjälper operatören att styra en boms riktning och läge vid manuell positionering av borrmaskinen, genom att på skärmen rita ut den aktuella borrplanens hål samt hålens riktningar tillsammans med borrmaskinens riktning. De tre instanserna av *Auto* beräknar s.k. flyttorder för respektive bom givet borrplanen, efter det att operatören har aktiverat automatisk borrning och positionering för en eller flera bommar. För varje hålposition tar *Auto* hänsyn till eventuell risk för kollision med de övriga två bommarna. Till detta tar den hjälp av en inbyggd strategi för kollisionsdetektering, som beskrivs närmare i avsnitt 2.4.

Bom 1, 2 och 3 i Figur 2.5 är speglingar av motsvarande process i respektive **APP**-modul (se även Figur 2.4). Bomprocesserna i **APP** och **MASTER** kommunicerar med varandra över CAN-nätet och skickar med jämna mellanrum färska givarvärden till de processer som behöver uppdateras med färska data. Utöver processerna i Figur 2.5 körs även processer som sköter kommunikationen med CAN-nätet och andra processer som är mer administrativa till karaktären.

2.4 Kollisionsdetektering

På tidigare generationers borriggar detekteras kollisionsrisken mellan bommarna med hjälp av en förenklad vektormodell enligt figuren nedan. Strategin formulerades 1984 av Ingvar Carlvik och finns noggrant beskriven i [Carlvik, 1984]. Den förenklade vektormodellen togs fram för kollisionsdetektering när motsvarande borriggars styrsystem var centraliserat, processorkraft var dyrbart och höga krav ställdes på enkla och beräkningssnåla modeller.



Figur 2.6:Förenklad vektormodell av två bommar.

I vektormodellen ovan kallas de undre vektorerna *bomkropp*, vilket motsvarar länk 1–6 i trådmodellen i Figur 2.3 (se sid. 9) och utgör den kraftigaste delen av bommen som sitter monterad i borriggen. De övre vektorerna kallas *matare* och motsvarar länk 11–13 i trådmodellen i Figur 2.3. Mataren är den del av bommen där borrmaskin och borrstål sitter monterat i en slädkonstruktion för att kunna matas fram och tillbaka vid borrning.

För att en beräknad förflyttning ska verkställas vid automatisk positionering av bommarna, måste följande villkor vara uppfyllda:

1) Avståndet mellan bomspetsarna måste överstiga ett givet värde i y- resp. z-led. (se a och b i Figur 2.6.)
2) Minsta avståndet mellan en boms matare och en annan boms bomkropp (c och d i Figur 2.6.) måste vara större än ett givet säkerhetsavstånd.

Om villkoren inte uppfylls larmar automatiken och överlåter till operatören att ta över förflyttningen av bommarna, tills de befinner sig på ett säkert avstånd från varandra och villkoren ovan uppfylls i styrprogramvaran.

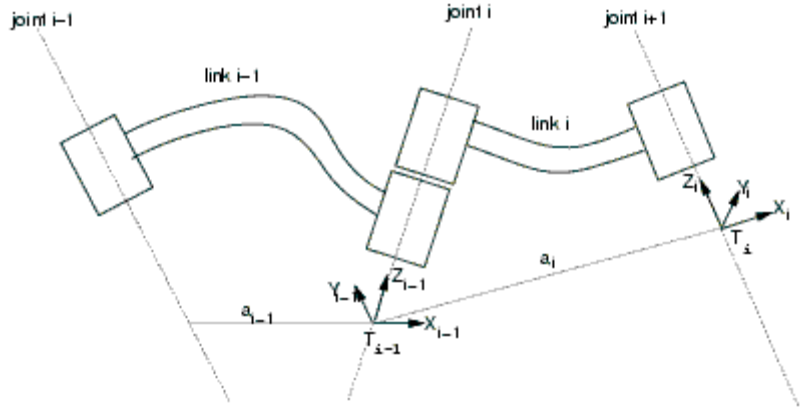
3 Teoretisk bakgrund

I det här avsnittet görs en genomgång av den teori som ligger till grund för examensarbetet. Först behandlas hur en robotarms positioner kan beräknas med matematiska verktyg. Därefter beskrivs hur objekt modelleras i tre dimensioner, varefter några metoder att utföra kollisionskontroll mellan objekt presenteras.

3.1 Kinematik

Det första problemet som måste lösas när en robotarm av något slag används är att bestämma riktning och position för armens *verktyg*, d.v.s. den del av robotarmen som utför en uppgift i sin omgivning. I problemet ingår även att räkna ut *var* armen som håller i verktyget befinner sig. Denavit & Hartenberg har formulerat regler för hur en robotarms geometri beskrivs med fyra parametrar på ett systematiskt sätt utifrån armens fysiska leder. Denavit-Hartenbergs *DH-parametrar* är vanligt förekommande i dessa sammanhang och beskrivs bl.a. i [Spong, 1989] och [Stadler, 1995].

En robotarms leder indelas i två kategorier, *prismatiska* leder och *vridbara* leder. Prismatiska leder förskjuter armen i sid- eller längdled och vridbara leder roterar armen kring en punkt. Varje oberoende led (prismatisk eller vridbar) tillför en *frihetsgrad* till armen och ju fler frihetsgrader desto större rörlighet hos robotarmen. För att nå en godtycklig punkt i rummet inom armens räckvidd och med en godtycklig riktning hos verktyget, krävs sex frihetsgrader; tre för att nå punkten och ytterligare tre för att ställa in riktningen. En arm med fler än sex frihetsgrader kallas *kinematiskt redundant*.



Figur 3.1: Denavit-Hartenbergs standardnotation[3].

Denavit-Hartenbergs regler innebär att man betraktar armens frihetsgrader en åt gången och sätter upp en tabell innehållande både den storhet som är variabel för leden och de parametrar som anger ledernas inbördes avstånd. För varje länk hos armen identifieras följande storheter, med beteckningar från Figur 3.1 ovan:

[3] Bilden hämtad ur [Sciavicco, 1996].

- *länklängd*: Avståndet mellan z_{i-1} och z_i längs x_i.
- *länkvridning*: Vinkeln mellan z_{i-1} och z_i kring x_i.
- *länkförskjutning*: Avståndet från ledorigo T_{i-1} till x_i längs z_{i-1}.
- *ledvinkel*: Vinkeln mellan x_{i-1} och x_i kring z_{i-1}.

En tabell över ovanstående storheter för varje länk beskriver en robotarms möjliga rörelser och positioner. Tabellen används för att ta fram matriser för en arms leder och därigenom kunna beräkna en förflyttning mellan ett godtyckligt par av leder. Matriserna utgör en homogen avbildning[4] om de utformas på följande sätt:

$$H_i = \begin{bmatrix} R_i & t_i \\ 0 & 1 \end{bmatrix} \qquad (3.1)$$

I ekvation 3.1 ovan är R_i en 3x3-matris som utgör en rotation kring någon axel i ett fixt koordinatsystem *[x,y,z]* och t_i är en 3x1-vektor som utgör en förflyttning i x-, y- och z-led.

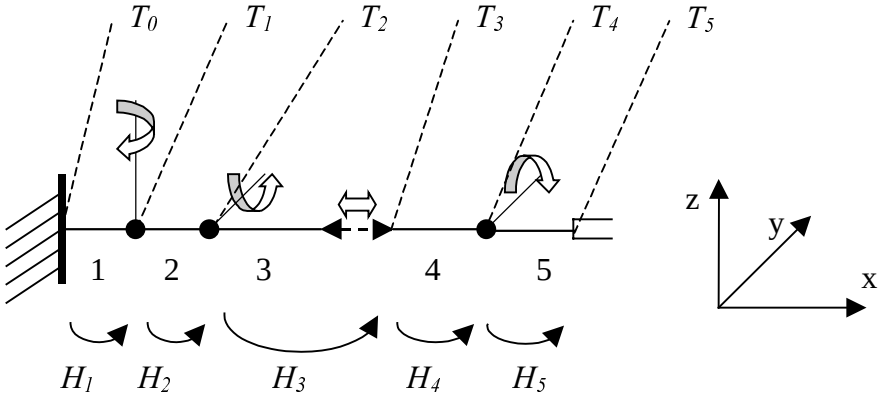
Om en arm har *n* länkar kan dessa numreras från 1 till *n*, där länk *1* utgår från armens infästning och länk *n* är verktyget. Då erhålls verktygets riktning och position genom att beräkna

$$T_n = \prod_{i=1}^{n} H_i \qquad (3.2)$$

Efter matrismultiplikationen i ekvation 3.2 ovan har T_n samma form som H_i , d.v.s. verktygets riktning relativt bommens origo ges av T_n:s tre första kolumner och rader, medan en vektor från bommens origo till verktygets lokala origo (T_n), ges av de tre första raderna i T_n:s fjärde kolumn.

Notera att H_i inte är konstant, utan varierar med den storhet som är variabel för led *i*. I de fall led *i* är prismatisk är R_i konstant och t_i varierar med längdförskjutningen. För en vridbar led gäller på motsvarande sätt att t_i är konstant och R_i varierar med ledvinkeln.

[4] En homogen avbildning innebär att längder och vinklar bevaras under avbildningen.



Figur 3.2: Robotarm bestående av fem länkar.

Figur 3.2 illustrerar en arm bestående av fem länkar. Vid armens infästning markeras dess origo med T_0. Varje led som tillför armen en frihetsgrad tilldelas ett lokalt origo, T_1–T_4. För att enkelt kunna beräkna verktygets riktning och position, tilldelas även verktygets *angreppspunkt* ett origo T_5. Transformationer mellan armens leder (och till angreppspunkten) sker med de fem *H*-matriserna, H_1–H_5, som erhålls med Denavit-Hartenbergs regler och beskrivs ovan. För att exempelvis beräkna hur länk 4 är placerad i rymden utförs matrismultiplikationen $T_3 = H_1 \cdot H_2 \cdot H_3$.

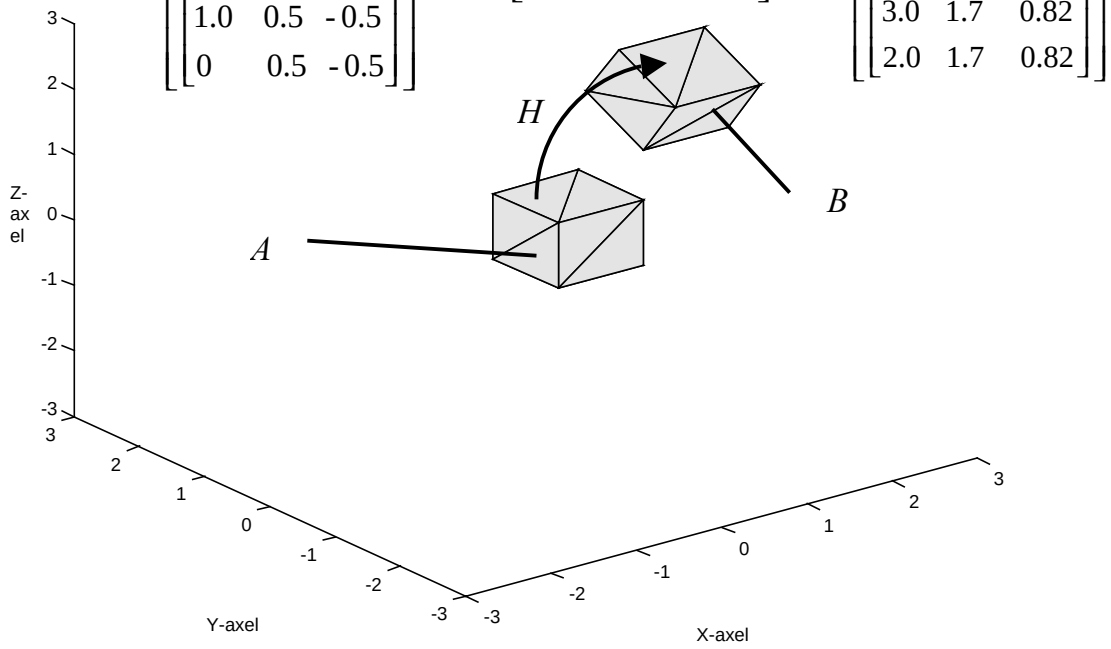
3.2 Modellering av 3D-objekt

Den vanligaste metoden att modellera grafiska objekt är att använda *polygoner*, d.v.s. linjer sammansatta till geometriska figurer, t.ex. trianglar. Polygonerna (trianglarna) används sedan som byggstenar för att forma större objekt.

Genom att bygga objekt med trianglar och spara trianglarnas hörn i en matris, kan en enkel matrismultiplikation transformera objektens rotation och position i rymden med hjälp av den matris som beskrivs i ekvation 3.1 ovan.

Figur 3.3 nedan illustrerar hur ett objekt uppbyggt av 12 trianglar formar en kub. Med matrisen *H* transformeras kubens koordinater från *A* till *B*. *H*-matrisen motsvarar en rotation kring x-axeln med 30 grader och en förflyttning av kuben i x-, y- och z-led.

$$A = \begin{bmatrix} \begin{bmatrix} 0 & -0.5 & -0.5 \\ 1.0 & -0.5 & -0.5 \\ 1.0 & 0.5 & -0.5 \end{bmatrix} \\ \vdots \\ \begin{bmatrix} 0 & -0.5 & -0.5 \\ 1.0 & 0.5 & -0.5 \\ 0 & 0.5 & -0.5 \end{bmatrix} \end{bmatrix} \quad H = \begin{bmatrix} 1.0 & 0 & 0 & 2.0 \\ 0 & 0.87 & -0.5 & 1.0 \\ 0 & 0.5 & 0.87 & 1.0 \\ 0 & 0 & 0 & 1.0 \end{bmatrix} \quad B = \begin{bmatrix} \begin{bmatrix} 2.0 & 0.82 & 0.32 \\ 3.0 & 0.82 & 0.32 \\ 3.0 & 1.7 & 0.82 \end{bmatrix} \\ \vdots \\ \begin{bmatrix} 2.0 & 0.82 & 0.32 \\ 3.0 & 1.7 & 0.82 \\ 2.0 & 1.7 & 0.82 \end{bmatrix} \end{bmatrix}$$



Figur 3.3: Transformation av trianglarna i A till B med matrisen H.

Matrisen *A*, som representerar trianglarna i kuben, innehåller 12 st. 3x3-matriser, vars koordinaterna [x,y,z] för hörnen i respektive triangel. För varje rad p_A i *A* ska transformationen $p_B = R \cdot p_A + t$ beräknas, där *R* och *t* är den rotation och translation som beskrivs i ekvation 3.1. Transformationen åstadkoms genom att utöka *A* med en fjärde kolumn av ettor (A_1) och därefter utföra matrismultiplikationen nedan.

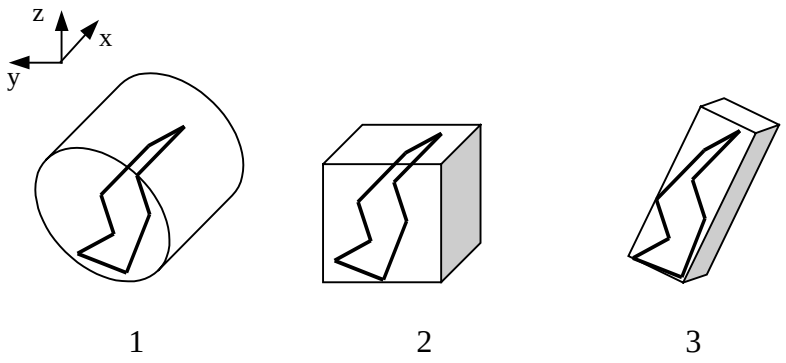
$$H \cdot A_1^T = \begin{bmatrix} 1 & 0 & 0 & 2.0 \\ 0 & 0.87 & -0.5 & 1.0 \\ 0 & 0.5 & 0.87 & 1.0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 0 & 1.0 & 1.0 & & 0 & 1.0 & 0 \\ -0.5 & -0.5 & 0.5 & \dots & -0.5 & 0.5 & 0.5 \\ -0.5 & -0.5 & -0.5 & & -0.5 & -0.5 & -0.5 \\ 1 & 1 & 1 & & 1 & 1 & 1 \end{bmatrix} = \cdots$$

$$\cdots = \begin{bmatrix} 2.0 & 3.0 & 3.0 & & 2.0 & 3.0 & 2.0 \\ 0.82 & 0.82 & 1.7 & \dots & 0.82 & 1.7 & 1.7 \\ 0.32 & 0.32 & 0.82 & & 0.32 & 0.82 & 0.82 \\ 1 & 1 & 1 & & 1 & 1 & 1 \end{bmatrix} = B_1^T$$

I den resulterande matrisen, B_1^T, stryks den fjärde raden (som alltid innehåller ettor) varefter matrisen transponeras till den slutgiltiga matris *B* som innehåller de nya koordinaterna för kubens trianglar.

3.3 Kollisionsdetektering mellan objekt i 3D

Kollisionsdetektering implementeras ofta med hjälp av *omslutande volymer* som begränsar utsträckningen av de aktuella objekten. Sådana omslutande volymer kan vara sfäriska, cylindriska eller rektangulära (se exempel nedan i Figur 3.4). Volymernas syfte är att ge garantier för att objekten inte löper risk för kollision. Om två omslutande volymer kolliderar, är det nödvändigt att göra en ytterligare kontroll av de omslutna objekten för att undersöka om en kollision äger rum eller inte.



Figur 3.4: Tre metoder att omsluta objekt i tre dimensioner för kollisionskontroll.

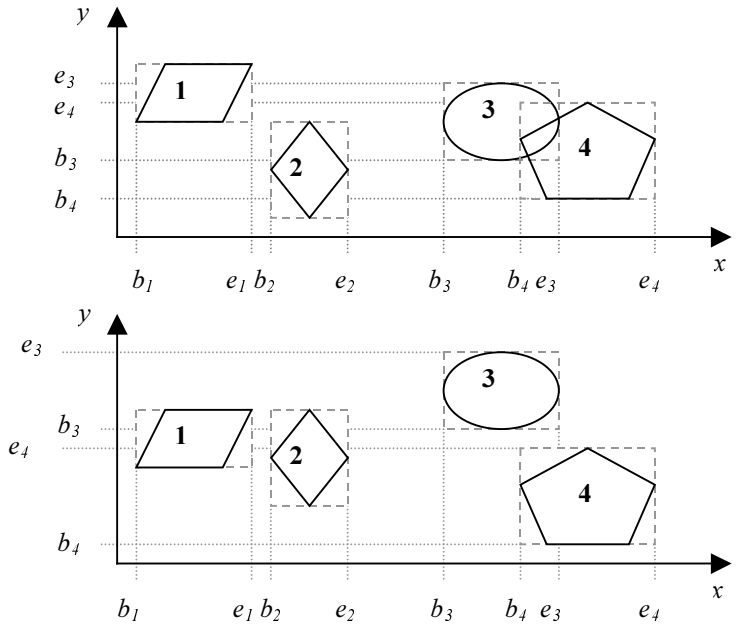
Figur 3.4 illustrerar olika sätt att kapsla in ett objekt i en omslutande volym för att detektera risk för kollision mellan olika objekt. Den *första* metoden i figuren implementeras ofta genom att definiera ett centrum med en längdriktning i objektet och sätta ett minsta tillåtet avstånd kring dess centrumaxel. Risk för kollision detekteras då av att ett annat objekt befinner sig närmare än det tillåtna avståndet.

Den *andra* metoden är en variant där den omslutande volymen har samma orientering som koordinatsystemets referensaxlar (eng. *axis aligned bounding box* [Lin, 1997]). Kollisionsdetektering görs genom att jämföra koordinaterna för volymens hörn med närliggande volymers koordinater, vilket är detsamma som en projicering av volymerna på x-, y- respektive z-axeln. Kollisionsrisk mellan två objekt inträffar då deras volymers projektioner överlappar varandra i alla tre dimensioner, i annat fall föreligger ingen risk för kollision.

Den *tredje* metoden är en vidareutveckling av metod 2, där den omslutande volymen intar samma riktning som objektet (eng. *oriented bounding box* [Lin, 1997]). Även här projiceras den omslutande volymen på en axel, men nödvändigtvis inte en koordinataxel. Om en axel finns, där två volymers projektioner på axeln inte överlappar varandra, kallas den för *separerande axel*[5],

[5] Fri översättning av *seperating axis* som används i [Lin, 1997].

vilket medför att det inte finns någon risk för kollision mellan de båda objekten. I [Gottschalk, 1996] beskriver författaren teorier bakom riktade, omslutande volymer, samt en algoritm för att beräkna överlappning mellan volymerna. Han gör även en jämförelse mellan riktade, omslutande volymer och andra typer av volymer, såsom metod 1 och 2.



Figur 3.5: Projektion av omslutande volym.

I Figur 3.5 illustreras två fall av projektion av omslutande volymer på axlar i två dimensioner. I figuren används *metod två*, d.v.s. de omslutande volymerna är riktade efter koordinatsystemets axlar. Båda bilderna i figuren visar samma resultat på x-axeln: två av projektionernas intervall överlappar (intervallen b_3–e_3 och b_4–e_4), vilket kan tyda på kollision mellan två objekt. Men projektionen av samma objekt på y-axeln visar att objekten 3 och 4 endast kolliderar i den övre bilden, där projektionerna av de båda volymerna överlappar i både x- och y-led. I den undre bilden kolliderar inget objekt med något annat, eftersom inga omslutande volymers projektioner överlappar i både x- och y-led.

4 Mjukvara för kollisionskontroll

Många grupper på universitet världen över bedriver forskning på algoritmer för datorgrafik och simulering. I detta avsnitt görs en presentation av några algoritmer som bygger på resultat av forskningsarbete kring kollisionsdetektering. Algoritmerna finns publicerade på Internet och får användas fritt i studiesyfte. För att använda dem i kommersiella produkter krävs dock någon form av licens eller tillåtelse från upphovsmännen.

De algoritmer som valts ut i presentationen nedan ger en uppfattning om vilka angreppssätt som använts på problemet med att detektera kollision mellan simulerade objekt. En översikt över nedanstående och andra liknande system görs i [Lin, 1997]. I presentationen nedan ligger tyngdpunkten på RAPID i avsnitt 4.3, som visade sig vara det bäst lämpade systemet för vårt problem.

4.1 Gjk

Paketet GJK bygger på algoritmer för att beräkna avstånd mellan par av konvexa objekt/månghörningar[6]. Algoritmerna utvecklades ursprungligen av Gilbert, Johnson och Keerthi (GJK) vid Oxfords universitet. Stephen Cameron vid samma universitet, har senare implementerat algoritmerna i ett paket i C där en användare själv definierar geometriska objekt, uppbyggda av en mängd av hörn och kanter. Användaren definierar även sina egna primitiver för att traversera objekten, d.v.s. stega igenom objektens hörn och kanter i en viss ordning med funktioner som FORALLvertices etc. I [Cameron, 1997] och [Cameron, 1998] beskrivs användningsområde och funktionalitet hos GJK.

Kommentar: GJK är i och för sig ett litet och ganska användbart paket, men i vår tillämpning är avstånd mellan objekt inte så intressant. Dessutom har paketet en låg abstraktionsnivå vad gäller objekten, vilket gör paketet komplicerat att utnyttja.

4.2 I-Collide

I-COLLIDE är ett lite större programpaket som låter användaren definiera en objektsrymd med ett stort antal konvexa objekt. Objektens positioner och riktningar sätts av användaren, som sedan kan göra ändringar av ett eller flera objekts placering i rymden. Systemet håller reda på alla definierade objekt och uppdaterar automatiskt en lista med kolliderande objekt. Användaren har också möjlighet att få beräknade avstånd mellan objekt som anses intressanta. Paketet innehåller även stöd för funktioner i grafikbiblioteket OpenGL, som ofta används vid visualisering av datorsimuleringar. En komplett beskrivning av I-COLLIDEs funktioner och finesser ges i [Cohen, 1997].

[6] För konvexa objekt gäller att varje rät linje mellan två av objektets hörn, går igenom objektet eller ligger på ytan av detsamma.

Kommentar: Med tanke på att en modell av en borrigg aldrig kommer att bli så stor, verkar I-COLLIDE vara ett onödigt stort och därmed minneskrävande programpaket. Dess funktioner kommer inte till sin fulla rätt i en så pass liten miljö med ett litet antal objekt i rymden.

4.3 Rapid

RAPID är en förkortning av Rapid and Accurate Polygon Interference Detection och är ett programpaket avsett att användas för att detektera kollision mellan objekt vid datorsimulering och visualisering av objekt i tre dimensioner. RAPID är utvecklat och implementerat av forskare och doktorander vid University of North Carolina (UNC). Kollisionskontrollen i paketet använder tekniken med riktade, omslutande lådor, som är utvecklad av samma forskare.

Hela paketet är implementerat i C++ och bygger på en klass, `RAPID_model`, som byggs upp av ett valfritt antal trianglar. Trianglarna har ingen inbördes ordning eller placering, utan ligger i modellen som en s.k. ”polygonsoppa”[7].

Kommentar: RAPID som helhet är mycket tilltalande, dels på grund av att det är relativt litet och enkelt att använda och dels för att det ligger till grund för andra paket för kollisionsdetektering. Det används bl.a. av I-COLLIDE som beskrivits ovan, vilket betyder att det är flitigt använt och därför borde vara någorlunda befriat från fel. Paketet är effektivt enligt [Gottschalk, 1996] där forskarna vid UNC låtit olika paket göra samma kollisionsdetektering och jämfört prestanda dem emellan. Nedan beskrivs mer ingående hur RAPID används.

Gränssnittet mot användaren/programmeraren består av ett fåtal metoder hos klassen `RAPID_model`:

- `RAPID_model()`–används för att skapa nya objekt av klassen.
- `BeginModel()`–initierar modellbygge.
- `AddTri(p1, p2, p3, id)`—lägger till en triangel i modellen, där triangelns tre hörn uttrycks med vektorerna `p1`, `p2` och `p3`. Den sista parametern, `id`, är ett unikt tal för den aktuella triangeln.
- `EndModel()`–slutför uppbyggnaden av modellen.
- `~RAPID_model()`–används för att destruera objekt av klassen.

Därutöver finns en metod för att kontrollera om ett par av objekt kolliderar:

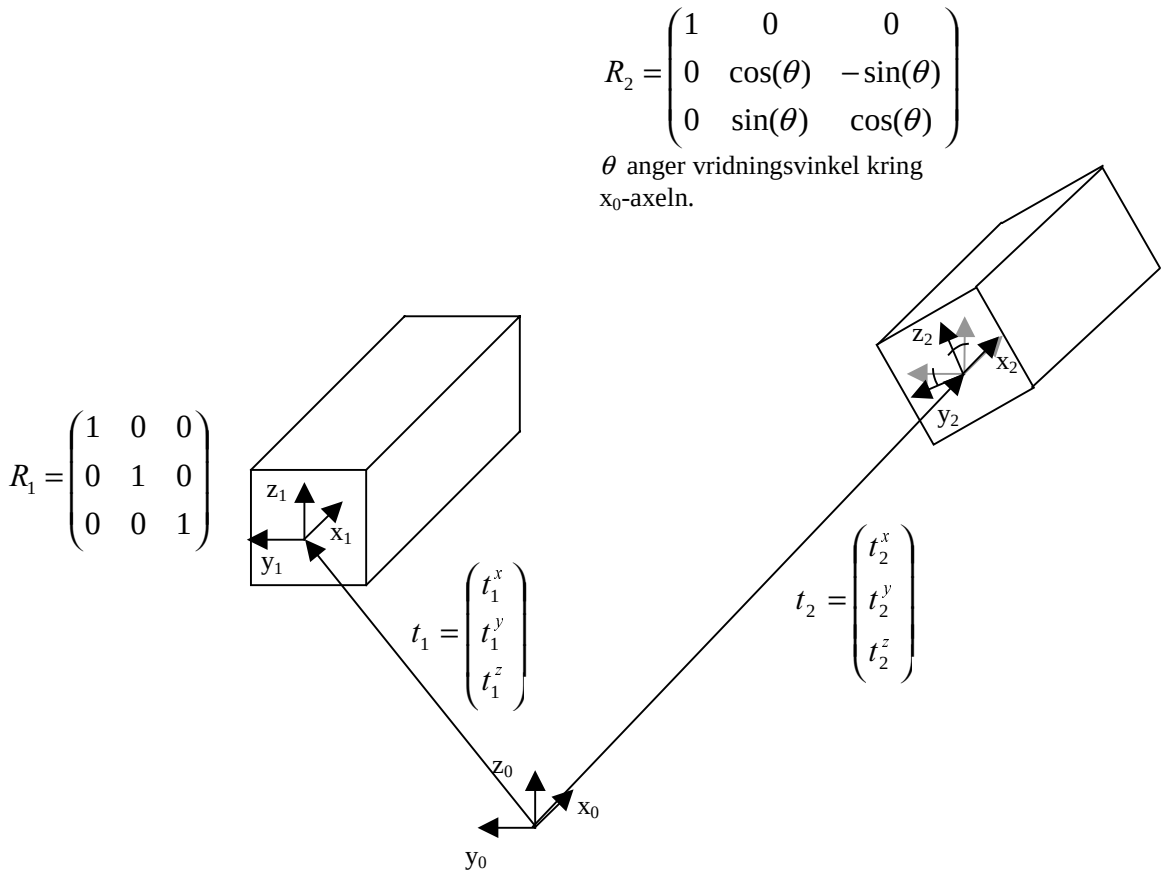
```
RAPID_Collide(R1[3][3], T1[3], s1, model1,
              R2[3][3], T2[3], s2, model2, flag)
```

Argumenten `model1` och `model2` är pekare till de två objekt som ska kollisionstestas. `R1` och `R2` är två ortogonala, normerade rotationsmatriser[8], där `R1` anger rotationen för `model1` och `R2` för `model2`.

[7] Fri översättning av det engelska begreppet *polygon soup* för en oordnad mängd av polygoner.

På motsvarande sätt anger `T1` och `T2` respektive modells position i rummet, uttryckt i ett gemensamt koordinatsystem för alla objekt i rymden. Parametrarna `s1` och `s2` ger möjlighet att förstora/förminska modellerna med en faktor större än 0. Den sista parametern, `flag`, anger om *alla* eventuella kollisioner mellan objekten ska undersökas, eller om funktionen ska avbryta testet så fort den första kollisionen påträffas, vilket ger betydligt kortare exekveringstid.

Returvärden från `RAPID_Collide` ges i två globala variabler; `RAPID_num_contact` som anger antal kollisioner mellan de testade objekten och `RAPID_contact` som är en lista med par av `id`:n för kolliderande trianglar.



Figur 4.1: Två volymer relaterade till ett gemensamt origo.

Figur 4.1 illustrerar hur rotationsmatrisen *R* och translationsvektorn *t* för en låda används i RAPID. *R* beskriver hur lådans lokala origo är roterat relativt ett globalt origo och *t* är en vektor som anger var lådans origo är placerat i rymden. I

[8] För en rotationsmatris *R* måste följande gälla: $R \cdot R^t = E$, där R^t är transponatet till *R* och *E* är enhetsmatrisen.

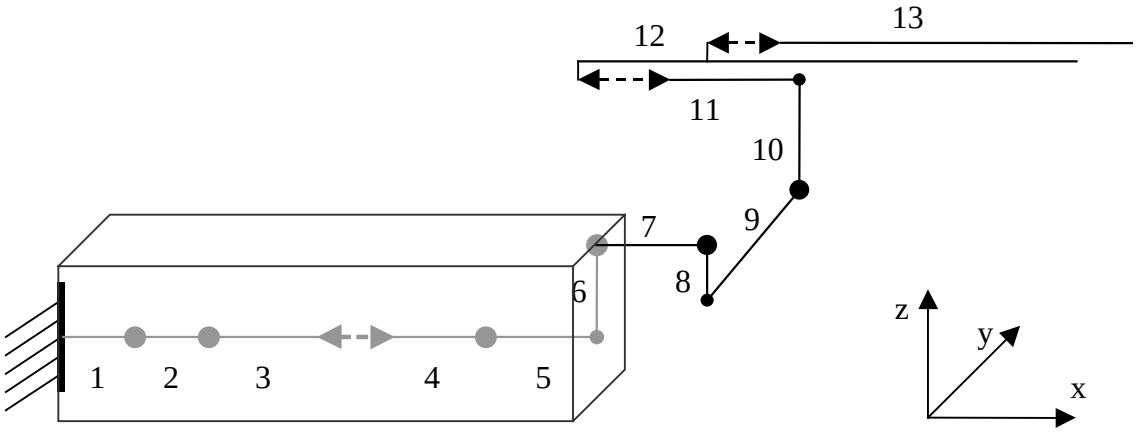
vår tillämpning beräknas R och t med hjälp av den kinematiska beskrivning av riggens bommar som finns i styrprogramvaran.

5 Lösning

I avsnittet nedan presenteras en lösning till problemet med kollisionskontroll mellan bommarna på *Boomer*. Först ges en inblick i hur implementeringen genomförts. Därefter presenteras en utvärdering av kollisionskontrollens prestanda i målmiljön, följt av en diskussion om lösningens utbyggbarhet. Sist ges exempel på alternativa lösningar till problemet.

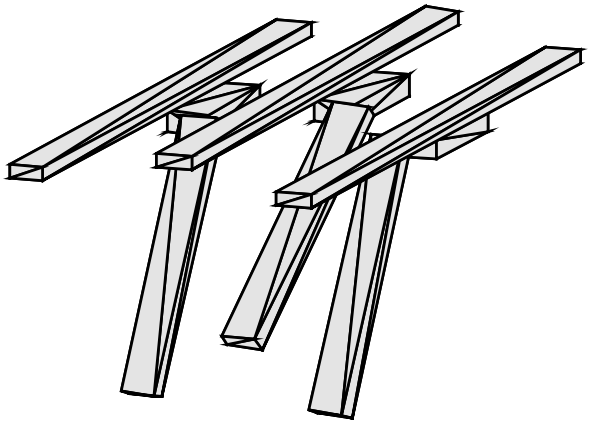
5.1 Implementering

Den nya strategi för kollisionskontroll som tagits fram bygger på teorier och idéer från simulering och modellering inom datorgrafik. I princip bygger strategin på att bommens delar innesluts i ett antal omslutande volymer och kapslar in utstickande detaljer i en säkerhetszon för att undvika kollisioner mellan bommarna. De omslutande volymerna har utformats som *lådor* kring den trådmodell som finns tillgänglig i styrsystemets programvara och som innehåller information om länklängder hos den aktuella riggens bommar. Med hjälp av trådmodellen kan programvaran bygga en polygonmodell av riggens bommar, bestående av trianglar i RAPID. Varje låda motsvarar ett objekt i C++ som dels håller reda på lådans riktning och placering i rymden och dels innehåller ett RAPID-objekt uppbyggt av tolv trianglar på samma sätt som beskrivs i avsnitt 3.2, *Modellering av 3D-objekt*, på sidan 15. I bilagorna 8.1 sist i rapporten redovisas klassdiagram som visar hur kollisionskontrollen implementerats i praktiken.



Figur 5.1: Bomkroppen omsluten av en låda.

Figur 5.1 ovan visar hur bomkroppen i trådmodellen omsluts av en låda. Den totala modellen av bommarna består av tre lådor per bom och Figur 5.2 illustrerar de totalt nio lådor som används för kollisionskontroll.



Figur 5.2: Polygonmodell med tre bommar.

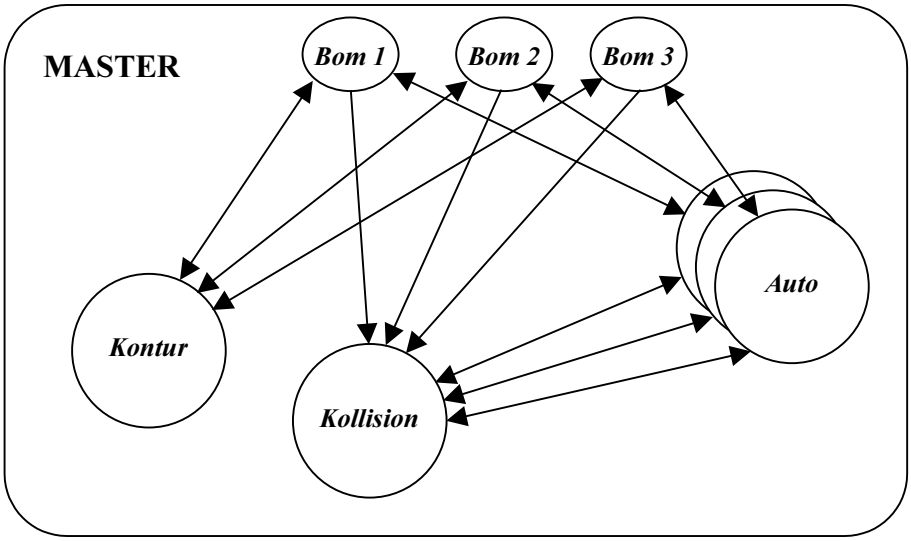
För att testa och utvärdera modellen av bommarna och lättare förstå kollisionskontrollen, kopplades först RAPID ihop med Matlab, där modellen kan ritas upp grafiskt och åskådliggöras på ett enkelt sätt. Matlab har även inbyggt de matrismultiplikationer som krävs för att transformera de omslutande volymernas koordinater i enlighet med aktuella värden för bommarnas ledvariabler. I Matlab implementerades också ett grafiskt användargränssnitt för att kunna mata in värden då modellen och kollisionsdetekteringen i RAPID testades på en PC.

I målmiljön bröts sedan kopplingen till Matlab och de funktioner som implementerats där implementerades istället i C++. I målmiljön saknas dock både behov och möjlighet att visualisera objekten grafiskt, varför en hel del av Matlabs funktionalitet kunde försummas.

Den kärna som används för kollisionskontroll kan illustreras med följande pseudokod. De delar av koden som endast användes tillsammans med Matlab är *kursiverade*.

```
main()
   Skapa lådor med RAPID-objekt;
   While (1)
      Hämta bomvariabler från Matlab;
      Uppdatera lådornas position (R & t);
      Gör kollisionskontroll mellan objekten;
      If (Antal kollisioner > 0)
        Skicka meddelande till berörda Auto-
        instanser;
      Skicka data till Matlab;
      Plotta modellen;
   End;
End.
```

I målmiljön exekveras kollisionskontrollen i en egen process i mastermodulen, vilket illustreras i Figur 5.3. Detta är en flexibel lösning som tillåter att kollisionskontrollen kan förfinas och uppdateras om behov skulle finnas i framtiden.



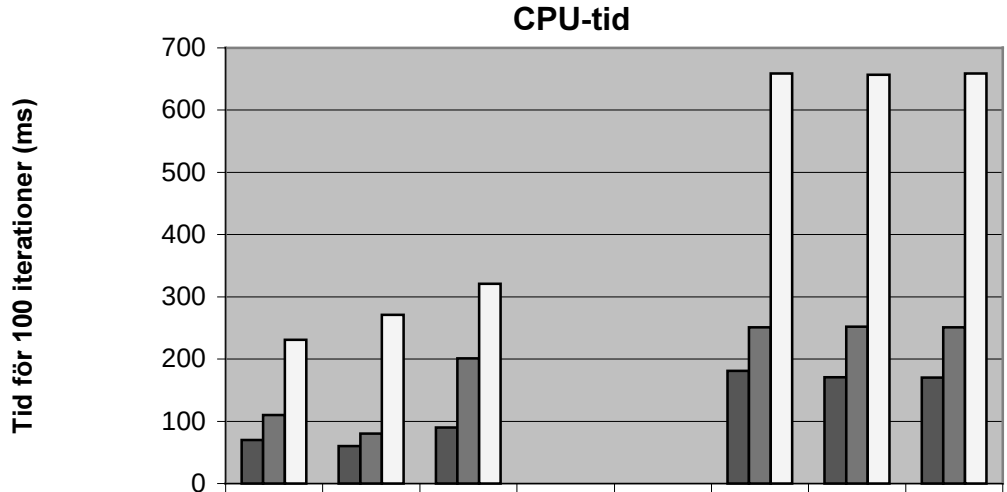
Figur 5.3: Processer i mastermodulen efter utökning med ny kollisionskontroll.

5.2 Utvärdering

En stor del av utvärderingen av den nya strategin är att utreda om processorkraften i **MASTER** är tillräcklig för att använda RAPID. I utvärderingen av prestanda hos modulen är det en viktig uppgift att analysera om modellen kan utökas eller om en enklare strategi bör tas fram. Därför gjordes tidtagningar för att ta reda på om den nya kollisionskontrollens ”varvtider” var tillräckligt korta för att krav på realtidsegenskaper uppfylldes när systemet kördes i målmiljön. Färdiga rutiner för att göra tidtagning och läsa av belastning på systemets processor fanns redan inbyggt i systemet. Tidsmätningarna utfördes både i målmiljön och i en simulerad miljö, i syfte att jämföra tidsegenskaper och belastning av processorn i de båda miljöerna.

För att kunna testa och felsöka det styrsystem som används på borriggarna, finns en simulator utvecklad för att enkelt kunna studera riggens olika moduler samt deras samverkan i systemet. Simulatorn körs i Windows-miljö på en PC och använder samma källkod för styrsystemet i målmiljön, för systemets alla moduler. En avgörande skillnad mellan de båda miljöerna är att systemets alla moduler simuleras i Windows-miljön, inklusive knappar på operatörspanelen. Processorn i målmiljön däremot har endast ett fåtal processer som behöver dela på processorns beräkningskapacitet, vilket är viktigt att ha i åtanke vid jämförelse av försökens utfall.

Diagram 5.1 nedan visar tidsåtgång för kollisionskontroll, dels i simulerad miljö i Windows NT (kolumn 1–3) och dels på ordinarie hårdvara i RTKernel (kolumn 6–8). I tabellen under diagrammet redovisas de tider som noterats vid tidtagning av kollisionskontroll för tre olika uppsättningar givarvärden hos bommarna.



	1	2	3	4	5	6	7	8
0 kollisioner	70	60	90			181	171	170
1 kollision	110	80	201			251	252	251
6 kollisioner	231	271	321			659	657	659

Diagram 5.1: Resultat från tidtagning av kollisionskontroll.

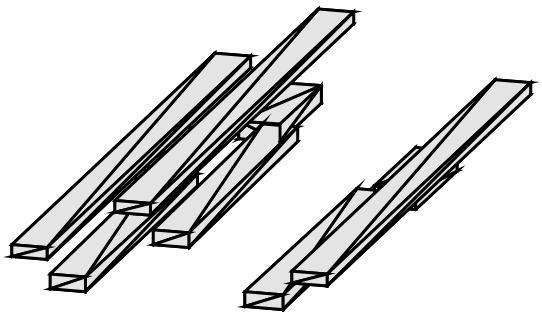
Tiderna i tabellen är uppmätta genom att använda en tidtagningsmekanism i styrsystemet. Varje tidsmätning är upprepad tre gånger för att försöka utpeka slumpmässighet eller omständigheter som påverkat tidtagningen. Vid varje försök utfördes parvis kollisionsdetektering av alla kombinationer av lådor i polygonmodellen, utom de lådor som är monterade intill varandra och *ska* kollidera. Kollisioner mellan intilliggande lådor undveks genom att helt enkelt utesluta anrop till `RAPID_Collide(…, box`*`i`*`, …, box`*`j`*`, …)` där `box`*`i`* och `box`*`j`* innesluter länkar som är monterade i varandra på den fysiska bommen.

I diagrammet syns en tydlig spridning mellan tiderna uppmätta i Windows NT, medan motsvarande tider i målmiljön är i stort sett konstanta. Spridningen beror på att exekveringen i Windows NT påverkas av interna processer som är högre prioriterade än vår simulator. Ett annat fenomen som är värt att notera är att förhållandet mellan tiderna i de olika miljöerna inte är linjärt. I fallet *utan* kollisioner är tiderna i målmiljön cirka *2.5* gånger så stora som de i Windows NT. Med *en* kollision är samma faktor knappt *2*.[9] Orsaken till dessa skillnader kan bero på hur bra koden är optimerad för respektive processor eller att den stora slumpmässighet som finns i beteendet hos Windows NT, exempelvis p.g.a. användningen av virtuellt minne, gör det omöjligt att bestämma en noggrannare faktor mellan de olika miljöerna.

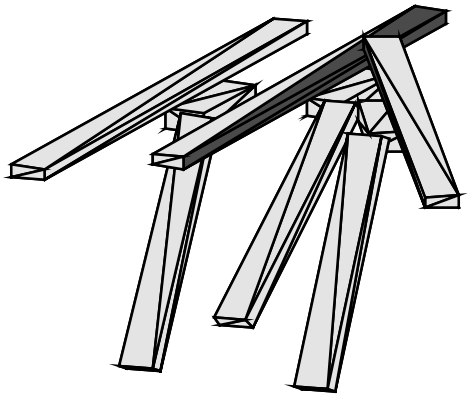
Tidtagningen är gjord under 100 upprepningar av respektive försök för att få större noggrannhet på den genomsnittliga tiden för varje iteration. Tidtagningsmekanismen i styrsystemet mäter nämligen tiden i hela

[9] (181+171+170) / (70+60+90) = 2,4 och (251+252+251) / (80+110+201) = 1,9

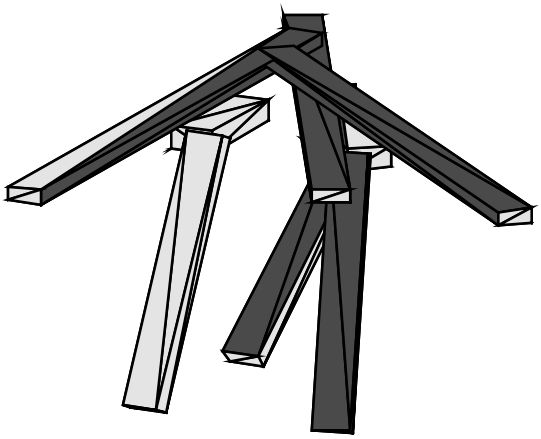
millisekunder, vilket är ett ganska grovt mått i detta sammanhang. De tre tidtagningarna är genomförda med tre olika uppsättningar *manuellt* inmatade givarvärden, vilka ger noll, en respektive sex kollisioner mellan lådorna i modellen. Uppsättningarna av givarvärden illustreras i Figur 5.4–Figur 5.6 nedan, där de trianglar som ingått i en kollision är markerade med mörkare färg.



Figur 5.4: Ingen kollision mellan lådorna i modellen.



Figur 5.5: En kollision mellan två lådor i modellen.



Figur 5.6: Sex kollisioner mellan lådorna i modellen.

Sammanfattningsvis kan sägas att ett test av kollision mellan borriggens bommar kommer att ta knappt två millisekunder i målmiljön, om ingen risk för kollision finns. I extrema fall kan svarstiden för kollisionstest vara upp emot sju millisekunder enligt Diagram 5.1. Dessa tider ligger med god marginal inom de krav som har satts för att godta en säker strategi för kollisionskontroll av borriggar. Den periodtid som används för att uppdatera givarvärden hos styrsystemet är längre än de tider som nämnts ovan, vilket betyder att kollisionskontrollen alltid kommer att rapportera kollisionsrisk för de mest aktuella värdena som finns tillgängliga i systemet.

5.3 Utbyggbarhet

För att kunna testa RAPIDs koncept inom tidsramen för ett examensarbete, är den nuvarande implementerade versionen av kollisionsdetekteringen anpassad för endast en bomstorlek. Med en tämligen liten arbetsinsats kan dock implementeringen utvidgas och generaliseras till alla förekommande bomtyper hos *Boomer*, då dess styrprogramvara innehåller en trådmodell av bommarna som konfigureras vid systemstart.

För att kollisionskontrollen ska fungera på ett betryggande sätt på en borrigg i drift, har storleken hos modellens lådor en avgörande betydelse. Därför krävs empiriska undersökningar för att ta reda på hur lång tid det tar för en bom att stanna efter det att kollisionskontrollen har larmat om kollisionsrisk mellan två bommar i rörelse. Exempelvis kan för *små* lådor medföra att kollisionskontrollen inte larmar i tillräckligt god tid för att ett meddelande om kollisionsrisk ska hinna propagera genom systemet och få de båda bommarna att stanna. Av den anledningen har förberedelser gjorts i programvaran för att ge servicepersonal möjlighet att ställa in lådornas storlek via operatörspanelen. På så vis kan kollisionskontrollen anpassas till den aktuella borriggens övriga inställningar och egenskaper, som t.ex. max hastighet vid förflyttning av en bom i sidled.

Kollisionskontrollen är i huvudsak implementerad i en klass som kapslar in de data som behövs för att använda RAPID. Klassen är en generell klass för en låda

i modellen, där ett lokalt origo definieras i en av lådans kortsidor. Klassen kan ärvas för att lägga till nya metoder och attribut hos en låda. Till exempel är den generella klassen ärvd till en specialiserad klass för att modellera bomkroppen. Den specialiserade klassen behövde funktioner för att kunna ändra den omslutande lådans längd, eftersom bomkroppen innehåller en extension för förlängning av bommen. På motsvarande sätt kan andra specificerade lådor ärvas och ges speciella egenskaper som är nödvändiga för att modellera bommens delar på ett så bra sätt som möjligt.

5.4 Alternativa lösningar

Tidigare i rapporten har nämnts några andra system för kollisionsdetektering av objekt i tre dimensioner. Systemen erbjuder sina egna lösningar på hur modeller av objekt ska byggas för att kunna användas i systemet. I [Lin, 1997] ges en översikt över olika metoder att beskriva objekt. Bland dem kan nämnas implicita ytor, där ett objekt ges av en funktion $f : R^3 \mapsto R$, där de punkter *(x,y,z)* som ger *f(x,y,z) = 0* definierar objektets yta. Punkter som ger *f(x,y,z) > 0* ligger utanför objektet och punkter där *f(x,y,z) < 0* befinner sig inuti objektet. Att avgöra om en given punkt i rymden befinner sig i ett visst objekt är således väldigt enkelt, vilket är metodens största fördel.

Den stora nackdelen med implicit definierade objekt är dock att objektsrymden är svår och tidsödande att söka av, då alla möjliga punkter i rymden måste löpas igenom och kontrolleras så att de befinner sig i maximalt *ett* objekt. Annars föreligger en kollision mellan objekt i rymden. En sådan beskrivning av objekt ger heller inte samma intuitiva inblick i modellen som en polygonbeskrivning som t.ex. RAPID använder. En egenskap till polygonmodellers fördel är också att de kan visualiseras på ett enkelt sätt, då de flesta grafikbibliotek innehåller stöd för att rita polygoner på en skärm.

RAPID erbjuder således en intuitiv modell för bommen och en modell som går att förfina och bygga ut på ett enkelt och konkret sätt.

6 Sammanfattning och slutsatser

Den ursprungliga strategin för att lösa uppgiften i examensarbetet var att använda teori och metoder för liknande områden inom robotik och sedan applicera dessa på problemet. Vår ansats var att en modernare och mer verklighetsnära modell än den befintliga, förenklade vektormodellen, skulle förbättra kollisionskontrollen.

Många artiklar rörande problemet med kollisionsdetektering inom datorgrafik och simulering i tillämpningar som Virtual Reality (VR) finns att tillgå på bl.a. Internet. Inom VR har mycket arbete lagts ned på att hitta snabba och effektiva algoritmer för att detektera kollision mellan objekt i tre dimensioner. Det är som bekant viktigt att en datorsimulerad hand i en VR-applikation inte flyter igenom en stol som användaren vill flytta på i sin simulerade omgivning. För att ge användaren en äkta känsla är det viktigt att detekteringen av eventuella kollisioner går tillräckligt snabbt, så att användarens önskningar får acceptabla svarstider. Detta drog vi nytta av genom att återanvända liknande funktioner och tillämpa dessa på bästa sätt.

Efter att ha jämfört olika algoritmer och programpaket från olika forskningsgrupper, fastnade vi för ett system som bäst verkade uppfylla de krav som vi ställt upp för en modern kollisionsdetektering. Systemet skulle vara enkelt att använda för en programmerare, lätt att utveckla och expandera till en modell för borriggar samt vara minnessnålt och ge en snabb exekvering i målmiljön.

Det system som bäst uppfyllde kraven heter RAPID och är implementerat i C++. Det är utvecklat av forskare och doktorander vid *University of North Carolina*, USA. En stomme för kollisionskontroll med hjälp av RAPID implementerades i C++ och ett användargränssnitt gjordes i Matlab för visualisering av modellen och värdeinmatning till densamma. I Matlab utfördes även allt nödvändigt beräkningsarbete för kollisionskontrollen.

Farhågorna med att kollisionskontrollen med RAPID inte skulle fungera fanns i vårt medvetande, men eftersom försöken med RAPID och Matlab slog väl ut gick vi vidare och implementerade en Matlab-oberoende version i C++ och gjorde tester av programvaran i målmiljön på befintlig hårdvara; annars skulle en egen variant som byggde på teorierna och idéerna bakom RAPID ha implementerats. De problem som stötts på under arbetets gång har inte rört själva kollisionskontrollen, mycket tack vare att konceptet bakom RAPID är enkelt att förstå och erbjuder en intuitiv modell. Däremot har det tagit tid att sätta sig in i det verktyg som används för utveckling av programvara vid företaget och lära sig att utnyttja verktyget och dess kraftfulla finesser för felsökning på rätt sätt.

Den lösning som presenteras i rapporten uppfyller både de krav som ställts upp i problemformuleringen för arbetet och de krav som företaget ställt på realtidsapplikationer i sina produkter. Om det fortsätter att fungera som tänkt kommer kollisionskontrollen med RAPID därför att implementeras så att den även kontrollerar bommarna då *operatören* kör bommarna på riggen och inte endast då programmet för automatisk positionering styr, som det tidigare var planerat.

7 Referenser

Cameron, Stephen: *Enhancing GJK: Computing Minimum and Penetration Distances between Convex Polyhedra*, Oxford University Computing Laboratory, 1997

Cameron, Stephen: *Computing the Distance between Objects*, http://www.comlab.ox.ac.uk/oucl/users/stephen.cameron/distances.html, 1998

Carlvik, Ingvar: *BVX77. Strategy for choosing route between two boom positions*, Atlas Copco MCT AB, Nacka 1984, XRS 84.49

Cohen, J. D. / Lin, M. C. / Dinesh, M. / Ponamgi, M. K.*: I-Collide: An Interactive and Exact Collision Detection System for Large-Scale Environments*, University of North Carolina, USA 1997 ftp://ftp.cs.unc.edu/pub/users/manocha/PAPERS/COLLISION/paper3dint.ps.Z

Gottschalk, S. / Lin, M. C. / Manocha D.: *OBBTree: A Hierachical Structure for Rapid Interference Detection*, University of North Carolina, USA 1996, http://www.cs.unc.edu/~geom/OBB/OBBT.html

Lin, M. C. / Gottschalk S*.: Collision Detection between Geometric Models: a Survey*, University of North Carolina, USA 1997 ftp://ftp.cs.unc.edu/pub/users/manocha/PAPERS/COLLISION/cms.ps.gz

Sciavicco, L. / Siciliano, B.: *Modeling and Control of Robot Manipulators*, McGraw-Hill, USA 1996, ISBN 0-07-057217-8

Spong, Mark W. / Vidayasagar, M.: *Robot Dynamics and Control*, John Wiley & Sons, Canada 1989, ISBN 0-417-61243-X

Stadler, Wolfram: *Analytical Robotics and Mechatronics*, McGraw-Hill, Singapore 1995, ISBN 0-07-113792-0

8 Bilaga

8.1 Klassdiagram