NTNU
Det skapende universitet

Styresystem for autonome fartøy

Autonom konstruksjon av tredimensjonale baner

Øystein Solheim Havstad

Master i ingeniørvitenskap og IKT
Innlevert: juni 2017
Hovedveileder: Sven Fjeldaas, MTP

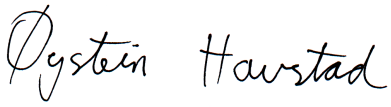
Norges teknisk-naturvitenskapelige universitet
Institutt for maskinteknikk og produksjon

Forord

Denne masteroppgaven er skrevet våren 2017, og er en fortsettelse på arbeidet starta i min prosjektoppgave høsten 2016. Den markerer avslutninga på min femårige utdannelse på NTNU ved linja IKT, produktutvikling og materialer. Denne oppgaven passer meg bra da den kombinerer kunnskap om geometri, mekanikk og fartøy med datamaskiner og programmering. Foruten å gi meg mulighet til å arbeide på et så spennende fagområde som autonome fartøy.

Jeg vil gjerne takke veileder Sven Fjeldaas for fortreffelig, nyttig og motiverende veiledning gjennom hele arbeidet med oppgaven. Samt en takk til mine medstudenter Stefan Bui og Tony Gjendahl.

Trondheim, 11.06.2017



Øystein Havstad

Sammendrag

Denne masteroppgaven omhandler styresystem for autonome fartøy og deres baner. Den etablerer en programvare som gjør det mulig å eksperimentere med algoritmer for autonom konstruksjon av tredimensjonale baner. Konstruksjonsverktøyet inkludere både et interaktivt grafisk brukergrensesnitt og et grensesnitt for bruk av anna programvare i autonom modus. Programvaren er basert på en geometrisk modellerer kalt GeoMod og utvikla ved hjelp av programmeringsspråket C++ og applikasjonsrammeverket Qt. Dynamisk linking av programvaren er innført slik at den passer inn i GeoMods modulære designfilosofi. Verktøyet kan holde orden på og arbeide med flere baner. Fartøys posisjon og orientering langs banene kan beregnes. Det er utvikla et nytt format for lagring av geometriske data på fil. Forslag til en metode for å parametrisere geometrien i filene er tatt med, og implementasjonen er påbegynt. I tillegg er det utført et innledende forsøk med en algoritme som automatisk konstruerer en bane mellom bevegelige hindre. Algoritmen viser konseptet, men mye arbeid gjenstår fortsatt. Videre bør det arbeides med utvikling av mer avanserte algoritmer. Samt med finpuss av verktøyet med tilhørende rammeverk.

Summary

This thesis deals with a control system for autonomous vehicles and their trajectories. It establishes a software which makes it possible to experiment with algorithms for autonomous design of tree-dimensional trajectories. The software is based on a geometric modeller named GeoMod. It is developed using the programming language C++ and the Qt application framework. The design tool includes both an interactive graphical user interface and an interface to be used by external software and autonomous algorithms. Adaption of dynamic linking makes the software fit into the modular design philology of GeoMod. The tool makes it possible to design and manages multiple trajectories. The vehicle's position and orientation is calculated. A new file format is developed to store geometric data. A method to parametrize geometry is proposed, and work on the implementation has commenced. Additionally, preliminary trails with an algorithm designing trajectories between moveable obstacles have been made. The algorithm is a proof of concept, and much work persists developing more advanced algorithms. In addition to polishing out quirks in the design tool and the included framework.

Innhold

Figurliste

Programkodeliste

1 Introduksjon

Autonome undervannsfartøy er av stor interesse både i forskningsarbeid, militært bruk og til installasjon, vedlikehold og inspeksjon av industrielle installasjoner [2, s. 186]. De kan utføre observasjoner ved hjelp av kamera og utstyres med en manipulator for å plukke opp gjenstander eller betjene verktøy. Teknologiske nyvinninger gjør autonome fartøy stadig mer attraktive. Autonome fartøy har allerede overtatt flere oppgaver fra sine fjernstyrte søskenfartøy, men nye arbeid gjenstår. Teknikker og algoritmer må utvikles og testes. Metoder og rammeverk kan være med på å lette arbeidet.

Autonome fartøy bør kunne modifisere gitte baner og konstruere nye, for deretter å følge banene. Baner bør kunne gis både i fritt rom og relativt til flater. Hovedformålet med denne masteroppgaven er å etablere en programvare som gjør det mulig å eksperimentere med algoritmer for autonom konstruksjon av baner. Den formelle oppgavebeskrivelsen er gjengitt i sin helhet i vedlegg B. Programvaren er basert på den geometriske modelleren GeoMod og et konstruksjonsverktøy for tredimensjonale baner. Modelleren er utvikla av veileder, Sven Fjeldaas, med hjelp av tidligere masterstudenter. Utvikling av konstruksjonsverktøyet ble påbegynt i min prosjektoppgave høsten 2016, og videreføres i denne oppgaven. Oppgaven er den siste i en lang rekke masteroppgaver ved institutt for maskinteknikk og produksjon under hovedtittelen "Styresystem for autonome fartøy". Denne våren er det tre studenter som arbeider hver sin oppgave under denne hovedtittelen.

Oppgaveteksten inneholder i tillegg mer konkrete deloppgaver. Deloppgavene er ment som en hjelp til å nå hovedmålsetninga, og er med på å forbedre rammeverket, og legge forholdene til rette for å lettere kunne eksperimentere med autonome algoritmer. Første deloppgave lyder "Innføre dynamisk linking av konstruksjonsverktøy for baner". Dynamisk innlasting og linking av verktøy og modeller under kjøretid er en viktig del av de modulære designprinsippene bak programvaren GeoMod. Ved å innføre dynamisk linking i konstruksjonsverktøyet sørger man for at verktøyet er bygd på riktige premisser og at det oppfører seg bra sammen med resten av programmet.

Neste del omhandler justering av detaljer i programvaren og innføring av ny funksjonalitet. Første punkt går ut på å innføre baner med orientering i tillegg til posisjon. Koblinga mellom banenes romlige geometri og fartøy som kjører på banen må opprettes. Posisjon og orientering skal beregnes etter metode spesifisert av veileder. Videre skal det innføres et skille mellom planlagte- og faktiske baner, samt overganger mellom disse og utfallsrommet for videre navigasjon. For å gjøre det

mulig å ta vare på og overføre baner, skal det arbeides med lagring av geometriske data på fil. Ny funksjonalitet må innføres, og som en del av arbeidet skal muligheter for parametrisk styring av geometri vurderes.

I tillegg kommer en deloppgave som inneholder et eksperiment med en autonom algoritme. Det skal gjøres et innledende forsøk med en algoritme som forsøker å legge baner mellom bevegelige hindre. Oppgaven spesifiserer at kuber og pyramider kan tjene som hindre. Koden utvikla i dette forsøket kan nyttes som et rammeverk for videre eksperimentering med nye algoritmer.

Den opprinnelige oppgaveteksten inneholder punktet "Reetablere en tidligere versjons program for skulpturerte flater". Denne deloppgaven er svært omfattende. I samtaler med veileder ble det bestemt å se bort fra denne deloppgaven. Arbeid med å oppgi baner relativt til flater blir som følger også utsatt. Flere detaljer er lagt til kapittel 5.

Teksten følger stort sett samme oppbygning som oppgaveteksten, og hver deloppgave har sitt eget delkapittel. Først introduseres tidligere arbeid og den geometriske modelleren GeoMod. Videre tar teorikapittelet for seg matematikk og geometri brukt i arbeidet med programvaren. Videre introduseres programmeringsspråket C++ og Qt-biblioteket. Kapittel 4 skal gi leseren en oversikt over den nåværende tilstanden til konstruksjonsverktøyet. Kapittel 5 introduserer skulpturerte flater og gjør kort rede for hvorfor deloppgaven med dette temaet ble valgt bort. Videre følger ett kapittel for hver av deloppgavene. Hvert kapittel introduserer deloppgaven, kommer om nødvendig med en utledning av matematikk og anna teori, før selve løsninga og implementasjonen legges fram. Kapitlene avsluttet ved å ta opp begrensninger og komme med forslag til videre arbeid. Teksten avsluttes med et oppsummeringskapittel som inkluderer ei vurdering av framtidige muligheter.

Utviklingsarbeidet er utført på en 64-bit datamaskin med en oppdatert utgave av operativsystemet Arch Linux med kjerneversjon 4.11 og skrivebordsmiljøet GNOME versjon 3.24. Utviklingsmiljøet Qt Creator versjon 4.3.0 er benytta sammen med versjon 5.8.0 av Qt-biblioteket og GCC (GNU Compiler Collection) versjon 6.3.1. Maskinen har en Intel Core i7-6700K prosessor med 16GB minne og en NVIDIA GeForce GTX 1070 grafikkprosessor. Alle bilder, figurer og illustrasjoner er egenprodusert dersom det ikke er oppgitt noe anna. Flesteparten er skjermbilder fra GeoMod. Noen dem har blitt beskåret og fått lagt til uthevinger ved hjelp av bildebehandlingsprogrammet GIMP. Figurene som illustrer geometri og den bakenforliggende matematikken, i teorikapittelet og ellers, er laga ved hjelp av det interaktive geometriprogrammet GeoGebra.

2 Tidligere arbeid

2.1 GeoMod

Arbeidet i denne masteroppgaven utføres på en geometrisk modellerer kalt GeoMod. Modelleren er utvikla av veileder, Sven Fjeldaas, med hjelp av tidligere studenter. Den kan styre og visualisere modeller av fritt bevegelige ledda og glidende mekanismer. Programmeringsspråket C++ er brukt sammen med applikasjonsrammeverket Qt. GeoMod er ifølge dokumentasjonen [7] en kompakt, rask og brukervennlig modeller basert på et sett transformasjoner avleda fra tensoralgebra, matematiske verktøy fra relativitetsteorien og romlige dynamiske datastrukturer. Modelleren er designa for syntetisk programmering og fjernstyring av komplekse maskiner, med fokus på autonome undervannsfartøy utstyrt med manipulator.

2.1.1 Geometri i GeoMod

Programvaren er delt inn i to deler. Biblioteket 'MaxLib' inneholder den grunnleggende funksjonaliteten, inkludert geometri, matematikk og tegnerutiner. GeoMod og Qt Creator-prosjektet 'entrance_06' inneholder det grafiske brukergrensesnittet sammen med funksjonalitet for å laste inn programtillegg, også kalt plugins, dynamisk. Dette delkapittelet forklarer hvordan de mest brukte geometriske objektene i GeoMod og MaxLib virker. Figurene illustrerer prinsipper og er forenkla til to dimensjoner, programvaren arbeider selvfølgelig i tre dimensjoner.

Vektorer og utvida baser
GeoMod har sin egen implementasjon av en matematisk vektor. Klassen 'MathVec' implementerer en vektor i tre dimensjoner med x- y- og z-komponenter. I tillegg finnes klasen 'IDMthVec' som utvider vektorkonseptet ved å legge til en tekststreng som virker som en identifikasjon eller navn. Vektorklassen i standardbiblioteket til C++ eller i Qt-biblioteket kan ikke brukes til dette formålet. Den implementerer en sekvensiell liste eller konteiner, ikke en matematisk vektor. De fleste av GeoMods geometriske objekter har koordinater oppgitt relativt til sin egen base. En vilkårlig base B_1 har origo i $\vec{O}_{B_1}$ og ortonormale aksevektorer $\vec{x}_{B_1}$, $\vec{y}_{B_1}$ og $\vec{z}_{B_1}$. Implementasjonen er lagt til klassen 'ExtBasis' i 'MaxLib/math/extbasis<.h/.cpp>'.

Modeller og transformasjonsgrupper
Geometriske objekter i GeoMod er vanligvis programmert inn som en egen C++-klasse som arver klassen 'Models::ModelData'. Mange modeller er koda som et vanlig C++-program, og kan dermed utnytte programmeringsspråkets muligheter og fleksibilitet. Basismodeller som kuber og pyramider følger med, og kan brukes som de er, eller danne inspirasjon for utvikling av nye modeller. En modell kan

inneholde komplisert geometri. Delgeometrier som hører sammen må transformeres sammen, og lagres derfor i egne transformasjonsgrupper. Transformasjonsgrupper er implementert som instanser av klassen 'Tgroup' i MaxLib. Derav kommer forkortelsen 'tgruppe', som er mye brukt videre i teksten.

Noder, kanter og flater
Katalogen 'MaxLib/net_g/' inneholder en rekke klasser for elementære geometriske objekter. Objektene danner grunnlaget for å bygge opp den sammensatte geometrien som behøves av for eksempel baner og polyeder. En node, som definert av klassen 'GeomNode', representerer et punkt i det tredimensjonale rommet. Den har et navn og er gjengitt av en vektor med koordinatverdier. Klassen 'GeomEdge' beskriver geometriske kanter. En kant er ei navngitt, rett eller kurva linje mellom to noder. Kanten er retta. Positiv retning går fra halenoden (eng: tail node) til hodenoden (eng: head node). Flater er implementert i 'GeomRegion', og er gitt implisitt som et nettverk av pekere til kantene som omslutter flata. På dette stadiet av utviklingsarbeidet, er ikke flater veldig relevant for arbeidet med konstruksjon av baner. De inngår likevel i arbeidet med lagring på fil, og en mer detaljert beskrivelse er lagt til kapittel 9.1. Maxlib har også støtte for kurver. Et utvalg kurvetyper er implementert i 'MaxLib/curves/curve<.h/.cpp>'. En kurve er assosiert med en vanlig rettlinja kant, og går mellom de samme nodene som kanten.

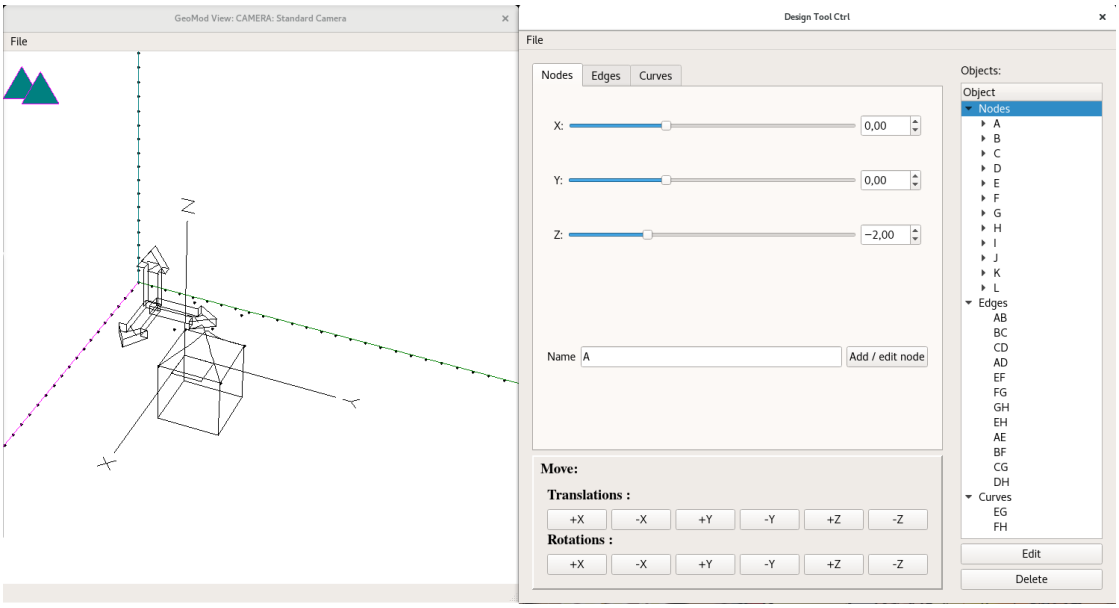
2.2 Bevegelseskontroll og innledende støtte for baner

Martin L. Furevik arbeidet i sin masteroppgave våren 2016 [8] med styring av undervannsfartøy, inkludert manipulator, langs baner i GeoMod. Verktøyet 'MotionCtrl' ble utvikla og introduserte initiell støtte for baner. Dette på et tidspunkt før arbeidet med konstruksjonsverktøyet var påbegynt. Banene er implementert i form av vanlige modeller med spesielle lister over kanter og noder som danner selve banen. Verktøyet lar maskiner og roboter følge banen med støtte for både posisjon og orientering gitt av støttekanter. Verktøyet har i tillegg mulighet for å legge til nye noder og kanter under kjøretid. Men funksjonalitet for å endre eksisterende geometri i banene er ikke inkludert.

2.3 Konstruksjonsverktøy for tredimensjonale baner

I min prosjektoppgave høsten 2016 ble det utvikla et konstruksjonsverktøy for tredimensjonale baner for bruk sammen med de geometrisk modelleren GeoMod. Verktøyet er implementert i et interaktivt grafisk brukergrensesnitt, og prøver å etterligne et Kuhlmann-tegnebord med arbeidsflate og hovedlinjal. Figur 1 viser verktøyets hovedbrukergrensesnitt sammen med tegneområdet. Banene bygges opp

av et utvalg enkle geometriske objekter. Etter arbeidet med prosjektoppgaven er det implementert støtte for noder, kanter og kurver. Et objekttre gir oversikt over objektene. Annen programvare har også mulighet til å bruke verktøyet i autonom modus, og støtte for å håndtere flere baner er inkludert. Flere detaljer finnes i rapporten fra prosjektoppgaven [11] og i kapittel 4 "Oversikt over konstruksjonsverktøyet".



Figur 1: Skjermbilde som viser ei typisk arbeidsøkt med konstruksjonsverktøyet utvikla i prosjektoppgaven. Tegneflata vises til venstre og hovedkontrollpanelet til høyre.

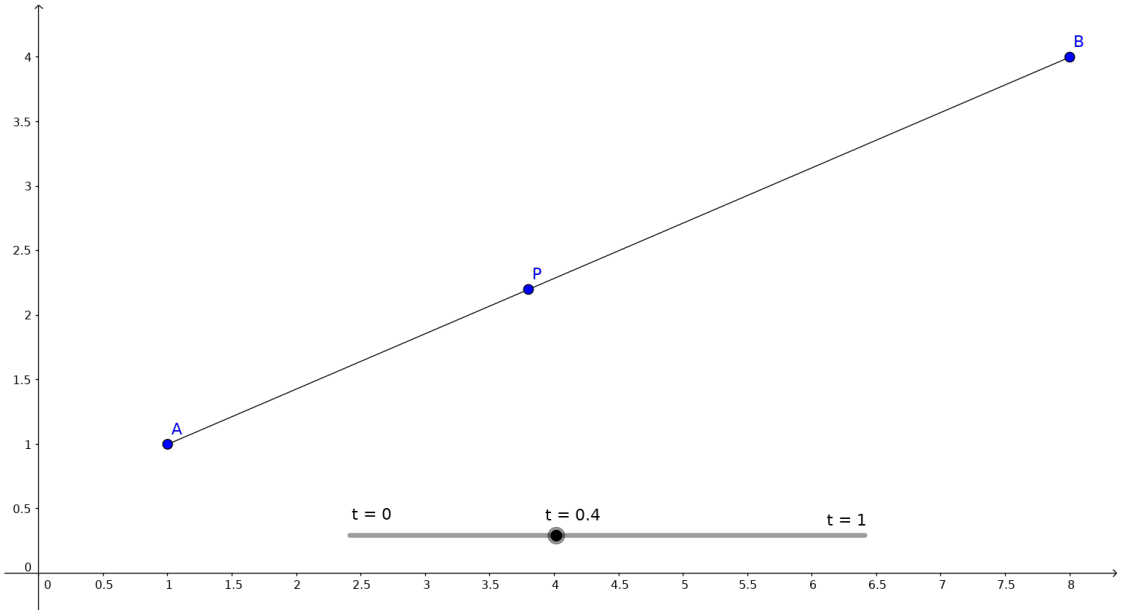
3 Teori

3.1 Lineær interpolasjon

Punkt på rette linjer kan beregnes ved hjelp av lineær interpolasjon som beskrevet i [5, s. 25]. La A og B være to ulike punkt i det tredimensjonale rommet. Punkt P på linja mellom A og B, bestemt av parameteren t, er gitt av ligning 1.

$$P(t) = (1 - t)A + tB \tag{1}$$

Ved $t = 0$ er $P = A$, og ved $t = 1$ er $P = B$. Når t er mellom 0 og 1, ligger P på linjestykket mellom A og B. For andre verdier av t ligger P på forlengelsen av linjestykket. Figur 2 viser forholdet mellom linja og parameteren (t).



Figur 2: Viser forholdet mellom parameteren t og punktet P på den interpolerte linja mellom A og B.

3.2 Bezierkurver

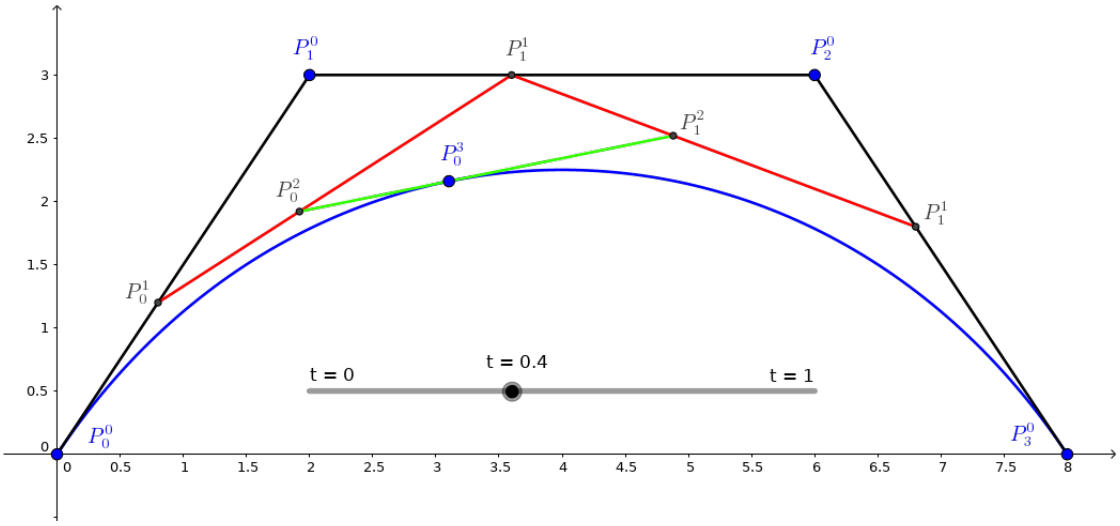
Kurver har mange bruksområder innen kunst, arkitektur, matematikk og industriell design. Bezierkurver er mye brukt innen datagrafikk og i systemer for datamaskin assistert konstruksjon. Tradisjonell design bestod for det meste av rette linjer, sirkelsektorer og andre kjeglesnitt. Flere detaljer og mer komplisert geometri kan konstrueres ved å ta i bruk kurver. Enkle baner kan bygges opp av rette linjer. Men behov for kurva linjer i banene viser seg raskt.

de Casteljaus algoritme
Punkt på ei bezierkurve kan beregnes etter de Casteljaus algoritme [25, s 151]. Kurven parametriseres med parameteren $t \in [0, 1]$. Algoritmen følger den rekursive formelen i ligning 2.

$$P_i^j = (1-t)P_i^{j-1} + tP_{i+1}^{j-1} \tag{2}$$

Notasjonen P_i^j viser til punkt nummer i ved iterasjon j. For ei kurve med kontrollpunktene $P_0^0, P_1^0, P_2^0, ..., P_n^0$ er $j = 0, 1, 2, ..., n$ og $i = 0, 1, 2, ..., n - j$.

Konstruksjon av ei kubisk bezierkurve etter de Casteljaus algoritme er vist i figur 3. Kurven har fire kontrollpunkt (P_0^0, P_1^0, P_2^0 og P_3^0), og algoritmen generer



Figur 3: Konstruksjon av bezierkurve med fire kontrollpunkt etter de Casteljaus algoritme

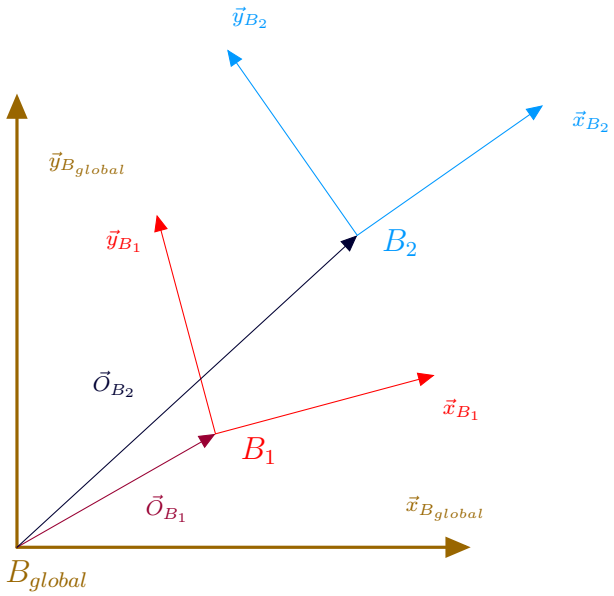
dette triangulære punktsettet:

$$\begin{array}{cccc} P_0^0 & P_1^0 & P_2^0 & P_3^0 \\ P_0^1 & P_1^1 & P_2^1 & \\ P_0^2 & P_1^2 & & \\ P_0^3 & & & \end{array} \tag{3}$$

Punktet P_0^3 er det ønska punktet på kurven for den valgt verdien av t. En geometrisk betraktning av algoritmen viser gjentatte lineære interpolasjoner mellom kontrollpunkta. Algoritmen kan brukes med vilkårlig mange kontrollpunkter, men i praksis går det ei grense ved fem eller seks punkter. Vanligvis benyttes kubiske kurver (med fire kontrollpunkter), og komplekse kurver konstrueres ved å legge flere kubiske kurver etter hverandre.

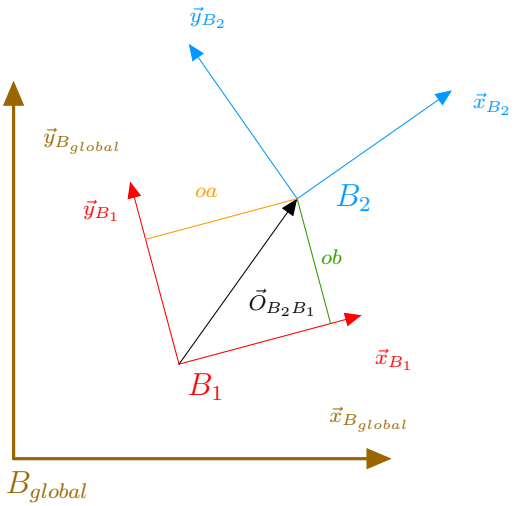
3.3 Transformasjon av punktkoordinater

Modellene og transformasjonsgruppene i GeoMod har koordinater oppgitt i sin egen lokale base. Under samhandling mellom modeller er det ofte behov for å transformere koordinater fra en base til en anna. Dermed er det nødvendig å kunne transformere et punkt $\vec{P}_{B_2} = (P_x, P_y, P_z)$ fra base B_2 til B_1. Figur 4 viser to utvida baser B_1 og B_2 i forhold til hverandre og i forhold til det globale koordinatsystemet B_{global}.



Figur 4: Viser basene B_{global}, B_1 og B_2 med deres akser og offsetvektorer i to dimensjoner.

Først beregnes offsetvektoren $\vec{O}_{B_2B_1}^{B_glob} = \vec{O}_{B_2} - \vec{O}_{B_1}$ relativ til det globale koordinatsystemet (B_{glob}). Deretter skalarprojiseres offsetvektoren langs aksene til B_1 for å danne $\vec{O}_{B_2B_1}^{B_1} = [oa, ob, oc]$ slik ligning 4 og figur 5 viser.



Figur 5: Viser skalarprojeksjonene oa og ob av $\vec{o}_{B_2B_1}$ langs aksene $\vec{x}_{B_1}$ og $\vec{y}_{B_1}$ respektivt.

$$\begin{aligned} oa &= \vec{x}_{B1} \cdot \vec{O}_{B_2B_1}^{B_glob} \\ ob &= \vec{y}_{B1} \cdot \vec{O}_{B_2B_1}^{B_glob} \\ oc &= \vec{z}_{B1} \cdot \vec{O}_{B_2B_1}^{B_glob} \end{aligned} \tag{4}$$

Så beregnes skaleringsfaktorene a, b, ..., i etter ligning 5. Ligning 6 bruker skaleringsfaktorene til å bestemme $\hat{\vec{P}} = (\hat{P}_x, \hat{P}_y, \hat{P}_z)$. Til slutt legges offsetvektoren mellom basene til, og $\vec{P}_{B_2} = \hat{\vec{P}} + \vec{O}_{B_2B_1}^{B_1}$.

$$\begin{aligned} a &= \vec{x}_{B1} \cdot \vec{x}_{B_2} \quad b = \vec{y}_{B1} \cdot \vec{x}_{B_2} \quad c = \vec{z}_{B1} \cdot \vec{x}_{B_2} \\ d &= \vec{x}_{B1} \cdot \vec{y}_{B_2} \quad e = \vec{y}_{B1} \cdot \vec{y}_{B_2} \quad f = \vec{z}_{B1} \cdot \vec{y}_{B_2} \\ g &= \vec{x}_{B1} \cdot \vec{z}_{B_2} \quad h = \vec{y}_{B1} \cdot \vec{z}_{B_2} \quad i = \vec{z}_{B1} \cdot \vec{z}_{B_2} \end{aligned} \tag{5}$$

$$\begin{aligned} \hat{P}_x &= aP_x + bP_y + cP_z \\ \hat{P}_y &= dP_x + eP_y + fP_z \\ \hat{P}_z &= gP_x + hP_y + iP_z \end{aligned} \tag{6}$$

3.4 Programmeringsspråket C++

C++ er et programmeringsspråk til alminnelige formål med fokus på design og bruk av typerike, lettvekte abstraksjoner [28]. Det er godt egna for resursbegrensa applikasjoner innenfor systemprogrammering, innebygde systemer, spillmotorutvikling og programvareinfrastruktur. C++ er en utvidelse av programmeringsspråket C med inspirasjon fra det tidlige objektorienterte språket Simula. Utviklinga av starta med Bjarne Stroustrups doktorgradsarbeid i 1979 ved å introduser arv og klasser til C. Det opprinnelige målet var å kombinere C sin effektivitet og fleksibilitet for systemprogrammering med Simulas objektorienterte konsepter som klasser, arv og virtuelle funksjoner. Videre har språket blitt standardisert av International Organization for Standardization (ISO) med siste versjon, C++14, utgitt desember 2014 [15]. Språket er støtta på de fleste plattformer med kompilator fra flere leverandører. Blant anna fra Free Software Foundation, Microsoft, Intel, og IBM.

Sentrale begrep i et objektorientert programmeringsspråk er arv, objekter og klasser. En klasse er en brukerdefinert datatype som definerer medlemsvariabler og medlemsfunksjoner. Klassekonseptet muliggjør innkapsling av data sammen med kontroll over initialisering og grensesnitt utad. Objekter opprettes som en instans av en klasse. Klassen definerer objektets struktur og oppbygging, men hver instans har sin egen utgave av medlemsvariablene. Ved hjelp av arv kan opprette nye klasser som utvider en eksisterende klasse ved å legge til nye medlemmer.

Språket støtter, i likhet med C, bruk av pekere. En peker er en spesiell datatype som "peker til" en verdi lagra et anna sted i minnet. Minneadressa er lagra i pekeren, ikke selve verdien. Konseptet kan sammenlignes med stikkordslista i ei bok. Ett stikkord er assosiert med et sidetall, altså stikkordet "peker til" sida og teksten (verdiene) som er lagra der. Pekere brukes ofte for å unngå tids- og plasskrevende kopiering av data når objekter skal lagres i lister eller sendes som argument til funksjoner.

3.5 Qt-rammeverket

Qt er et kryssplattform applikasjonsutviklingsrammeverk for skrivebord, mobil og innebygde systemer [20]. Qt er ikke et programmeringsspråk i seg selv men et rammeverk bygd oppå C++. Bindinger til andre programmeringsspråk finnes også. Plattformene Linux, OS X, Windows, Android, iOS og Sailfish OS med flere er støtta. Utvikling starta i 1990 av nordmennene Haavard Nord and Eirik Chambe-Eng og deres firma Trolltech, seinere sogt til Nokia og Digia. I dag eies Qt av firmaet The Qt Company, og programvaren tilbys både under en kommersiell lisens (Qt

Commercial License) og under lisenser for fri programvare (GPLv3, GPLv2 og LGP3) [18].

Rammeverket blir som regel brukt i forbindelse med grafiske brukergrensesnitt (GUI), men inneholder også et standardbibliotek med vanlige datatyper, tekststrenger, oppbevaringsklasser, internasjonalisering og trådstøtte. Samt biblioteker for grafikk, multimedia, nettverk, mobil, web, lagring og filhåndtering [22]. Det danner blant anna grunnlaget for noen av de største skrivebordsmiljøa brukt sammen med Unixvarianter som Linux og BSD. Eksempelvis KDE [31], LXQt [30] og Liri [17]. GeoMod er bygd med C++ oppå Qt-rammeverket.

3.5.1 Signals and Slots

Signals and slot er et konsept som gjør at objekter, især GUI-widgeter, kan kommunisere med hverandre. Et objekt kan sende ut et signal når den interne tilstanden har blitt endra. Et slot er en funksjon som skal kalles som svar på et bestemt signal. Qt-widgetene har mange forhåndsdefinerte slot, men det er vanlig at utviklere oppretter sine egne slots for å håndtere signaler av interesse [23]. Når brukeren for eksempel klikker på knappen 'Avslutt', skal programrutinen 'exit()' utføres. Dette løses ved å koble knappens 'clicked()'-signal sammen med programmets 'exit()'-slot. For å dra nytte av konseptet i egne objekter, må klassen arve fra QObject. Signaler og slots kobles sammen og fra hverandre ved å kalle QObjects funksjoner 'connect(...)' og 'disconnect(...)' som eksemplifisert i listing 1.

```
connect    ( exitBtn , SIGNAL(clicked()) , this , SLOT(exit()));
disconnect( exitBtn , SIGNAL(clicked()) , this , SLOT(exit()));
```

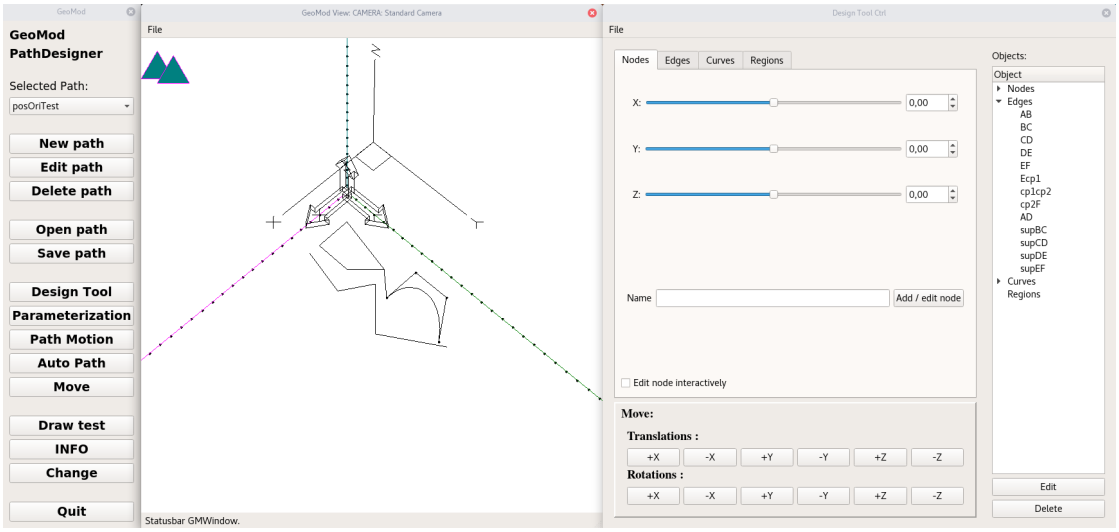
Listing 1: Eksempel på til- og frakobling av signaler og slots.

Signals and Slots gjør det lett å implementere designmønsteret 'Observer' som, ifølge Gamma [9, kap. 5.8, s. 326], definerer en en-til-mange-avhengighet mellom objekter. Slik at når et objekt endrer tilstand, får alle de avhengige objektene beskjed, og mulighet til å oppdater seg automatisk. Mønsteret er nyttig når endringer av et objekt påvirker mange andre, kanskje også objekter man ikke vet om på dette tidspunktet. Dette er med på å unngå tett kobling mellom objekter og letter gjenbruk av kode.

4 Oversikt over konstruksjonsverktøyet

Konstruksjonsverktøyet for tredimensjonale baner ble utvikla i arbeidet med min prosjektoppgave høsten 2016. Det muliggjør konstruksjon av geometri i baner via

et interaktivt grafisk brukergrensesnitt. Verktøyet prøver å etterligne et Kuhlmann-tegnebord med arbeidsflate og hovedlinjal. Oversiktsbilde i figur 6 gir et eksempel på ei typisk arbeidsøkt med verktøyet. Tegneområdet og geometrien i banene tegnes opp i GeoMods grafiske visning. Et sett grafiske paneler, basert på Qt-Widgets, styrer verktøyet. GeoMod har ikke hatt et konstruksjonsverktøy av denne typen tidligere.



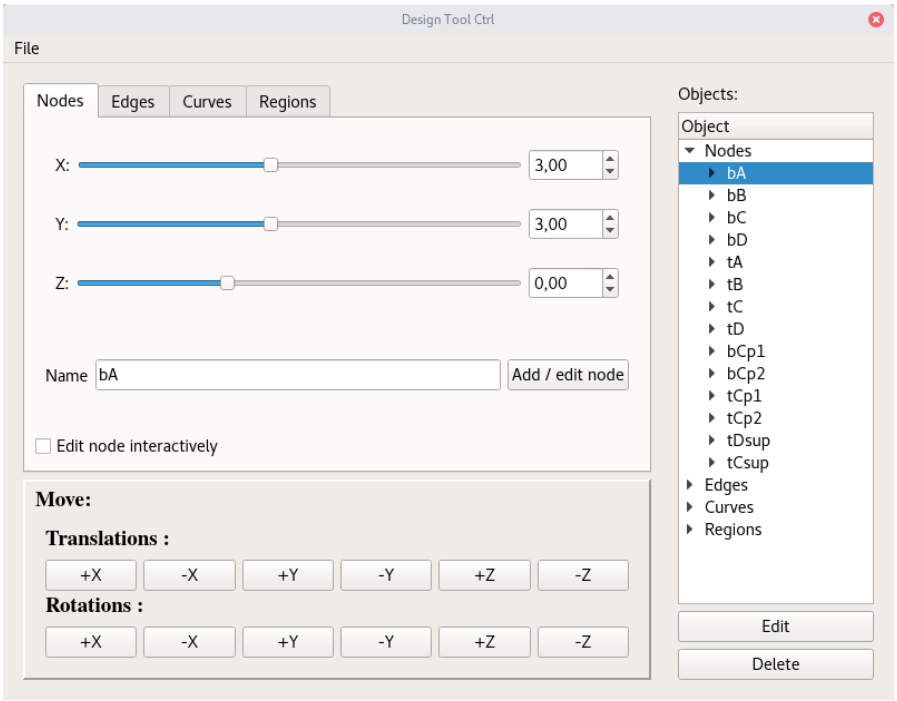
Figur 6: Oversiktsbilde av konstruksjonsverktøyet. Både tegneflata, med en eksempelbane, og de viktigste kontrollpanela er vist.

Brukerens første møte med konstruksjonsverktøyet er knapperada vist i figur 7. Nedtrekksboksen øverst i vinduet brukes til å velge banen verktøyet arbeider på. Her kan man legge, endre på og slette baner. Mer om valg og endring av baner kapittel 8 “Flere baner”. Det er her brukergrensesnittet for å lagre og laste baner til og fra fil aksesseres. Filbehandling er beskrevet i kapittel 9 “Lagring på fil”. De fem neste knappene starter ulike delverktøy gjennomgått nedenfor. Knappene ’Draw test’, ’INFO’ og ’Change’ er tatt med grunnet tradisjon i GeoMod, men har på dette tidspunktet ingen funksjon i konstruksjonsverktøyet. Vinduet lukkes ved å trykke ’Quit’.

Kontrollpanelet ’DesignTool’ inneholder det grafiske brukergrensesnittet for å styre designverktøyet. Verktøyet er vist i sin helhet i figur 8-11. Kontrollpanelet består av tre hoveddeler. Øverst til venstre er et sett med faner som brukes til å opprette og endre et utvalg geometriske objekter. Foreløpig er støtte for noder, kanter og kurver implementert. Støtte for flater er delvis implementert, verktøyet

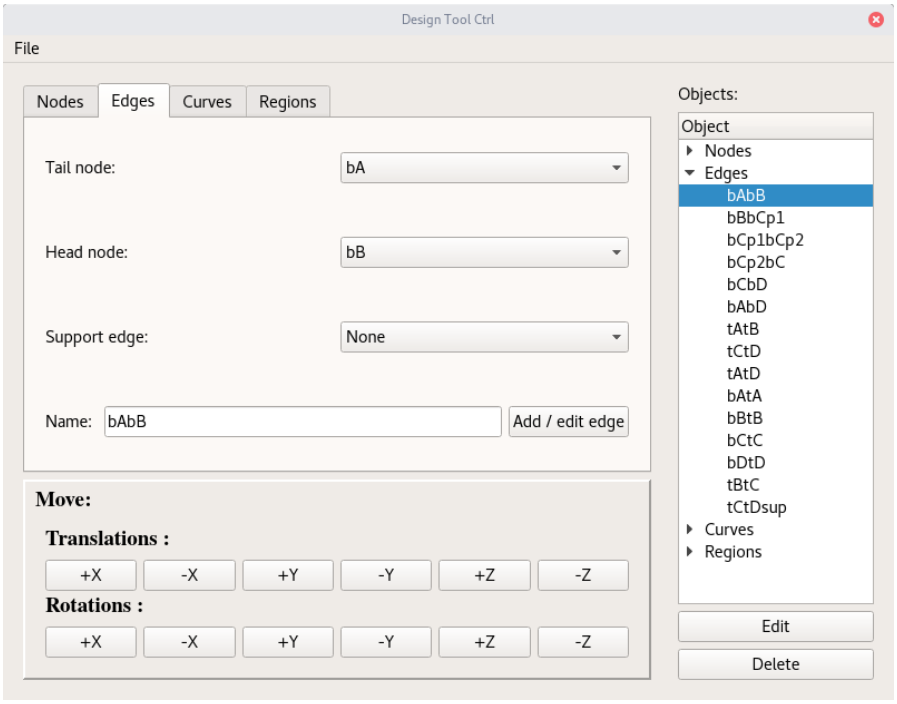
Figur 7: Inngangsporten til konstruksjonsverktøyet med mulighet til å velge bane og knapper for å starte delverktøy.

kan opprette men ikke endre flater. Nederst er det plassert et grunnleggende flytteverktøy som styrer DesignTool. Translasjon foregår langs DesignTools egne akser, rotasjon likedan. Helt til høyre er en oversikt over alle de geometriske objektene som danner den valgte banen. Oversikten er organisert som et tre med objekttype på toppnivå.

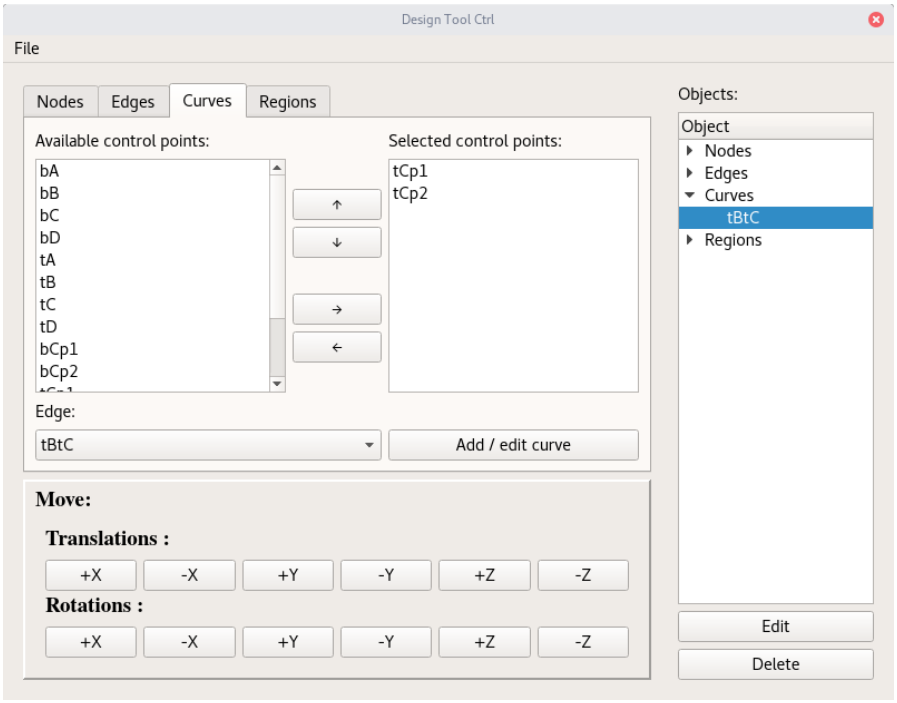


Figur 8: DesignTools fane for å opprette og endre noder.

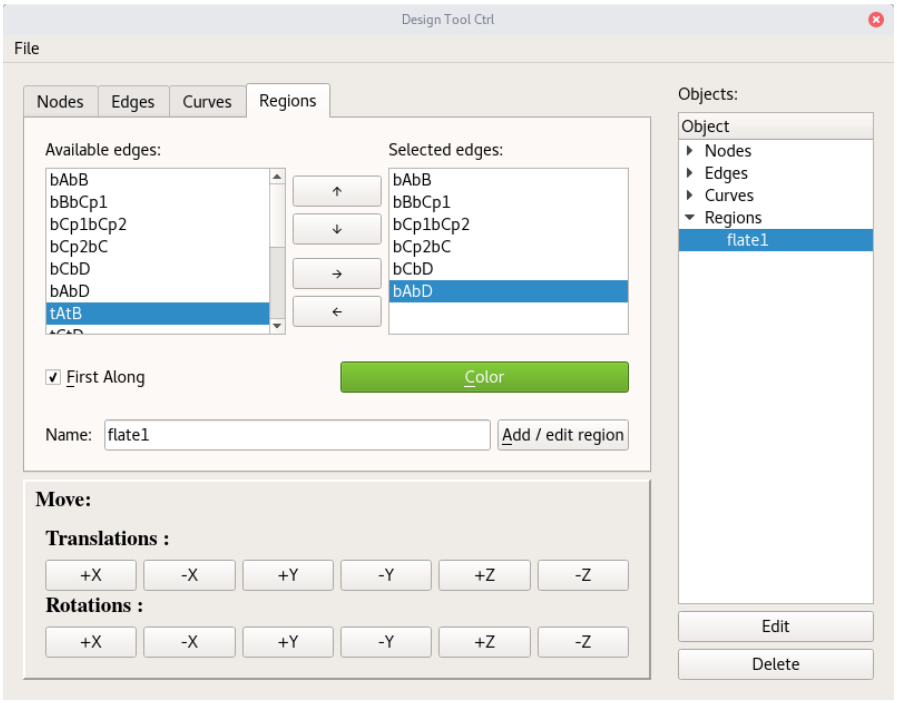
Parametrisering av geometri i konstruksjonsverktøyet er diskutert i kapittel 10 “Parametrisk styring av geometri”. Implementasjonen er påbegynt, men ferdigstilling krever mer tid og arbeid enn det rammene rundt denne masteroppgaven tillater. Utvikling av et grafisk verktøy er også påbegynt, og aksesseres via knappen ’Parameterization’. Bevegelse styres av verktøyet ’PathMotion’. Det beregner posisjon og orientering langs banene etter framgangsmåten beskrevet i kapittel 7 “Posisjon og orientering”. Et innledende forsøk med å automatisk legge baner mellom bevegelige hindre er utført og kan prøves ut via verktøyet bak knappen ’Auto Path. Detaljer finnes i kapittel 11 “Baner mellom bevegelige hindre”.



Figur 9: DesignTools fane for å opprette og endre kanter.



Figur 10: DesignTools fane for å opprette og endre kurver.



Figur 11: DesignTools fane for å opprette og endre flater.

Konstruksjonsverktøyet kan også benyttes av anna programvare i autonom modus. Klassen 'Path' inneholder funksjoner for å opprette, endre og slette geometriske objekter. Klassen 'PathDesArea' supplerer med funksjoner for å opprette og behandler baner samt hjelpefunksjoner for å konvertere koordinater mellom forskjellige baser. Når man utvikler programvare for å bruke verktøyet i autonom modus kan man bruke sammen framgangs- og arbeidsmåte som i det grafiske brukergrensesnittet. Funksjonene i Path og PathDesArea er lista opp nedenfor og mange av dem tilsvarer innputt- og klikk-kombinasjoner i det grafiske grensesnittet.

Relevante funksjoner i klassen Path ('geom_path_des/path.h'):

```
//Function to manage name and description
void setName(QString newName);
QString getName();
void setDescription(QString newDescription);
QString getDescription();


//Functions to manage nodes
int getNumberOfNodes();
GeomNode* getNode(int index);
```

```
GeomNode* getNodeByName(QString name);
GeomNode *getNodeByIDMthVec(const IDMthVec *vec);
GeomNode* addNode(double x, double y, double z, QString ID);
void editNode(GeomNode *node, double x, double y, double z);
void deleteNode(GeomNode *toBeDeleted);
int getNodeIndex(GeomNode *node);
QList<GeomNode*>::iterator getNodesBegin();
QList<GeomNode*>::iterator getNodesEnd();

//Functions to manage edges
int getNumberOfEdges();
GeomEdge *getEdge(int index);
GeomEdge *addEdge(GeomNode *tail, GeomNode* head, QString name);
GeomEdge *addEdge(GeomEdge *edge);
GeomEdge *getEdgeByName(QString name);
void deleteEdge(GeomEdge *toBeDeleted);
int getEdgeIndex(GeomEdge *edge);
QList<GeomEdge*>::iterator getEdgesBegin();
QList<GeomEdge*>::iterator getEdgesEnd();

//Functions to manage curves
int getNumberOfCurves();
BezierCurve *getCurve(int index);
BezierCurve *addCurve(GeomNode *startnode, GeomNode* endnode,
    GeomNode *controlPoint1, GeomNode *controlPoint2, QString name);
BezierCurve *addCurve(GeomEdge *edge, QList<IDMthVec*>
    controlPointList);
BezierCurve *getCurveByName(QString name);
void deleteCurve(BezierCurve *toBeDeleted);
QList<BezierCurve*>::iterator getCurvesBegin();
QList<BezierCurve*>::iterator getCurvesEnd();

//Functions to manage regions
int getNumberOfRegions();
GeomRegion* getRegion(int index);
GeomRegion* addRegion(std::list<GeomEdge*> edges, bool first_along,
    QString name);
GeomRegion* getRegionByName(QString name);
//void deleteRegion(GeomRegion* toBeDeleted);  //not implemented yet
QList<GeomRegion*>::iterator getRegionsBegin();
QList<GeomRegion*>::iterator getRegionsEnd();

//Functions to manage parameters
int getNumberOfParameters();
Parameter *addParameter(Parameter *param);
Parameter *getParameterByName(QString name);
//void deleteParameter(); //not implemented yet
QMap<QString, Parameter*>::iterator getParametersBegin();
QMap<QString, Parameter*>::iterator getParametersEnd();
```

```
QList<QString> getParameterNames();


//Function related to saving to / loading from file
bool isEmpty();
void makeJsonObject(QJsonObject &json);
bool loadFromJsonObject(const QJsonObject &json);
```

Relevante funksjoner i klassen PathDesArea ('geom_path_des/path_des_area.h'):

```
//Functions to manage paths
void setCurrentPath(Path *path);
Path *getCurrentPath();
int getNumberOfPaths();
Path *getPath(int index);
Path *createNewPath();
void deletePath(Path* toBeDeleted);
QList<Path*>::iterator getPathsBegin();
QList<Path*>::iterator getPathsEnd();


//Beregner skaleringsfaktorene som beskriver utbases rotasjon i
    innbase
//argumentene a, b, c, ... g, h brukes til returverdier
//Noatasjoner:
    //xi, yi, zi: Vektorer som beskriver innbase
    //xu, yu, zu: Vektorer som beskriver utbase
//Formeldefinisjon:
    //xu = a * xi + b * yi + c * zi
    //yu = d * xi + e * yi + f * zi
    //zu = g * xi + h * yi + i * zi
static void beregnSkaleringsfaktorerForRoteringAvBase(const ExtBasis
    *innbase, const ExtBasis *utbase, double *a, double *b, double *c,
     double *d, double *e, double *f, double *g, double *h, double *i)
    ;

//Koverterer koordinatene til punktet Pinn fra utbase til innbase
static MathVec konverterPunktFraInnbaseTilUtbase(const ExtBasis *
    innbase, const ExtBasis *utbase, MathVec Pinn);

//Beregner utbase sin relative offset i mellom inn- og utbase
static MathVec beregnRelativOffset(const ExtBasis *innbase, const
    ExtBasis *utbase);
```

5 Baner på skulpturerte flater

Ei skulpturert flate, implementert som en bikubisk lapp (engelsk: bicubic patch), er ei flate bygd opp av kubiske kurver. Flata opprettes ved å kjøre en generator langs to skinneganger kalt direktriser. Både generatoren og direktrisene er kubiske kurver. Flata parametriseres ved to parametere, ofte u og v, den ene parameteren langs direktrisene og den andre langs generatoren. Bruk av skulpturerte flater gjør det mulig å modellere kompliserte flater som kan innta mange former. Moderne bilkarosseri er et godt eksempel, men skulpturerte flater brukes også i design av håndverktøy, forbrukerelektronikk som radioer og datautstyr i tillegg til mange andre produkter. For en komplett beskrivelse av skulpturerte flater og den bakenforliggende matematikken henvises det til Fjeldaas hefte "Basic Computational Geometry: Curves and Surfaces" [6] eller ei lærebok på temaet dataassistert konstruksjon, slik som Duncan Marshs "Applied Geometry for Computer Graphics and CAD" [25].

Dagens versjon av GeoMod støtter ikke skulpturerte flater. Det fantes støtte i en tidligere versjon som bør gjeninnføres i den nye versjonen. Men dette krever en hel del arbeid. Den gamle koden finnes fortsatt, men bærer preg av å være gammel. Den er prosedyreorientert, og mye omskriving må til før den passer inn i den nye objektorienterte versjonen av GeoMod. Dette kommer ikke som noen overraskelse da datostemplinger i kodefilene daterer implementasjonen til årtusenskiftet.

Opprinnelig skulle arbeidet med å reetablere en tidligere versjons program for skulpturerte flater være en del av denne masteroppgaven. Men i samtaler med veileder ble det enighet om at arbeidsmengden er for stor. Arbeidet med å reetablere støtte skulpturerte flater ender opp i samme størrelsesorden som en hel masteroppgave. Denne oppgaven inneholder i tillegg flere andre deloppgaver som krever mye tid og arbeid. Så dermed ble det vedtatt å se bort fra dette punktet i den opprinnelige oppgaveteksten. Konsentrasjonen rettes mot de andre deloppgavene og tilrettelegging for eksperimentering med autonom konstruksjon av baner.

På dette tidspunktet oppretter konstruksjonsverktøyet baner fritt ute i rommet, uavhengig av andre geometriske objekter. Når støtte for skulpturerte flater er lagt til GeoMod bør konstruksjonsverktøyet utvides slik at baner også kan gis relativt til flater. Videre trengs det verktøy for å generere en bane som ligger på ei flate og som går på tvers av generator og direktrisene. Gjerne beregna basert på skjæring mellom den skulpturerte flata og et plan. Man kan også ta et skritt videre og beregne skjæringa mellom to skulpturerte flater. Dette er en rimelig stor oppgave i seg selv, og er noe framtidige masterstudenter kan få bryne seg på.

6 Dynamisk linking

Dynamisk linking er en metode for å dele applikasjoner og subsystem inn i mindre deler som kan kompileres, testes, gjenbrukes, administreres, rulles ut og installeres separat [27, kapp 4.1]. Flere av disse prinsippene er nyttige for et modulært system som GeoMod. Et mål er at hovedprogrammet kun skal inneholde grunnleggende funksjonalitet for å modellere geometri. Mens mer spesialiserte verktøy, modeller og visninger skal kunne lastes inn, uten avbrudd, mens programmet kjører. GeoMod har verktøy for å laste inn tre hovedtyper programtillegg eller plugins dynamisk. 'Views Manager' behandler ulike visninger og kameramoduler, 'Database Manager' arbeider på frittstående geometriske modeller slik som pyramider og kuber. 'Tools Manager' laster inn mer kompliserte verktøy. Verktøy består ofte av grafiske brukergrensesnitt som kan opprette, manipulere og styre geometriske modeller, både modeller som følger med verktøyet og eksisterende GeoMod-modeller.

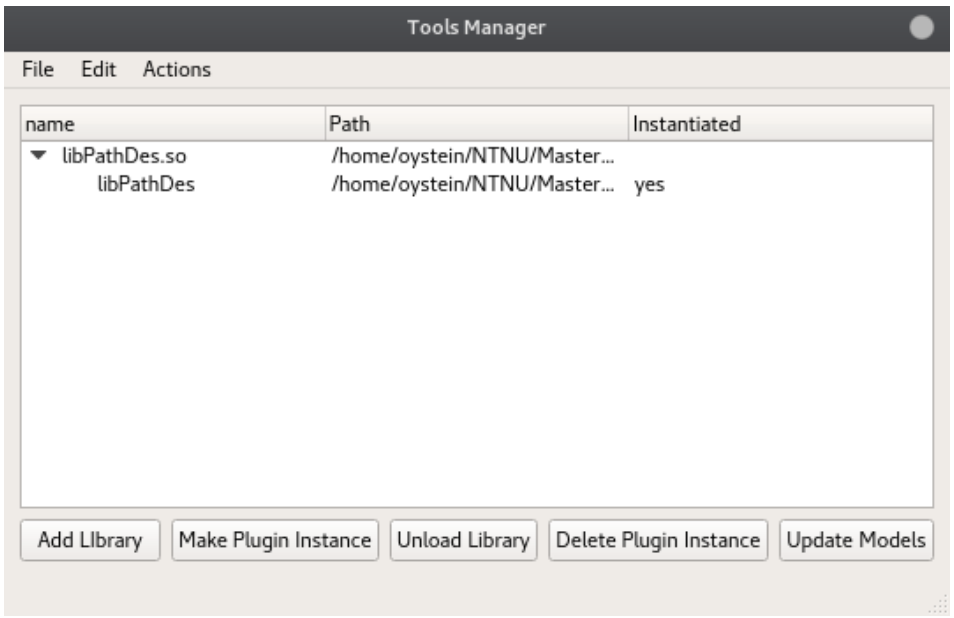
Arbeidet med den geometriske modelleren GeoMod har pågått i flere tiår. Datostemplinger i kodefiler viser at mye av grunnfunksjonalitet i dagens versjon av GeoMod ble utvikla rundt år 2000. Historisk var GeoMod basert på Qt versjon 3 og tett knytta opp mot Linux/UNIX. Utdatert funksjonalitet har blitt bytta ut, omskrevet og modernisert. Henviser til Aukland og Finnstrom sin masteroppgave [1] for flere detaljer rundt moderniseringsprosessen og dagens implementasjon av dynamisk linking. Tony Gjendahl har i sin prosjektoppgave, høsten 2016, videreført arbeidet og omgjort mange av GeoMods modeller og verktøy fra statisk til dynamisk linka bibliotek.

Plattformuavhengighet har vært et mål gjennom hele prosessen. De tre studentene som i sine prosjekt- og masteroppgaver arbeider GeoMod i 2016-2017 har alle benytta UNIX-lignende operativsystem. Nærmere bestemt Ubuntu, Arch Linux og Mac OS X. Med jevne mellomrom har det vært nødvendig å teste programvaren på Microsoft Windows, da det er dette operativsystemet NTNU MTP og veileder benytter. Noen framtidige studenter kommer mest sannsynlig også til å benytte Windows. Stort sett virker programmet på alle plattformene, mye takket være kryssplattformstøtte i Qt og C++. Ved noen tilfeller har testing på Windows vært med på å avdekke feil i programmet, ofte små feil som må fikses uansett plattform.

Utvikling av konstruksjonsverktøyet for baner har pågått siden starten av min prosjektoppgave høsten 2016. Den modulære designfilosofien bak GeoMod og idéen om dynamisk linking har ligget i bakhodet helt siden starten. Men for å lette utviklingsarbeidet, og gi prosjektoppgaven en mykere start, ble verktøyet linka inn statisk sammen med hovedprogrammet i Qt Creator-prosjektet 'entrance_06'.

Tilretteleggingа for dynamisk linking består i hovedsak av å holde koblinger til resten av GeoMod på et minimum. Koblinger geometrirelatert kode i MaxLib er nødvendig, men koblinger til det grafiske brukergrensesnittet i entrance_06 er forsøkt unngått. Verktøyet lever for seg selv i ei egen mappe og benytter kun standardfunksjonalitet i Qt. Det er en fordel at den dynamisk linka utgaven har samme kodebase som den statisk linka utgaven, og at de to versjonene kan brukes om hverandre.

Implementasjonen av dynamisk linking i konstruksjonsverktøyet for baner er lagt til qmake-prosjekt med navn 'PathDes' plasser i 'Tools/99_PathDes'. Implementasjonen er inspirert av, og basert på Gjendahls 'TestTool' og 'TestTool_If'. Koblinga mellom det opprinnelige konstruksjonsverktøyet og GeoMods system for dynamisk linking er lagt til omslutningsklassen med navn 'PathDes_If'. Den implementerer PluginInterface, altså den arver fra den abstrakte klassen PluginIterface og implementerer alle virtuelle funksjoner. Omslutningsklassen oppretter et objekt av klassen 'PathDesArea' og et av 'PathDesCtrl', og gjør dem tilgjengelig for resten av programmet. Figur 12 viser brukergrensesnittet Tools Manager som brukes til å laste inn konstruksjonsverktøyet dynamisk.



Figur 12: Viser Tools Manager-vinduet med konstruksjonsverktøyet lasta inn og initialisert

Klassen PathDes_If kommer med fullverdige implementasjoner av funksjoner relevant for plugins som skal lastes inn ved hjelp av Tools Manager. Det er i hovedsak tre funksjoner som er interessante: 'newInstance()' oppretter en ny instans av pluginen, og returnerer en peker til denne. Funksjonen 'getTool()' returnere pekeren til PathDesCtrl, konstruksjonsvektøyes grafiske kontrollpanel. Den tredje funksjonen 'getModels()' returnerer ei liste av modellpekere som skal legges til i GeoMods globale modellliste. Først og fremst gjelder dette modellen for konstruksjonsområdet, PathDesArea, i tillegg til eventuelle sekundære modeller. For øyeblikket er det inkludert en modell brukt til visualisering av posisjon og orientering (mer om posisjon og orientering i kapittel 7). I framtida kan flere modeller inkluderes, hvis nødvendig. Kompilatoren krever at hele interfacet implementeres. PathDes_If inkluderer dermed også en minimal implementasjon av andre funksjoner. De produserer en debugutskrift som forteller at funksjonen har blir kalt, og returnerer nullpeker hvis nødvendig.

Symbolske lenker sikrer at den dynamisk linka utgaven av konstruksjonsverktøyet har samme kodebase som den statisk linka utgaven. Ei symbolsk lenke inneholder en tekststreng som operativsystemet automatisk tolker som en sti til ei fil eller mappe. Lenker til mappene 'autopath', 'ctrl_path_des', 'geom_path_des' og 'parameterization' er oppretta fra 'GeoMod/path_des/' til 'Tools/99_PathDes/PathDes/'. Symbolske lenker er også støtta på Microsoft Windows fra og med Windows Vista, men kun på NTFS-partisjoner. Microsoft har implementert symbolske lenker for å gjøre migrasjon fra UNIX enklere, og lenkene oppfører seg på samme måte som på UNIX [26]. Symbolske lenker er ikke en del av den tradisjonelle arbeidsflyten på Windows, og oppsettet kan være noe kronglete. For å opprette lenker kjøres kommandoen 'mklink' i en terminal med administratorrettigheter. Dermed kan det være at Windows-brukere skal lage en kopi av mappene i stedet. Det kan by på utfordringer med å holde begge versjonene oppdatert. Ei enkel løsning er å bestemme seg for å kun gjøre endringer på ett sted. Når den statisk linka versjonen fases ut forsvinner også denne problematikken.

Plassering av MaxLib-headerfiler bøy på noen utfordringer. Tradisjonen i GeoMod er å oppgi stien til lokale headerfiler relativt til gjeldene kodefil. Dette fungerer ikke for konstruksjonsverktøyet, da de to utgavene aksessere de samme kodefilene fra forskjellige mappesiter i ulike qmake-prosjekter. Problemet er løst med et flagg og preprosessormakroer. Uttrykket 'DEFINES += DYNAMIC_LINKING' i prosjektfila til den dynamisk linka utgaven setter flagget, og resten av kodefilene omslutter #include-setningen sine med en #ifdef-setning på formen vist i listing 2. Hvis den statisk linka versjonen fases ut, vil det ikke lengre være behov for denne makroen.

```
#ifdef DYNAMIC_LINKING
  #include "../../../MaxLib/net_G/geomnode.h"
  #include "../../../MaxLib/net_G/geomedge.h"
  #include "../../../MaxLib/math/vec.h"
#else
  #include "../../MaxLib/net_G/geomnode.h"
  #include "../../MaxLib/net_G/geomedge.h"
  #include "../../MaxLib/math/vec.h"
#endif
```

Listing 2: Preprosessormakroer som gjør at både den statisk og dynamisk linka utgaven av konstruksjonsverktøyet finner de nødvendige headerfilene til MaxLib.

Løsninga som bruker flagg og makroer er ikke ideell. Headerfilene til MaxLib bør i stedet legges inn ved å bruke qmake-variabelen 'INCLUDEPATH'. Variabelen spesifiserer hvilke mapper som '#include'-uttrykk skal søke igjennom når prosjektet kompileres [24]. Dette kan godt gjøres i resten av GeoMod også. Både i hovedprogrammet og i andre plugins.

Plugin av typen tool trenger mer fleksibilitet med hensyn på modeller enn det de andre plugintypene krever. Det er i hovedsak to klasser med plugins som skaper utfordringer: Plugins som oppretter modeller etter initialisering, og plugins som har mer enn én modell. Den første utfordringa ble løst ved å legge til en ny knapp i Tools Manager. Knappen 'Update Models' henter ut modellene fra pluginen og oppdaterer den globale modellista. Opprinnelig kunne en verktøyplugin kun inneholde én modell. Konstruksjonsverktøyet inneholder allerede flere modeller. For å gjøre det mulig å legge alle til i den globale modelllista, er funksjonen 'getModels()' i PluginInterface utvida til å returnere en vektor med pekere til modeller i stedet for en peker til en enkelt modell. Kode for å sikre bakoverkompatibilitet med andre plugins er også inkludert.

Fortsatt foregår mesteparten av utviklingsarbeidet på den statisk linka versjonen av konstruksjonsverktøyet. Arbeidet blir lettere, og man kan raskt teste ut nyimplementerte funksjoner og feilrettinger. For å gjøre det mer attraktivt å arbeide med verktøy og modeller som er linka inn dynamisk, trengs det en enklere metode med færre klikk for å velge og laste inn plugins. En mulighet er å bruke ei konfigurasjonsfil med liste over plugins som skal lastes inn automatisk når programmet starter. Seinere kan man utvide, og lage grafiske brukergrensesnitt for å opprette og organisere plugginlister, både den som lastes inn når programmet starter, men også lister over plugins som skal lastes inn for å utføre bestemte arbeidsoppgaver.

7 Posisjon og orientering

Noen baner beskriver veien et fartøy skal kjøre, andre baner forteller robotarmer og verktøymaskiner hvordan de skal føre verktøyet sitt for å utføre ønska arbeid. Enten det er å plukke opp gjenstander, føre måleinstrumenter og kamera, eller det er å skjære eller frese bort materiale fra et arbeidsstykke. Man kan sammenligne banen med en vei eller jernbanelinje. I seg selv er den kun ei samling geometri som befinner seg et sted i det tredimensjonale rommet. For å legge til rette for videre eksperimentering med autonome fartøy som følger baner bør det utvikles rutiner for å beregne og lagre et fartøys posisjon langs en bane.

I tillegg til å vite hvor på banen fartøyet befinner seg, må man vite hvordan fartøyet skal orienteres. Undervannsfartøyet må kanskje legge seg over på siden for å komme seg igjennom en trang passasje mellom to hindre. Vinkelen mellom verktøyet og arbeidsstykke er viktig for fleraksa freser og andre numerisk kontrollerte verktøymaskiner som skjærer bort materiale. Arbeidsstykket må få den ønska formen, og avkutta materiale må få plass til å ramle bort. For MIG-sveising er både posisjonen og orienteringa til sveiseverktøyet viktig for å oppnå et godt resultat. Sveisetråden må treffe arbeidsstykket på riktig sted med riktig vinkel for å fordele dekkgassen optimalt og unngå forurensninger som svekker sveisens styrke.

Konstruksjonsverktøyet trenger støtte for baner med posisjon i tillegg til orientering. I arbeidet med denne masteroppgaven har det blitt utvikla et underverktøy med navn 'PathMotion' som danner ei kobling mellom bane og fartøy. Verktøyet tilbyr både funksjonalitet for å beregne posisjon og orientering langs banen. Orientering beregnes etter en metode spesifisert av veileder, basert på støttegeometri. Detaljerte beskrivelser følger.

7.1 Beregning av posisjon

Verktøyet kan beregne posisjon både langs rettlinja og kurva kanter. Posisjonen parametriseres med parameter t som følger standardkonvensjonen med vilkårlig mange steg fra $t = 0$ til $t = 1$. Informasjon om en kant finnes i objekter av GoeMod-klassen GeomEdge. Verktøyet bestemmer om kanten er ei kurve ved å sjekke om funksjonen 'GeomEdge::getCurve()' returnerer nullpeker. Hvis kanten er rettlinja, hentes endenodene ut, og punktet bregnes ved hjelp av lineær interpolasjon som beskrevet i teorikapittelet (kap. 3.1). Et utvalg av kurvetyper er implementert i 'Maxlib/curves/curve.cpp'. Punkt på kurver bergenes i henhold til kurvetypen, og lagres automatisk i lista 'toolPathPoints' når kurven tegnes. For bezierkurver beregnes punktene etter deCasteljaus algoritme som beskrevet i teorikapitlet (kap. 3.2). Posisjonen gjengis som en offsetvektor i forhold til en base. Man har valget

om offsetvektoren skal beregnes relativ til banens lokale base eller i forhold til GeoMods globale koordinatsystem.

7.2 Støttekanter for å bestemme orientering

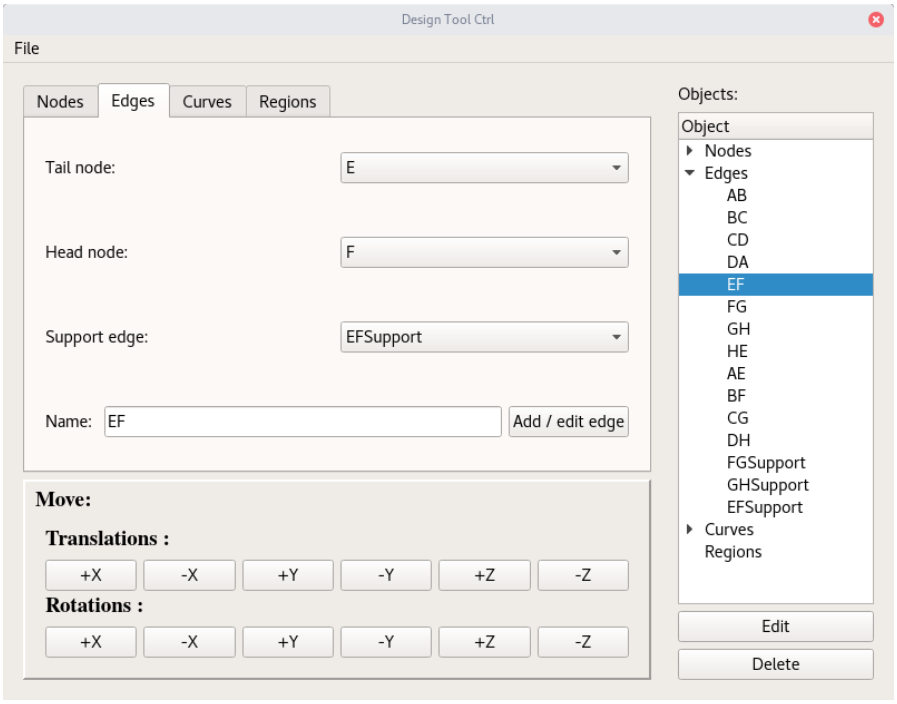
Gitt kun en bestemt posisjon langs banen finnes det uendelig mange orienteringer. For å få ei entydig orientering er flere grensebetingelser nødvendig. Her er denne utfordringa løst ved å bruke støttekanter til å framtvinge orientering. Støttekanter opprettes som en del av konstruksjonsarbeidet og er en del av banen. Framover i dette kapittelet omtales to typer kanter: banekanter, ofte forkorta 'kanter', og støttekanter. Vanligvis er støttekanter rette linjer mellom to punkt, men kurva kanter er også støtta. Hvis en kant ikke er assosiert med en støttekant, kan ikke orientering beregnes, så den opprinnelige orientering beholdes. Det er ingen forskjell på implementasjonen av de to kanttypene, begge er vanlige instanser av klassen GeomEdge. Bare bruksområde er forskjellig, og dermed er det heller ikke implementert noen restriksjoner på bruk og blanding av de to kanttypene. Det er opp til brukeren å vise sunn fornuft og bare bruke støttekanter slik de er tenkt. Støttekantene i eksempelbanen er markert med gul farge i figur 15.

Konstruksjonsverktøyets er utvida med funksjonalitet for å registrere støttekanter på banekanter. Støttekanten opprettes som en vanlig frittstående kant. Deretter kan den registreres på en banekant ved hjelp av nedtrekksboksen i DesignTools kantfane, vist i figur 13. Støttekanten kan registreres enten når banekanten opprettes, eller legges til på et seinere tidspunkt ved å bruke funksjonaliteten for å redigere kanter.

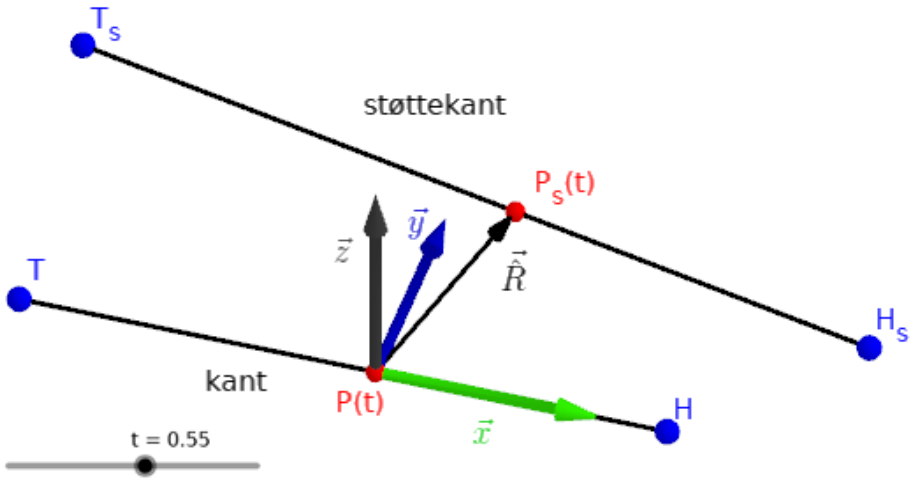
7.3 Beregning av orientering

Orientering beskrives ved hjelp av en ortonormal vektorbase som danner ei orienteringsramme. Tradisjonelt legges x-aksen i kjøretøyets 'framover'-retning, det vil ofte si i fartsretninga eller i fartøyets lengste retning. Z-vektoren legges i retning 'oppover' i den grad 'oppover' lar seg definere. Den gjenstående y- eller 'sideveis'-aksen må stå vinkelrett på de to andre. Dermed er også denne retninga bestemt, da det er ønskelig å ha et høyrehåndssystem.

Orientering beregnes for en gitt verdi av parameteren $t \in [0, 1]$. Først beregnes kurvens tangentvektor $\vec{T}$. Den estimeres ved å ta differansen mellom to nærliggende



Figur 13: Muligheter for registrering av støttekanter via nedtrekksboks i DesignTools kantfane.



Figur 14: Beregning av orienteringsramme ved hjelp av støttekant.

punkt på kurven (ligning 7). Lengden må normaliseres.

$$\begin{aligned} \vec{\hat{T}} &= P(t+\delta) - P(t) \\ \vec{T} &= \frac{1}{|\vec{\hat{T}}|} \cdot \vec{\hat{T}} \end{aligned} \tag{7}$$

Videre trengs en hjelpevektor som peker mot støttekanten. Retningsvektoren $\vec{R}$ beregnes utfra differansen mellom et punkt $P(t)$ på banekanten og et samsvarende punkt $P_s(t)$ på støttekanten (ligning 8). Også her må lengden normaliseres.

$$\begin{aligned} \vec{\hat{R}} &= P_s(t) - P(t) \\ \vec{R} &= \frac{1}{|\vec{\hat{R}}|} \cdot \vec{\hat{R}} \end{aligned} \tag{8}$$

Ved hjelp av disse to vektorene kan orienteringsramma bestemmes utfra ligning 9.

$$\begin{aligned} \vec{x} &= \vec{T} \\ \vec{z} &= \vec{R} \times \vec{x} \\ \vec{y} &= \vec{x} \times \vec{z} \end{aligned} \tag{9}$$

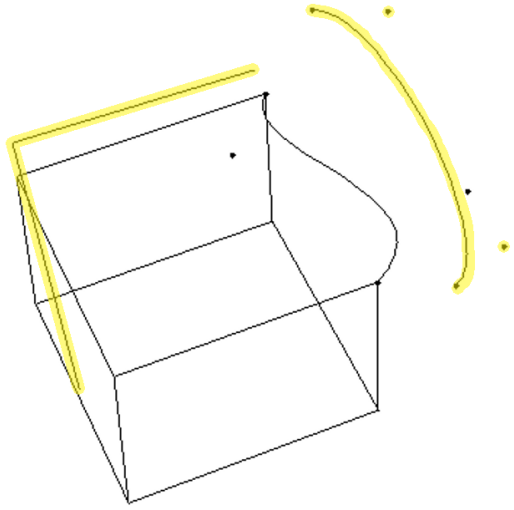
Tangentvektoren brukes uforandra til å bestemme framoverretninga ($\vec{x}$). Oppoverretninga $\vec{z}$, normalt på $\vec{x}$ og $\vec{R}$, bestemmes av et vektorprodukt. Retningsvektoren $\vec{R}$ står ikke nødvendigvis vinkelrett på $\vec{x}$ og $\vec{z}$, og kan derfor ikke brukes som $\vec{y}$ direkte. Ett nytt vektorprodukt sørger for å gi oss den siste vektoren i høyrehåndssystemet. Aksevektorene får alle enhetslengde, da både $\vec{T}$ og $\vec{R}$ har lengde én.

Framgangsmåten har noen likhetstrekk med metoden brukt i [16] for å utlede kamera-ramma brukt i OpenGL og fagområdet datagrafikk (engelsk: Computer Graphics). Punktet på banekanten er analogt med kameraposisjonpunket, og punktet på støttekanten med senter-av-oppmerksomhet-punket. Da blir retningsvektoren ($\vec{R}$) analog med aksen kameraet ser langs ([16, $\vec{w}$]), og tangentvektoren ($\vec{T}$) med kameraets vertikale retning ([16, $\vec{y}$]).

Fordelen med å estimere tangenten framfor å benytte mer nøyaktig matematikk gjør at metoden fungerer for alle typer linjer og kurver. Alt men trenger er muligheten for å beregne punkt på kurven for ulike parameterverdier. Beregning av nøyaktig tangent krever en egen implementasjon for hver kurvetype. Dette finnes for øyeblikket ikke i GeoMod. Man kan, hvis ønskelig, legges til en virtuell metode 'getTangent(...)' i klassen 'Curve'. Da må hver kurvetype implementere sin variant av metoden. Så vil arv og polymorfi i C++ sørge for at implementasjonen til den gitte kurvetypen blir benytta.

7.4 Eksempelbane

Demonstrasjon skal vise bruksområdene til PathMotion og mulighetene verktøyet tilfører. Til demonstrasjon brukes banen vist i figur 15. Banen danner en modifisert kube hvor en av sidekantene er bytta ut med en kubisk kurve. Den er en modifisert utgave geometrien Furevik bruker i sin 'MotionCtrl, vist i [8, figur 6.2]. En av de rette støttekantene er bytta ut med en kurve for å vise at PathMotion kan nytte kurva støttekanter. En bane rundt om på en kube er ved første øyekast mer relevant for roboter og verktøymaskiner enn for kjøretøy som beveger seg rundt i rommet. Undervannsfartøy utstyrt med manipulator kan komme opp i en situasjon hvor de må føre verktøy langs en lignende bane. Uansett fungerer banen godt i denne demonstrasjonen, da den både viser overgangen mellom flere kanter i et knutepunkt, og hvordan støttekanter virker.



Figur 15: Banen brukt for å demonstrere PathMotion. Den danner en modifisert kube hvor en av sidekantene er bytta ut med en kubisk kurve. Støttegeometri er utheva med gul farge.

7.5 PathMotion

Informasjon om posisjon og orientering må lagres et sted. En mulighet er å la posisjon og orientering være en del av selve banen, og lagre det direkte i Path-klassen. Men det kan tenkes situasjoner der en bane benyttes av flere verktøy eller kjøretøy

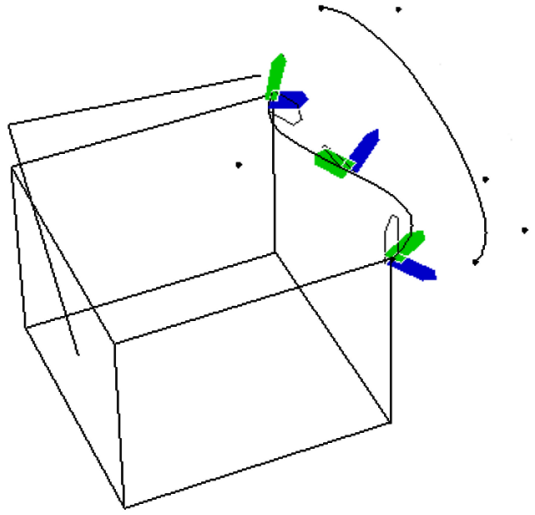
samtidig. Av disse hensyn, bør posisjon og orientering lagres utenfor baneklassen. En mulighet er å la fartøy lagre denne informasjonen selv. Men hvis hvert fartøy skal gjøre dette på egenhånd, er det en fare for hver modellutvikler ender opp med å finne opp hjulet på nytt. I motsatt fall, hvis utviklerne samarbeider, kan det oppstå duplisert kode med små endringer relatert til de ulike fartøya. For å komme seg rundt disse utfordringene er det utvikla et standardverktøy for å holde rede på posisjon og orientering.

Verktøyet tilbyr en koblingsklasse mellom banen og fartøys- eller vektøysmodellen, samt et grafisk kontrollpanel og et grensesnitt for autonome algoritmer. Koblingsklassen 'PathMotion' beregner og holder styr på posisjon og orientering langs banen. Klassen assosieres med en bane ved hjelp av en peker til instans av baneklassen Path. Den lagrer også parameterverdien og en peker til den valgte banekanten. Funksjoner for å beregne posisjon og orientering etter framgangsmåten beskrevet ovenfor er inkludert. I tillegg tilbys funksjonalitet for å hente ut gjeldene kant og parameterverdi. Dessuten kan den også liste opp alle kanter som møtes i en node.

Et demonstrasjonsverktøy, forma som et aksekors, er inkludert. Modellen har ei grønn pil i x-retning, ei blå pil i y-retning. I z-retningen er pila hvit. Figur 16 viser PathMotion og demonstrasjonsverktøyet i bruk. Bevegelse er forsøkt illustrert ved å legge tre skjermbilder oppå hverandre. Et bilde for parameterverdi $t = 0.0$, et for $t = 0.5$ og sist $t = 1.0$. Den grønne pila følger den kurva banekanten, mens den blå hele tida er retta mot støttekanten.

Når fartøyet beveger seg langs banen, vil det etter en tid nå et knutepunkt i enden av den valgte kanten. Nå må det velges hvilken kant fartøyet skal fortsette å kjøre på. PathMotion tilbyr funksjonalitet for å liste opp nabokanter i begge ender av den valgte kanten. Funksjonaliteten kan nyttes både i det inkluderte grafiske brukergrensesnittet og av autonome algoritmer i egenutvikla programvare. Den nye kanten velges via funksjonen 'PathMotion::setCurrentEdge(...)'.

Det viser seg at retningen en kant transverseres er av betydning i overgangen fra en kant til den neste. I overgangen ønsker man å beholde samme posisjon mens parameterverdien går fra $t = 1$ på den ene kanten til $t = 0$ på den neste. Omvendt for overganger i motsatt ende av kanten. Det er opp til banekonstruktøren å legge ut noder og bestemme hvordan kanter skal koble dem sammen. Han må passe på og ta hensyn for å oppnå kontinuerlige overganger mellom nabokanter. Nabokanter må ha en node felles. Gap mellom kantene gjør det umulig å komme seg til neste kant. Det viser seg også at retninga kantene kobles sammen har betydning. Ideelt



Figur 16: Viser posisjon og orientering langs eksempelbanen representert ved et aksekors. Bevegelse er forsøkt gjengitt ved å legge tre skjermbilder oppå hverandre for ulike verdier av parameteren t .

sett kobles kanter sammen 'hode-mot-hale'. Dette lar seg greit gjøre for enkle baner, for eksempel en bane som beskriver en vei fra A til B med et begrensa antall sidespor. Men mer kompliserte baner, inkludert eksempelbanen vist ovenfor, har sykler. Da ender noen kanter opp med å ligge 'hode-mot-hode' eller 'hale-mot-hale'. Da kan overgangen fra parameter $t = 1$ på den første til $t = 0$ på den neste kanten gi posisjon feil ende av kanten. Dette medfører en uheldig effekt for fartøyet som skal følge banen. Kontinuerlig overgang mellom er viktig, så problemet må løses.

Utfordringa er løst ved å gjøre det mulig å transversere kanter begge veier. Implementasjonen inkluderer også programlogikk for å automatisk velg retning beregna utfra kantenes sammenkoblingsmetode. I PathMotion kan transverseringsretninga til den valgte kanten settes. Valget ligger mellom 'framover' og 'revers'. En kant er som kjent retta fra hale- til hodenoden (jamfør "Geometri i GeoMod", kap. 2.1.1). Dermed er det naturlig at en kant som transverseres framover gir posisjon på halenoden for parameterverdi $t = 0$, og posisjon på hodenoden for $t = 1$. Motsatt i revers. Videre er det behov for et begrep som kan navngi endenodene basert på parameterverdi, uavhengig av transverseringsretning. For enkelhets skyld, og i mangel på bedre ordlegging, er det tenkt en sammenligning mellom en kant og ei tallinje. På tallinja ligger 0 til venstre for 1, dermed brukes begrepa 'venstre' og

'høyre' om endene til kanten. Den venstre noden ligger i den enden av kanten der parameteren $t = 0$, uavhengig av retninga. Høyre node ligger ved $t = 1$.

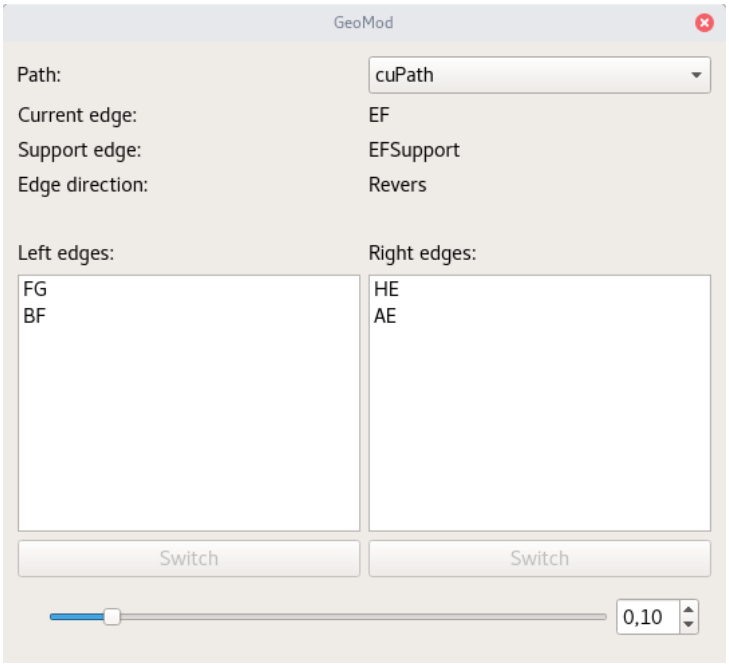
Logikken som automatisk sørger for naturlige overganger er lagt til bytteknappen i det grafiske brukergrensesnittet. Logikken skal avgjøre transverseringsretninga til kanten man bytter til. For overganger som finner sted ved høyreenden av den valgte kanten må man første bestemme høyrenoden. Den er bestemt av retninga til den valgte kanten. Hvis retninga er framover, er settes høyrenoden lik kantens hodenode. Er retninga revers settes den lik halenoden. Så sammenliknes høyrenoden med nabokantens noder. Er høyrenoden lik halenoden til naboen, skal den nye kanten transverseres framover. Hvis ikke skal man gå i revers. Til slutt settes den valgte kanten til den nye sammen med den beregna retninga. Slideren med parameterverdi stilles tilbake til 0. Logikken for bytter i venstreenden av kanten følger samme struktur, men utfører sammenligninger med noden i venstre ende, og setter parameteren til 1 i stedet. Programkoden, gjengitt i listing 3, tar med alle detaljene og forsøker å gjøre det lettere å lese.

```
void switchRightClicked() {
  GeomNode *rightNode;
  if(isCurrentEdgeDirectionForward()) {
    rightNode = getCurrentEdge()->headNode();
  } else {
    rightNode = getCurrentEdge()->tailNode();
  }

  bool forward = ( rightNode == nextEdge->tailNode() );

  setCurrentEdge(nextEdge, forward);
  sliderP->setValue(0.0);
}
```

Listing 3: Programkode med logikken som automatisk sørger for naturlig overgang til neste kant på høyresida.

7.6 Grafisk brukergrensesnitt

PathMotion kan styres av det grafiske brukergrensesnittet, 'PathMotionCtrl', vist i figur 17. Kontrollpanelet lar brukeren styre en modells bevegelse langs en bane. Slideren nederst i vinduet styrer bevegelsen langs den valgte kanten ved å endre verdien til parameteren t. Bytte mellom nabokanter skjer via kantlistene, ei for hver enda av den valgte kanten med tilhørende bytteknapper. Bytteknappene blir aktivert når slideren kjøres helt til endes. Øverst inneholder brukergrensesnittet merkelapper som gir informasjon om valgt kant, hvilken støttekant den er assosiert med, og retninga den transverseres.



Figur 17: Det grafiske brukergrensesnittet 'PathMotionCtrl' som brukes for å bevege en modell langs en bane.

Kantlistene oppdaterer seg automatisk hver gang konstruksjonsverktøyet endrer geometrien i banene. Så hvis noen legger til en ny, eller fjerner en kant kobla til den valgte kanten, vises oppdateringa i listene med en gang. Skulle uforutsette endringer i bane oppstå, så oppdaterer kantlistene seg også hver gang slideren når et av endepunkta sine. Dette kan skje hvis noen endrer banen ved å bruke rutiner utenfor konstruksjonsverktøyets kontroll.

Det grafiske grensesnittet i PathMotionCtrl er utvikla for å kunne assosieres med og styre én instans av PathMotion. Men for øyeblikket er verktøyet noe forenkla. Det styrer demonstrasjonsverktøyet langs banen valgt i nedtrekksboksen. Dette fungerer bra til demonstrasjonsformålet, men bør utvides i framtida. En utvidelsesmulighet er å opprette et nytt "lag" i brukergrensesnittet hvor man kan velge én modell og én bane, for så å koble dem sammen ved å opprette en instans av PathMotion med tilhørende kontrollpanel. Da kan hvert modell-bane-par styres uavhengig av hverandre med hvert sitt grafiske kontrollpanel. En anna mulighet er å ha et generelt kontrollpanel som passes til og bygges inn i modellen selv. Det gir utvikleren av modellen flere tilpasningsmuligheter og bedre kontroll over modellspesifikke aspekter ved transversering av banen.

PathMotion, og særlig GUI-et i PathMotionCtrl, henter mye inspirasjon fra Fureviks 'MotionCtrl' [8, kap. 5.2]. Målet med PathMotion er å tilby utvida, men i hovedsak tilsvarende, funksjonalitet som Fureviks verktøy. MotionCtrl ble utvikla før arbeidet med konstruksjonsverktøyet for baner var påbegynt. Derfor inneholder de to noe overlappende funksjonalitet (oppretting av noder og kanter, for å nevne noen). Det viste seg å være lettere å lage en ny utgave, framfor å videreutvikle eller skrive om Fureviks verktøy. På denne måten kan begge verktøya være med GeoMod videre, og brukes hver for seg. Videre skiller PathMotion seg fra MotionCtrl ved å dele beregninger og logikk fra det grafiske brukergrensesnittet. Det gjør PathMotion bedre egna for bruk sammen med autonome algoritmer.

7.7 Grensesnitt for autonome algoritmer

Fartøy må få beskjed når posisjon og orientering endres. PathMotion tilbyr to grensesnitt ut mot fartøy, verktøy og andre modeller. Grensesnitta tilbyr ulike metoder som forsøker å balansere grad av kontroll opp mot enkel bruk. Det enkle grensesnittet tilbyr liten grad av kontroll, og lar PathMotion ta hele oppgaven. Mens den avanserte overlater mer av kontrollen til brukeren.

Den enkle metoden innebærer å gi PathMotion full kontroll over en utvida base. Modellen sender en peker til basen via funksjonen 'PathMotion::setToolBaisisP(...)'. Hver gang parameteren, og dermed posisjon og orientering, endres vil PathMotion oppdatere basen. Posisjon i det globale koordinatsystemet beregnes og tilordnes den utvida basens offsetvektor. Basens aksevektorer settes lik de beregna orienteringsvektorene. Hvis man ønsker posisjon gitt i et anna koordinatsystem må det avanserte grensesnittet benyttes. Den enkle metoden er særlig nyttig tidlig i arbeidet med å utvikle nye fartøy- og verktøymodeller. Den tilbyr rask og enkel testing, og brukes derfor av det inkluderte demonstrasjonsverktøyet. Etterhvert som modellen ferdigstilles trengs det mer detaljert kontroll med hvordan modellen skal reagere på endinger i posisjon og orientering. Da er tiden inne for å gå over til den avanserte metoden.

Grensesnittet i den avanserte metoden baserer seg på Qt signals and slots. PathMotion tilbyr en rekke signaler modeller kan koble seg til. Hver gang parameteren endres, beregnes ny posisjon som sendes ut med signalet 'positionChanged(MathVec)'. Modeller som lytter på dette signalet mottar posisjonen via signalets parameter i form av en offsetvektor. Hvis den valgte kanten er assosiert med en støttekant, sendes det også ut et signal med oppdatert orientering. Orienteringa beregnes som tre retningsvektorer, og signalet 'orientationChanged(ExtBasis)' sender den ut via parameteren av typen ExtBasis. Videre kan modellen selv ut-

føre nødvendige transformasjoner og beregninger. For eksempel skal orienteringa tolkes ulikt avhengig om banen følges av et kjøretøy eller en verktøymaskin. Kjøretøyet vil ofte legge orienteringas x-vektor i fartsretninga, mens verktøymaskinen gjerne vil rette verktøyet langs z-vektoren. I tillegg kan man få beskjed om at parameteren eller den valgte kanten er endra via signalet 'parameterChanged()' og 'currentEdgeChanged()' respektivt. Listing 4 lister opp hele grensnittet med nødvendige detaljer.

```
void parameterChanged();
void currentEdgeChanged();
void positionChanged(MathVec offset);
void orientationChanged(ExtBasis orientation);
```

Listing 4: Signaler brukt av PathMotions avanserte grensesnitt for autonome algoritmer

7.8 Finere kontroll med flere støttekanter

Fartøy som beveger seg i trange områder, eller verktøymaskiner som maskinerer ut fine detaljer i et hjørne trenger nøyaktig kontroll over orientering. Da blir én støttekant for hver banekant altfor grovt. Det trengs en metode som gjør det mulig å justere orientering enda finere. I mange tilfeller kan dette løses ved at man deler opp kant i mindre deler og gi hver av dem sin egen støttekant. Dette krever en del merarbeid, man i må konstruere nye noder og kanter langs den opprinnelige kanten i tillegg til nye støttenoder og -kanter. Særlig hvis den opprinnelige kanten allerede er kort, blir dette fort tungvint. Å kunne assosiere flere støttekanter med én banekant gir ei lignende men bedre løsning. Man må fortsatt konstruere de nye støttekantene, men banekantene forblir som de er. Tidsmangel har ikke gjort det mulig å implementere og teste denne funksjonaliteten ennå, men tanker om mulige implementasjonsveier legges fram nedenfor.

Ta for eksempel kanten e med lengde $|e| = 3$ og de tre støttekantene s_1, s_2 og s_3 hver med lende én. I stedet for en tekststreng med navnet på støttekanten, lagres ei liste over støttekanter. Videre trengs en metode for å holde orden på hvilke støttekant en gitt parameterverdier på banekanten svarer til. Ved å ta utgangspunkt i kantenes lengde, kan man dele opp støttekantenes virkeområde. Grovt forklart, med utgangspunkt i eksemplet, kan parameter $t_e \in [0, \frac{1}{3}]$ tilordnes s_1, $t_e \in [\frac{1}{3}, \frac{2}{3}]$ til s_2 og $t_e \in [\frac{2}{3}, 1]$ til s_3. Støttekanten traverseres med en skalert parameter. For s_1 i eksempelet brukes parameteren $t_{s_2} \in [0, 1]$, hvor $t_{s_2} = 0$ svarer til $t_e = \frac{1}{3}$ og $t_{s_2} = 1$ til $t_e = \frac{2}{3}$. Man får finere kontroll på orientering, og merarbeidet går kun ut på å opprette flere støttekanter og legge dem til i lista. Videre må detaljer rundt grenseoppdeling formaliseres for å oppnå en korrekt implementasjon.

Behov for mer eksplisitt kontroll med parameterverdiene som bestemmer overgangen mellom støttekantene kan være ønska. Ei løsning som gir mer kontroll, men som også kompliserer konstruksjonsarbeidet, er å oppgi parametergrenser sammen med støttekantene. Separate lister med samsvarende indekser kan brukes til å lagre verdiene av t_e som svarer til $t_s = 0$ og $t_s = 1$. Traversering av støttekanten skaleres utfra disse verdiene. Denne løsninga trenger mer bokholderi, men lar seg gjennomføre. Å oppgi alle parametergrensene manuelt er langtekkelig og kjedelig for et menneske. Datamaskiner er derimot laga for å utføre beregninger, så autonome algoritmer med kunstig intelligens bør ikke ha dette problemet.

7.9 Flere utvidelsesmuligheter

For å holde rede på hvor man han vært tidligere, kan man innføre ei ordna liste med peker til de forrige kantene man beveget seg langs. Tilsvarende liste for framtidige kanter kan brukes av et planleggingsprogram til å fortelle hvilken rute fartøyet skal ta og hvilke kanter som må følges. Kanskje bør dette også utvides med muligheter for å ta opp, lagre og spille av sekvenser.

8 Flere baner og skille mellom dem

Autonome fartøy må evaluere flere ulike baner for å nå sine mål og utføre oppgaven sin best mulig. Flere baner planlegges og settes opp mot hverandre for å finne den bane som for øyeblikket er best egna. Et utvalg alternative baner kan tas vare på for bruk i framtidige evalueringer. Fartøyet må også holde orden på hvor det er og hvor det har vært. Faktisk fulgt bane lagres og holdes oppdatert. Planlagt og faktisk fulgt bane sammenlignes for å avgjøre om fartøyet er på rett kurs. Uforutsette situasjoner kan oppstå og gjøre den valgte banen mindre gunstig. Avvik beregnes, og en overgangsbane konstrueres. Utfallsrom for videre navigasjon er også av interesse. Simuleringer av fartøyets posisjon etter en gitt tid ved forskjellige motor- og omgivelseparametere kan lagres og evalueres.

Særlig under eksperimentering med algoritmer for automatisk konstruksjon av baner er det nyttig å arbeide med flere baner. Variasjon i algoritmens parametere resulterer i ulike baner. Utvikleren må lett kunne visualisere og sammenligne banene. Både for å finne feil og svakheter ved algoritmen og for å evaluere ulike parametere. De beste banene kan lagres for seinere bruk, ugunstige baner kan velges bort og slettes. Kanskje behøves manuell finpuss for å gjøre banen optimal.

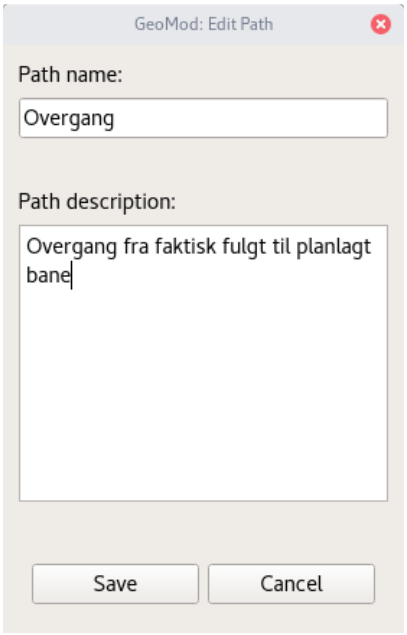
Konstruksjonsverktøyet har støtter flere baner og mulighet for å raskt bytte mellom dem. Alle banene er synlig i konstruksjonsverktøyets arbeidsområde. Verktøyet

kan opprette nye baner og ta bort baner som ikke brukes lengre. Informasjon om banene kan redigeres. Det er mulig å lagre baner til fil, og lastes inn igjen på et seinere tidspunkt. Nedtrekksboksen i konstruksjonsverktøyets startvindu, i vist i figur 18, velger den aktive banen. Altså hvilken bane konstruksjonsverktøyet arbeider på. Når den aktive banen endres, oppdateres resten av verktøyet seg umiddelbart. Oppdateringa er godt synlig i objekttreet i DesignTool.

Figur 18: Konstruksjonsverktøyets startvindu lar brukeren velge aktiv bane.

I tillegg til geometri, har hver bane et navn og en beskrivelse. Både navn og beskrivelse er implementert som en standard tekststreng lagra i baneobjektet. Endringer utføres enten i det grafiske brukergrensesnittet vist i figur 19 eller av ekstern programvare via funksjonene 'Path::setName(...)' og 'Path::setDescription(...)'. Det er ingen formelle krav til hverken navn eller beskrivelse. Men sunn fornuft og beste praksis bør følges. Navnet bør være kort, og brukes i hovedsak for å identifisere og skille ulike baner fra hverandre. Programvare kan nytte strengsammenligning og

regulære utrykk på navnet for å hente ut baner for bruk sammen med autonome algoritmer. I beskrivelsen kan brukeren utdype og skrive hva bane skal brukes til, hvorfor den ble konstruert, og eventuelle hensyn tatt under arbeidet. Autonom programvare kan bruke beskrivelsen til å identifisere seg selv og parameterne som ble brukt for å konstruere banen.



Figur 19: Vindu for endring av en banes navn og beskrivelse

Videre kan det utvikles rutiner og brukergrensesnitt for mer systematisk organisering av baner. Forskjellige klasser baner, eksempelvis planlagte, faktiske og overganger, kan behandles av skreddersydde verktøy. Baner kan utvides til å inneholde flagg og typebeskrivelser som forteller hensikten med banen og hvilke verktøy den skal behandles i. En grafisk visning som illustrerer utfallsrommet for videre navigasjon bør også utvikles. Men på dette tidspunkt og under denne oppgavens rammer var det ikke hensiktsmessig å sette av mye tid og arbeid på utvikling av slike brukergrensesnitt.

Algoritmer for automatisk konstruksjon av baner er ønska. Situasjoner der den faktiske banen avviker fra den planlagte kan oppstå. Da er det gunstig å automatisk kunne beregne en fortsettelse fra den faktiske banen tilbake til den planlagte. Korteste vei mellom sluttpunktet på den ene banen og startpunktet på den andre er ei rett linje. Å følge den rette linja vil være enkelt og intuitivt men medfører problemer med treghetskrefter og høy akselerasjon. Zhijie Xu m.fl. [32] bruker

bezierkurver til å beskrive overgangen mellom baner til en industriell robot som arbeider med limfuger. De undersøker hvordan hastigheten påvirkes av ulike banetype og -parametere. For et autonomt undervannsfartøy er hastighetsendringer av større betydning enn absolutt hastighet, likevel er resultatene i [32] nyttige. Fartøyet må bevege seg jevnt slik at manipulator med verktøy og/eller kamera kan utføre jobben sin optimalt.

9 Lagring av geometriske data på fil

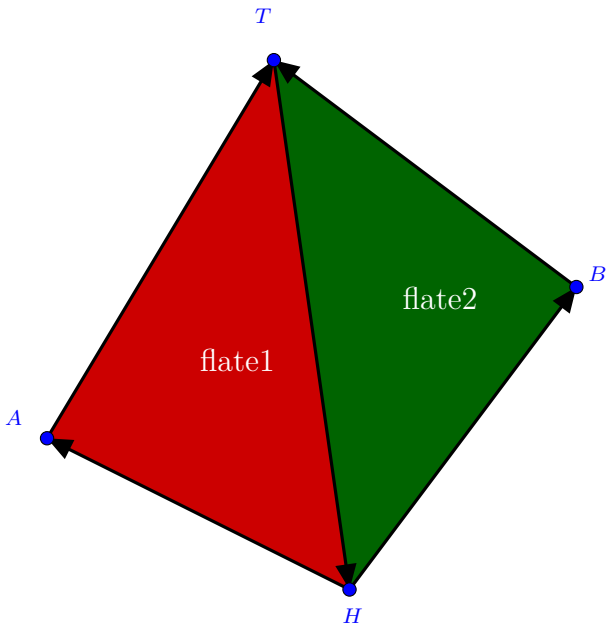
Lagring av geometriske data på fil er nyttig i mange sammenhenger. Det gjør det mulig for konstruktøren å ta vare på geometrien i mellom arbeidsøkter og programkjøringer. Flere versjoner av geometrien kan beholdes lagra separat i forskjellige filer. Det gjøre det mulig å ta vare på, sammenligne og gjenbruke resultater fra forskjellige algoritmer og forskjellige utgaver av samme algoritme. En bane påbegynt i av en algoritme kan lagres på fil og brukes som innputt til en anna algoritme. Videre kan autogenererte baner lastes inn i konstruksjonsverktøyet for en siste manuell finpuss. Autonome fartøy bør også kunne lagre og laste inn baner. Filer kan brukes til å overføre baner mellom fartøy eller mellom fartøy og datamaskiner som genererer baner. Slik kan fartøy ta med seg ei samling baner for bruk i arbeidet sitt.

Til å begynne med er det konstruksjonsverktøyet for baner som har størst behov for å kunne lagre geometri på fil. Baner oppretta i konstruksjonsverktøyet skiller seg ut fra tradisjonelle GeoMod-modeller slik som kuben, pyramiden og roboten. Disse modellene er definert i C++-kodefiler, kompilert og linka inn i programmet. Geometrien i banene er derimot oppretta under kjøretid, og eksisterer bare i datamaskinens primærminne. Data i primærminnet blir som kjent borte når datamaskinen eller programmet avsluttes. For å kunne ta vare på data og jobbe videre, må banene kunne lagres til og lastes inn fra filer på datamaskinens harddisk eller et anna sekundærlager.

9.1 Geometri

Før man kan se på muligheter for lagring på fil må man forstå hvordan GeoMod strukturerer geometrien internt. I primærminnet er geometrien lagra som geometriske objekter med pekere mellom seg. Geometri som hører sammen lagres i transformasjonsgrupper (forkorta tgruppe). I ei tgruppe ligger det en peker til et nettverk av typen GeomNet. Her ligger det lister over noder, kanter og flater. En node representerer et punkt i det tredimensjonale rommet, gjengitt av tre koordinatverdier. En kant ligger mellom to noder, og har retning fra halenoden (eng: tail node)

til hodenoden (eng: head node).



Figur 20: Kanten 'TH' fra node 'H' til 'T' danner grensa mellom flatene 'flate1' og 'flate2'. På positiv side av 'TH' ligger 'flate2', og 'flate1' på negativ side.

Topologien i nettverket er gitt implisitt. Hver kant har lagra peker til neste kant i ei syklisk enkeltlenka liste. Flata 'flate2' i figur 20 er gitt ved 'TH' → 'HB' → 'BT' → 'TH'. Det er lagt til rette for at en kant kan representere grensa mellom to flater. I og med at kanten er retta, er det definert ei positiv og ei negativ side. Ei flate på den positive sida har normalvektor som peker i samme retning tommelen i høyrehåndsregelen for rotasjon ([29], regel nummer to). De andre fingrene krummer rundt flata i kantenes retning. Peker tommelen i motsatt retning, ligger flata på den negative sida. Kanten 'TH' i figur 20 danner grensa mellom flatene 'flate1' og 'flate2'. På positiv side ligger 'flate2', og 'flate1' på negativ side. Hver kant inneholder en peker til både positiv og negativ flate, samt peker til neste kant rundt flata. En boolsk verdi forteller om retningen til den neste kanten går langs med eller imot rotasjonsretningen til flata.

Kurver i GeoMod er alltid assosiert med en kant. Et kurveobjekt opprettes ved hjelp av funksjonen 'GeomEdge::make<CurveType>()' og lagres i kanten via en peker. Kurven går mellom de samme nodene som kanten. Flere kurvetyper er støtta, og flesteparten av dem styres ved hjelp av kontrollpunkter gitt av noder.

9.2 Gammelt filformat

Det finnes et gammelt filformat, brukt i tidligere versjoner av GeoMod, for å lagre geometrien i ei tgruppe på fil. Formatet består av ei samling klasser i 'MaxLib/-net_I/' og to funksjoner i 'MaxLib/net_T/tgroup.<h/cpp>'. Det ble gjort forsøk på å ta den gamle koden inn i den nye programversjonen og passe den til konstruksjonsverktøyet for baner. Lavnivåsrutiner fra standardbiblioteket til C++ benyttes til å åpne, skrive til og lese fra filer. Alle verdier blir lagra binært ved å dumpe innholdet i primærminnet direkte til fil ved hjelp av funksjonen 'write(char*, std::streamsize)' i 'std::ofstream'. Verdier blir lest inn ved hjelp av funksjonen 'read(char*, std::streamsize)' i 'std::ifstream'. Det gamle filformatet omtales heretter som IndexNet-formatet.

Lagringsrutinene må utføre en del forberedelser før selve lagringa kan begynne. Den interne strukturen basert på C++-pekere kan ikke lagres direkte på fil, objektene må adresseres på en anna måte. Tgruppas nettverk må konverteres til et anna format som egner seg bedre til lagring. IndexNet-formatet har valgt å bruke de naturlige tallene for å adressere objektene. Første seg i lagringsprosessen er å opprette en avbildning, i form av 'std::map<GeomNode*, unsigned int>', fra nodene til de naturlige talla. Tilsvarende avbildninger opprettes også for kanter og flater. Neste steg er å bygge opp lister med kanter og flater konvertert fra GeomEdge / GeomRegion til IndexEdge / IndexRegion. IndexXxx-klassene inneholder samme informasjon som GeomXxx-klassene, men alle pekerne er bytta ut med heltallsreferanser i henhold til avbildningene.

Når nettverket er konvertert, starter selve lagreprosessen. Først lagres tgruppas base ved å dumpe objektet av typen 'ExtBasis' direkte til fila. Deretter lagres nodene. Nodene inneholder ingen pekere, og kan derfor lagres direkte. Først skrives antall noder. Deretter skrives det for hver node, antall tegn i nodenavnet, så selve navnet og de tre koordinatverdiene. Neste steg er å skrives antall kanter, så løkker men igjennom kantlista og skriver alle IndexEdge-objektene direkte fra minne til fil. Tilsvarende for flater. Lagring av kurver var ikke implementert i det opprinnelige programmet. Kurver ble forsøkt implementert, men IndexNet-formatets svakheter gjorde det svært utfordrende.

Under arbeidet kom det fram en rekke svakheter ved formatet. Noen skapte problemer allerede på dette utviklingsstadiet, andre kommer til å gjøre det i framtida. Navn-attributtet skapte utfordringer helt fra starten. Tekststrenger av typen std::string har variabel lengde og man må ta spesielle hensyn når man lagrer den til fil. Opprinnelig var det kun noder som fikk navnet sitt lagra, men formatet ble

utvida til også å lagre kantnavn før arbeidet med det gamle formatet ble avslutta. Programdelen som konverterer kantene ble utvida til å la legge kantnavnene i ei eiga liste med indekser som samsvarer med kant-lista. Under lagring skrives antall tegn, så selve tekststrengen rett før IndexEdge-objektet skrives. Samme framgangsmåte kan også brukes ved lagring av flater.

Mer alvorlige svakheter finnes også. Formatet har liten eller ingen toleranse for filer med feil format, feil blir verken oppdaga eller håndtert, og krasj forekommer. Vanlige feilårsaker er avbrudd i lagringsprosessen, filer som er åpna og endra i feil program, eller at programmet blir bedt om å åpne ei fil av en feil type. Ofte fører dette til at en søppelverdi blir lest inn i variabelen for antall noder. Verdien blir tolka som et stort tall (f.eks.: 2 133 525 324), og programmet forsøker å opprette dette antallet GeomNode-objekter. Vanligvis blir datamaskinens minne fylt opp, og maskinen blir uresponsiv helt til operativsystemet avslutter hele GeoMod-programmet.

Videre er binærformatet tett knytta arkitekturen til datamaskinen som oppretter filene. Rekkefølgen datamaskiner lagrer sifrene i et binærtall er kalt 'endianness'. Datamaskiner som følger Intels x86-akretektur følger standarden 'little-endian' og lagrer de minst signifikante sifra først. Andre maskiner, bygd på for eksempel IBMs z/Architecture, nytter 'Big-endian' og lagrer de mest signifikante sifra først. En måte å komme seg rundt dette problemet er å følge en standard konsekvent. Big-endian er mest vanlig i forbindelse med overføring av data over nettverk. Rutiner for å konvertere mellom lokalet binærformat og nettverkbinærformatet finnes i programmeringsspråkets standardbibliotek [10].

Formatet er også vanskelig å utvide og vedlikeholde. Etter hvert som støtte for nye geometriske objekter blir lagt til i konstruksjonsverktøyet, må filformatet utvides. Dette er en stor utfordring fordi hver minste endring i filformatet gjør gamle file ubrukelige. Å opprettholde bakoverkompatibilitet krever stor innsats. Rutiner for å håndtere eldre versjoner må opprettholdes, og forskjellige filformatversjoner må holdes fra hverandre og bokføres. Formatet er dermed ikke særlig godt egna til eksperimentering, og innføring av parametrisert geometri blir vanskelig.

Under utviklingsarbeidet er det ofte nyttig å kunne lese og endre filer manuelt. Filer kan inspiseres for å forstå hvordan programmet faktisk virker eller for å søke etter feil. Av og til er det også nyttig å kunne gjøre små manuelle endringer for å rette opp eller komme seg rundt en feil i programmet. Mennesker har vanskelig for å lese, forstå og skrive binærdata. Eksterne verktøy for å behandle binærfiler finnes, men er knotete å bruke. Tatt de ovenfornevnte utfordringene og svakhetene

i betraktning ser ikke binærformatet attraktivt ut. Tida er inne for å undersøke mulighetene for et revidert filformat, basert på rein tekst.

9.3 JSON-formatet

JSON (JavaScript Object Noation) er et lettvekts standardformat for datautveksling. Både mennesker og datamaskiner kan lett lese og skrive formatet. Det er basert på et subsett av programmeringsspråket JavaScript. Formatet er tekstbasert og helt uavhengig av programmeringsspråk. Men bruker likevel konvensjoner som minner om C-familien av programmeringsspråk (inkludert C, C++, C#, Java, JavaScript, Perl, Python med flere). Disse egenskapene gjør JSON godt egna som format for datautveksling [14]. Formatet følger en åpen standard definert i 'RFC 7159' [3] og 'ECMA-404' [13].

JSON-formatet bruker seks forskjellige datatyper for å lagre og strukturere data. Data lagres i par av nøkler og verdier. Alle nøkler er tekststrenger og må være unike. Nøkkel og verdi er skilt med et kolon (:). Og et komma (,) skiller nøkkel-verdi-par og tabellelementer fra hverandre. De seks datatypene er boolske verdier (bool), desimaltall (double), tekststrenger (string), tabeller (array), objekter (object) og nullverdier (null). Boolske verdier representeres ved teksten 'true' eller 'false'. Tall er representert som dobbeltpresisjons flyttall. Tekstrenger er omslutta av anførselstegn (") og støtter unicode-standarden. tabeller omsluttes av firkantparenteser ([...]) og objekter med krøllparentesers ({ ... }). Alle filer inneholder minst et objekt. Listing 5 viser et eksempel på ei JSON-fil.

```
{
  "tekststreng":    "Hei verden!",
  "tallverdi":      42.0,
  "boolskverdi":    false,
  "tabell":         [
                        1.2,
                        2.3,
                        3.4
                    ],
  "objekt":         {
                        "medlemsTekst": "Tekst",
                        "medlemsTall": 14.67
                    }

}
```

Listing 5: Eksempel som illustrerer JSON-formatet.

Støtte for JSON finnes allerede i QT, og er en del av Qt Core. Qts JSON-støtte tilbyr et C++ API for å tolke, endre og lagre JSON-data. [21]. JSON-implementasjonen bygger i hovedsak rundt klassene 'QJsonDocument', 'QJson-Object', ' QJsonArray' og 'QJsonValue'. Nøkkel-verdi-par aksesseres ved hjelp av firkantparenteser, altså C++ operatoren '[]', på samme måte som man henter ut verdier av en tabell. En verdi henta ut fra et JSON-objekt må konverteres til den riktige typen ved hjelp av funksjoner som 'toDouble()', toString()', 'toArray()', og 'toObject()'. Det er vanskelig å beskrive hvordan disse Qt-objektene skal brukes for å lagre og laste JSON-filer i en sammenhengende tekst. Et eksempel med virkelig programkode gir en bedre illustrasjon. Listing 6 viser kode for å lagre eksempelfila ovenfor. Innlasting av den samme fila er vist i listing 7. Hvis et mer omfattende eksempel er ønska, finnes dette i Qt-dokumentasjonen [19]. Her demonstreres lagring av karakterdata fra et dataspill i JSON-formatet.

```
QJsonObject json;

json["tekststreng"]   = "Hei verden!";
json["tallverdi"]     = 42.0;
json["boolskverdi"]   = false;

QJsonArray jsonTabell;
jsonTabell.append(1.2);
jsonTabell.append(2.3);
jsonTabell.append(3.4);
json["tabell"] = jsonTabell;

QJsonObject jsonObjekt;
jsonObjekt["medlemsTekst"] = "Tekst";
jsonObjekt["medlemsTall"]  =  14.67;
json["objekt"] = jsonObjekt;

QJsonDocument jsonDokument(json);
QFile fil("filnavn.json");
fil.write(jsonDokument.toJson());
fil.close();
```

Listing 6: Viser programkode som nytter Qt-biblioteket for å skrive JSON-fila fra eksempelet ovenfor.

```
QFile fil("filnavn.json");
fil.open(QIODevice::ReadOnly);
QByteArray data = fil.readAll();

QJsonDocument jsonDokument( QJsonDocument::fromBinaryData(data) );
QJsonObject   json = jsonDokument.object();

QString tekst       = json["tekststreng"].toString();
```

```
double   tall        = json["tallverdi"].toDouble();
bool     boolskverdi = json["boolskverdi"].toBool();

QJsonArray jsonTabell = json["tabell"].toArray();
for(int i = 0; i < jsonTabell.size(); ++i) {
  double tabellverdi = jsonTabell[i].toDouble();
  //Gjoer noe med tabellverdien...
}

QJsonObject jsonObjekt = json["objekt"].toObject();
QString medlemsTekst = jsonObjekt["medlemsTekst"].toString();
double  medlemsTall  = jsonObjekt["medlemsTall"].toDouble();
```

Listing 7: Viser programkode som nytter Qt-biblioteket for å lese inn JSON-fila fra eksempelet ovenfor.

Qt sin implementasjon av JSON tillater både å lagre filene i standard tekstformat i tillegg til et eget binærformat. Binærformatet ligger nærme Qt sin interne representasjon av dataene og tilbyr bedre ytelse og lavere filstørrelse. Vanligvis benyttes tekstformatet, fordi lesbarhet er en stor fordel, og ytelsen er i de fleste tilfeller god nok.

9.4 Nytt filformat

GeoMod trenger et nytt og moderne filformat for å lagre geometri. Det nye filformatet må kunne gjøre alt det gamle kan og mer. Flest mulig av det gamle formatets svakheter (diskutert i delkapittel 9.2) bør unngås. Formatet må være fleksibelt og utvidbart. Videre må det kunne lagre alle de geometriske objektene som finnes i dag, med mulighet for seinere å legge inn støtte for nye objekttyper. I denne oppgaven er det utvikla et nytt filformat basert på JSON-formatet. En bane lagra i det nye filformatet er lista opp i sin helhet i vedlegg D.

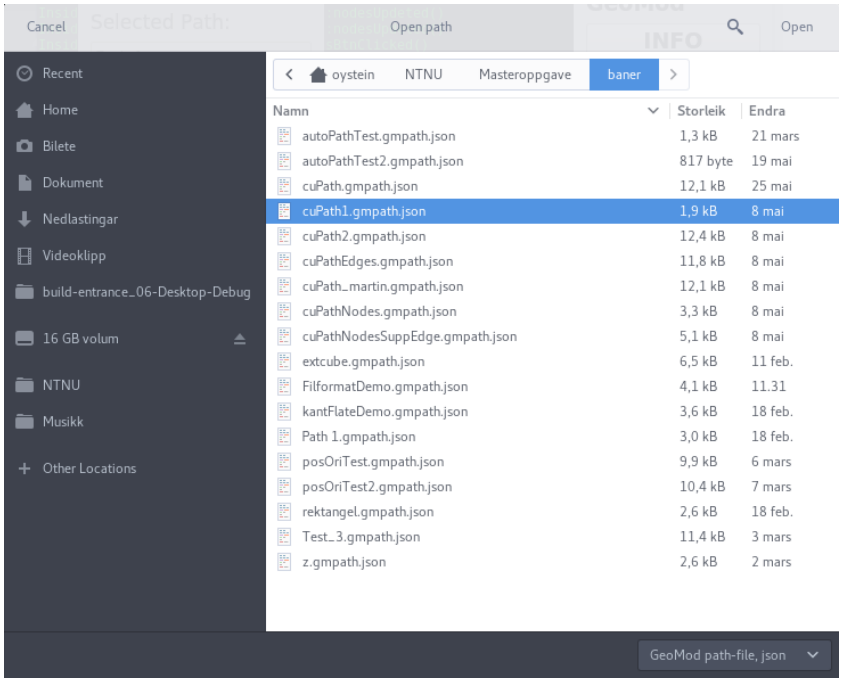
Banen lagres som et JSON-objekt. Først lagres grunnleggende informasjon om banen slik som navn, beskrivelse og base. Geometriske objekter slik som noder, kanter, kurver og flater lagres som underobjekter i lister eller tabeller etter objekttype. Lagring av objekter foregår sekvensielt i medlemsmetoden 'makeJsonObject(...)' i klassen Path. Nodene behandles først, deretter kanter, så kurver med flater til slutt. For hver objekttype løkker programmet gjennom bannes lister, for så å konvertere hvert enkelt geometrisk objekt til et JSON-objekt som legges til i JSON-tabellen. Sist skrives alle tabellene til banes JOSN-objekt. Konverteringa går ut på å legge det geometriske objektets egenskaper inn i JOSNs nøkkel-verdi-par. Hvis man bruker nodene som eksempel, lagres koordinatverdiene under nøklene 'x', 'y' og 'z'. Nodenavnet lagres under 'name' og typebeskrivelsen under 'type'.

For å kunne lagre geometri på fil og seinere laste den inn igjen i en ny instans av programmet, kanskje på en anna datamaskin, er det nødvending å kunne identifiser de geometriske objektene. Internt, mens programmet kjører, har hvert objekt sin egen unike adresse i datamaskinens primærminne. Programmet bruker C++-pekere til lagre denne adressa og legge objektene i lister. Men minneadresser er dårlig egna til å identifisere objekter på tvers av programkjøring og datamaskiner. En mulighet er å gi hvert objekt en unik identifikator basert på heltallverdier. Det vil fungere godt nok, men for brukeren, vil tallverdiene gi liten mening og se ut som de er valgt ut mer eller mindre tilfeldig. Brukere med bakgrunn fra skolefag som matematikk og mekanikk er godt kjent ved bruk av navn for å identifisere geometriske objekter. For eksempel kan ei lærebok i elementær geometri fortelle at rektangelet $ABCD$ er bygd opp av linjene mellom punktene A, B, C og D. Av denne grunn er støtte for navn bygd dypt inn i GeoMod og MaxLib sine geometriske objekter. Konstruksjonsverktøyet presenter allerede brukeren objektene ved å vise objektnavnet. Dermed er det også naturlig å bruke unike navn for å identifisere og koble objekter sammen i det nye filformatet.

Innlasting fra fil krever flere steg enn lagring. Først må objektene opprettes, så må nettverket kobles sammen på nytt. Grunnleggende informasjon slik som navn, beskrivelse og base leses inn og lagres i banen. Nodene er uavhengig av andre objekter, og er dermed rett fram å laste inn. Kantene inneholder informasjon om nettverket, og må lastes inn i to steg. I første steg opprettes kantobjektene og får tilordna navn, hode- og halenode. Videre lastes flatene inn. Før steg nummer to av kantinnlastinga kan starte, må alle kant- og flateobjektene være oppretta. Steg nummer to kobler kantnettverket sammen og registrerer flatene på både kantens positive og negative side. Til slutt lastes kurvene inn, og assosieres med riktig kant. Innlastinga finner sted i metoden 'loadFromJsonObject(...)' i klassen Path.

En hjelpekasse tilbyr funksjoner for lagring og lasting av de fleste objektene. Klassen 'JsonHelper' inneholder statiske medlemsfunksjoner som kan kalles rett fra klassen uten å først opprette et objekt. Hjelpeklassen er med å gi lagringsrutinene en løsere kobling til konstruksjonsverktøyet. Hvert objekt har et typefelt for å støtte flere varianter av samme objekttype. Dette er med på å øke fleksibiliteten og gi formatet gode utvidelsesmuligheter. Lagre- og lasterutiner for nye objekttyper kan implementeres. Baneklassens rutiner for å igangsette lagring og lasting kan utvides til å sjekke for objekttype og kalle rutinen relevant for den gitte objekttypen. Støtte for parametriserte noder kan for eksempel legges til ved å sette typefeltet til 'parametized' og implementere egne funksjoner for lagring og lasting.

Lagring av flater er implementert i det nye filformatet. Og for å gjøre eksperimentering med det nye filformatet lettere er det implementert grunnleggende støtte for flater i konstruksjonsverktøyet. Fana 'Regions' (vist i figur 11, kap. 4) lar brukeren velge kantene som skal omkranse flata fra ei liste. Verktøyet kan lage nye flater, men endring og sletting av flater er foreløpig ikke implementert. Disse funksjonen kan og bør implementeres seinere, men de er ikke direkte relevant for denne delen av oppgaven.



Figur 21: Viser Qt sin fildialog slik den ser ut under GNOME på Arch Linux. Dialogen brukes til å åpne banefiler.

Lasting og lagring igangsettes i fra konstruksjonsverktøyets hovedvindu ved å trykke knappene 'Open path' og 'Save path' respektivt. Qt sin innebygde fildialog 'QFileDialog' lar brukeren bla i gjennom kataloger, velge filer og oppgi filnavn. Dialogen er vist i figur 21 og nytter skrivebordsmiljøets eller operativsystemets standardutforming. Qt sin implementasjon av JSON-formatet gjør det mulig å lagre filene både i et binærformat og som rein tekst. For rein tekst, har banefilene har endelsen '.gmpath.json'. Binærfiler bruker endelsen '.gmpath.bin'. Heldigvis er man ikke lengre begrensa til filnavn med lengde '8.3' slik man var på gamle MS-DOS-partisjoner. Første del av endelsen ('.gmpath') forteller brukeren at denne fila er ei banefil oppretta av GeoMods konstruksjonsverktøy for baner. Andre del av endelsen ('.json' eller '.bin') er med på å fortelle operativsystemet hvilke program

som skal assosieres med fila. Forsøk på å åpne binærfiler gir ofte en advarsel om ukjent filtype, mens JSON-filer åpnes i en vanlig tekstredigeringsverktøy slik som Windows Notepad eller GNU Emacs. Grunnet GeoMods modulære natur, er det vanskelig å få operativsystemet til å åpne banefiler direkte i konstruksjonsverktøyet.

Karv og behov for filformater i GeoMod har endra seg siden begynnelsen av 2000-tallet. Det har datamaskiner og programvare også. Det gamle filformatet har gjort sin nytte og kan pensjonere seg med god samvittighet. Raskere datamaskiner har flytta fokuset bort fra et reint ytelsesperspektiv over til mer fokus på brukervennlighet og plattformuavhengighet. Det nye formatet dekker alle behovene konstruksjonsverktøyet for baner har i dag. I tillegg er det lesbar for mennesker og basert på et format laga for utveksling av data både lokalt og over internett. Formatet er også utvidbart, det støtter flere typer av en klasse geometriske objekter, og nye objekter kan legges til ved behov. Delkapittelet avsluttes med en oppfordring til å ta formatet i bruk i andre deler av GeoMod.

10 Parametrisk styring av geometri

For å vise fram fleksibiliteten og utvidbarheten til det nye filformatet, er det utført et innledende forsøk med parametrisert geometri i banene. Et utvalg grunnklasser er implementert. Alle med mulighet for å lagres på fil. Utvikling av grafiske brukergrensesnitt er så vidt påbegynt, men arbeidet er på et veldig tidlig stadium, og langt ifra ferdig. Ferdigstilling av støtte for fullstendig parametrisert geometri i GeoMod er nok grunnlag for opptil flere prosjekt- og masteroppgaver i framtida.

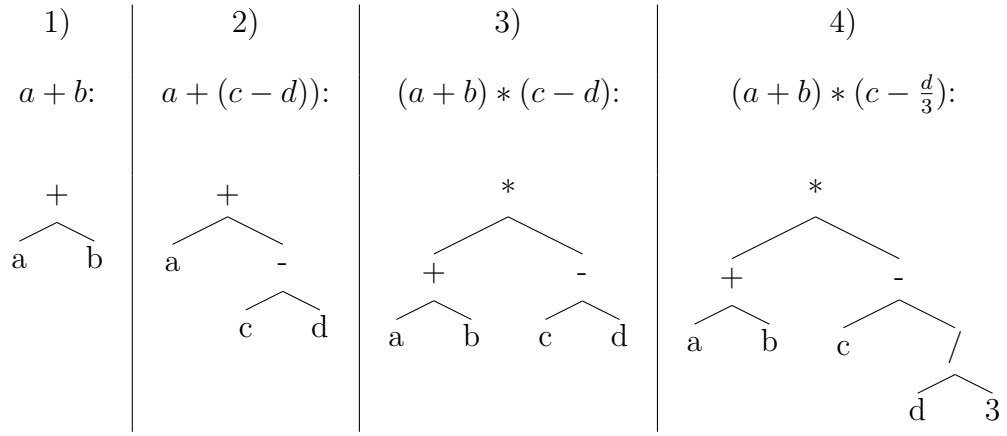
Parametrisering av geometri kan utføres på mange måter avhengig av fagområde og ønska resultat. Første møte med parametrisering er for mange kalkulus grunnkurs på universitetet. Matematiske kurver parametriseres ofte på formen $x(t) = f(t)$ og $y(t) = g(t)$. En anna variant av parametrisering er metoden brukt i kapittel 7 og 11 for å parametrisere rette linjer ved lineær interpolasjon mellom to punkt. Parameteren t bestemmer punktet $P(t)$ på linja, gitt av ligninga $P(t) = (1-t)A+tB$. Videre kan kjente geometriske objekter, slik som kuber, kuler, pyramider osv., parametriseres med parametere som karakteriserer objekttypen. Kuler kan parametriseres utfra radius eller diameter, en boks er bestemt av høyde, bredde og dybde. Polyedre kan bestemmes av skjæringa mellom plan parametrisert via ett sett normalvektorer. Mer kompliserte, men standardiserte, objekter som for eksempel skruer, bolter og mutre, kan parametriseres med parametere i henhold til standarden. Det vil si, geometrien til en parametrisert metrisk bolt kan for eksempel genereres utfra parameterne størrelse $M8$, gjengestigning $1,5mm$, og lengde

$30mm$. Dette kan igjen tas et steg videre, til utvikling av systemer for kunnskapsbasert ingeniørvitenskap (engelsk: Knowledge-based Engineering, KBE). Ta for eksempel et KBE-system for konstruksjon av bokhyller. Bokhylla kan parametriseres ved høyde, bredde, dybde, hylleavstand og lastekapasitet. Utfra parameterne og den innebygde kunnskapen, kan programmet bygge opp geometrien, bestemme antall hyller, sette dimensjonene på materialene og avgjøre om det trengs ekstra støtte mellom hyllene.

10.1 Parametriseringsmetode

I dette innledende forsøket introduseres en anna parametriseringsmetode. Det er utført eksperimenter med en metode for å parametrisere geometri ved hjelp av parametriserte koordinatverdier i nodene. Metoden er av en enkel matematisk type, inspirert av elementær algebra. En parametrisert node bestemmes av tre parametriserte uttrykk. Ett for hver koordinatretning. Et kvadrat i planet kan parametriseres med uttrykket a, og konstrueres via nodene $A = (-a, -a)$, $B = (a, -a)$, $C = (a, a)$ og $D = (-a, a)$. Ved å endre verdien til parameteren a, endrer kvadratet størrelse, men formen forblir den samme. Tre parametertyper er inkludert: konstanter, variable og enkle uttrykk. Konstanter har en fast tallverdi som tilordnes de opprettes. Variable inneholder også en tallverdi, men har også mulighet for å endre verdien når som helst. Et enkelt utrykk er definert som binært uttrykk med en operator og to operander. Operandene er andre parametere, og for øyeblikket er operatorer for addisjon, subtraksjon, multiplikasjon og divisjon implementert. Eksempel på et enkelt uttrykk er $a + b$. Mer kompliserte uttrykk kan bygges opp ved å koble sammen flere enkle uttrykk.

Et enkelt utrykk kan ses på som et lite binært tre. Rotnoden er operatoren, de to løvnodene er operandene. Siden uttrykk også kan brukes som operander, kan mer kompliserte uttrykk bygges opp ved å koble flere trær sammen. Nedenfor er det lista opp et utvalg eksempeluttrykk. Utrykka er nummerert $1 - 4$, så er de gitt på en algebraisk form, før trestrukturen er illustrert. Det vil også være behov for utrykk av kun én operand slik som trigonometriske funksjoner, logaritmer og eksponentialutrykk. Dette er helt rett fram å implementere, men har ikke blitt inkludert i det innledende forsøket på grunn av tidspress og fordi funksjonaliteten strengt tatt ikke er nødvendig for å vise fram konseptet.



10.2 Implementasjon

Implementasjonen er lagt til katalogen 'path_des/parametrization' med klassen 'Parameter' i bunn. Klassen bestemmer grensesnittet og alle de andre parametertypene arver fra denne. Aleine fungerer den som en konstantparameter. En parameter med konstant verdi er ikke særlig nyttig i seg selv, men er tatt med for å ligge nærmere matematikken, og passer fint inn i basisklassen. En variabel parameter finnes i klassen 'VariableParameter'. Den utvider Parameter med mulighet for å endre parameterverdien. Enkle uttrykk er implementert i klassen 'SimpleExpression'. Den viktigste delen av grensesnittet er lista opp i listing 8. Hver parameterinstans identifiseres via et unikt navn i form av en tekststreng, og tallverdien hentes ut ved å kalle 'getValue()'.

```
double getValue();  //returnerer parameterens tallverdi
QString getName();  //returnerer parameterens unike navn i form av en
    teksstreng
QString getType();  //returnerer en tekststreng som representer
   parametertypen
QString toString(); //returnerer en utskriftsvennlig
   tekstrepresentasjon av parameteren, inkludert navn, type og
   tallverdi

//Brukes til aa lagre parameteren paa fil
void writeToJson(QJsonObject &json);
void readFromJson(QJsonObject &json, QMap<QString, Parameter*>
   paramMap);
```

Listing 8: Lister opp den viktigste delen av grensesnittet til parameterklassen. Alle funksjonen er definert som virtuelle.

Støtte for arv og virtuelle funksjoner i programmeringsspråket C++ er utnytta for å implementere den valgte parametriseringsmetoden. Polymorfi gjør at man

kan ha en peker til en generell parameter uten å bry seg om den bestemte typen. Programmet bruker stort sett bare pekere til basisklassen Parameter. Det forventes som regel at objektet en Parameter-peker peker til er en instans av en underklasse som for eksempel VariableParameter. Virtuelle funksjoner sørger for at funksjonen implementert i den faktiske objekttypen blir kalt, uansett hvilken type pekeren har. Dermed vil kall til operandenes getValue() i SimpleExpression alltid evalueres i henhold til operandens faktiske parametertype.

Programkode for å implementere eksempelutrykka gitt ovenfor er tatt med i listing 9. Merk at alle parameterobjektene opprettes ved hjelp av kodeordet 'new', og lagres i pekere av type Parameter. Uttrykkstrær bygges nedenifra og opp.

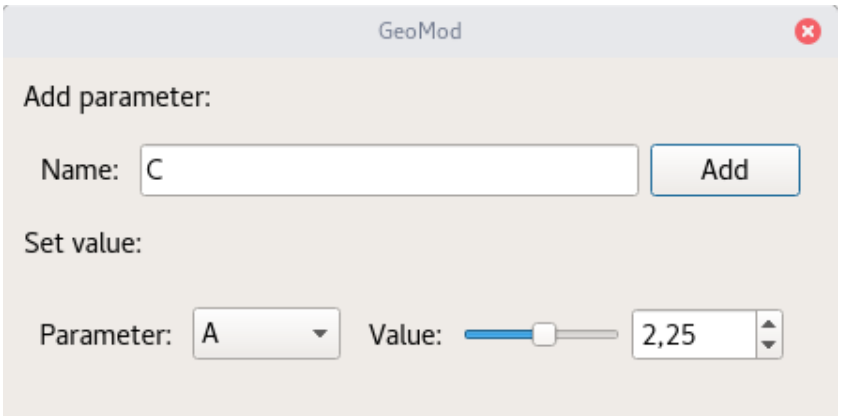
```
//Opprett variablene
Parameter *a = new VariableParameter("A", 1.5);
Parameter *b = new VariableParameter("B", 2.75);
Parameter *c = new VariableParameter("C", 3.25);
Parameter *d = new VariableParameter("D", 4.2);

//Utrykk 1)
Parameter *a_pluss_b = new SimpleExpression("a_pluss_b", a, b, "+");

//Utrykk 2)
Parameter *c_minus_d = new SimpleExpression("c_minus_d", c, d, "-");
Parameter *exp2 = new SimpleExpression("exp3", a, c_mius_d, "+");

//Utrykk 3)
Parameter *exp3 = new SimpleExpression("exp3", a_pluss_b, c_minus_d,
    "*");

//Utrykk 4)
Parameter *d_div_3 = new SimpleExpression("d_div_3", d, new Parameter
    ("konst3", 3), "/");
Parameter *c_minus_d3 = new SimpleExpression("c_minus_d3", c, d_div_3
    , "-");
Parameter *exp4 = new SimpleExpression("exp4", a_pluss_b, c_minus_d3,
     "*");
```

Listing 9: Programkode for å opprette de parametriserte uttrykka fra eksempelet ovenfor.

For å innføre parametriseringa i GeoMod er klassen 'ParamGeomNode' oppretta. Den arver i fra og utvider den vanlige nodeklassen GeomNode i MaxLib. Noden har et parametrisert uttrykk for hver av de tre koordinatretningene. Når en parameterverdi endres, oppdateres noden automatisk. Ellers oppfører den seg som en

vanlig node. Klassen GeomNode er utvida med funksjonen 'getType()' som returnerer en tekststreng for å identifisere nodetypen. Vanlig noder returnerer 'default', ParamGeomNode returnerer 'parameterized'. Konstruksjonsverktøyet skal ta seg av oppretting og organisering av parametriserte noder.

Sammenkobling og automatisk oppdatering av parametere er utført ved hjelp av Qts 'signals ands slots'-mekanisme. Når uttrykk bygges opp blir alle parameterobjektene automatisk kobla sammen. Når en parameter endrer verdi, sendes signalet 'valueChangend()' ut, og endringa flytter seg oppover i uttrykkstreet. Hvert parameterobjekt beregner og mellomlagrer sin egen verdi. En alternativ implementasjonsmetode er å kun lagre verdiene til variable og konstante parametere, for så å evaluere alle uttrykk på nytt hver gang noen ber om verdien til et sammensatt uttrykk. Dette reduserer behovet for kommunikasjon, men mellomlagring av parameterverdier er likevel den mest effektive metoden. Man trenger ikke å evaluere kompliserte uttrykk mange ganger. Parameterverdien endres sjelden i forhold til hvor ofte en del av programmet ber om tallverdien til et uttrykk. Tegnerutinen ber for eksempel om alle koordinatverdiene hver gang kameraet skal oppdateres. Ulempen er at alle mellomregninger også tar opp plass i minnet.



Figur 22: Grafisk brukergrensesnitt for å opprette og endre verdien til variable parametere.

Parameterne opprettes som, og oppfører seg som vanlige geometriske objekter i banene. De identifiseres utfra et unikt navn på samme måte som andre geometriske objekter, og skal dukke opp i konstruksjonsverktøyets objekttreet. For å gjøre parametriseringa tilgjengelig for sluttbruker, må det opprettes grafisk brukergrensesnitt i flere nivåer. Sammen med det innledende forsøket er det inkludert et enkelt brukergrensesnitt for å opprette og endre verdien til parametere. Grensesnittet er vist

i figur 22. Her kan nye parametere legges til ved å gi dem et navn i tekstfeltet, og trykke knappen 'Add'. Verdien til eksisterende parametere kan endres ved å velge parameteren i nedtrekksboksen, og sette verdien ved hjelp av slideren. Flere vinduer er nødvendig for å bygge opp sammensatte uttrykk og legge til parametriserte noder. På dette tidspunktet er integrasjonen mellom parametrisering og konstruksjonsverktøyet noe mangelfull. Det viste seg å være for tidkrevende til å bli med i denne masteroppgaven. Det ble fokusert mer på teorien og prinsippene bak samt at parametriseringa skal virke sammen med det nye filformatet.

Den valgte parametriseringsmetoden minner på mange måter om en enkel kalkulator. Enkle og mer kompliserte uttrykk kan bygges opp og brukes til å konstruere geometriske objekter. Det vil være særlig nyttig ved manuelt konstruksjonsarbeid. Parametriseringa ligger tett opp mot ingeniørers tankemåte. Dimensjoner, avstander, vinkler og så videre navngis og brukes i beregninger før den konkrete tallverdien er bestemt. Dette muliggjør rask prototyping. Først kan geometrien konstrueres med ukjente dimensjoner, så kan man eksperimentere med forskjellige verdier og se hva som gir best resultat.

11 Baner mellom bevegelige hindre

Autonomt arbeidene undervannsfartøy kan bli nødt til å navigere krevende farvann. Enten fartøyet skal kartlegge havbunnen eller utforske et skipsvrak, vil det møte hindre på veien. Algoritmer for å kjenne igjen og navigere rundt hindre er nødvendig. Havstrømmer og bølger kan være med på å gi hindra bevegelse, så algoritmene må ta hensyn til bevegelige hindre. Formålet med denne masteroppgaven er å legge fram et grunnlag for videre eksperimentering med algoritmer for autonom konstruksjon av baner. For å finne ut om man nærmer seg denne målsetninga, skal det gjøres et innledende forsøk med en algoritme som forsøker å legge en bane mellom bevegelige hindre. Den inkluderte algoritmen viser konseptet. Men nødvendige forenklinger er gjort for passe inn i denne oppgavens rammer når det kommer til bruk av tid og arbeidskraft. Utvikling av mer avanserte algoritmer som tar flere av de nødvendig hensyn for trygg navigasjon kan danne grunnlaget for videre forskning og framtidige masteroppgaver. Forhåpentligvis kan rammeverket utvikla i denne oppgaven være til hjelp i dette arbeidet.

11.1 Automatisk oppdatering

Fartøy må kunne navigere rundt bevegelige hindre. Arbeid med bevegelse på datamaskiner utføres ofte ved bruk av tidsskritt. Parametere forandres etter funksjoner

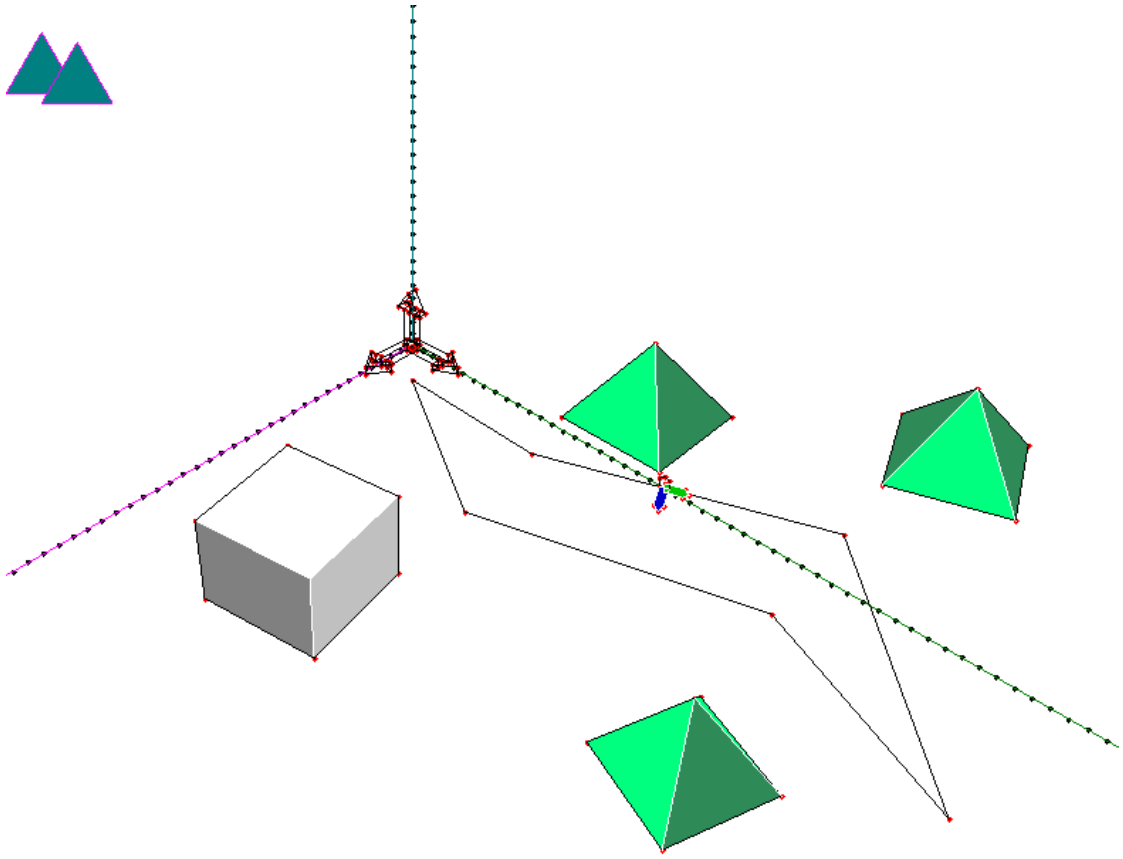
av tidsskritta, beregninger utføres på nytt, og geometrien oppdateres. Med et rammeverk for oppdatering på plass, trenger ikke algoritmer for autonom konstruksjon av baner å forholde seg til tidskritta direkte. Algoritmene kan generere banen utfra hindrets nye geometri og posisjon etter at oppdateringa er utført.

Det grafiske området tegnes på nytt hver gang geometrien i den simulerte verden endres. Dermed er tegnerutinene et naturlig sted å legge sammenkoblinga med rutiner for automatisk oppdatering av genererte baner. Prinsippene rundt 'Signals and slots'-mekanismen til Qt gjør det mulig å koble objekter sammen og få tilstandsendringer i et objekt til å påvirke et anna. Klasser som skal benytte prinsippene må arve ifra Qt-klassen 'QObject' og inkludere preprosessormakroen 'Q_OBJECT' i starten av sin egen klassedefinisjon. Opprinnelig er det ingen klasser i MaxLib som arver QObject. Så modifikasjoner er nødvendig før man kan ta i bruk signals and slots. Baner tegnes opp ved hjelp av en rutine i klassen Tgroup. Dermed er det denne klassen som har blitt utvida til å arve QObject. Videre er makroen Q_OBJECT og signalet 'drawRoutineCalled()' lagt til. Tegnerutinen sender ut signalet, og gjør det mulig å koble til et slot fra konstruksjonsalgoritmen for å oppdatere banen.

11.2 Bevis-av-konsept algoritme

Det er utført et innledende forsøk med en algoritme som legger baner mellom bevegelige hindre. En demonstrasjon er vist i figur 23. Her genereres en bane automatisk mellom fire hindre i form av en kube og tre pyramider. Algoritmen har fått navnet 'NodePairPath', og som navnet tilsier konstruerer den baner utfra vilkårlig mange par av eksterne av noder. Implementasjonen er lagt til katalogen 'path_des/autopath'. Her kan man også legge framtidige algoritmer for automatisk konstruksjon av baner. Innputt til algoritmen er par av noder på hindra banen skal legges mellom, samt en bane med start- og sluttnode. Utputt er den genererte banen med kanter fra start- til sluttnoden som unngår hindra Den tar i bruk konstruksjonsverktøyets grensesnitt for autonom modus for å opprette noder og kanter i banen. Nodene plasseres automatisk på den interpolerte rette linja mellom hindra. Støttekanter for å bestemme orientering genereres også (mer om orientering og støttekanter i kapittel 7 "Posisjon og orientering"). Algoritmen består i hovedsak at to deler. En del bygger opp banens struktur når nye nodepar legges til. Den andre plasserer geometrien i rommet og oppdaterer banen hver gang hindre flytter på seg.

Videre i teksten omtales flere typer noder utfra hvilken rolle de spiller i algoritmen. Alle er vanlige GeoMod-noder, altså en instans av klassen GeomNode. Noder



Figur 23: Viser den automatisk genererte banen mellom to nodepar og fire hindre.

på hindre kalles eksterne noder og er en del av GeoMod-modellen som danner hinderet. De autogenererte nodene langs banen kalles kort og greit autonoder. Hver autonode har en tilhørende støttenode brukt av støttekantene for orientering. I tillegg har banen en start- og sluttnode. Startnoden angir fartøyets initiell posisjon, sluttnoden målposisjonen. En god del bokholderi er nødvendig for å holde styr på alle nodene. To lister med pekere danner parene av eksterne noder. Algoritmen er avhengig av å vite posisjonen en ekstern node har i det globale koordinatsystemet. For seg selv inneholder noder kun koordinater relativ til sin lokale base, dermed må det også lagres en peker til nodenes base. Sist men ikke minst er pekere til de autogenererte nodene og de assosierte støttenodene lagra i separate lister. Listene kobles sammen ved hjelp av samsvarende indekser. Det vil si at autonode med indeks i i sin liste hører sammen med de eksterne nodene lagra separat med indeks i i sine lister.

Banens struktur oppdateres når eksterne nodepar legges til. Rutinen 'addExternalNodePair(...)' tar inn pekere til to eksterne noder og basene deres som argument. Argumentene lagres i listene, så oppretters det en auto- og en støttenode. Nodene navngis 'autoPathNode$X$' og 'autoPathSupportnode$X$' respektivt. 'X' følger nodeparets interne nummerering i listene. Når første nodepar legges til, opprettes to kanter. En fra startnoden til autonoden, og en fra autonoden til sluttnoden. De er navngitt 'autoPathStartedge' og 'autoPathEndedge' respektivt. Analogt genereres støttenoden 'autoPathSupportNode$X$', sammen med støttekantene 'autoPathStartedgeSupport' og 'autoPathEndedgeSupport'. Etterhvert som flere nodepar legges til, legges det til nye kanter mellom den nye autonoden og den forrige. Den nye kanten får navnet 'autoPathEdge$X$', og kanten som går til sluttnoden modifiseres slik at halenoden nå peker til den nye autonoden. Støttekanter legges til og modifiseres likedan.

Med banens struktur på plass mangler kun autonodene sin posisjon i rommet. Banen oppdateres ved at rutinen "updatePath()' beregner ny posisjon til alle autonodene, kantene følger med av seg selv. Rutinen kalles automatisk hver gang banen tegnes opp på nytt, men det er også mulig å kalle den manuelt. For hvert eksterne nodepar beregnes posisjonen til både auto- og støttenodene. Posisjonen til de eksterne nodene i banens lokale base beregnes ved hjelp av konstruksjonsverktøyets hjelperutine 'konverterPunktFraInnbaseTilUtbase(...)'. Videre interpoleres ei rett linje mellom disse to punktene. Parameterne t og t_{supp} brukes for å beregne to punkt på linja. Autonoden plasseres på det ene punktet, støttenoden på det andre. I demonstrasjonen er parameterne $t = 0,5$ og $t_{support} = 0,25$ valgt på forhånd.

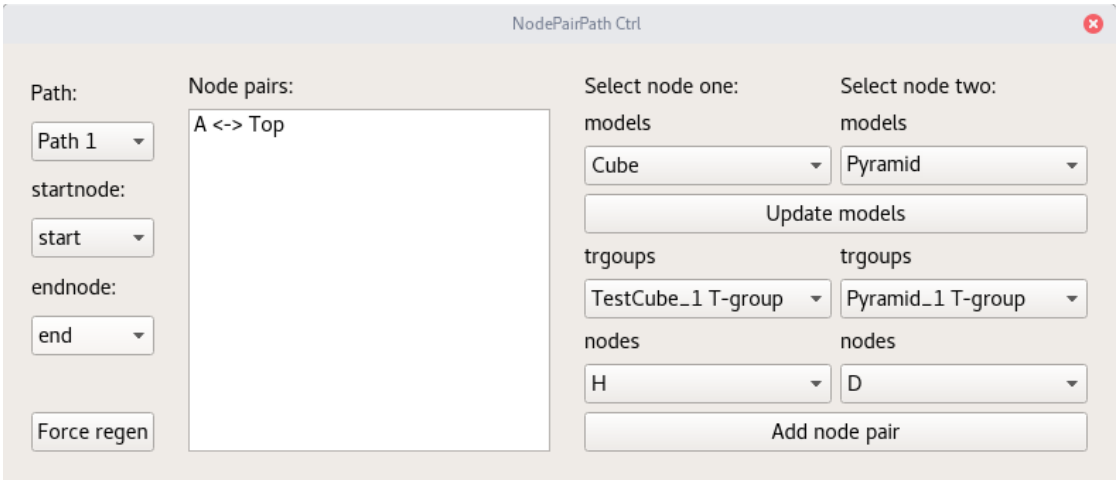
```
void setStartnode(GeomNode *node);
void setEndnode(GeomNode *node);
GeomNode *getStartnode();
GeomNode *getEndnode();

void setParameter(double param);
void setSupportParameter(double param);
double getParameter();
double getSupportParameter();

void addExternalNodePair(GeomNode *nodeOne, ExtBasis *nodeOneBasis,
    GeomNode *nodeTwo, ExtBasis *nodeTwoBasis);
void updatePath();
```

Listing 10: Liste over relevante funksjoner som tilbys av NodePairPath.

11.3 Grafisk brukergrensesnitt

I likhet med resten av konstruksjonsverktøyet, kan NodePairPath både brukes direkte av anna programvare eller styres gjennom et grafisk brukergrensesnitt. Figur 24 viser det grafiske brukergrensesnittet 'NodePairPathCtrl'. Her kan brukeren legge til nodepar ved å velge ut noder i andre GoeMod-modeller. Til venstre finnes nedtrekksbokser som lar brukeren velge hvilken bane verktøyet skal arbeide på, samt start- og sluttnode. Knappen 'Force regen' tvinger fram ei oppdatering av banen ved å kalle oppdateringsrutinen. Det skal ikke være nødvendig å bruke den, men den er der likevel i tilfellet brukeren ikke stoler på programmet og mistenker at det har oppstått en uforutsett feil baneoppdateringa. Midt i vinduet listes de valgte nodepara opp. Til høyre finnes verktøy for å velge ut noder ifra modellene og legge til et nytt nodepar. Første seg er å velge modellen av hindra i de øverste paret av nedtrekksbokser. Så velges den transformasjonsgruppa som inneholder nodene. Mange modeller har kun én transformasjonsgruppe, eller har alle de relevante nodene i den første gruppa. Da kan man la standardvalget i nedtrekksboksen stå. Sist velges nodene i de nederste nedtrekksboksene, og nodeparet legges til ved å trykke knappen 'Add node pair'.

Nedtrekksboksene for valg av modell skal liste opp alle modellene i GeoMods hovedmodelliste 'GenViewVariables::ModelsPtr'. Her er alle modeller inkludert, både de som er linka inn statisk og de som er linka inn dynamisk. Ideelt sett skulle NodePairPathCtrl ha oppdatert nedtrekksboksene sine automatisk når en ny modell lastes inn dynamisk. Men for øyeblikket finnes det ingen funksjonalitet i



Figur 24: Grafisk brukergrensesnitt for styre algoritmen NodePairPath ved å velge hindre banen skal konstrueres mellom.

GeoMod som forteller andre deler av programmet at modelllistene har blitt endra. Dette kunne med fordel vært implementert enten i Database Manager ved å sende ut et Qt-signal 'databaseUpdated()' når en ny modell lastes inn og en ny forekomst opprettes. Eller via signaler i klassen 'Models', som ModelsPtr er en instans av. Skjønt faller dette utenfor rammene til denne oppgaven. Derfor må nedtrekksboksene oppdateres manuelt etter man har lasta inn en ny modell dynamisk. Knappen 'Update models' tvinger fram oppdateringa, og får den nye modellen med på lista.

I arbeid med grafiske brukergrensesnitt er det lett å gjøre feil. Kanskje trykker man en knapp ned to ganger, eller man har markert feil element i en nedtrekksboks. Da er det behov for å angre eller gjøre om valgene ved hjelp av rutiner for endring og fjerning av nodepar. Dessverre er ikke slike rutiner inkludert på dette tidspunktet. Endring av nodepar innebærer en del bokholderi. De eksisterende kantene må kobles sammen på nytt og noder må legges til og fjernes fra listene. Alt i alt innebærer dette mer utviklings- og testingsarbeid enn det som er hensiktsmessig for en bevis-av-konsept algoritme.

11.4 Framtidige muligheter

Fram til nå er det kun utført innledende forsøk med algoritmer for å legge baner mellom hindre. Demonstrasjonen og bevis-av-konsept algoritmen NodePairPath bygger opp rammeverket for videre utvikling, men algoritmen i seg selv har mange svakheter og et stort potensial for forbedring.

Den geometriske formen til et hinder kan være svært kompleks. Å arbeide med hele den komplekse geometrien er ikke ideelt. Det øker både kompleksiteten og kjøretida til algoritmene. Samtidig som det øker vanskelighetsgraden og tidsforbruket til utvikleren. Den nåværende løsninga hvor et hinder er representert som ett enkelt punkt har mange svakheter. Første svakhet ligger i hvordan punkt velges ut. Et punkt kan bare representere en liten del av hindret, og hvilke punkt som velges er av stor betydning. For øyeblikket innebærer valg av punkt manuelt arbeid. Utvikling av kunstig intelligens for å velge punkt på hinderet er ikke trivielt, og neppe den beste bruken av tid og arbeid. Dermed er en anna representasjon av hindringer ønska.

Et naturlig steg videre er å omslutte hindre i enkle, forhåndsgitte geometriske objekter som kuler, kuber, kuboider og lignede objekter Matematikk for å beregne skjæringspunkt mellom linjer, plan og disse objektene er kjent og overkommelig å implementere. Ei omsluttende kule kan konstrueres ved å identifisere den lengste avstanden innad i hindret, for så å bruke denne lengden som diameter i kula. Det er et mål å legge den omsluttende geometrien så tett innpå den virkelige som mulig. Ei kule vil overestimere utstrekninga til mange hindre. Lange og smale hindre, som for eksempel en stav eller trestamme, er særlig utsatt. En boks eller kuboide er i mange tilfeller bedre egna. Beregning av en minimal omsluttende boks involverer en mer omfattende framgangsmåte. Heldigvis finnes allerede flere algoritmer som løser dette. Den iterative framgangsmåten til Chan et.al [4] er både nøyaktig og effektiv, og kan med fordel implementeres i konstruksjonsverktøyet for baner.

For enkle hindre, brukt til testing og eksperimentering, er kuber, pyramider og lignende modeller som følger med GeoMod tilstrekkelig. Men før eller seinere er det behov for modeller som representerer hindre henta fra virkeligheten. Da kan avansert bildebehandling og metoder for å gjenkjenne og rekonstruere tredimensjonal geometri utfra todimensjonale bilder være nyttig. Hirano et.al. [12] bruker "shape-from-silhouette"-metoder (SFS) for å konstruere en voxelbasert 3D-modell fra silhuettbilder. For å unngå voxler i konkave områder, brukes teknikker for "Space carving". Til slutt konverteres voxelmodellen til et triangulært mesh for bruk i CAD-programvare ved hjelp av "Marching Cubes"-algoritmen. Denne framgangsmåten kan implementeres i GeoMod for å konstruere geometriske modeller av hindre automatisk.

For å kunne navigere rundt hindre automatisk, må fartøyet identifisere og hente informasjon om hindra. En metode for å identifisere hinder ved å slå sammen data fra flere sensorer er foreslått av Zhang et.al i [33]. Først analyseres sensorverdiene, så nyttes teori for å avgjøre om det finnes hindre foran fartøyet. Så estimeres

hinderets posisjon og fart for å danne en unngåelsesstrategi. hindre og sensorer kan simuleres i GeoMod, og konstruksjonsverktøyet danner et godt rammeverk for eksperimentering med algoritmer av denne typen.

Så langt har man bare fokusert på hindra og deres utstrekning. Fartøyets størrelse er av også betydning, og algoritmene som konstruerer baner bør ta hensyn. Små fartøy kan presse seg igjennom mindre åpninger, og kan ofte gjennomføre rasker manøvreringer enn større fartøy. I utgangspunktet kjenner ikke banegeneratoren til fartøyet. Her ligger et par utfordringer knytta til koblinga mellom fartøysmodellen og algoritmen som konstruerer banen. En mulighet er å innføre en eksplisitt kobling til fartøysmodellen. Men en tett kobling er ikke ønska. Det vil være bedre å sende en innkapsla versjon av fartøysgeometrien til algoritmen. Innkapslinga kan utføres etter samme metode som beskrevet ovenfor for innkapsling av hindre. Uansett bør man innføre en sikkerhetsmargin som bestemmer minste avstand mellom fartøy og hindre. Marginen kan enten gis som en prosentandel av fartøyets utstrekning eller som en absolutt avstand. Fartøyet må også vurdere hvilken vei det skal ta for å unngå hinderet. I de fleste tilfeller er fartøyets mål bare å unngå hinderet, og hvilke side av hinderet fartøyet tar er likegyldig. Planleggingsalgoritmer må finne ut hva som gir mest mening i hvert enkelt tilfelle.

Virkelige fartøy må følge fysikkens lover, og baner bør konstrueres rundt fartøyets fysiske begrensinger. Rette linjer er vel og bra så lenge fartøyet skal beveger seg i én retning. Men så fort fartøyet nærmer seg et knutepunkt, bør rette linjer gå over i kurver. Algoritmen bør altså passe på å konstruer kontinuerlige overganger. Først må man påse at selve banen er kontinuerlig, dernest må man passe på at både banens første- og andrederiverte også er kontinuerlig. Diskontinuitet i den andrederiverte fører til høy akselerasjon og store påkjenninger for fartøyet. Videre bør man også sette av tid til å finjustere konstruksjonen av støttekanter. For øyeblikket har støttekantene og autokantene felles start- og sluttnode, egne støttenoder bør opprettes og posisjoneres hensiktsmessig. I nærheten av knutepunkt, bør man kanskje bruke flere støttekanter for å få bedre kontroll med orienteringa og oppnå mykere overganger.

12 Oppsummering og veien videre

Arbeidet med konstruksjonsverktøyet for baner har kommet et godt stykke på vei siden starten av prosjektoppgaven høsten 2016. Justeringer og finpuss av verktøyet har blitt utført, og ny funksjonalitet har blitt lagt til. Verktøyet kan opprette, endre og fjerne geometriske objekter i banene. Baner er bygd opp av noder, kanter og kurver kobla sammen. Både et grafisk brukergrensesnitt og grensesnitt for bruk

av programvare i autonom modus er med. Verktøyet danner et godt utgangspunkt for videre eksperimentering med algoritmer for autonom konstruksjon av baner. Det kan arbeide på flere baner. Verktøyet oppretter baner og legger dem i ei liste. Ulike baner skilles fra hverandre ved hjelp av brukeroppretta navn og beskrivelse. Brukeren velger gjeldene bane som verktøyet arbeider på.

Dynamisk linking er innført, og verktøyet finnes nå i to utgaver. Både den statisk og dynamisk linka utgaven har samme kodebase. Brukeren har valget om verktøyet skal linkes inn statisk eller dynamisk. For øyeblikket er den statiske versjonen mest brukt. Men på sikt skal den statiske versjonen fases ut til fordel for den dynamiske. Bruk av den dynamisk linka utgaven kan gjøres mer attraktivt ved å forbedre brukergrensesnittet for innlasting av plugins i GeoMod.

Delverktøyet PathMotion lar fartøysmodeller bevege seg langs baner. Det oppretter en kobling mellom banen og fartøyet. Posisjon beregnes langs rette eller kurva banekanter, og orientering er gitt ved hjelp av støttekanter. Verktøyet skal være en hjelp til utvikleren av fartøysmodeller. Det tilbyr to grensesnitt og gir utvikleren varierende grad av kontroll. Selvfølgelig er det muligheter for utvidelser og forbedringer. Muligheter for mykere overganger og finere kontroll over orientering er diskutert. Men man kan også ta det lenger og utvide verktøyet til å holde rede på hvor fartøyet har vært, og utvikle algoritmer for ruteplanlegging.

Konstruksjonsverktøyet har behov for å kunne lagre baner og geometriske data på fil. Det er innført et helt nytt filformat som lagrer data som rein ASCII-tekst etter JSON-standarden. Geometriske objekter kapsles inn og lagres som JSON-objekter i tabeller organisert etter objekttype. Filformatet dekker alle behova konstruksjonsverktøyet for baner har i dag. I tillegg er formatet fleksibelt og lett å utvide ved å legge til støtte for nye geometriske objekter. Andre deler av GeoMod-programvaren kan med fordel ta i bruk det nye formatet. Muligheter for parametrisk styring av geometri er vurdert. En parametriseringsmetode er lagt fram, og arbeidet med å implementere metoden i konstruksjonsverktøyet er påbegynt. Innføring av parametrisert geometri er en stor oppgave, og mye arbeid gjenstår. Først må metoden implementeres og testes, så må det utvikles grafiske brukergrensesnitt for å gjøre funksjonaliteten tilgjengelig for brukeren.

Det er utført innledende forsøk med en algoritme som automatisk konstruerer baner mellom bevegelige hindre. Algoritmen viser fram konseptet og danner et rammeverk for videre eksperimentering. Den legger banenoder på den interpolerte rette linja mellom utvalgte noder på par av hinder. Rutiner for automatisk oppdatering av banene når hindre beveger seg er utvikla. Den inkluderte algoritmen

er kun ment til å vise fram konseptet, og har som følger flere svakheter. Videre bør mer avanserte algoritmer utvikles. Banene som konstrueres bør nytte kurva kanter for bedre posisjonskontroll. Dessuten bør overganger mellom kanter holdes kontinuerlige for å unngå store akselerasjoner og minske påkjenningene på fartøyet.

Selv om mye har blitt utretta i arbeid med denne masteroppgaven, gjenstår det fortsatt mye arbeid. Videre testing kommer til å avdekke feil som må utbedres. Og det vill alltid være mulighet for finpuss og justeringer, både i det grafiske brukergrensesnittet og i rammeverket rundt. Mye forskning gjenstår rundt temaet "Styresystem for autonome fartøy", og videreutvikling av GeoMod og konstruksjonsverktøyet for baner danner grunnlag for opptil flere framtidige masteroppgaver.

Referanser

[1] Sigbjørn Aukland, Christian Anders Finnstrom, and Sven Fjeldaas, *Programvare for autonome arbeidende undervannsfartøy - Plattformuavhengig innlasting av dynamiske programvarebibliotek under kjøretid*, (2015).

[2] Robert Bogue, *Underwater robots: A review of technologies and applications*, Industrial Robot **42** (2015), no. 3, 186–191.

[3] T. Bray, *The JavaScript Object Notation (JSON) Data Interchange Format*, `https://tools.ietf.org/html/rfc7159`, March 2014.

[4] Ck Chan and St Tan, *Determination of the minimum bounding box of an arbitrary solid: An iterative approach*, **79** (2001), no. 15, 1433–1449.

[5] Gerald E. Farin, *Curves and Surfaces for CAGD : A Practical Guide*, The Morgan Kaufmann Series in Computer Graphics and Geometric Modeling, vol. 5th ed, Morgan Kaufmann, San Francisco, CA, 2002 (English).

[6] Sven Fjeldaas, *Basic Computational Geometry: Curves and Surfaces*, preliminary issue ed., NTNU, The Norwegian University of Science and Technology Department of Machine Design and Materials Technology, February 2011.

[7] ———, *GeoMod: General -> Introduction (GeoMod/entrance_06/hlp_ent/g_introduction.html)*, `GeoMod/entrance_06/hlp_ent/g_introduction.html`, 2015.

[8] Martin Lygre Furevik, *Trajectory Control for Autonomous Working Submersibles*, (2016).

[9] Erich Gamma, *Design patterns : Elements of reusable object-oriented software*, Addison-Wesley professional computing series, Addison-Wesley, Reading, Mass., 1995.

[10] GNU, *Byteorder(3) - Linux manual page*, `http://man7.org/linux/man-pages/man3/byteorder.3.html`, October 2016.

[11] Øystien Havtad, *Prosjektoppgave ved Institutt for produktutvikling og materialer, Norges Teknisk- Naturvitenskapelige Universitet: Styresystem for autonome fartøy*, (2016).

[12] Daisuke Hirano, Yusuke Funayama, and Takashi Maekawa, *3D shape reconstruction from 2D images*, Computer-Aided Design and Applications **6** (2009), no. 5, 701–710.

[13] Ecma International, *Final draft of the TC39 "The JSON Data Interchange Format" standard - ECMA-404*, http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf, October 2013.

[14] _____, *JSON*, http://json.org/, 2013.

[15] International Organization for Standardization., *ISO/IEC 14882:2011 - Information technology – Programming languages – C++*, https://www.iso.org/standard/50372.html, December 2014.

[16] Ken Joy, *On-Line Computer Graphics Notes: THE CAMERA TRANSFORM*, http://graphics.idav.ucdavis.edu/education/GraphicsNotes/Camera-Transform/Camera-Transform.html, December 1999.

[17] Liri, *Liri*, https://liri.io/, 2017.

[18] The Qt Company Ltd, *Qt - Legal | FAQ*, https://www.qt.io/faq/, 2014-09-04T14:56:47+00:00.

[19] _____, *JSON Save Game Example | Qt Core 5.8*, http://doc.qt.io/qt-5/qtcore-json-savegame-example.html, 2016.

[20] _____, *About Qt - Qt Wiki*, https://wiki.qt.io/About_Qt, February 2017.

[21] _____, *JSON Support in Qt | Qt Core 5.8*, http://doc.qt.io/qt-5/json.html, 2017.

[22] _____, *Qt Overviews | Qt 5.8*, https://doc.qt.io/qt-5/overviews-main.html, 2017.

[23] _____, *Signals & Slots | Qt Core 5.8*, http://doc.qt.io/qt-5/signalsandslots.html, 2017.

[24] _____, *Variables | qmake Manual*, http://doc.qt.io/qt-5/qmake-variable-reference.html#includepath, 2017.

[25] Duncan Marsh, *Applied Geometry for Computer Graphics and CAD*, London, GBR: Springer London, London, 2005.

[26] Microsoft, *Symbolic Links (Windows)*, https://msdn.microsoft.com/en-us/library/windows/desktop/aa365680(v=vs.85).aspx, 2017.

[27] Tommi Mikkonen, *Dynamic Linking*, Chichester, UK: John Wiley & Sons, Ltd, Chichester, UK, 2007.

[28] Bjarne Stroustrup, *The C++ Programming Language Forth Edition*, April 2014.

[29] Camilo Tafur and Dan MacIssac, *Right-hand Rules*, `http://physicsed.buffalostate.edu/SeatExpts/resource/rhr/rhr.htm`, 2003.

[30] The LXDE Team, *About / LXQt*, `http://lxqt.org/about/`, 2017.

[31] KDE Webmasters, *KDE - About KDE*, `https://www.kde.org/community/whatiskde/`, 2017.

[32] Zhijie Xu, Shuai Wei, Nianfeng Wang, and Xianmin Zhang, *Trajectory Planning with Bezier Curve in Cartesian Space for Industrial Gluing Robot*, Intelligent Robotics and Applications: 7th International Conference, ICIRA 2014, Guangzhou, China, December 17-20, 2014, Proceedings, Part II (Xianmin Zhang, Honghai Liu, Zhong Chen, and Nianfeng Wang, eds.), Springer International Publishing, Cham, 2014, pp. 146–154.

[33] Zhang Rubo, Gu Guochang, and Zhang Guoyin, *AUV obstacle-avoidance based on information fusion of multi-sensors*, Intelligent Processing Systems, 1997. ICIPS '97. 1997 IEEE International Conference on **2** (1997), 1381–1384.

Vedlegg

A Avtale om gjennomføring av masteroppgave

NTNU

Avtale om gjennomføring av masteroppgave

Denne avtalen bekrefter at masteroppgavens tema er godkjent, at et veilederforhold er etablert, og at partene (student, veileder og institutt) er kjent med og har akseptert gjeldende retningslinjer for gjennomføring av masteroppgaven. Avtalen er videre regulert av lovverk, studieforskrift og studieplanen for masterprogrammet.

1. Personopplysninger

Etternavn, fornavn **Havstad, Øystein Solheim**	Fødselsdato **08. februar 1993**
E-post **oysteish@stud.ntnu.no**	Telefon

2. Institutt og studieprogram

Fakultet **Fakultet for ingeniørvitenskap**	
Institutt **Institutt for maskinteknikk og produksjon**	
Studieprogram **Master i ingeniørvitenskap og IKT**	Studieretning **IKT, produktutvikling og materialer**

3. Avtalens varighet

Oppstartsdato **15. januar 2017**	Innleveringsfrist* **11. juni 2017**
Hvis avtale om deltidsstudier, angi prosent:	

** Inkludert 1 uke ekstra p.g.a påske*
All veiledning må være gjennomført innenfor avtaleperioden.

4. Arbeidstittel for oppgaven

Styresystem for autonome fartøy *Autonom konstruksjon av tredimensjonale baner*

5. Veiledning

Veileder **Sven Fjeldaas**

Normert veiledningstid er **25 timer** for 30 studiepoengs (siv.ing) og **50 timer** for 60 studiepoengs (realfag) masteroppgaver.

6. Tematisk beskrivelse

Autonome fartøy bør kunne modifisere gitte baner og konstruere nye, for deretter å følge banene. Baner bør kunne gis både i fritt rom og relativt til flater. Oppgaven går ut på å etablere en programvare som gjør det mulig å eksperimentere med algoritmer for autonom konstruksjon av baner. Det finnes en noe utvidet problembeskrivelse i en separat tekst. Denne binnes inn som vedlegg til besvarelsen.

7. Andre avtaler

Tilleggsavtale	**Ikke aktuelt**
Søknad om godkjenninger (REK, NSD)	**Ikke aktuelt**
Risikovurdering (HMS) gjennomført	**Ikke aktuelt**

Vedlegg (oversiktsliste)

8. Underskrifter

Vilkår	Dato	Underskrifter
Jeg har lest og akseptert gjeldende retningslinjer for masteroppgaven		Studenten
Jeg påtar meg ansvaret for veiledning av studenten etter gjeldende retningslinjer		Veileder
Institutt/Fakultet godkjenner opplegget for masteroppgaven		Fakultet/Institutt

PS: Avtalen ble signert i januar og levert til instituttet før jeg rakk å skanne inn den signerte utgaven.

B Utvida problembeskrivelse

Autonome fartøy bør kunne modifisere gitte baner og konstruere nye, for deretter å følge banene. Baner bør kunne gis både i fritt rom og relativt til flater.

Oppgaven går ut på å etablere en programvare som gjør det mulig å eksperimentere med algoritmer for autonom konstruksjon av baner.

Den geometriske modellereren GeoMod er tilgjengelig og egnet for oppgaven, men kandidaten bør legge forholdne til rette ved å:

- Innføre dynamisk linking av konstruksjonsverktøy for baner.
- Justere detaljer i programvaren, blant annet ved å innføre:
  - Baner med orientering i tillegg til posisjon.
  - Et skille mellom planlagte- og faktiske baner, overganger mellom disse og utfallsrommet for videre navigasjon.
  - Lagring av geometriske data på fil.
  - Vurdere muligheten for parametrisk styring av data lagret på fil.
- Reetablere en tidligere versjons program for skulpturerte flater.

Kandidaten bør gjøre innledende forsøk med en algoritme som forsøker å legge baner mellom bevegelige hindre. Kuber og pyramider kan tjene som hindre.

C Risikovurdering

NTNU	Kartlegging av risikofylt aktivitet	Utarbeidet av	Nummer	Dato
		HMS-avd.	HMSRV2601	22.03.2011
		Godkjent av		Erstatter
HMS		Rektor		01.12.2006

Enhet: MTP **Dato: 15.01.17**

Linjeleder: Torgeir Welo

Deltakere ved kartleggingen (m/ funksjon): Øystein Havstad (student), Sven Fjeldaas (ansvarlig veileder)
(Ansv. veileder, student, evt. medveiledere, evt. andre m. kompetanse)

Kort beskrivelse av hovedaktivitet/hovedprosess: Masteroppgave Øystein Havstad. Styresystemer for autonome fartøy: Autonom konstruksjon av tredimensjonale baner

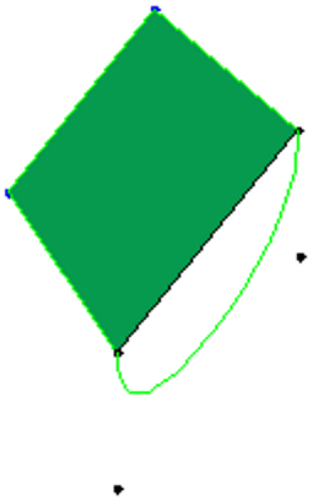
Er oppgaven rent teoretisk? (JA/NEI): JA *«JA» betyr at veileder innestår for at oppgaven ikke inneholder noen aktiviteter som krever risikovurdering. Dersom «JA»: Beskriv kort aktiviteteten i kartleggingsskjemaet under. Risikovurdering trenger ikke å fylles ut.*

Signaturer: *Ansvarlig veileder:* Sven Fjeldaas *Student:* Øystein Havstad

ID nr.	Aktivitet/prosess	Ansvarlig	Eksisterende dokumentasjon	Eksisterende sikringstiltak	Lov, forskrift o.l.	Kommentar
0	Utvikling av programvare: Vanlig kontorarbeid		Ikke relevant	Ikke relevant	Ikke relevant	

D Demonstrasjon av det nye filformatet

Vedlagt er et eksempel på ei fullstendig fil i det nye formatet. Fila beskriver geometrien vist nedenfor. Les om det nye filformatet i kapittel 9.4.



```
{
    "basis": {
        "offset": {
            "x": 0,
            "y": 0,
            "z": 0
        },
        "x": {
            "x": 1,
            "y": 0,
            "z": 0
        },
        "y": {
            "x": 0,
            "y": 1,
            "z": 0
        },
        "z": {
```

```
            "x": 0,
            "y": 0,
            "z": 1
        }
    },
    "curves": [
        {
            "controlPoints": [
                "cp1",
                "cp2"
            ],
            "curvetype": "bezier",
            "edge": "BC",
            "type": "default"
        }
    ],
    "description": "Eksempelbane laga for aa demonstrere det nye
    filformatet",
    "edges": [
        {
            "color": {
                "alpha": 1,
                "blue": 0,
                "green": 255,
                "red": 0
            },
            "headNode": "B",
            "name": "AB",
            "negRegion": "flate",
            "negRegionNextEdge": "AD",
            "negRegionNextEdgeAlong": true,
            "posRegion": "nullptr",
            "posRegionNextEdge": "nullptr",
            "posRegionNextEdgeAlong": true,
            "supportEdge": "",
            "tailNode": "A",
            "type": "default"
        },
        {
            "color": {
                "alpha": 1,
                "blue": 0,
                "green": 255,
                "red": 0
            },
            "headNode": "C",
            "name": "BC",
            "negRegion": "flate",
            "negRegionNextEdge": "AB",
```

```
            "negRegionNextEdgeAlong": false,
            "posRegion": "nullptr",
            "posRegionNextEdge": "nullptr",
            "posRegionNextEdgeAlong": true,
            "supportEdge": "",
            "tailNode": "B",
            "type": "default"
        },
        {
            "color": {
                "alpha": 1,
                "blue": 0,
                "green": 255,
                "red": 0
            },
            "headNode": "D",
            "name": "CD",
            "negRegion": "flate",
            "negRegionNextEdge": "BC",
            "negRegionNextEdgeAlong": false,
            "posRegion": "nullptr",
            "posRegionNextEdge": "nullptr",
            "posRegionNextEdgeAlong": true,
            "supportEdge": "",
            "tailNode": "C",
            "type": "default"
        },
        {
            "color": {
                "alpha": 1,
                "blue": 0,
                "green": 255,
                "red": 0
            },
            "headNode": "D",
            "name": "AD",
            "negRegion": "nullptr",
            "negRegionNextEdge": "nullptr",
            "negRegionNextEdgeAlong": true,
            "posRegion": "flate",
            "posRegionNextEdge": "CD",
            "posRegionNextEdgeAlong": false,
            "supportEdge": "",
            "tailNode": "A",
            "type": "default"
        }
    ],
    "name": "FilformatDemo",
    "nodes": [
```

```
        {
            "name": "A",
            "type": "default",
            "x": 3,
            "y": 3,
            "z": 0
        },
        {
            "name": "B",
            "type": "default",
            "x": 3,
            "y": 7,
            "z": 0
        },
        {
            "name": "D",
            "type": "default",
            "x": 8,
            "y": 4,
            "z": 0
        },
        {
            "name": "C",
            "type": "default",
            "x": 9,
            "y": 8,
            "z": 0
        },
        {
            "name": "cp1",
            "type": "default",
            "x": 5,
            "y": 9,
            "z": 0
        },
        {
            "name": "cp2",
            "type": "default",
            "x": 11,
            "y": 10,
            "z": 0
        }
    ],
    "parameters": [
    ],
    "regions": [
        {
            "color": {
                "alpha": 1,
```

```
                "blue": 6,
                "green": 154,
                "red": 78
            },
            "entryEdge": "AB",
            "entryEdgeAlong": false,
            "name": "flate",
            "type": "default"
        }
    ]
}
```

FilformatDemo.gmpath.json