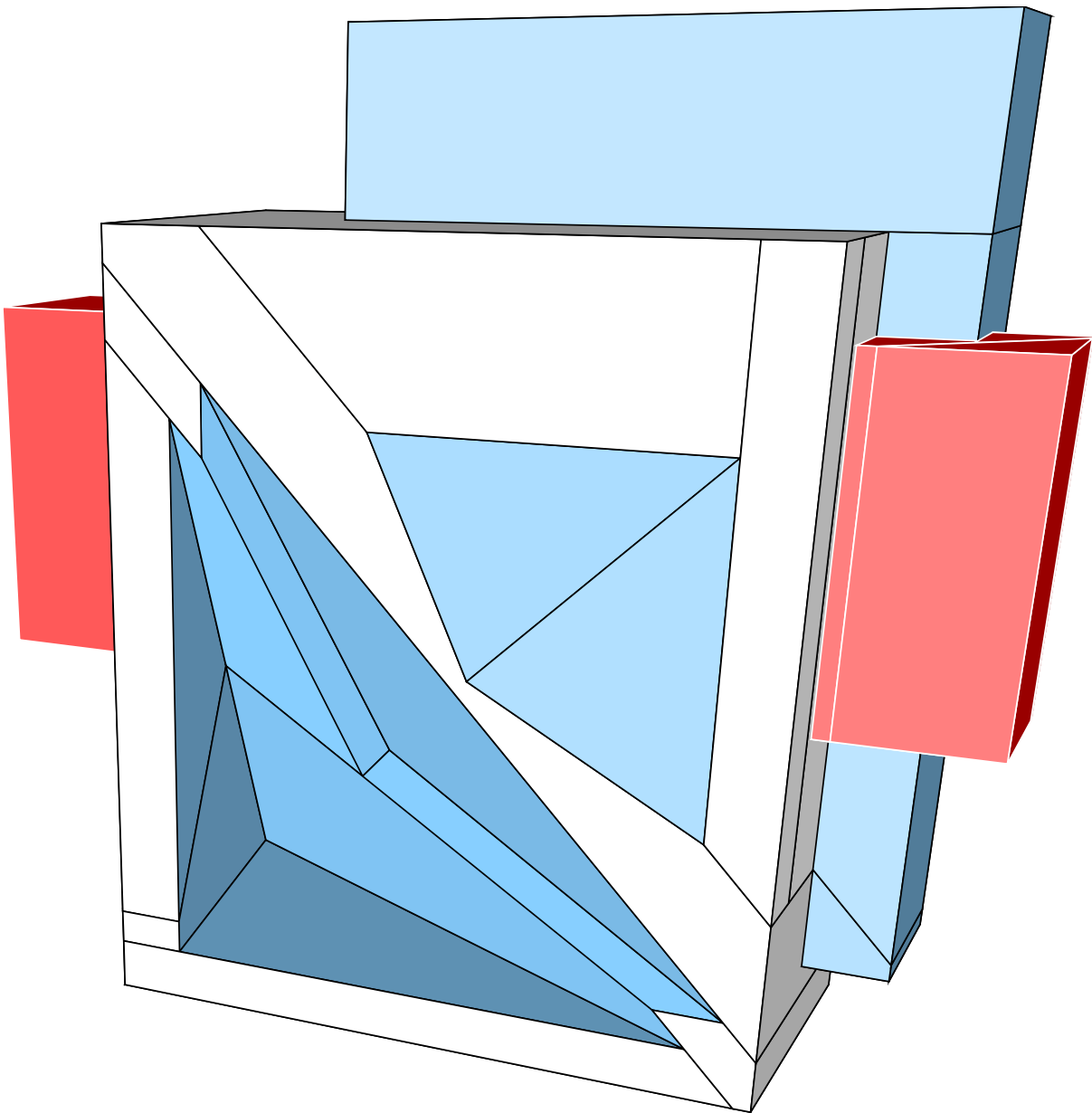


Perspektivisk afbildning

K-Opgave i Datalogi 0GA

Lars Holm Nielsen
Jørgen H. Seland

17. december 2003



Datalogisk Institut
Københavns Universitet

Indhold

1	Sammenfatning	4
1.1	Læsevejledning	4
1.2	Konventioner	4
2	Problemorienteret analyse	5
2.1	Indlæsning	5
2.2	Scene	5
2.3	Projektion	6
2.4	Udskrivning	6
3	Programmeringsovervejelser	7
3.1	Datafil til scene	7
3.2	Scene til polygoner	8
3.3	Polygoner til træ	10
3.4	Træ til billede	11
4	Programbeskrivelse	16
4.1	Datastrøm	16
4.2	Moduler	16
4.3	Hovedprogram (program.sml)	20
4.4	Oversigt over køretiden	20
5	Brugervejledning	20
5.1	Scenebeskrivelse	20
5.2	Analyse af scenebeskrivelse	21
5.3	Generering af perspektivisk billede	21
5.4	Fejlmeldinger	21
6	Afprøvning	22
6.1	Krav	23
6.2	Ækvivalensklasser for inddata	26
6.3	Afprøvningstilfælde	28
	Litteratur	33
A	Kodekonventioner	34
B	Kildekode	34
C	Afprøvningsscener	54

1 Sammenfatning

Vores formål med denne opgave har været at lave et program, som ud fra en beskrivelse af en række genstande i rummet kan lave en perspektivisk afbildning af disse genstande. Vores ambitioner har været at lave et program, der løser problemet på en enkelt og elegant måde, samt gennem intensiv afprøvning af programmet at dokumentere at programmet fungerer korrekt.

Den følgende rapport indeholder vores løsning på problemet. Vi bruger dog en anden metode, end den der er angivet i brevkassen til at oversætte planer i rummet til planer i skærmfladen. Dette skyldes at brevkassens metode først kom til efter, at vi havde designet og implementeret vores udgave.

Gennem den eksterne afprøvning har vi vist, at programmet overholder specifikationen på alle vigtige punkter. Vi har dog ikke formået at bringe alle fejlmeldinger op på et klart nok niveau.

1.1 Læsevejledning

Rapporten er skrevet, så den kan læses af en almindelig studerende på Datalogi 0 GA. Det forudsættes at opgaveteksten er gennemlæst og forstået, samt at læseren har kendskab til at multiplicere en vektor med en matrix.

1.2 Konventioner

Vektorer er skrevet med pil over (fx " \vec{v}_{op} "), punkter med store bogstaver (fx " P "), og matricer med store fede bogstaver (fx " \mathbf{M}^{-1} ").

Vi refererer til en vektors enkelte komponenter ved at bruge vektoren som sænket skrift, fx vil \vec{v}_{ops} x-komponent blive $x_{\vec{v}_{op}}$.

2 Problemorienteret analyse

Overordnet kan problemstillingen opdeles i

- En indlæsningsdel, som varetager indlæsningen af en scenebeskrivelse.
- En scenedel, som varetager repræsentationen af en række objekter i rummet.
- En projektionsdel, som genererer den perspektiviske afbildning af scenen.
- En udskrivningsdel, som varetager udskrivningen af den perspektiviske afbildning.

2.1 Indlæsning

Det er et krav fra opgaven, at scenebeskrivelsen noteres ved brug af S-udtryk (jf. 2.1 i opgaveteksten), og at alle mål skal specificeres i centimeter.

2.1.1 Fejl i inddata

Opgaveteksten specificerer ikke hvorvidt, om inddata kan indeholde fejl, men med vores kendskab til programmering ved vi at chancerne er meget store. Inddata skal derfor overholde syntaksen for S-udtryk og dybdesyntaksen. Gør den ikke det, skal der gives en let forståelig fejlmelding.

Fejl i S-udtryks-delen af syntaks er imidlertid defineret som at ligge udenfor opgavens problemområde (jf. afsnit 2.1 i opgaveteksten).

2.2 Scene

En scene indeholder en række objekter, som kan være af forskellige former. For at gøre beskrivelsen af objekter så generel som mulig, ønsker vi at adskille placering og form. Dette vil lette eventuelle justeringer i en scene, da man derved kan flytte objekter udelukkende ved at ændre deres position. Man skal altså fx kun flytte et punkt i stedet for otte punkter i rummet for at flytte et parallelepipedum uden at ændre dets form.

Opgaven specificerer, at vi kun skal behandle to typer objekter: tetraeder og parallelepipeder. Det vil dog ikke være noget problem at repræsentere andre typer objekter, da de alle kan beskrives med en stedvektor og en form (fx kan en kugle specificeres med en stedvektor og en radius).

2.2.1 Tetraeder

En tetraeder specificerer vi med en stedvektor til en hjørnespids, samt en vektor fra denne hjørnespids til hvert af de tre andre hjørnespidser (jf. ovenstående ønske om at adskille placering og form).

2.2.2 Parallelepipedum

Et parallelepipedum specificerer vi på næsten samme måde som en tetraeder. En stedvektor til en hjørnespids og en vektor til hvert af de tre nabohjørner¹. En stor fordel ved at specificere det på denne måde, er at vi sikre at brugeren virkelig angiver et parallelepipedum, og ikke bare en arbitrær kasse hvor sidefladerne to og to ikke er parallelle, bevidst eller ubevidst.

¹Hjørner forbundet med en kant, er nabohjørner til hinanden.

2.2.3 Farvelægning af objekter

Der er flere muligheder for at farvelægge objekter. Ser vi udelukkende på tetraeder og parallelepiped, kunne man vælge at give hver sideflade en farve. Det vil imidlertid være omstændeligt at definere hvilken sideflade, der skal have en given farve. Ydermere skal farven af forskellige type objekter, defineres på forskellig måde.

En langt simplere metode, som vi har valgt, er at give hele objektet en farve. Vi giver godt nok afkald på noget funktionalitet, men til gengæld kan alle typer objekter nu specificeres udfra stedvektor, form og farve.

2.3 Projektion

Det er et krav, at vi skal bruge centralprojektion til at opbygge den perspektiviske afbildning af scenen. Centralprojektion angives, med

Øjepunkt Dette må ikke ligge i billedplanet da projektionen i dette tilfælde ikke er defineret.

Op-vektor² Denne må ikke være parallel med normalvektoren til billedplanet, og må ikke være lig med $\vec{0}$.

Op-vektoren eksisterer for at gøre transformationen af billedplanet i rummet til billedplanet på skærmen entydig, og angiver en vektor i scenens koordinatsystem, der skal pege opover på billedfladen.

Billedplan Et gyldigt plan i rummet.

Når et given punkt P skal projiceres ind på billedplanet, dannes synslinien $P_{\text{øje}}P$, som skæres med billedplanet. Skæringspunktet er da afbildningspunktet på billedplanet. Dog må P ikke ligge i et plan gennem øjepunktet, som er parallelt med billedplanet (herefter “øjeplanet”), da synslinien derved ikke skærer billedplanet.

Projiceringen til billedplanet giver scenen projiceret på et plan i rummet. Vi skal imidlertid bruge punkternes koordinater i planets koordinatsystem, da det er indholdet af planet, der skal repræsenteres i uddata. For at hente disse koordinater, skal planet sammenfalde med skærmens billedflade. Dette kræver i de fleste tilfælde en transformation mellem de respektive koordinatsystemer.

2.3.1 Hvilke objekter skal tegnes

Matematisk set kan centralprojektion bruges til at tegne alle punkter i rummet på nær dem i øjeplanet. Imidlertid vil punkter bagved³ øjepunktet blive spejlet ved projektionen (jf. figur 1). Derfor vil vi ikke tegne polygoner, der ligger bagved øjepunktet.

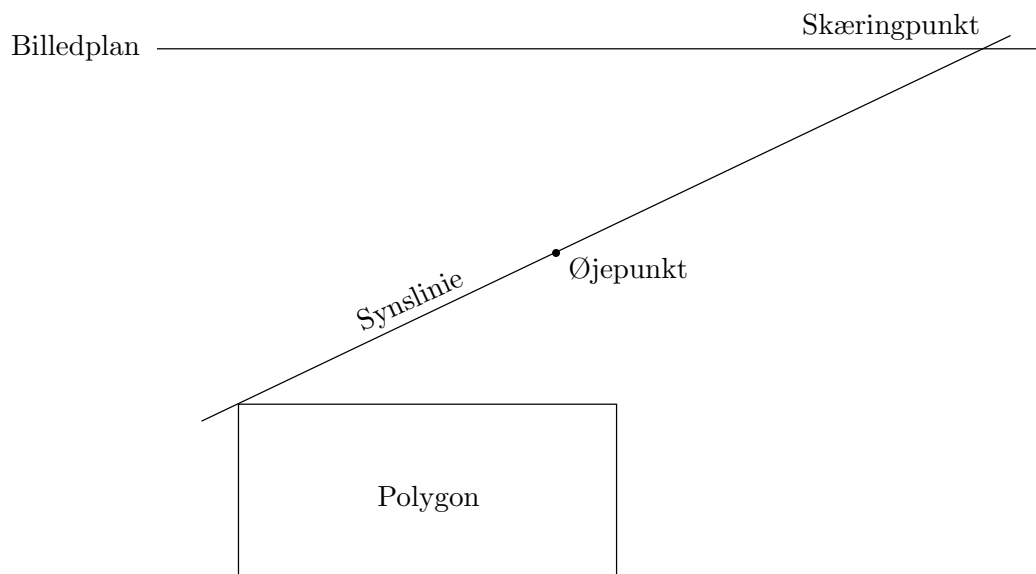
Matematisk set vil punkter, jo tættere på øjeplanet de ligger, blive afbilledet med større og større koordinatværdier i billedplanet. De der ligger meget tæt på øjeplanet, vil derfor få meget store koordinatværdier. Dette skyldes at synslinien vil konvergere med øjeplanet.

2.4 Udskrivning

Opgaven specificerer, at uddataformatet skal være i FIG 3.2 format. Dette format giver mulighed for mange typer tegneredskaber med mange egenskaber. Imidlertid skal vi kun kunne tegne polygoner udfyldt med en farve. Netop polylinie tegnereskabet opfylder dette krav, og vi kan derfor begrænse os til at anvende dette.

²Jf. k-opgave brevkassen, som betragtes som en fortsættelse af opgaveteksten iflg. opgaven

³“Bagved” defineres i forhold til øjeplanet som det modsatte halvrum af det billedplanet ligger i.



Figur 1: Illustration af hvordan en polygon spejles, når den ligger bagved billedplanet.

3 Programmeringsovervejelser

Ud fra analysen kan man læse et antal transformationer, dataene skal gennem inden de er egnet for den endelige projektion, hver dikteret af den type inddata det næste trin skal bruge:

1. Datafil til scene, dvs. indlæsning af scene-definition.
2. Scene til polygoner
3. Polygoner til træ
4. Træ til billede

3.1 Datafil til scene

Iflg. opgavebeskrivelsen skal dette trin udføres vha. det eksterne modul “A”, der analyserer S-udtryk og ud fra dem danner et træ af S-udtryk i et nemt processerbart format. Dette modul forholder sig imidlertid ikke til dybdesyntaks, så det bliver op til vores program at tjekke.

Eksakt hvordan dette tjek skal forløbe, overlader vi til den enkelte implementering, da det er meget tæt knyttet til det enkelte programmeringssprog.

Den følgende dybdesyntaks bruges til at definere en scene (i BNF-form⁴, nonterminals⁵ er i kursiv, faktisk tekst i gåseøjne):

*scene = form**

form = tetraeder | parallelepipedum

tetraeder = “tetra” vektor vektor vektor vektor farve

parallelepipedum = “para” vektor vektor vektor vektor farve

vektor = “vekt” tal tal tal

⁴Bacchus-Naur Form - jf. [Foley et. al, 1997].

⁵Elementer i sproget der kan reduceres til underelementer.

farve = "farve" heltal heltal

Parametrene for de forskellige udtryk er som følger (se afsnit om "Scene til polygoner" for de specifikke variables geometriske betydning):

tetraeder og parallelepipedum: P , \vec{v}_1 , \vec{v}_2 , \vec{v}_3 og f .

vektor: x, y og z-værdi

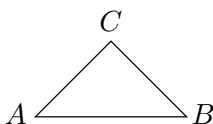
farve: indeks for kantfarve, indeks for fyldfarve, begge i intervallet $[-1, 543]^6$.

3.2 Scene til polygoner

BSP-træ-algoritmen forholder sig til plan og polygoner⁷, og vi skal derfor have de større former over i dette format, inden de kan smides ind i træet.

Alle polygonerne i vores scener er sluttede, så vi definerer et polygon som en liste af tre eller flere punkter der i) ligger i det samme plan og hvor ii) ingen punkter er ens, der iii) definerer en linje, der afgrænser en flade i rummet uden tykkelse, og hvor der iv) implicit skal trækkes en linje mellem det sidste og det første punkt.

Hvis man for eksempel har polygonen (A, B, C) , bliver det som på figur 2.



Figur 2: Illustration af en polygon

Eftersom polygonerne løsrives fra deres former, skal de også indeholde information om hvilken farve, de skal tegnes med.

Hver form skal oversættes til polygoner på den måde, der er beskrevet i de følgende afsnit.

3.2.1 Tetraeder

Tetraederen er defineret ved et punkt, tre vektorer og en farve. Disse kalder vi hhv. P , \vec{v}_1 , \vec{v}_2 , \vec{v}_3 og f . Ud fra punktet og vektorerne kan vi danne de følgende fire hjørnepunkter (jf. fig 3):

$$\begin{array}{ll} A = P & C = P + \vec{v}_2 \\ B = P + \vec{v}_1 & D = P + \vec{v}_3 \end{array}$$

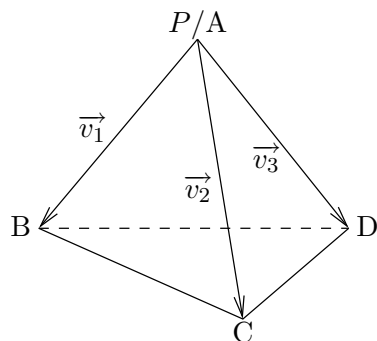
En tetraeder har fire sider, og giver derfor de følgende polygoner - alle med farve f :

$$(A, B, C) \quad (A, D, B) \quad (A, C, D) \quad (B, D, C)$$

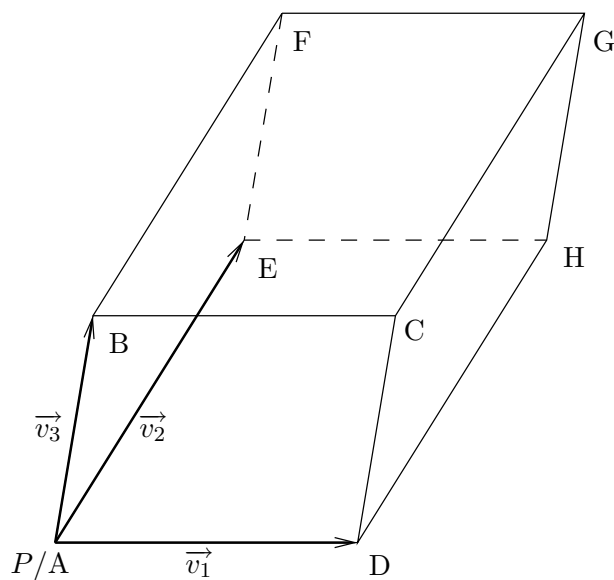
3.2.2 Parallelepipedum

Parallelepipedumet er også defineret ved et punkt, tre vektorer og en farve. Hvis vi bruger den samme benævnelse som oven for, bliver parallelepipedumets otte hjørnepunkter som følger (jf. 4):

$$\begin{array}{llll} A = P & C = B + \vec{v}_1 & E = P + \vec{v}_2 & G = F + \vec{v}_1 \\ B = P + \vec{v}_3 & D = P + \vec{v}_1 & F = B + \vec{v}_2 & H = E + \vec{v}_1 \end{array}$$



Figur 3: Illustration af tetraeder.



Figur 4: Illustration af parallelepipedum

Et parallelepipedum har seks sider, og danner følgende polygoner alle igen med farven f :

$$\begin{array}{lll} (A, D, C, B) & (C, D, H, G) & (H, G, F, E) \\ (G, F, B, C) & (A, B, F, E) & (D, A, E, H) \end{array}$$

3.2.3 Bemærkninger

Planaritet Et parallelepipedum introducerer et potentielt problem, idet det har firsidige polygoner: Da et plan er entydigt defineret ved tre punkter, er tresidige polygoner altid er planare⁸, mens firsidige polygoner må have punkter, der ikke nødvendigvis ligger i det samme plan. Imidlertid giver vores definition af parallelepipedumet (se definitionen af parallelepipedumets punkter) at de firsidige polygoner nødvendigvis er planare. Derfor er ikke-planare polygoner ikke et problem inden for det understøttede sæt forme.

Punktrækkefølge Vi har sørget for at nummerere punkterne i rækkefølge konsekvent mod uret⁹, af to grunde:

1. For polygoner med over tre punkter er dette et krav, for at de skal blive korrekte.
2. Normalvektorerne vil pege i den samme retning for alle fladerne. Hvis vi fx ønsker at benytte prikprodukt til belysning, vil dette blive meget nemmere hvis normalvektorerne peger i den samme retning

Det er værd at notere, at hvis \vec{v}_1 , \vec{v}_2 og \vec{v}_3 bliver specificeret i modsat rækkefølge (med uret, set fra P), af det vi har antaget her, vil normalvektorerne pege i den modsatte retning. Det vil imidlertid være konsekvent over alle formens flader, og pkt. 2 holder derved fortsat.

Punktoverlap Eftersom polygonerne skal have en udstrækning (jf. 3.2), skal de to følgende punkter være overholdt:

1. Ingen af vektorerne \vec{v}_1 , \vec{v}_2 eller \vec{v}_3 må være lig med $\vec{0}$.
2. \vec{v}_1 , \vec{v}_2 eller \vec{v}_3 skal være forskellige fra hinanden.

3.3 Polygoner til træ

På dette trin i processen har vi nu en liste af polygoner, hver med en farve. Vi skal nu opbygge et BSP-træ ud af disse.

Vores BSP-træ er sammensat af deltræer, som er i en af følgende forme:

- En knude, der består af to deltræer, et plan der adskiller deltræerne samt en liste af polygoner der ligger i planet.
- Det tomme træ

Opbygningen laves med følgende algoritme:

1. Vi tager den første polygon i listen af polygoner, der endnu ikke er en del af træet, og udregner ligningen for det plan, der indeholder denne polygon.

⁶Dette er `xf`'s udvalg af farver.

⁷Sådan som den er beskrevet i opgaveteksten. Det er imidlertid også mulig at beskrive fx en mængde generelle konvekse volumer i et sådant træ

⁸Dvs. at alle punkterne ligger i det samme plan.

⁹set udefra

2. Vi opdeler så de resterende polygoner i tre grupper — A, B og C — i forhold til deres punkters afstand til planet. Nærmere bestemt den skalar d der for et givet punkt P , tilfredsstiller ligningen

$$A(x_P + dx_{\vec{N}}) + B(y_P + dy_{\vec{N}}) + C(z_P + dz_{\vec{N}}) + D = 0,$$

hvor \vec{N} er normalvektoren og A , B , C og D koefficienterne i planligningen til α .

- De polygoner der har alle sine punkter på planet ($d = 0$), går i gruppe B.
- De polygoner der har alle sine punkter enten på planet eller på den ene side af planet ($d \leq 0$) går i A.
- De polygoner der har alle sine punkter enten på planet eller på den anden side af planet ($d \geq 0$) går i C.
- De polygoner der har punkter både med $d < 0$ og $d > 0$, køres gennem splitte-algoritmen nedenunder, og resultatet smides i hhv. A og C.

3.3.1 Splitte-algoritme

Vi starter med tre tomme polygoner A, B og C, for hhv. $d \leq 0$, $d = 0$ og $d \geq 0$. Derefter gør vi følgende, for hvert punkt i polygonen der skal splittes:

1. Vi klassificerer det foregående punkt og dette punkt i en af grupperne $d < 0$, $d > 0$. Hvis $d = 0$, regnes det til samme side, som det sidste punkt der ikke lå i $d = 0$ ¹⁰.

Hvis alle punkterne i polygonen ligger i $d = 0$, lægges de i B, og det næste punkt i algoritmen springes over.

2. Hvis det foregående punkt og dette punkt er på forskellig side af planet, så lægger vi skæringspunktet mellem planet og vektoren mellem de to punkter i både A og C.

Hvis skæringspunktet sammenfalder med et af punkterne ($t \in \{0, 1\}$ i skæringspunkt-formelen), udelades skæringspunktet fra den aktuelle mængde, dvs. det foregående punkts mængde ved $t = 0$ og det aktuelle punkts mængde ved $t = 1$ (se fig 6b for et eksempel).

Hvis det aktuelle punkt er det første i polygonen, bruges det sidste punkt i polygonen som “det foregående”.

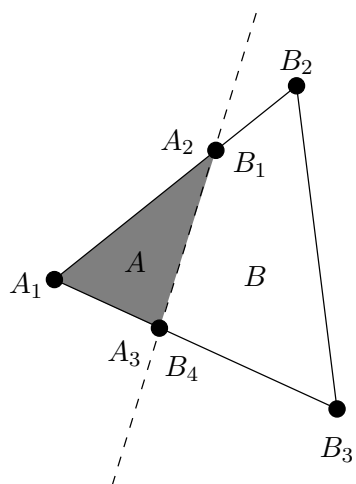
Planaritet Selv om splitte-algoritmen må lave polygoner med ret mange punkter i komplicerede scener, vil den give planare polygoner som resultat, hvis de oprindelige polygoner er planare. Dette er fordi et nyt punkt dannes på linjen mellem de oprindelige punkter, og dermed også i det samme plan.

3.4 Træ til billede

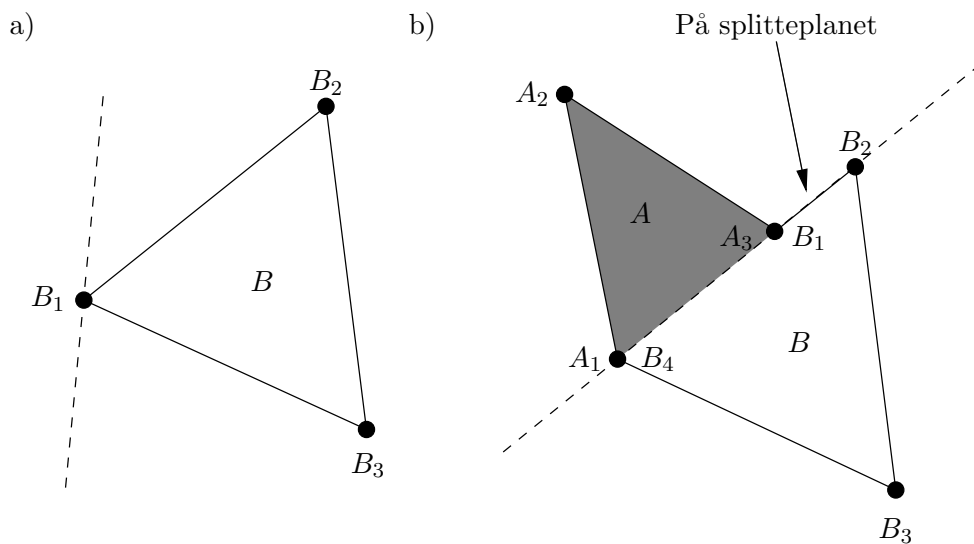
For at danne et billede ud fra vores træ af polygoner, udfører vi en in-order traversering af træet, begyndende ved rod-knuden:

1. Regn ud på hvilken side af knudens plan øjepunktet ligger.
2. Udfør denne algoritme på det undertræ, der er på den modsatte side af øjepunktet.
3. Projicer alle polygonerne i denne knude og skriv dem til filen.

¹⁰Figur 6a viser vigtigheden af at gruppere $d = 0$ ’erne med de nærliggende punkter. Havde vi fx automatisk lagt dem i A, ville vi i dette tilfælde fået en polygon uden areal

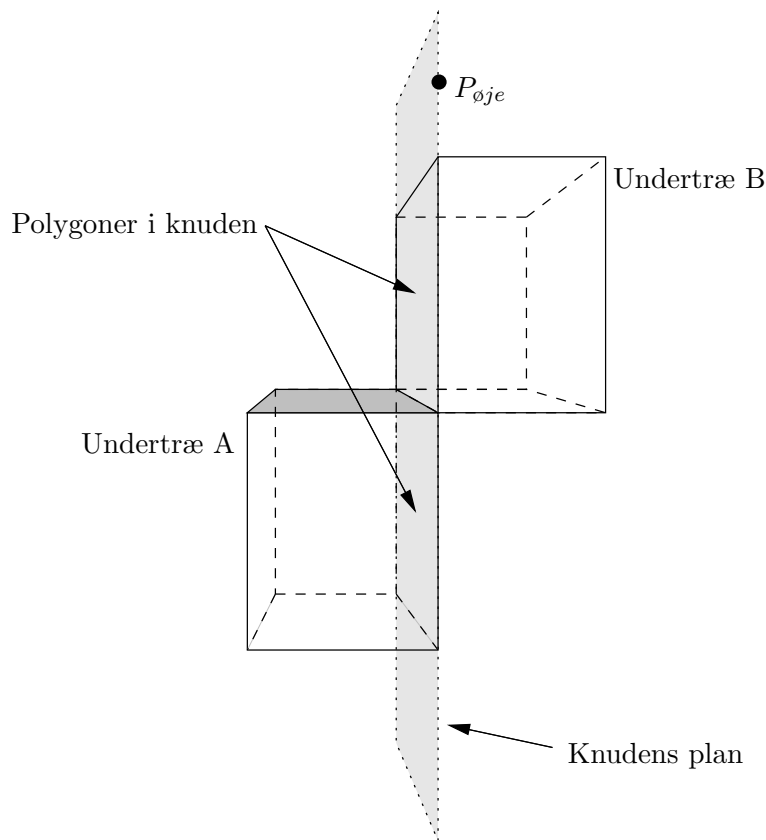


Figur 5: En tresidig polygon splittet i to nye polygoner A og B, med hhv. tre og fire sider.



Figur 6: Polygon med hhv. punkt og linjesegment på planet.

Figur 7: Situationen hvor øjepunktet ligger i knudens plan. α er ikke tegnet ind, da det er uden betydning for tegnerækkefølgen.



4. Udfør denne algoritmen på det undertræet, der er på den samme side af denne knudens plan som øjepunktet.

Hvis øjepunktet ligger i knudens plan (dette specielle tilfælde udelukkes i “4.1 Fra BSP-træ til malerækkefølge” i opgaveteksten), er det ligegyldigt hvilket undertræ vi behandler først: Vi kan se ud fra figur 3.4, at i en sådan situation overlapper undertræ A og B ikke hinanden. Den polygon der er i knudens plan og nærmest $P_{\text{øje}}$, vil ikke blive overlappet af den, der er fjernere — uanset tegnerækkefølge — da polygonerne ikke har nogen tykkelse, og derfor ikke vil være synlige set fra siden, som på figur 3.4 (jf. 3.2).

3.4.1 Klipping

Det der er bagved, på og lige foran øjeplanet giver os problemer hhv. med realismen og med matematikken. Vi har to løsninger:

- Vi definerer os et klippeplan i en afstand d fra øjet. Vi kører så alle polygonerne gennem splitte-algoritmen (og forkaster det, der er på den forkerte side) inden vi begynder projiceringsprocessen.
- Vi gør som overfor, men bruger α i stedet for at definere et nyt plan. Dette gør, at man ikke kan flytte klippeplanet uafhængig af α .

Vi vælger at bruge α som klippeplan, da vi 1) regner uafhængigheden som et mindre problem, da dette udelukkende er interessant, hvis man skal vise objekter, der er på den samme side af billeplanet som øjepunktet, og 2) dette tager endnu en komplicerende faktor ud af programmet.

3.4.2 Formel

Iflg. de krav der er fremsat i analysen, skal vi bruge en formel, der

1. projicerer alle punkter i scenen over på α ,
2. translaterer scenen, sådan at α s origo flyttes til skærmfladens origo,
3. retter α s normalvektor, \vec{N} efter den positive Z-akse, og
4. retter op-vektoren \vec{v}_{op} opover på skærmfladen, dvs. i XY-planet.

Projicering Projicering af scenen er givet i stor del i opgaveteksten, men vi gengiver den her for kompletthedens skyld. For et givet punkt P ønsker vi at finde punktet P' , hvor linjen mellem øjepunktet $P_{\emptyset je}$ og P skærer α . Vi definerer P' ved

$$\overrightarrow{P_{\emptyset je}P'} = t\overrightarrow{P_{\emptyset je}P}, \quad (1)$$

og sætter ind i planligningen for α for at finde parameteren t for skæringen. Vi får derved:

$$t = -\frac{Ax_{P_{\emptyset je}} + By_{P_{\emptyset je}} + Cz_{P_{\emptyset je}} + D}{Ax_{\overrightarrow{P_{\emptyset je}P}} + By_{\overrightarrow{P_{\emptyset je}P}} + Cz_{\overrightarrow{P_{\emptyset je}P}}}.$$

Ved indsætning i 1, får vi koordinaterne for skæringspunktet.

Translatering Translateringen til det nye origo er todelt: Først finder et passende origo for α . Vi vælger det punkt på α , der er nærmest $P_{\emptyset je}$. Dette punkt har de følgende fordele:

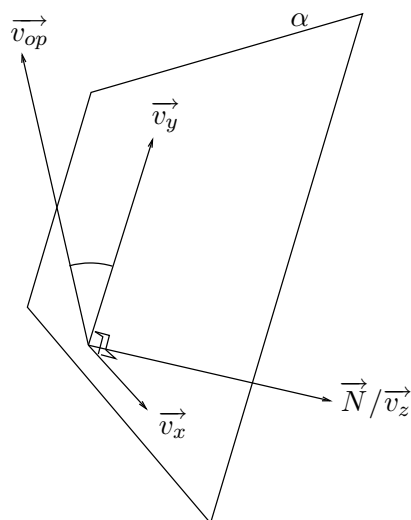
1. Billedets centrum vil flytte på sig, hvis man flytter på øjet, hvilket antagelig er hvad en typisk bruger ville forvente.
2. Billedets centralakse er givet ved linjen $P_{\emptyset je} + t\vec{N}$, hvilket gør det nemt at manipulere hvad, der vil være i centrum af billedet: Planet vil give synsretningen, og øjet vil give betragterens ståsted.

Det nærmeste punkt i α fra $P_{\emptyset je}$ er givet ved skæringspunktet $P_{\alpha origo}$ mellem linjen $P_{\emptyset je} + t\vec{N}$ og α . \vec{N} s koordinater er givet i α s planligning som $\langle A, B, C \rangle$, og vi får:

$$\begin{aligned} A(x_{P_{\emptyset je}} + tA) + B(y_{P_{\emptyset je}} + tB) + C(z_{P_{\emptyset je}} + tC) + D &= 0 \\ Ax_{P_{\emptyset je}} + By_{P_{\emptyset je}} + Cz_{P_{\emptyset je}} + D + t(A^2 + B^2 + C^2) &= 0 \\ t &= -\frac{Ax_{P_{\emptyset je}} + By_{P_{\emptyset je}} + Cz_{P_{\emptyset je}} + D}{A^2 + B^2 + C^2} \end{aligned}$$

$P_{\alpha origo}$ er da givet ved indsætning i linjeligningen.

Andre trin i translateringen er nemt; vi subtraherer simpelthen $P_{\alpha origo}$ fra alle punkterne i scenen.



Figur 8: Opbygning af nyt koordinatsystem

Rotation De sidste to punkter kan vi udføre på to måder:

1. Først dreje om X- og Y-aksen for at rette \vec{N} efter Z-aksen, derefter dreje om Z-aksen for at rette \vec{v}_{op} opover.
2. Se på drejningerne som en overgang mellem koordinatsystemer, og danne en matrix der udfører transformationen.

Vi vælger den sidste løsning, da den matematisk set er nemmere. Imidlertid skal man bruge et komplet defineret koordinatsystem for at bruge denne metode, og det kræver en ekstra vektor i tillæg til \vec{N} og \vec{v}_{op} . Den danner vi med den følgende metode (illustreret på figur 8; vi kalder akse-vektorerne i det nye system for \vec{v}_x , \vec{v}_y og \vec{v}_z):

1. Vi sætter $\vec{v}_z = \vec{N}$.
2. Vi danner så vores X-akse, ved at tage krydsproduktet $\vec{v}_z \times \vec{v}_{op}$. Dette kan vi gøre, da vi ønsker et ortogonalt koordinatsystem, dvs. et system hvor akserne står vinkelret på hinanden.
3. Vi danner til sidst Y-aksen, ved at tage krydsproduktet $\vec{v}_z \times \vec{v}_x$ ¹¹.

Det vi nu skal lave, er en lineær transformation mellem koordinatsystemerne. Iflg. [Messer, 1994] kan vi, hvis vi kender resultatet \vec{v}_x , \vec{v}_y og \vec{v}_z af en lineær transformation T på enhedsvektorerne, opstille en matrix der udfører den samme lineære transformation på alle andre vektorer. Denne matrix er givet ved

$$\mathbf{M} = [\vec{v}_x \quad \vec{v}_y \quad \vec{v}_z],$$

der er matricen dannet ved at bruge de transformerede enhedsvektorerne som søjler. Transformationen T er da givet ved

$$T(\vec{x}) = \mathbf{M}\vec{x}.$$

¹¹I modsætning til hvad man kunne tro, kan vi *ikke* bruge \vec{v}_{op} som Y-akse, da den kan stå i alle orienteringer i forhold til \vec{N} undtaget parallelt, og derfor ikke garanterer ortogonalitet (jf. fig 8).

Dog ønsker vi ikke at ændre på længden til vektorene i scenen ved transformationen, og vi skal derfor sørge for at vores “transformerede enhedsvektorer” fortsat har længde 1. Vi skal derfor normalisere¹² \vec{v}_x , \vec{v}_y og \vec{v}_z inden vi opstiller matrixen.

T er imidlertid den transformation vi skal udføre, hvis vi ønsker at oversætte en vektor udtrykt i α s koordinatsystem og til det nuværende¹³, men det er den modsatte operation vi er ude efter — at gøre α s koordinatsystem til det gældende. At lave vores matrix om, til at udføre dette er imidlertid nemt: For en matrix der udfører en lineær transformation, er den inverse transformation givet ved den transponerede matrix, dvs. matrixen med rækker og søjler byttet om.

$$M^{-1} = \begin{bmatrix} x_{\vec{v}_x} & y_{\vec{v}_x} & z_{\vec{v}_x} \\ x_{\vec{v}_y} & y_{\vec{v}_y} & z_{\vec{v}_y} \\ x_{\vec{v}_z} & y_{\vec{v}_z} & z_{\vec{v}_z} \end{bmatrix}$$

Vi udfører så transformationen ved at multiplicer alle punkterne i scenen med denne matrix. For et givet punkt P_i får vi det roterede punkt P'_i ved

$$P'_i = M^{-1}P_i$$

4 Programbeskrivelse

4.1 Datastrøm

Datastrømmen gennem programmet er angivet på figur 9. Hele scenen går gennem de forskellige operationer som en enhed, undtagen de to sidste trin, hvor operationerne udføres på per-polygon-niveau (først projiceres og tegnes den første polygonen, derefter projiceres og tegnes den anden etc.).

4.2 Moduler

Vi opdeler programmet i moduler, som er løseligt centreret om datatyperne. Bemærk, at vi vil se bort fra konstanter i udledning af størrelsesordenen af køretiden for centrale funktioner i programmet.

4.2.1 Basismoduler

De følgende tre moduler er relative simple i forhold til de andre, og er grundigt dokumenteret i kildekoden. Vi vil derfor kun kort nævne dem her.

Vektor3D Indeholder datatypen for en vektor og funktioner til at behandle tredimensionale vektorer.

Matrix3x3 Indeholder datatypen for en 3x3 matrix og funktioner til behandling af 3x3-matricer.

Plan Indeholder `plan` typen og plan-manipuleringsfunktioner:

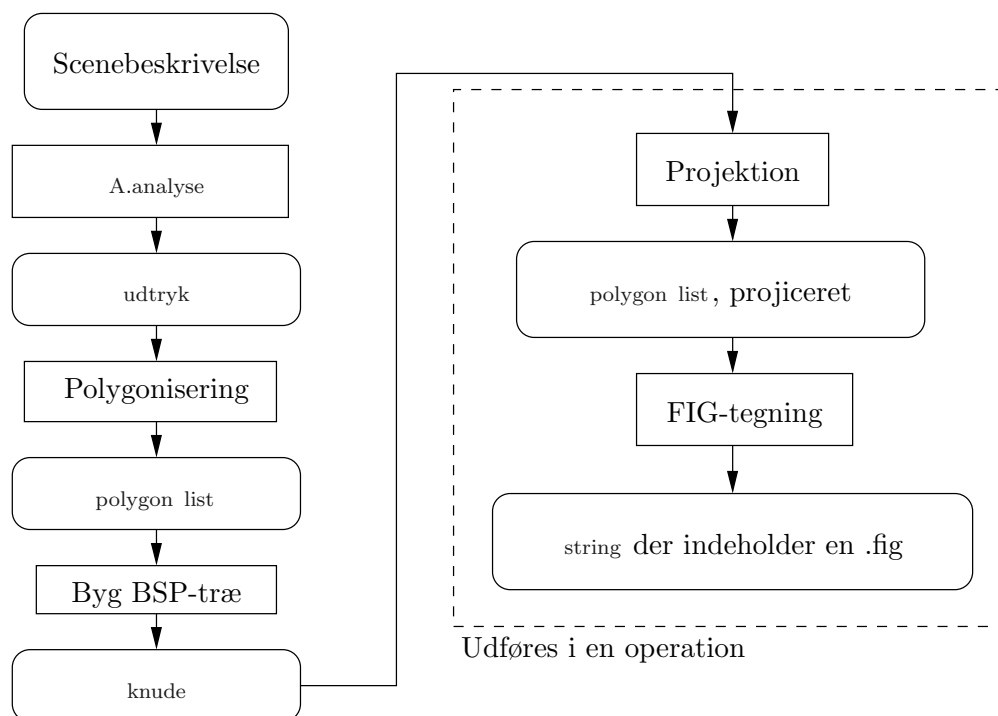
FIG Indeholder datatypen `fig` der repræsenterer en `.fig`-fil, og funktioner til at manipulere denne.

4.2.2 Polygon

I modulet er typen `polygon` defineret. Modulet indeholder generelle polygon-manipuleringsfunktioner samt funktioner til at oversætte en scenebeskrivelse i form af et udtryk til en liste af polygoner (jf. afsnit 3.1).

¹²En vektor normaliseres til længde 1 ved at multiplicere den med $\frac{1}{l}$, hvor l er vektorens nuværende længde.

¹³Med “nuværende” menes det koordinatsystem, vektorens koordinater for øjeblikket er udtrykt i.



Figur 9: Datastrøm i programmet: Almindelige kasser er operationer, afrundede kasser er data.

Køretid for splitte-algoritmen Splitte-algoritmen (jf. afsnit 3.3.1) er implementeret i Polygon modulet i funktionen `SplitPolygon`. Denne funktion er ret central da den både bruges i opbygningen af BSP-træet, samt i central projektionen. Det er derfor nødvendigt at kende størrelsesordenen af køretiden for algoritmen, for at kunne sige noget om størrelsesordenen af køretiden for BSP-træet og centralprojektionen.

```

(**
 * Splitter polygonet
 *
 * @type polygon * plan -> polygon option * polygon option * polygon option
 *)
fun SplitPolygon (p as Polygon (farve , ps) , alpha) =
  let
    val startside = FindSidstePunktMedSide (ps , alpha)
  in
    if startside <> 0 then
      let
        val (pa , pb) = KlassificerPunkter (startside , hd (rev ps) , ps
          , alpha)
        in
          (* Undgå at lave ikke-polygoner *)
          (if length pa > 2 then SOME (Polygon (farve , pa)) else NONE,
            NONE,
            if length pb > 2 then SOME (Polygon (farve , pb)) else NONE)
        end
      else
        (NONE , SOME(p) , NONE)
    end
  end
  
```

Udledning af køretid Vi ønsker nu at bestemme køretiden, T , af `SplitPolygon`. Lad n være antallet af punkter i en polygon. Først kaldes funktionen `FindSidstePunktMedSide`, som rekursivt kalder sig selv i alt n gange. Hvis alle punkter ligger i planet α er `startside = 0`, hvorved `SplitPolygon` returnerer med det samme. Altså bliver køretiden T af samme størrelsesorden som $\mathcal{O}(n)$.

Hvis derimod alle punkter *ikke* ligger i planet, kaldes `KlassificerPunkter`. Et af parametrene til `KlassificerPunkter` vender listen af punkter med biblioteksfunktionen `rev`. Dennes køretid er i størrelsesordenen $\mathcal{O}(n)$ (jf. [Hansen et. al, 1999] side 257), men kaldes imidlertid på samme niveau som `FindSidstePunktMedSide`, og bidrager derfor ikke til størrelsesordenen af køretiden som sådan, da vi allerede er oppe på $\mathcal{O}(n)$.

`KlassificerPunkter` kalder to funktioner, `TilfoejTilSide` og `TilfoejSkaering`, som begge er har en køretid af størrelsesordenen $\mathcal{O}(1)$, og er derfor ikke af betydning. Vi ser til sidst på `KlassificerPunkter` selv, og finder at den kalder sig selv rekursivt i alt n gange. Vi får derved at køretiden for kaldet til `KlassificerPunkter` er af sammen størrelsesorden som $\mathcal{O}(n)$, på samme måde som `rev` og `FindSidstePunktMedSide`.

Køretiden for splitte-algoritmen bliver da,

$$T = T_{\text{FindSidstePunktMedSide}} + T_{\text{rev}} + T_{\text{KlassificerPunkter}} = n + n + n = 3n.$$

T er med andre ord af størrelsesordenen $\mathcal{O}(n)$ — den samme som hvis alle polygoner ligger i knudes plan.

Vi har i gennemsnit kun 3-4 punkter pr. polygon. Når vi senere skal bruge køretiden for splitte-algoritmen, kan vi derfor som en tilnærmelse bruge, at køretiden er af størrelsesordenen $\mathcal{O}(4) = \mathcal{O}(1)$. Det er dog samtidig klart, at denne tilnærmelse ikke holder, hvis programmet senere udvides med former, som har polygoner med mange punkter. Ydermere kan komplekse scener bevirke, at antallet af punkter i en polygon bliver så stort, at tilnærmelsen heller ikke holder her.

4.2.3 BSPTræe

Modulet indeholder typen for en BSP-træknude samt funktioner til behandling af BSP-træer. Specielt funktionerne `BygTrae` og `FoldMaler` skal nævnes her.

Opbygning af BSP-træ Selve implementeringen af opbygningen af BSP-træet følger programmeringsovervejelserne i afsnit 3.3, dog med den modifikation af vi køre splitte-algoritmen på alle polygoner, i stedet for kun dem som skal splittes. Dette gør vi, da koden til at opdele polygoner i de omtalte grupper, næsten vil være identisk med store dele af splitte-algoritmen.

```

fun BygTrae [] = TomtTrae
  | BygTrae (p :: ps) =
      let
          val plan = Polygon.PolygonPlan p
          val (a, b, c) = OpdelPolygoner (ps, plan, [], [], [])
      in
          Knude (BygTrae a, BygTrae c, plan, p :: b)
      end

```

Vi ønsker nu finde størrelsesordenen af køretiden for `BygTrae`. Lad n være antallet af polygoner, og f brøkdelen af de polygoner som skal opdeles som *ikke* ligger i en knudens plan¹⁴. I værste tilfælde ligger der kun en polygon i knudens plan, hvilket medfører at $f = 1$. Vi bruger derfor

¹⁴Knudens plan beregnes ud fra den første polygon p i listen. De resterende polygoner ps , opdeles med funktionen `OpdelPolygoner`. f er altså brøkdelen af polygonerne i ps som ikke ligger i samme plan som p . Det er samtidig klart, at f er forskellig i hvert rekursivt kald.

som en tilnærmelse at $f = 1$, hvorved f er ens for alle kald til `BygTrae`. Fra programbeskrivelsen af splitte-algoritmen ved vi, at vi i vores tilfælde kan bruge at splitte-algoritmens køretid er i størrelsesordenen $\mathcal{O}(1)$. Køretiden for opdelingen af m polygoner i forhold til knudens plan, vil da være af størrelsesordenen $\mathcal{O}(m)$ (jf. funktionen `OpdelPolygoner`). Ved det første kald til `BygTrae`, er tidsforbruget da $T_n = n - 1$, når vi ser bort fra de to rekursive kald `BygTrae a` og `BygTrae c`. Antag nu at f_a er brøkdelen af de $f \cdot (n - 1)$ polygoner, som er på den ene side af knudens plan, og at f_c er brøkdelen af de $f \cdot (n - 1)$ polygoner, som er på den anden side af knudens plan. Da vil køretiden ved de to rekursive kald til `BygTrae a` og `BygTrae c`, være givet ved,

$$T_{n-1} = f_a \cdot f \cdot (n - 2) + f_c \cdot f \cdot (n - 2) = f \cdot (n - 2)$$

eftersom der gælder at $f_a + f_b = 1$. Køretiden for opbygning af BSP-træet, idet vi har sat $f = 1$, bliver da

$$T = T_n + T_{n-1} + \dots + T_1 = (n - 1) + (n - 2) + (n - 3) + \dots + 1 = \frac{1}{2} \cdot (n - 1) \cdot (n).$$

Vi får altså at køretiden for `BygTrae` er af samme størrelsesordenen som $\mathcal{O}(n^2)$.

Traversering af BSP-træ I afsnit 3.4 er gennemgået, hvordan vi kommer fra et BSP-træ til et billede. I algoritmen projicerer vi polygonerne, mens vi traverserer træet i malerækkefølge. Vi har valgt at lave en mere generel funktion, da vi senere kunne være interesseret i at traversere træet i malerækkefølge, mens vi udfører en given operation på polygonerne i træet. Funktionen fungerer på tilsvarende måde som biblioteksfunktionerne `foldr` og `foldl`, idet den akkumulerer en funktion `f` fra en startværdi `b` over polygonerne i BSP-træet i malerækkefølge.

```

fun FoldMaler f b TomtTrae oejepunkt = b
  | FoldMaler f b (Knude (vTrae, hTrae, plan, polygoner)) oejepunkt =
      let
          val oejeside = Plan.Sammenlign (plan, oejepunkt)
          val b2 = FoldMaler f b (if oejeside > 0 then vTrae
                                else hTrae) oejepunkt
          val b3 = foldr f b2 polygoner
      in
          FoldMaler f b3 (if oejeside > 0 then hTrae else vTrae)
                        oejepunkt
  end

```

Vi ønsker nu at bestemme størrelsesordenen af køretiden for `FoldMaler`. Lad m være antallet af knuder i BSP-træet, n antallet af polygoner i BSP-træet og k størrelsesordenen af køretiden for `f`. Vi har at hver knude i gennemsnit indeholder $\frac{n}{m}$ polygoner, samt at $n \geq m$ idet hver knude mindst indeholder én polygon.

Vi ser midlertidigt bort fra størrelsesordenen af køretiden af de rekursive kald. Først udregnes hvilken side af knodes plan øjepunktet ligger på. Størrelsesordenen af køretiden for dette kald er $\mathcal{O}(1)$, hvorved det ikke har betydning for størrelsesordenen af køretiden af `FoldMaler`. Efter at have fundet den akkumulerede værdi af det ene undertræ, kalder `FoldMaler` nu `foldr` på polygonerne i knuden. Eftersom `foldr` akkumulerer funktionen `f` over alle polygonerne i knuden, og da størrelsesordenen af køretiden for `f` er k , har vi at størrelsesordenen af dette kald er $\mathcal{O}(k \cdot \frac{n}{m})$.

Da vi besøger hver knude præcis én gang, gennem de rekursive kald, og eftersom størrelsesordenen af køretiden af hvert kald¹⁵ er $\mathcal{O}(k \cdot \frac{n}{m})$, får vi størrelsesordenen af køretiden for `FoldMaler` til $\mathcal{O}(k \cdot \frac{n}{m} \cdot m) = \mathcal{O}(k \cdot n)$. Det interessante ved dette resultat er, at `FoldMaler` er uafhængig af antallet af knuder i BSP-træet.

¹⁵Når vi ser bort fra de to rekursive kald.

4.2.4 CentralProjektor

Indeholder `projektion` typen samt funktioner til at projicerer en polygon ind på billedplanet.

Projicering af én polygon Funktion `Projicer` projicerer en polygon ind på billedplanet, ud fra et øjepunkt, en op-vektor og et billedplan. Den implementerer metoden som er beskrevet i afsnit 3.4.2 og 3.4.1. Vi ønsker nu ud fra et kort ræsonnement at bestemme størrelsesordenen af køretiden for `Projicer`. Ræsonnementet¹⁶ er, at vi i gennemsnit kun har 3-4 punkter pr. polygon, og derfor som en tilnærmelse bruger at størrelsesordenen af køretiden er $\mathcal{O}(1)$. Det skal stadig stå klart, at denne tilnærmelse ikke kan bruges når der kommer for mange punkter polygonerne.

4.3 Hovedprogram (program.sml)

Programmet indeholder de to påkrævede funktioner, `BygTrae` af typen `udtryk -> BSPtree` og `GenererFig` af typen `BSPtree -> projektion -> string`.

`GenererFig` kalder `BSPTrae.FoldMaler` med `ProjicerOgUdskrivPolygon` som den akkumulerende funktion. Størrelsesordenen af køretiden for `ProjicerOgUdskrivPolygon` er $\mathcal{O}(1)$, idet vi igen bruger at vi i gennemsnit kun har 3-4 punkter pr. polygon. Vi får derved at størrelsesordenen af køretiden er $\mathcal{O}(n)$ for projiceringen af n polygoner i et BSP-træ på et billedplan¹⁷.

4.4 Oversigt over køretiden

For overblikkets skyld, bringer vi her en opsummering af køretiden for centrale funktioner i programmet.

Beskrivelse	Funktion	Køretid ¹⁸
Splitte-algoritmen	<code>Polygon.SplitPolygon</code>	$\mathcal{O}(1)$
Opbygning af BSP-træ	<code>BSPTrae.BygTrae</code>	$\mathcal{O}(n^2)$
Projicering af BSP-træ	<code>GenererFig</code>	$\mathcal{O}(n)$

5 Brugervejledning

Programmet kan ud fra en rumlig scenebeskrivelse generere et perspektivisk billede i FIG 3.2 Format, når den suppleres med information om hvordan genstandene i scenebeskrivelsen skal projiceres. Selve programmet ligger i filen `~di030517/k-opgave/program/program.sml`. For at køre programmet, går man ind i mappen hvor `program.sml` ligger, og skriver `mosml programl.sml` i kommandofortolkeren. Derefter vil funktionerne `BygTrae` og `GenererFig` være til rådighed.

5.1 Scenebeskrivelse

Programmet skal bruge en scenebeskrivelse, for at kunne generere et perspektivisk billede. Denne scenebeskrivelse skal anvende notationen som er defineret i afsnit 3.1. Her er et eksempel på en scenebeskrivelse.

```
(scene
  (para (vekt 0.0 5.0 5.0) (vekt 0.0 0.0 10.0) (vekt 10.0 0.0 0.0) (vekt
    0.0 10.0 0.0) (farve 4 20))
```

¹⁶Da størrelsesordenen af køretiden udledes på sammen måde som ved splitte-algoritmen (jf. afsnit 4.2.2), vil vi ikke gå i detaljer her.

¹⁷Jf. den fundne størrelseorden af køretiden, $\mathcal{O}(k \cdot n)$, for `BSPTrae.FoldMaler`, hvor k er størrelsesordenen af køretiden for den akkumulerende funktion (her $\mathcal{O}(1)$).

¹⁸ n er antallet af polygoner.

```
(tetra (vekt 0.0 0.0 5.0) (vekt 5.0 0.0 5.0) (vekt 0.0 5.0 5.0) (vekt
-0.5 -0.5 5.0) (farve 1 20))
)
```

5.2 Analyse af scenebeskrivelse

Før end scenebeskrivelse kan omsættes til et perspektivisk billede, skal den først analyseres, og laves om til et udtryk af funktionen `A.Analyse`. Herefter skal udtrykket omsættes til en intern repræsentation vha. funktionen `BygTrae`. Følgende eksempel viser hvordan en scenebeskrivelse, omdannes til en intern repræsentation for programmet. Det forudsættes her, at variabelen `scenebeskrivelse` er defineret, og indeholder en streng med en scenebeskrivelse.

```
lofn > cd ~/di030517/k-opgave/program
/home/disk22/di030516/k-opgave/program
lofn > mosml program.sml
Moscow ML version 2.00 (June 2000)
Enter 'quit();' to quit.
[opening file "program.sml"]
```

her udskrives en masse information om modulerne

```
[closing file "program.sml"]
- val bsptree = BygTrae (A.analyse scenebeskrivelse);
```

5.3 Generering af perspektivisk billede

Vi er nu klar til at generere et perspektivisk billede ud fra scenebeskrivelsen, under den forudsætning at vi allerede har bestemt et øjepunkt, en op-vektor og et billedplan. Antag at variabelen `bsptree` indeholder den interne repræsentation for programmet af scenebeskrivelse, at (x_1, y_1, z_1) er stedvektor for øjepunktet, at (x_2, y_2, z_2) er op-vektoren, og at (A,B,C,D) beskriver billedplanet.

```
- val projektion = ((x1,y1,z1),(x2,y2,z2),(A,B,C,D)) : projektion;
> val projektion = ((1.0, 1.0, 1.0), (3.0, 3.0, 3.0), (0.0, 0.0, 1.0, 30.0)) :
(real * real * real) * (real * real * real) * (real * real * real * real)
- GenererFig bsptree projektion;
```

Svaret vil være en streng, formateret i formatet FIG 3.2. Denne streng skal gemmes i en fil, førend den kan læses af programmet `xfig`.

5.4 Fejlmeldinger

Følgende er en liste fejlmedinger fra programmet. Ved hver fejlmeding er en kort beskrivelse af hvad der forårsagede fejlmedingen.

- **Uventet: Forventede et flydende tal** - Hvis der i et `vekt` udtryk forekommer et underudtryk, vil denne fejl blive rejst, fx `(vekt (farve 1 2) 3 4)`.
- **Uventet: Forventede "vekt" med tre parametre** - Hvis et `vekt` udtryk ikke har præcis tre elementer, rejses denne fejl.
- **Uventet: Forventede farve indeks** - Hvis der i et `farve` udtryk forekommer et underudtryk, vil denne fejl blive rejst, fx `(farve (vekt 1 2 3) 4)`.
- **Uventet: Forventede "farve" med to parametre** - Hvis et `farve` udtryk ikke har præcis to elementer, rejses denne fejl.

- **Uventet: Forventede "tetra" eller "para" med fem parametre** - Hvis et `tetra` eller `para` udtryk ikke har præcis fem elementer, rejses denne fejl.
- **Uventet: Forventede "scene"** - Hvis det øverste udtryk ikke er `scene` rejses denne fejl, fx (`udtryk (scene)`).
- **Fejlformateret: Ugyldigt flydende tal** - Hvis et tal i et `vekt` udtryk, ikke er et flyende tal, rejses denne fejl.
- **Fejlformateret: Ugyldigt heltal** - Hvis et tal i et `farve` udtryk, ikke er et heltal, rejses denne fejl.
- **Fejlformateret: Farveindeks udenfor [-1,533]** - FIG 3.2 formatet understøtter ikke farver udenfor det angivne interval.
- **Kunne ikke beregne polygonens normalvektor, da den ikke har mindst tre punkter** - Skyldes at to af vektorerne som definerer hjørnerne i en form er sammenfaldne eller en af dem er nulvektoren.

Linjen $\langle 0.0, 10.0, 0.0 \rangle + t \langle 0.5, -10.5, 0.0 \rangle$ er parallel med planet $(0.0, 0.0, 1.0, 0.0)$

- Øjepunktet ligger i planet.

- **Ugyldig plan** - Rejses hvis normalvektoren for et plan er nulvektoren.
- **Vektor med længde 0 kan ikke normaliseres** - Skyldes at to af vektorerne som definerer hjørnerne i en form er sammenfaldne eller en af dem er nulvektoren

6 Afprøvning

Opgaven kræver en ekstern afprøvning. Den almindelige strategi ved ekstern afprøvning er at danne ækvivalensklasser for inddata med hensyn til specifikationen¹⁹ (eller et sæt krav udledt af denne) og finde repræsentative eksempler for disse, for derefter igen at anvende specifikationen på eksemplernes uddata for at afgøre hvorvidt programmet er i overholdelse.

Vi ønsker at effektivisere denne strategi ved kun at rette hver enkelt krav i vores kravsat mod de relevante kvaliteter ved de relevante ækvivalensklassers uddata (som f.eks. farven på en bestemt form i en given sammenhæng). Dette medfører at vi undgår at inddrage hele kravsettet for hvert eneste afprøvningstilfælde.

Vi ender på den følgende strategi:

1. Vi formulerer en liste af præcise krav, som analysen, programmeringsovervejelserne og opgaveteksten stiller til programmet.

Hver krav ledsages (hvis det ikke er umiddelbart givet) af en beskrivelse, af hvilke kvaliteter ved uddata der skal være tilstede, for at kravet skal kunne regnes som overholdt.

2. Vi udformer så ækvivalensklasser for inddata, sådan at vi fokuserer på de områder der er relevante mht. kravsettet.

3. Til hver ækvivalensklasse opstiller vi afprøvningstilfælde for repræsentative medlemmer fra klassen, med tilhørende beskrivelse af de forventede uddata for hvert tilfælde.

Disse forventede uddata er summen af de kvaliteter som de enkelte krav forventer af ækvivalensklassens uddata.

¹⁹I vores tilfælde er specifikationen summen af opgavetekst, analyse og programmeringsovervejelser.

4. Vi opstiller så tabeller, der viser i hvilken grad, programmet tilfredsstiller de enkelte afprøvningstilfælde.
5. Ved at betragte de efterspurgte kvaliteter i hver enkelt krav, kan vi til sidst afgøre, hvorvidt det enkelte krav er overholdt.

Vi har hertil lavet et program, der kører vores afprøvningstilfælde automatisk (`ekstern-afproevning.sml`). Resultaterne af de enkelte afprøvningstilfælde skal dog tjekkes manuelt.

En konsekvens af vores strategi er, at et krav der henviser til en bestemt ækvivalensklasse, kan være overholdt, selv om klassens afprøvningstilfælde fejler, så længe den del af afprøvningstilfældene der er relevant for kravet lykkes. Imidlertid påvirker dette ikke situationen ved fuldstændig overholdning af kravene: Alle kravene er overholdt hvis og kun hvis alle afprøvningstilfældene lykkes. I en situation hvor dette ikke er tilfældet, skal man uanset hvad gå dybere ned i afprøvningsresultaterne, og vi regner derfor strategien som god nok.

6.1 Krav

Ud fra opgaveteksten, analysen og programmeringsovervejelserne har vi formuleret de følgende krav. Konsekvenser (noteret med “ \Rightarrow ”) og evt. måde at teste kravet på er angivet, hvor dette ikke er umiddelbart oplagt.

De forventede kvaliteter ved uddata for de enkelte ækvivalensklasser, er også oplyst, der hvor dette ikke fremgår klart af kravet og/eller konsekvenserne.

1. Mål skal være angivet i cm (jf. 2.1, overholdt)

Afprøves ved ækvivalensklassen 2b i 6.2.2, ved at kontrollere målene til de polygoner der ligger på billedfladen, da disse bliver projiceret på sig selv.

Note om overholdelse: Afprøvningstilfældet for den aktuelle ækvivalensklasse giver forkerte mål (2.1cm i stedet for 2.0cm) i xfigs brugergrænsnit, men regner vi 2.0cm over til $\frac{1}{1200}$ tommer, får vi

$$\frac{2.0 \cdot 1200}{2.54} \frac{1}{1200} \text{tommer} \approx 945 \frac{1}{1200} \text{tommer}.$$

Undersøgelse af fig-filen afslører, at de aktuelle punkter har FIG-målene 945, og vi regner derfor vores program som korrekt i denne henseende

2. Syntaks for s-udtryk skal overholdes (jf. 2.1.1, overholdt)

Bryde hver enkelt krav for syntaks i s-udtryk, og verificér at det bliver opdaget.

Det angives imidlertid i opgaveteksten at ligge udenfor opgavens problemområde (jf. “2.1 Indlæsning” i opgaveteksten). Vi regner derfor dette krav som overholdt, og laver ingen afprøvning for det.

3. Dybdesyntaks skal overholdes (jf. 2.1.1, ikke overholdt)

Bryde hver enkelt krav i dybdesyntaksen, og verificér at det bliver opdaget.

Afprøves ved underklasserne af ækvivalensklasse 2 i 6.2.1: Giver de overhovedet en fejl, er dette krav overholdt.

Note om overholdelse: Det eneste der fejler, er tilfældet hvor \vec{v}_{op} er parallel med \vec{N}_α , hvor vi i stedet for en fejlmelding, får en tom scene, samt at “2.5” bliver godtaget som et gyldigt heltal (2) i farveangivelse.

4. Fejlmeldinger skal være forståelige (jf. 2.1.1, ikke overholdt)

Afprøves gennem alle ækvivalensklasserne i 6.2.1.

5. Et parallelepipedum skal kunne specificeres (jf. 2.2, overholdt)

⇒Et parallelepipedum skal dukke op, hvis det er specificeret og synlig.

Afprøves af alle klasser der indeholder synlige parallelepipedumer, fx gruppen 2(a)iA i 6.2.2 (origo i markøren er et parallelepipedum).

6. Et tetraeder skal kunne specificeres (jf. 2.2, overholdt)

⇒Et tetraeder skal dukke op, hvis det er specificeret og synligt.

Afprøves af alle klasser der indeholder synlige parallelepipedumer, fx gruppen 2(a)iA i 6.2.2 (akseindikatorerne i markøren er tetraedre).

7. En form skal kunne specificeres ud fra et punkt og tre vektorer (jf. 2.2.2, 2.2.1, overholdt)

Specificer de understøttede forme ud i fra et punkt og tre vektorer og verificer at de dukker op.

Afprøves ved underklasserne af ækvivalensklasse 2a og 2b i 6.2.2: Hvis en scene har forme, er de specificeret ud fra et punkt og tre vektorer, da dette er den eneste måde at specificere forme på.

8. $P_{\phi j e}$ må ikke ligge i α (jf. 2.3, overholdt)

⇒En projektion hvor $P_{\phi j e}$ ligger i α giver en fejl.

Afprøves ved ækvivalensklasse 4a i 6.2.1.

9. \vec{v}_{op} må ikke være parallel med \vec{N}_α (jf. 2.3, ikke overholdt)

⇒En projektion der \vec{v}_{op} er parallel med \vec{N}_α giver en fejl.

Afprøves ved ækvivalensklasse 4b i 6.2.1.

10. \vec{v}_{op} skal angive hvad der er op (jf. 2.3, overholdt)

Verificer i en scene der er nem at orientere i, at \vec{v}_{op} faktisk angiver hvad der er op.

Afprøves ved gruppen 2(a)iA i 6.2.2.

11. Det genererede billede skal være en perspektivisk afbildning²⁰ (jf. 2.3, overholdt)

Overholdelse af dette krav fremgår af om underklasserne til ækvivalensklasse 2a og 2b i 6.2.2 producerer data der opfylder de grundlæggende kriterier for perspektiviske afbildninger:

(a) Genstandenes afbildning skal blive mindre desto længere væk de er.

(b) Parallelle linjer skal, hvis de forlænges, mødes, forudsat at de ikke også er parallelle med billedfladen.

12. Programmet skal generere uddata i FIG 3.2-format (jf. 2.4, overholdt)

Verificer at uddata for de forskellige klasser af inddata stemmer overens med FIG 3.2-formatet (specificeret i afsnit 5.1 i opgaveteksten).

Da xfig er streng med sine inddata, og giver fejlmeldinger i alle de tilfælde, hvor den skal udføre korrektioner, regner vi FIG 3.2 som overholdt, hvis xfig accepterer filen uden fejlmeldinger

Afprøves ved alle ækvivalensklasserne i 6.2.2.

²⁰Dette krav er en omformulering af kravet om brug af centralprojektion.

13. **Farveindekser skal ligge i området $[-1, 533]$ (jf. 3.1, overholdt)**
⇒Farveindekser udenfor dette område producerer en fejl.
Afprøves ved ækvivalensklasse 2(c)iii i 6.2.1.
14. **Farveindeks skal give den forventede farve (jf. 3.1, overholdt)**
⇒Forme i uddata skal have de farver, der blev angivet i inddata.
Afprøves ved grupperne 2(a)iB og 2(a)iA i 6.2.2.
15. **Hele formen skal kunne farvelægges (jf. 2.2.3, 3.2, overholdt)**
Dette krav afprøves gennem gruppe 2(a)iB i i 6.2.2: Formen skal have den aktuelle farve over alle sine flader.
16. **Forme uden volumen må ikke forekomme (jf. 3.2, overholdt)**
Forme specificeret med \vec{v}_1, \vec{v}_2 eller \vec{v}_3 lig $\vec{0}$, eller hvor to eller flere af vektorerne er ens, vil have polygoner med sammenfaldende punkter, og er iflg. det refererede afsnit ikke tilladt.
Afprøves ved underklasserne til ækvivalensklasse 3 i 6.2.1: Alle disse skal give en fejl.
17. **Skjulte flader skal ikke være synlige (jf. 3.4.2-3 i opgaveteksten, overholdt)**
Verificer at dette er tilfældet i et almindelig tilfælde, men lav også en scene med cyklisk overlappning.
Afprøves ved ækvivalensklassen 2(a)i og underklasserne af ækvivalensklasse 2(a)ii i 6.2.2: Her skal scenen have et "naturlig" udseende, dvs. at bagsiden af formerne, og dele af former der ligger bagved andre forme, ikke skal være synlige.
18. **Polygoner skal være usynlige set fra siden (jf. 3.4, ikke overholdt)**
Verificer at det er tilfældet, da dette er et kriterium for korrekt håndtering af "øjepunkt på splitteplan"-problematikken.
En måde at teste dette på, er at klippe en form hvor øjet ligger i bundfladens plan, mod billedplanet, sådan at det område hvor bundfladen er, ikke er overskygget af en af sidefladerne.
Afprøves ved ækvivalensklasse 2(b)i i 6.2.2.
Dette punkt kan udbedres ved at
(a) udfiltrere polygoner der ses fra siden, eller
(b) finde en specifik håndtering af det specielle tilfælde. En mulig løsning er at bruge en dybdesortering af de polygoner der ligger i det aktuelle splitteplan i forhold til deres afstand til øjepunktet.
19. **Den del af formerne, der er på $P_{\text{øje}}$ s side af billedplanet, skal ikke tegnes (jf. 3.4.1, overholdt)**
Afprøves ved underklasserne til ækvivalensklasse 2b i 6.2.2.
20. **Forme der er bagved øjeplanet skal ikke tegnes (jf. 2.3.1, overholdt)**
Følger som en umiddelbar konsekvens af krav 19, så kravet behøver ikke egen afprøvning.
21. **Centrum i billedet skal være det nærmeste punkt i α fra $P_{\text{øje}}$ (jf. 3.4.2, overholdt)**
Placer en markør på dette punkt og verificer at den havner i centrum af billedet.
Afprøves ved gruppen 2(a)iC i 6.2.2.

6.2 Ækvivalensklasser for inddata

For at gøre afprøvningen mere effektiv, opdeler vi inddata i ækvivalensklasser. Ækvivalensklasserne er baseret på relationen mellem inddata og det uddata giver: Hvis to forskellige inddata giver det samme uddata, tilhører de den samme ækvivalensklasse.

Anvendt direkte, ville dette for mange programmer give håbløst mange ækvivalensklasser, men programmet skal i en ekstern afprøvning sammenholdes med specifikationen, og denne dækker ikke alle aspekter ved uddata (fx differentierer vores specifikation ikke mellem forme der er 5 cm og 10 cm væk fra billedplanet, så længe formerne er på den samme side af det). Ækvivalensklasserne dannes derfor ud fra de dele af uddata som specifikationen siger noget om.

En vigtig kvalitet ved ækvivalensklasser er, at de altid udgør et system af ikke-overlappende mængder, og typisk er udarbejdet gennem gradvis opdeling i mindre og mindre dele (fx ved dannelse af *-ækvivalens fra k-ækvivalens i en endelige tilstandsmaskine [Epp, 1995]). Et sådan system egner sig meget godt til repræsentation i en hierarkisk struktur, og det er også dette vi har gjort i de følgende afsnit.

Idet ækvivalensklasser skal udarbejdes ud fra kriterier der deler inddata i ikke-overlappende mængder, er det naturligt at begynde med skættet mellem gyldige og ugyldige inddata.

6.2.1 Klasser af ugyldige inddata

1. Inddata med fejl i overfladesyntaks

Skal *ikke* afprøves, jf. “2.1 Indlæsning” i opgaveteksten.

2. Inddata med fejl i dybdesyntaks

Inddata der ikke følger dybdesyntaksen som defineret i 3.1. Disse grupperes igen efter hvilken nonterminal der indeholder fejl:

(a) Fejl i “scene”-syntaks

Ukendte elementer er den eneste fejl, der er omfattet af dette punkt.

(b) Fejl i “form”-syntaks

- i. Ukendte elementer
- ii. Fejl i antal elementer
- iii. Elementer i fejl format

(c) Fejl i “farve”-syntaks

- i. **Fejl i antal elementer**
- ii. **Elementer i fejl format (dvs. ikke heltal)**
- iii. **Elementer udenfor det tilladte område**

Her er det interessant at se på både et tilfælde med indeks over og under det tilladte område. (Mange guider i ekstern afprøvning anbefaler, at der kun er et bestemt gyldigt område i inddata, skal værdier på begge sider af dette område afprøves.)

(d) Fejl i “vektor”-syntaks

- i. Fejl i antal elementer
- ii. Elementer i fejl format (dvs. ikke flydende tal)

3. Inddata med ugyldige forme

I denne klasse har alle inddata der indeholder forme som det ikke er muligt for systemet at repræsentere. Den eneste type forme der ikke er tilladt, er forme uden udstrækning. Af disse findes det imidlertid fire typer:

(a) **Ugyldige tetraedre**

- i. Tetraedre med en el. flere nulvektorer
- ii. Tetraedre med en el. flere sammenfaldende vektorer

(b) **Ugyldige parallelepipedumer**

- i. Parallelepipedumer med en el. flere nulvektorer
- ii. Parallelepipedumer med en el. flere sammenfaldende vektorer

4. Inddata med ugyldig projektion

Projektionen kan også indeholde et antal forskellige typer fejl:

- (a) Projektion med $P_{\phi_{je}}$ i α
- (b) Projektion med \vec{v}_{op} parallel med \vec{N}_α

6.2.2 Klasser af gyldige inddata

I dette afsnit er der anvendt et skøn for at bringe antallet klasser ned på et realistisk niveau. Hvis vi havde et system for automatisk generering — og kontrol — af inddata, ville vi ikke have de samme restriktioner, men vi kan ikke se, hvordan vi skal lave et sådan system for dette program, da uddata i stor grad er billeder, og de aspekter ved uddata der skal kontrolleres, i stor grad er abstrakte kvaliteter ved disse billeder (fx hvilken retning en bestemt form synes at pege).

Den direkte følge af denne restriktion er, at mange underklasser ikke er taget med i alle de overordnede ækvivalensklasserne, da de regnes som vel afprøvet inden i den ene klasse de er taget med i. Dette er naturligvis ikke et idéelt scenario, da uforudsete interaktioner²¹ måske går uopdaget igennem. Vi har derfor forsøgt at udelade klasser, kun i de tilfælde hvor vi ikke kan se noget der måtte indikere at en interaktion skulle kunne opstå. Både mht. vores kendskab til programmer generelt²² samt specifikationen.

Vi begynder med at sortere scenerne efter om de indeholder forme:

1. Scener uden forme

Skal give tomt uddata, men ingen fejl.

2. Scener med forme(a) **Scener med forme foran billedplanet****i. Scener med enkel malerækkefølge**

Dette er scener hvor man kan, ved at tegne de oprindelige flader i en givet rækkefølge, sørge for at kravet 17 om skjulte flader er overholdt.

Punkterne heri er ikke egentlige ækvivalensklasser, da de har overlappende kvaliteter, udover de der er specificeret i de overordnede klasser. De skal i stedet betragtes som grupper af repræsentative medlemmer af klassen “Scener med enkel malerækkefølge”.

A. Markørscene med varierende op-vektor

Her skal vi bruge en scene med markører for de forskellige akse-vektorerne, samt origo, og med farver der gør det nemt at identificere de forskellige markørerne.

Repræsentative tilfælde her, er scener med op-vektor der peger langs hhv. positiv og negativ X, Y og Z-akse, samt en der peger med en 45° hældning.

²¹Virkninger der er resultat af en kombination af to el. flere faktorer

²²Dog ikke i vores kendskab til dette specifikke program; den kundskab må vi ikke på noget punkt benytte os af under ekstern afprøvning.

Derudover skal nogle af disse vektorer have en længde forskellig fra 1, da dette er tilladt iflg. kravene over.

Skal bruges til afprøvning af krav 10.

B. Gruppe af projektioner der giver fuldstændig dækning af en forms flader

Disse skal associeres med en scene med en tetraeder og en med et parallelepipedum.

Én sådan gruppe er de projektioner der afbilder formen fra punkter der er tilstrækkelig langt væk i retning af positiv og negativ X, Y og Z.

Skal bruges til afprøvning af krav 15.

C. Scene og projektion der placerer en markør ved det nærmeste punkt i α fra $P_{\emptyset j e}$

Vi skal bruge en scene med to forme, og projektioner der placerer det nærmeste punkt i α fra $P_{\emptyset j e}$ på spidsen af de enkelte forme.

Skal bruges til afprøvning af krav 21.

ii. Scener med kompliceret malerrækkefølge

Dette er scener hvor man kun kan opfylde krav 17 ved at tegne fladerne i en givet rækkefølge. Dette er et almindelig problemområde for programmer der skal tegne rumlige afbildninger, og derfor en aktuel klasse.

A. Scener hvor ingen forme skærer hinanden

De eneste scener med kompliceret malerrækkefølge hvor figurerne ikke skærer hinanden, er scener med cyklisk overlap.

B. Scener med forme der skærer hinanden

Her er det interessant at se på skæringer der resulterer i hhv. et punkt og en linje, da vi ved at BSP-algoritmer (hvilket er angivet i specifikationen at dette program skal bruge) typisk har problemer med scener der med forme som er sammenfaldende i alle punkter.

(b) Scener med forme der ligger på tværs af billedplanet

Her skal vi bruge en scene med forme, der alle har en mindre — og anderledes farvet — udgave indeni, for klart at vise, hvorvidt formen er blevet klippet mod billedfladen. Derudover skal formenes mål være kendt, da vi også skal bruge denne klassen til at afprøve at den korrekte måleenhed er blevet anvendt.

i. Scener med flader hvis plan indeholder $P_{\emptyset j e}$

Denne klassen eksisterer for at afprøve krav 18.

(c) Scener med forme bagved billedplanet

Disse skal alle give samme uddata: Ingen polylinjer i FIG-filen.

6.3 Afprøvningstilfælde

I de følgende tabeller er `.sud` i filnavne for inddata udeladt, og mapperne `gyldig` og `ugyldig` er forkortet hhv. `g` og `ug`.

6.3.1 Dybdesyntaks

klasse	inddata	forventede uddata	✓/×
2a	ug/2-b	Klar fejlmelding	✓
2(b)i	ug/2-b-i	Klar fejlmelding	✓
2(b)ii	ug/2-b-ii	Klar fejlmelding	✓
2(b)iii	ug/2-b-iii	Klar fejlmelding	✓
2(c)i	ug/2-c-i	Klar fejlmelding	✓
2(c)ii	ug/2-c-ii	Klar fejlmelding	×
2(c)iii	ug/2-c-iii	Klar fejlmelding	✓
2(c)iii	ug/2-c-iv	Klar fejlmelding	✓
2(d)i	ug/2-d-i	Klar fejlmelding	✓
2(d)ii	ug/2-d-ii	Klar fejlmelding	✓

2(c)ii fejler da “2.5” bliver fortolket som et gyldig heltal.

6.3.2 Ugyldige forme

klasse	inddata	forventede uddata	✓/×
3(a)i	ug/3-a-i	Klar fejlmelding	×
3(a)ii	ug/3-a-ii	Klar fejlmelding	×
3(b)i	ug/3-b-i	Klar fejlmelding	×
3(b)ii	ug/3-b-ii	Klar fejlmelding	×

Disse tilfælderne fejler da fejlmeldingen ikke er klar nok:

Fail: Vektor med længde 0 kan ikke normaliseres

6.3.3 Ugyldig projektion

klasse	inddata	forventede uddata	✓/×
4a	$\mathbf{g}/1\text{-a}$, $P_{\emptyset je} = (0, 10, 0)$, $\vec{v}_{op} = \langle 0, -1, 0 \rangle$, $\alpha = (0, 0, 1, 0)$	Klar fejlmelding	×
4b	$\mathbf{g}/1\text{-a}$, $P_{\emptyset je} = (0, 0, -10)$, $\vec{v}_{op} = \langle 0, 0, -1 \rangle$, $\alpha = (0, 0, 1, 0)$	Klar fejlmelding	×

Det første tilfælde fejler da den følgende fejlmeldingen ikke er klar nok:

Fail: Linjen $\langle 0.0, 10.0, 0.0 \rangle + \langle \sim 0.5, \sim 10.5, 0.0 \rangle$ er parallel med planet

Det andet tilfælde fejler da uddata er en gyldig, men tom FIG.

6.3.4 Scener uden forme

klasse	inddata	forventede uddata	✓/×
1	\mathbf{g}/tom , $P_{\emptyset je} = (0, 0, -10)$, $\vec{v}_{op} = \langle 1, 0, 0 \rangle$, $\alpha = (0, 0, 1, 0)$	En gyldig fig uden polylinjer	✓

6.3.5 Markørscene med varierende op-vektor

klasse	inddata	forventede uddata	✓/×
2(a)iA	$\mathbf{g}/1-\mathbf{a}$, $P_{\emptyset je} = (0, 0, -50)$, $\vec{v}_{op} = \langle 1, 0, 0 \rangle$, $\alpha = (0, 0, 1, 10)$	Rød tetraeder skal pege op	✓
2(a)iA	$\mathbf{g}/1-\mathbf{a}$, $P_{\emptyset je} = (0, 0, -50)$, $\vec{v}_{op} = \langle -2, 0, 0 \rangle$, $\alpha = (0, 0, 1, 10)$	Rød tetraeder skal pege ned, og pilene skal være lige så lange som i de andre tilfælde som bruger denne markøren	✓
2(a)iA	$\mathbf{g}/1-\mathbf{a}$, $P_{\emptyset je} = (0, 0, -50)$, $\vec{v}_{op} = \langle 0, 0.5, 0 \rangle$, $\alpha = (0, 0, 1, 10)$	Grøn tetraeder skal pege op, og pilene skal være lige så lange som i de andre tilfælde som bruger denne markøren	✓
2(a)iA	$\mathbf{g}/1-\mathbf{a}$, $P_{\emptyset je} = (0, 0, -50)$, $\vec{v}_{op} = \langle 0, -1, 0 \rangle$, $\alpha = (0, 0, 1, 10)$	Grøn tetraeder skal pege ned	✓
2(a)iA	$\mathbf{g}/1-\mathbf{a}$, $P_{\emptyset je} = (0, -50, 0)$, $\vec{v}_{op} = \langle 0, 0, 2.3 \rangle$, $\alpha = (0, 1, 0, 10)$	Blå tetraeder skal pege op, og pilene skal være lige så lange som i de andre tilfælde som bruger denne markøren	✓
2(a)iA	$\mathbf{g}/1-\mathbf{a}$, $P_{\emptyset je} = (0, -50, 0)$, $\vec{v}_{op} = \langle 0, 0, -1 \rangle$, $\alpha = (0, 1, 0, 10)$	Blå tetraeder skal pege ned	✓
2(a)iA	$\mathbf{g}/1-\mathbf{a}$, $P_{\emptyset je} = (0, 0, -50)$, $\vec{v}_{op} = \langle 1, 1, 0 \rangle$, $\alpha = (0, 0, 1, 10)$	Rød skal pege 45° op til venstre, grøn skal pege 45° op til højre	✓

6.3.6 Gruppe af projektioner der giver fuldstændig dækning af en forms flader

Tetraeder

klasse	inddata	forventede uddata	✓/×
2(a)iB	\mathbf{g}/tetra , $P_{\emptyset je} = (-50, 0, 0)$, $\vec{v}_{op} = \langle 0, 1, 0 \rangle$, $\alpha = (1, 0, 0, 10)$	Tetraeder med kun gule flader	✓
2(a)iB	\mathbf{g}/tetra , $P_{\emptyset je} = (50, 0, 0)$, $\vec{v}_{op} = \langle 0, 1, 0 \rangle$, $\alpha = (1, 0, 0, -10)$	Tetraeder med kun gule flader	✓
2(a)iB	\mathbf{g}/tetra , $P_{\emptyset je} = (0, -50, 0)$, $\vec{v}_{op} = \langle 0, 0, 1 \rangle$, $\alpha = (0, 1, 0, 10)$	Tetraeder med kun gule flader	✓
2(a)iB	\mathbf{g}/tetra , $P_{\emptyset je} = (0, 50, 0)$, $\vec{v}_{op} = \langle 0, 0, 1 \rangle$, $\alpha = (0, 1, 0, -10)$	Tetraeder med kun gule flader	✓
2(a)iB	\mathbf{g}/tetra , $P_{\emptyset je} = (0, 0, -50)$, $\vec{v}_{op} = \langle 0, 1, 0 \rangle$, $\alpha = (0, 0, 1, 10)$	Tetraeder med kun gule flader	✓
2(a)iB	\mathbf{g}/tetra , $P_{\emptyset je} = (0, 0, 50)$, $\vec{v}_{op} = \langle 0, 1, 0 \rangle$, $\alpha = (0, 0, 1, -10)$	Tetraeder med kun gule flader	✓

Parallelepipedum

klasse	inddata	forventede uddata	✓/×
2(a)iB	\mathbf{g}/para , $P_{\emptyset je} = (-50, 0, 0)$, $\vec{v}_{op} = \langle 0, 1, 0 \rangle$, $\alpha = (1, 0, 0, 10)$	Parallelepipedum med kun gule flader	✓
2(a)iB	\mathbf{g}/para , $P_{\emptyset je} = (50, 0, 0)$, $\vec{v}_{op} = \langle 0, 1, 0 \rangle$, $\alpha = (1, 0, 0, -10)$	Parallelepipedum med kun gule flader	✓
2(a)iB	\mathbf{g}/para , $P_{\emptyset je} = (0, -50, 0)$, $\vec{v}_{op} = \langle 0, 0, 1 \rangle$, $\alpha = (0, 1, 0, 10)$	Parallelepipedum med kun gule flader	✓
2(a)iB	\mathbf{g}/para , $P_{\emptyset je} = (0, 50, 0)$, $\vec{v}_{op} = \langle 0, 0, 1 \rangle$, $\alpha = (0, 1, 0, -10)$	Parallelepipedum med kun gule flader	✓
2(a)iB	\mathbf{g}/para , $P_{\emptyset je} = (0, 0, -50)$, $\vec{v}_{op} = \langle 0, 1, 0 \rangle$, $\alpha = (0, 0, 1, 10)$	Parallelepipedum med kun gule flader	✓
2(a)iB	\mathbf{g}/para , $P_{\emptyset je} = (0, 0, 50)$, $\vec{v}_{op} = \langle 0, 1, 0 \rangle$, $\alpha = (0, 0, 1, -10)$	Parallelepipedum med kun gule flader	✓

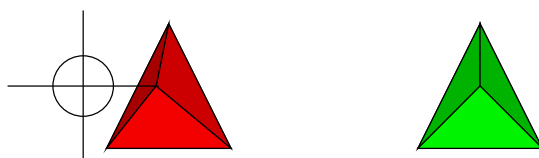
6.3.7 Scene og projektion der placerer en markør ved det nærmeste punkt i α fra $P_{\emptyset je}$

klasse	inddata	forventede uddata	✓/×
2(a)iC	$\mathbf{g}/\text{to-markoer}$, $P_{\emptyset je} = (0, -50, 0)$, $\vec{v}_{op} = \langle 0, 0, 1 \rangle$, $\alpha = (0, 1, 0, 10)$	Spids af rød tetraeder i centrum	✓
2(a)iC	$\mathbf{g}/\text{to-markoer}$, $P_{\emptyset je} = (5, -50, 0)$, $\vec{v}_{op} = \langle 0, 0, 1 \rangle$, $\alpha = (0, 1, 0, 10)$	Spids af grøn tetraeder i centrum	✓

Den anden afprøvningen fejlede, da den grønne spids ikke befandt sig på $(0,0)$ i FIG-filen, selvom man kan se, at den er i perspektivets centrum (jf. fig 10). Imidlertid fandt vi, at vi i linje 43 i `CentralProjektor.sml` anvendte `projiceret_p` i stedet for `flyttet_p`. Det er nu rettet, og afprøvningen fejler ikke mere.

6.3.8 Scene med kompliceret malerækkefølge hvor ingen forme skærer hinanden

klasse	inddata	forventede uddata	✓/×
2(a)iiA	$\mathbf{g}/\text{cyclisk}$, $P_{\emptyset je} = (0, -50, 0)$, $\vec{v}_{op} = \langle 0, 0, 1 \rangle$, $\alpha = (0, 1, 0, 10)$	Figur 9 i opgaveteksten	✓



Figur 10: Den grønne spids er ikke i centrum af billedet (indtegnet), men er tydelig i perspektivets centrum

6.3.9 Scene med kompliceret malerækkefølge med forme der skærer hinanden

klasse	inddata	forventede uddata	✓/×
2(a)iiB	g/skaering-linje, $P_{\emptyset je} = (50, -50, 0)$, $\vec{v}_{op} = \langle 0, 0, 1 \rangle$, $\alpha = (-1, 1, 0, 10)$	To parallelepipedumer der går ind i hinanden, og skærer i en linje	✓
2(a)iiB	g/skaering-punkt, $P_{\emptyset je} = (50, -50, -50)$, $\vec{v}_{op} = \langle 0, 0, 1 \rangle$, $\alpha = (-1, 1, 1, 10)$	Tre parallelepipedumer der går ind i hinanden, og skærer i et punkt	✓

6.3.10 Scener med forme der ligger på tværs af billedplanet

klasse	inddata	forventede uddata	✓/×
2b	g/skaering-linje, $P_{\emptyset je} = (0, -40, 0)$, $\vec{v}_{op} = \langle 0, 0, 1 \rangle$, $\alpha = (0, 1, 0, 1.5)$	Toppen af en tetraeder inden i en parallelepipedum der har fået sin øvre flade afskåret. Skæringens hjørner skal være i punkterne $(\pm 2.0cm, \pm 2.0cm)$.	✓
2(b)i	g/skaering-punkt, $P_{\emptyset je} = (2, -40, 0)$, $\vec{v}_{op} = \langle 0, 0, 1 \rangle$, $\alpha = (0, 1, 0, 1.5)$	Toppen af en tetraeder inden i en parallelepipedum der har fået sin øvre flade afskåret, og som mangler sin højre væg	×

Det andet tilfælde fejler da den højre væg ses.

6.3.11 Scener med forme der ligger bagved billedplanet

klasse	inddata	forventede uddata	✓/×
2b	g/para, $P_{\emptyset je} = (0, -40, 0)$, $\vec{v}_{op} = \langle 0, 0, 1 \rangle$, $\alpha = (0, 1, 0, -10)$	En gyldig fig uden polylinjer	✓

Litteratur

- [Epp, 1995] Epp, Susan S. *Discrete Mathematics with Applications, Second Edition*. PWS Publishing Company, 1995. Side 572–577.
- [Foley et. al, 1997] Foley, Van Dam, Feiner og Huges. *Computer Graphics: Principels and Practice, Second Edition in C*. Addison-Wesley, 1997. Side 461.
- [JavaDoc] JavaDoc 1.4 Standarden. <http://java.sun.com/j2se/1.4.2/docs/tooldocs/windows/javadoc.html>. Sun Microsystems, Inc., 2002
- [Messer, 1994] Messer, Robert. *Linear Algebra: Gateway to Mathematics*. HarperCollins College Publishers, 1994.
- [Hansen et. al, 1999] Hansen, M. R., Rischel, H. *Introduction to Prorgamming Using SML*. Addison-Wesley, 1999.

A Kodekonventioner

Vi bruger JavaDoc-formatet [JavaDoc] til at dokumentere kildekoden, dog med en proprietær tag `@type` tilføjet, og vi skriver `@raise` i stedet for `@exception`.

For symboler gælder de følgende regler:

1. Variable: Skrives med små bogstaver, men med stort bogstav ved ordskiller:

```
var variabel = 3;
var treOrdsVariabel = YES;
```

2. Typer: Skrives med små bogstaver, og med underscores ved ordskiller, fx:

```
type typisk_navn = ...
```

Dog er der en undtagelse fra denne regel: `BSPtree`, der er dikteret af opgaveteksten.

3. Funktioner, structures, functors og signatures: Skrives med små bogstaver, men med stort bogstav ved ordskiller og med stort for bogstav, fx:

```
fun Funk () = ...
fun EnAndenFunk () = ...
```

```
structure EnAndenStruct =
struct
...
end;
```

Endvidere skal symbolerne hvis mulig have danske navne.

B Kildekode

Selve programmet (program.sml)

```
1 load "Math";
2 load "Real";
3 load "Int";
4
5 use "Analyse.sml";
6 use "Afproevning.sml";
7 use "Vektor3D.sml";
8 use "Plan.sml";
9 use "Matrix3x3.sml";
10 use "Polygon.sml";
11 use "CentralProjektor.sml";
12 use "BSPTrae.sml";
13 use "FIG.sml";
14
15 (**
16  * Projicerer det givne polygon og tilføjer det til filen hvis det er noget
17  * igen efter projektionen.
18  *
19  * Udregner også en "belysningsverdi" der bruges til skravering for at gøre
20  * hælningerne i scenen nemmere at opfatte. Denne er baseret på cosinus til
21  * vinkelen mellem normalvektoren til polygonen og vektoren mellem en af
22  * polygonens punkter og øjepunktet.
23  *)
```

```
24 * Dette giver en illusion af at der er monteret en lyskaster på kameraet,
25 * da et polygon får en mørkere farve idet det vender væk fra øjepunktet.
26 *
27 * @type projektion -> polygon * fig -> fig
28 *)
29 fun ProjicerOgUdskrivPolygon (projektion as (oejepunkt, -, -))
30     (p as (Polygon (-, ps)), fig) =
31     case CentralProjektor.Projicer projektion p of
32     SOME(projiceret_p) =>
33         let
34             val n = Vektor3D.Norm (Polygon.NormalVektor p)
35             (* Et bedre resultat kan opnåes hvis vi bruger centrum
36              * af polygonen, i stedet for et tilfældig punkt *)
37             val l = Vektor3D.Norm (Vektor3D.Sub (oejepunkt, hd ps))
38             (*
39              * Vi bruger bare halvdelen af den dynamiske spændvidden
40              * på [0.0, 2.0] da meget lyse og meget mørke nuancer
41              * skjuler farvene.
42              *
43              * Derudover bruger vi abs, da enkelte konfigurationer
44              * kan "vende vrangsidens ud", og få normalvektorene
45              * til at pege i den modsatte retning.
46              *)
47             val lys = abs (Vektor3D.Prik (n, l)) + 0.5
48         in
49             FIG.TilfoejPolygon (fig, projiceret_p, lys)
50         end
51     | NONE => fig;
52
53 (**
54 * Opbygger et BSP-træ ud fra den scenedefinition det givne
55 * udtryk indeholder
56 *
57 * (jf. sektion 6, pkt. 3 i opgaveteksten)
58 *
59 * @type udtryk -> BSPtree
60 *)
61 fun BygTrae udtryk =
62     let
63         val polygoner = (Polygon.UdtrykTilPolygon udtryk) handle
64             Polygon.FejlformateretUdtryk (T (a, us), msg) =>
65                 raise Fail ("Fejlformateret: " ^ a ^ ": " ^ msg ^ "\n")
66             | Polygon.UvaentetUdtryk (T (a, us), msg) =>
67                 raise Fail ("Uventet: \" " ^ a ^ "\": " ^ msg ^ "\n")
68         in
69             BSPTrae.BygTrae polygoner handle
70             Polygon.ManglerPunkter (p, msg) =>
71                 raise Fail (msg ^ " (Har: " ^ Int.toString(p) ^ " punkter)")
72         end
73
74 (**
75 * Projicerer den scene det givne BSP-træ angiver og returnerer
76 * resultatet som en streng med FIG 3.2-filformat
77 *
78 * (jf. sektion 6, pkt. 4 i opgaveteksten)
79 *
80 * @type BSPtree -> projektion -> string
81 *)
82 fun GenererFig bsptrae (oejepunkt, vop, alpha) =
83     let
84         val f = ProjicerOgUdskrivPolygon (oejepunkt, vop, alpha)
85     in
86         FIG.toString (BSPTrae.FoldMaler f FIG.Start bsptrae oejepunkt)
87     end;
```

Vektor (Vektor3D.sml)

```
1 (*
```

```
2 * Grundlaeggende typer og funktioner for at arbejde med tredimensionale vektorer.
3 *
4 * @author: Jørgen H. Seland
5 * @version: $Revision: 1.11 $
6 *)
7
8 (**
9 * Angiver en 3D vektor
10 *)
11 type vektor3d = real * real * real
12
13 signature Vektor3D =
14   sig
15     val nul: vektor3d
16
17     val toString: vektor3d -> string
18     val Skalar: real * vektor3d -> vektor3d
19     val Prik: vektor3d * vektor3d -> real
20     val Kryds: vektor3d * vektor3d -> vektor3d
21     val Norm: vektor3d -> vektor3d
22     val Add: vektor3d * vektor3d -> vektor3d
23     val Sub: vektor3d * vektor3d -> vektor3d
24
25     val navn: string
26     val afproevninger: (string * (unit -> bool)) list
27   end
28
29 structure Vektor3D :> Vektor3D =
30   struct
31     (**
32      * Nulvektoren
33      *)
34     val nul = (0.0, 0.0, 0.0)
35
36     fun toString (x, y, z) = "<" ^ Real.toString x ^ ", " ^ Real.toString y ^ ", " ^
37       Real.toString z ^ ">"
38
39     (**
40      * @type real * vektor3d -> vektor3d
41      * @return Skalarproduktet af de givne værdier
42      *)
43     fun Skalar (s, (x, y, z)) : vektor3d =
44       (s * x, s * y, s * z)
45
46     (**
47      * @type vektor3d * vektor3d -> real
48      * @return Prikproduktet af de givne vektorer
49      *)
50     fun Prik ((ax, ay, az), (bx, by, bz)) : real =
51       (ax * bx + ay * by + az * bz)
52
53     (**
54      * @type vektor3d * vektor3d -> vektor3d
55      * @return Krydsproduktet af de givne vektorer
56      *)
57     fun Kryds ((ax, ay, az), (bx, by, bz)) : vektor3d =
58       (ay * bz - az * by, ~(ax * bz - az * bx), ax * by - ay * bx)
59
60     (**
61      * @type vektor3d -> real
62      * @return Længden af den givne vektor
63      *)
64     fun Laen ((x, y, z)) =
65       Math.sqrt(x * x + y * y + z * z)
66
67     (**
68      * @param v Vektor der skal normaliseres, må ikke være lig nul
69      * @type vektor3d -> vektor3d
```

```
69     * @return Den normaliserede udgave (længde lig 1) af den givne vektor
70     * @raise Fail hvis v = nul
71     *)
72     fun Norm (v) : vektor3d =
73         let
74             val l = Laen(v)
75         in
76             if l = 0.0 then raise Fail "Vektor med længde 0 kan ikke normaliseres"
77             else Skalar (1.0 / l, v)
78         end
79
80     (**
81     * @type vektor3d * vektor3d -> vektor3d
82     * @return Summen af vektorene
83     *)
84     fun Add ((ax, ay, az), (bx, by, bz)) : vektor3d =
85         (ax + bx, ay + by, az + bz)
86
87     (**
88     * @type vektor3d * vektor3d -> vektor3d
89     * @return Differencen af vektorene
90     *)
91     fun Sub ((ax, ay, az), (bx, by, bz)) : vektor3d =
92         (ax - bx, ay - by, az - bz)
93
94     (**
95     * Afprøvningstilfælde
96     *)
97     val navn = "Vektor3D"
98     val afproevninger =
99         [
100             ("Enkelt skalarprodukt",
101              fn () => Skalar (3.0, (1.0, 2.0, 3.0)) = (3.0, 6.0, 9.0)),
102             ("Enkelt prikprodukt",
103              fn () => Prik ((2.0, 3.0, 4.0), (5.0, 6.0, 7.0)) = (10.0 + 18.0 + 28.0)),
104             ("Enhedsvektorer for X og Y skal give Z",
105              fn () => Kryds ((1.0, 0.0, 0.0), (0.0, 1.0, 0.0)) = (0.0, 0.0, 1.0)),
106             ("Enkel normalisering",
107              fn () => Afproevning.TilnaermetLig(Laen (Norm (20.0, 40.0, ~10.0)), 1.0)),
108             ("Normalisering af nulvektor",
109              fn () => Afproevning.SkalFejle (Norm, nul)),
110             ("Enkel sum",
111              fn () => Add ((1.0, 2.0, 3.0), (4.0, 5.0, 6.0)) = (5.0, 7.0, 9.0)),
112             ("Enkel difference",
113              fn () => Sub ((1.0, 2.0, 3.0), (6.0, 5.0, 4.0)) = (~5.0, ~3.0, ~1.0))
114         ]
115     end;
```

Matrix (Matrix3x3.sml)

```
1 (*
2  * Grundlaeggende typer og funktioner for at arbejde med 3x3-matrixer.
3  *
4  * @author: Jørgen H. Seland
5  * @version: $Revision: 1.7 $
6  *)
7
8 (**
9  * Angiver en 3x3 matrix
10 *)
11 type matrixRaekke = real * real * real
12 type matrix = matrixRaekke * matrixRaekke * matrixRaekke
13
14 signature Matrix3x3 =
15     sig
16         val id: matrix
17
18         (* val toString: figfil -> string*)
```

```
19     val InvTransformation: vektor3d * vektor3d * vektor3d -> matrix
20     val Mul: matrix * vektor3d -> vektor3d
21
22     val navn: string
23     val afproevninger: (string * (unit -> bool)) list
24 end
25
26 structure Matrix3x3 :> Matrix3x3 =
27   struct
28     (**
29      * Identitets-matrixen
30      *)
31     val id = ((1.0, 0.0, 0.0),
32              (0.0, 1.0, 0.0),
33              (0.0, 0.0, 1.0))
34
35     (**
36      * Laver en transformations-matrix der ved multiplikation oversætter en
37      * vektor fra det nuværende koordinatsystem til det koordinatsystem
38      * der har de givne aksevektorer, udtrykt i det nuværende koordinatsystem.
39      *
40      * @type vektor3d * vektor3d * vektor3d -> matrix
41      * @param x Koordinatsystemets x-akse.
42      * @param y Koordinatsystemets y-akse.
43      * @param z Koordinatsystemets z-akse.
44      * @return Transformations-matrix
45      *)
46     fun InvTransformation (x, y, z) = (x, y, z)
47
48     (**
49      * Multiplicerer den givne vektor med den givne matrix.
50      *
51      * @type matrix * vektor3d -> vektor3d
52      *)
53     fun Mul (((m11, m12, m13), (m21, m22, m23), (m31, m32, m33)), (x, y, z)) =
54       (x * m11 + y * m12 + z * m13,
55        x * m21 + y * m22 + z * m23,
56        x * m31 + y * m32 + z * m33)
57
58     (**
59      * Afprøvningstilfælde
60      *)
61     val navn = "Matrix3x3"
62     val afproevninger =
63       [
64         ("Enkel multiplikation",
65          fn () => Mul (id, (1.0, 2.0, 3.0)) = (1.0, 2.0, 3.0))
66       ]
67   end;
```

Plan (Plan.sml)

```
1 (*
2  * Grundlaeggende typer og funktioner til at arbejde med planer i rummet.
3  *
4  * @author: Jørgen H. Seland
5  * @version: $Revision: 1.15 $
6  *)
7
8 (**
9  * Angiver et plan ud i fra koefficienterne A, B, C og D i planets ligning.
10 *)
11 type plan = real * real * real * real
12
13 signature Plan =
14   sig
15     val xy: plan
16     val xz: plan
```

```
17     val yz: plan
18
19     val toString: plan -> string
20     val PlanFraPunkt: vektor3d * vektor3d -> plan
21     val Skaeringspunkt: plan * vektor3d * vektor3d -> vektor3d
22     val Sammenlign: plan * vektor3d -> int
23     val NaermestePunkt: plan * vektor3d -> vektor3d
24
25     val navn: string
26     val afproevninger: (string * (unit -> bool)) list
27 end
28
29 structure Plan :> Plan =
30     struct
31         val xy = (0.0, 0.0, 1.0, 0.0)
32         val xz = (0.0, 1.0, 0.0, 0.0)
33         val yz = (1.0, 0.0, 0.0, 0.0)
34
35         (**
36          * @type plan -> string
37          *)
38         fun toString (a, b, c, d) =
39             "(" ^
40             Real.toString a ^ ", " ^
41             Real.toString b ^ ", " ^
42             Real.toString c ^ ", " ^
43             Real.toString d ^
44             ")"
45
46         (**
47          * Robust fortegns-operator
48          *
49          * Situationer hvor afstanden til planet er 0 kan i nogle tilfælde
50          * give +/-1 som resultat for Sammenlign, men siden resultatet siden
51          * bliver divideret med "u" i Afst, kan afstand give 0. Denne
52          * uoverensstemmelse giver fejl der det egentlig ikke er noget
53          * problem.
54          *
55          * Bivirkninge: Unøyaktigheder i BSP-træ-algoritmen ved meget små
56          * afstande (størrelsesorden 1/1000 mm)
57          *
58          * @type real -> int
59          *)
60         fun RobustFortegn x =
61             if x < ~0.00001 then ~1 else
62             if x > 0.00001 then 1 else 0
63
64         (**
65          * Udregner planet der har den givne normalvektor og der
66          * indeholder det givne punkt
67          *
68          * @type vektor3d * vektor3d -> plan
69          *)
70         fun PlanFraPunkt ((a, b, c), p) =
71             (a, b, c, ~(Vektor3D.Prik((a, b, c), p)))
72
73         (**
74          * Finder skæringspunktet mellem linjen med parameterfremstilling
75          * p + tv og planet.
76          *
77          * Anvender formelen angivet i afsnittet "Projicering" i programmerings-
78          * overvejelserne.
79          *
80          * @type plan * vektor3d * vektor3d -> vektor3d
81          * @param a A-koefficient i planets ligning
82          * @param b B-koefficient i planets ligning
83          * @param c C-koefficient i planets ligning
84          * @param d D-koefficient i planets ligning
```

```

85     * @param p linjens startpunkt p.
86     * @param v Linjens retningsvektor. Må ikke være lig med nul.
87     * @return skæringspunktet mellem linjen p + tv og planet.
88     * @raise Fail hvis linjen er parallel med planet, da der ikke findes
89     * noget skæringspunkt i dette tilfælde.
90     * @raise Fail hvis v = nul, da parameterfremstillingen i dette tilfælde
91     * angiver et punkt, ikke en linje.
92     *)
93 fun Skaeringspunkt ((a, b, c, d), p, v) =
94     let
95         val u = if v <> Vektor3D.nul then Vektor3D.Prik((a, b, c), v)
96                 else raise Fail "Parameterfremstillingen giver ikke en linje"
97         val t = if RobustFortegn u <> 0 then ~(Vektor3D.Prik((a, b, c), p) + d)
98                 / u
99                 else raise Fail ("Linjen " ^ Vektor3D.toString p ^ " + t" ^
100                                Vektor3D.toString v ^ " er parallel med planet "
101                                ^ toString (a, b, c, d))
102     in
103         Vektor3D.Add(p, Vektor3D.Skalar(t, v))
104     end
105 (**
106  * Finder kun fortegnet til afstanden til det givne plan.
107  *
108  * @type plan * vektor3d -> int
109  * @return -1 ved d < 0, 0 ved d = 0 og 1 ved d > 0
110  *)
111 fun Sammenlign ((a, b, c, d), p) =
112     RobustFortegn (~(Vektor3D.Prik((a, b, c), p) + d))
113 (**
114  * Finder afstanden til planet fra p, regnet i antal gange normalvektoren
115  * til planet skal tillægges p for at ramme planet.
116  *
117  * (Dette er første halvdel af nærmeste punkt-udregningen)
118  *
119  * @type plan * vektor3d -> real
120  * @param a A-koefficient i planets ligning
121  * @param b B-koefficient i planets ligning
122  * @param c C-koefficient i planets ligning
123  * @param d D-koefficient i planets ligning
124  * @param p Punktet p.
125  * @raise Fail hvis planets normalvektor er lig nul, da dette giver et
126  * ugyldig plan.
127  *)
128 fun Afst ((a, b, c, d), p) =
129     let
130         val u = if (a, b, c) <> Vektor3D.nul then a * a + b * b + c * c
131                 else raise Fail "Ugyldig plan"
132     in
133         ~ (Vektor3D.Prik ((a, b, c), p) + d) / u
134     end
135 (**
136  * Finder det punkt i planet der ligger nærmest p
137  *
138  * Anvender formelen angivet i afsnittet "Translatering" i
139  * programmerings overvejelserne.
140  *
141  * @type plan * vektor3d -> vektor3d
142  * @param a A-koefficient i planets ligning
143  * @param b B-koefficient i planets ligning
144  * @param c C-koefficient i planets ligning
145  * @param d D-koefficient i planets ligning
146  * @param p Punktet p.
147  * @raise Fail hvis planets normalvektor er lig nul, da dette giver et
148  * ugyldig plan.
149  *)
150

```

```
151     fun NaermestePunkt ((a, b, c, d), p) =
152         let
153             val t = Afst ((a, b, c, d), p)
154         in
155             Vektor3D.Add(p, Vektor3D.Skalar(t, (a, b, c)))
156         end
157
158     (**
159     * Afprøvningsstilfælde
160     *)
161     val navn = "Plan"
162     val afproevninger =
163     [
164         ("Enkelt skæringspunkt",
165          fn () => Skaeringspunkt (xy, (0.0, 0.0, ~10.0), (3.0, 2.0, 1.0)) =
166              (30.0, 20.0, 0.0)),
167         ("Skæringspunkt med parallel linje",
168          fn () => Afproevning.SkalFejle (Skaeringspunkt,
169              (xy, (0.0, 0.0, ~10.0), (3.0, 2.0, 0.0)))),
170         ("Enkelt nærmeste punkt",
171          fn () => NaermestePunkt (xy, (20.0, 10.0, ~10.0)) =
172              (20.0, 10.0, 0.0)),
173         ("Sammenligning 1",
174          fn () => Sammenlign ((0.0, 0.0, 1.0, 0.0), (0.0, 0.0, 0.0)) = 0),
175         ("Sammenligning 2",
176          fn () => Sammenlign ((0.0, 0.0, 1.0, 1.0), (0.0, 0.0, 0.0)) = ~1),
177         ("Sammenligning 3",
178          fn () => Sammenlign ((0.0, 0.0, 1.0, ~1.0), (0.0, 0.0, 0.0)) = 1)
179     ]
180     end;
```

Polygon (Polygon.sml)

```
1 (*
2  * Grundlaeggende typer og funktioner for at arbejde med polygoner.
3  *
4  * @author: Jørgen H. Seland
5  * @version: $Revision: 1.15 $
6  *)
7 type farve = int * int
8
9 (**
10 * Angiver en polygon. Det siste punkt regnes som implicit knyttet til
11 * det første.
12 *)
13 datatype polygon = Polygon of farve * vektor3d list
14
15 signature Polygon =
16     sig
17         exception UvaentetUdtryk of udtryk * string
18         exception FejlformateretUdtryk of udtryk * string
19         exception ManglerPunkter of int * string
20
21         val toString: polygon -> string
22         val UdtrykTilPolygon: udtryk -> polygon list
23         val NormalVektor: polygon -> vektor3d
24         val PolygonPlan: polygon -> plan
25         val SplitPolygon: polygon * plan -> polygon option * polygon option * polygon option
26
27         val navn: string
28         val afproevninger: (string * (unit -> bool)) list
29     end
30
31 structure Polygon :> Polygon =
32     struct
33         exception UvaentetUdtryk of udtryk * string
34         exception FejlformateretUdtryk of udtryk * string
```

```
35     exception ManglerPunkter of int * string
36
37     (**
38      * @type vektor3d list -> string
39      *)
40     fun UdskrivPunkter [] = ""
41       | UdskrivPunkter (p :: ps) =
42         Vektor3D.toString p ^ " " ^ UdskrivPunkter ps
43
44     (**
45      * @type polygon -> string
46      *)
47     fun toString (Polygon ((pen, bg), ps)) =
48       "[{" ^ Int.toString pen ^ ":" ^ Int.toString bg ^ "}, " ^ UdskrivPunkter ps
49         ^ "]"
50
51     (**
52      * @type udtryk -> real
53      *)
54     fun TolKReal (u as T (a, [])) =
55       (case Real.fromString(a) of
56        NONE => raise FejlformateretUdtryk (u, "Ugyldigt flydendetal.")
57        | SOME(r) => r)
58       | TolKReal u = raise UvaentetUdtryk (u, "Forventede et flydendetal.")
59
60     (**
61      * @type udtryk -> vektor
62      *)
63     fun TolKVektor (T ("vekt", [xu, yu, zu])) = (TolKReal xu, TolKReal yu, TolKReal
64       zu)
65       | TolKVektor u = raise UvaentetUdtryk (u, "Forventede \"vekt\" med tre
66         parametre.")
67
68     (**
69      * @type udtryk -> int
70      *)
71     fun TolKFarveIndeks (u as T (a, [])) =
72       (case Int.fromString(a) of
73        NONE => raise FejlformateretUdtryk (u, "Ugyldigt heltal.")
74        | SOME(i) => if (i >= ~1) andalso (i <= 533) then i
75                      else raise FejlformateretUdtryk (u, "Farveindeks udenfor
76                        [-1, 533]."))
77       | TolKFarveIndeks u = raise UvaentetUdtryk (u, "Forventede farveindeks.")
78
79     (**
80      * @type udtryk -> farve
81      *)
82     fun TolKFarve (T ("farve", [pu, bu])) = (TolKFarveIndeks pu, TolKFarveIndeks bu)
83       | TolKFarve u = raise UvaentetUdtryk (u, "Forventede \"farve\" med to parametre.
84         ")
85
86     (**
87      * @type udtryk -> polygon list
88      *)
89     fun TolKForm (T ("tetra", [pu, v1u, v2u, v3u, fu])) =
90       let
91         val farve = TolKFarve fu
92         val a = TolKVektor pu
93         val b = Vektor3D.Add(a, TolKVektor v1u)
94         val c = Vektor3D.Add(a, TolKVektor v2u)
95         val d = Vektor3D.Add(a, TolKVektor v3u)
96       in
97         [Polygon (farve, [a, b, c]), Polygon (farve, [a, d, b]),
98          Polygon (farve, [a, c, d]), Polygon (farve, [b, d, c])]
99       end
100     | TolKForm (T ("para", [pu, v1u, v2u, v3u, fu])) =
101       let
```

```

98         val farve = TolkFarve fu
99         val v1 = TolkVektor v1u
100        val v2 = TolkVektor v2u
101        val v3 = TolkVektor v3u
102
103        val a = TolkVektor pu
104        val b = Vektor3D.Add(a, v3)
105        val c = Vektor3D.Add(b, v1)
106        val d = Vektor3D.Add(a, v1)
107        val e = Vektor3D.Add(a, v2)
108        val f = Vektor3D.Add(b, v2)
109        val g = Vektor3D.Add(f, v1)
110        val h = Vektor3D.Add(e, v1)
111    in
112        [Polygon (farve, [a, d, c, b]), Polygon (farve, [g, f, b, c]),
113         Polygon (farve, [c, d, h, g]), Polygon (farve, [a, b, f, e]),
114         Polygon (farve, [h, g, f, e]), Polygon (farve, [d, a, e, h])]
115    end
116    | TolkForm u = raise UvaentetUdtryk (u, "Forventede \"tetra\" eller \"para\"
      med fem parametre")
117
118    (**
119     * @type udtryk list -> polygon list
120     *)
121    fun TolkFormListe [] = []
122    | TolkFormListe (f :: fs) = TolkForm f @ TolkFormListe fs
123
124    (**
125     * Udtrækker alle polygoner fra det givne sceneudtryk
126     *
127     * @type: udtryk -> polygon list
128     *)
129    fun UdtrykTilPolygon (T ("scene", us)) = TolkFormListe us
130    | UdtrykTilPolygon u = raise UvaentetUdtryk (u, "Forventede \"scene\".")
131
132    (**
133     * Beregner en normal vektor til polygonen sammensat af den givne liste af
      punkter,
134     * baseret på de tre første punkter.
135     *
136     * Det er brugerens opgave at sikre, at alle punkter ligger i de samme plan.
137     *
138     * @type vektor3d list -> vektor
139     * @raise ManglerPunkter Rejses hvis polygonen ikke har mindst tre punkter.
140     *)
141    fun NormalVektor (Polygon (_, v1::v2::v3::_)) =
142      Vektor3D.Kryds (Vektor3D.Sub (v1, v2), Vektor3D.Sub (v1, v3))
143    | NormalVektor (p as Polygon (_, vs)) =
144      raise ManglerPunkter (List.length vs,
145        "Kunne ikke beregne polygonens " ^ toString p ^ "
      normalvektor, da den ikke har mindst tre punkter.")
146
147    (**
148     * Beregner det plan, den givne polygon ligger i.
149     *
150     * @type polygon -> real * real * real * real
151     *)
152    fun PolygonPlan (p as Polygon (_, ps)) =
153      Plan.PlanFraPunkt (NormalVektor p, hd ps)
154
155    (**
156     * Finder siden til sidste punkt i listen der ikke ligger på det
157     * givne plan
158     *
159     * @type vektor list * Plan.plan -> int
160     * @return -1/1 hvis fundet, 0 hvis ingen slike punkter findes
161     *)
162    fun FindSidstePunktMedSide ([], _) = 0

```

```
163 | FindSidstePunktMedSide (p :: ps, alpha) =
164 |   let
165 |     val side = FindSidstePunktMedSide (ps, alpha)
166 |   in
167 |     if side <> 0 then side else Plan.Sammenlign (alpha, p)
168 |   end
169
170 (**
171  * Tilføjer til den liste i paret der passer med den givne side
172  *
173  * @type (polygon list * polygon list) * int * polygon ->
174  *       polygon list * polygon list
175  *)
176 fun TilfoejTilSide ((pa, pb), side, p) =
177 |   if side < 0 then (p :: pa, pb) else (pa, p :: pb)
178
179 (**
180  * Tilfoejer punktet på planet mellem forrige og gældende punkt hvis det
181  * findes
182  *
183  * @type (polygon list * polygon list) * int * int * polygon * polygon ->
184  *       polygon list * polygon list
185  *)
186 fun TilfoejSkaering (l as (pa, pb), forrige_side, side, forrige_p, p, alpha) =
187 |   if forrige_side = side then
188 |     l
189 |   else
190 |     let
191 |       val skaering =
192 |         Plan.Skaeringspunkt (alpha, forrige_p, Vektor3D.Sub (p, forrige_p
193 |           ))
194 |       val l2 =
195 |         if skaering <> forrige_p then TilfoejTilSide (l, forrige_side,
196 |           skaering)
197 |         else l
198 |     in
199 |       if skaering <> p then TilfoejTilSide (l2, side, skaering)
200 |     end
201
202 (**
203  * Klassificerer punkterne i forhold til planet
204  * Tilføjer de punkter som behøves for at danne komplette polygoner
205  *
206  * @type int * vektor list * plan -> vektor list * vektor list
207  *)
208 fun KlassificerPunkter (_, -, [], -) = ([], [])
209 |   KlassificerPunkter (forrige_side, forrige_p, p :: ps, alpha) =
210 |   let
211 |     val side = Plan.Sammenlign (alpha, p)
212 |     val justeret_side = if side <> 0 then side else forrige_side
213 |     val raest_l = KlassificerPunkter (justeret_side, p, ps, alpha)
214 |     val l = TilfoejTilSide (raest_l, justeret_side, p)
215 |   in
216 |     (* NB: Vi tilføjer skæringen efter punktet, da vi arbejder med cons,
217 |     ikke append *)
218 |     TilfoejSkaering (l, forrige_side, justeret_side, forrige_p, p, alpha)
219 |   end
220
221 (**
222  * Afprøvningstilfælde
223  *)
224 val navn = "Polygon"
225 val afproevninger =
226 |   [
227 |     ("Punkt med side, normaltilfælde",
228 |     fn () => FindSidstePunktMedSide ([ (0.0, 0.0, ~1.0),
```

```

229             (0.0, 0.0, 0.0),
230             (0.0, 0.0, 0.0)],
231             Plan.xy) = 1),
232 ("Punkt med side, ingen punkter",
233  fn () => FindSidstePunktMedSide ([ (0.0, 0.0, 0.0),
234                                   (0.0, 0.0, 0.0),
235                                   (0.0, 0.0, 0.0)],
236                                   Plan.xy) = 0),
237 ("Punkt med side, tom liste",
238  fn () => FindSidstePunktMedSide ([], Plan.xy) = 0),
239 ("Normalvektor med manglende punkte",
240  fn () => Afproevning.SkalFejle (NormalVektor,
241                                  Polygon((1, 1), [(0.0,0.0,0.0),
242                                                  (1.0,0.0,0.0)]))),
243 ("Tilføj skæring, samme side",
244  fn () => TilfoejSkaering (([], []), ~1, ~1,
245                           (0.0, 0.0, ~1.0),
246                           (0.0, 0.0, 1.0),
247                           Plan.xy) =
248                           ([], [])),
249 ("Tilføj skæring, forskellig side",
250  fn () => TilfoejSkaering (([], []), ~1, 1,
251                           (0.0, 0.0, ~1.0),
252                           (0.0, 0.0, 1.0),
253                           Plan.xy) =
254                           ([ (0.0, 0.0, 0.0)], [(0.0, 0.0, 0.0)])),
255 ("Tilføj skæring, forskellig side, sammenfalder med gældende punkt",
256  fn () => TilfoejSkaering (([], []), ~1, 1,
257                           (0.0, 0.0, ~1.0),
258                           (0.0, 0.0, 0.0),
259                           Plan.xy) =
260                           ([ (0.0, 0.0, 0.0)], [])),
261 ("Tilføj skæring, forskellig side, sammenfalder med forrige punkt",
262  fn () => TilfoejSkaering (([], []), ~1, 1,
263                           (0.0, 0.0, 0.0),
264                           (0.0, 0.0, 1.0),
265                           Plan.xy) =
266                           ([], [(0.0, 0.0, 0.0)])),
267 ("Klip triangel mod XY-planet",
268  fn () => SplitPolygon (Polygon((1, 1),
269                                (* Et triangel udstrakt i xz-planet *)
270                                [(10.0, 0.0, 5.0),
271                                 (~10.0, 0.0, 5.0),
272                                 (0.0, 0.0, ~5.0)]),
273                                Plan.xy) =
274                                (SOME (Polygon((1, 1),
275                                                [(5.0, 0.0, 0.0),
276                                                 (10.0, 0.0, 5.0),
277                                                 (~10.0, 0.0, 5.0),
278                                                 (~5.0, 0.0, 0.0)])),
279                                NONE,
280                                SOME (Polygon((1, 1),
281                                                [(5.0, 0.0, 0.0),
282                                                 (~5.0, 0.0, 0.0),
283                                                 (0.0, 0.0, ~5.0)]))))))
284 ]
285
286 end;
287
288 (**
289  * Splitter polygonet
290  *
291  * @type polygon * plan -> polygon option * polygon option * polygon option
292  *)
293 fun SplitPolygon (p as Polygon (farve, ps), alpha) =
294   let
295     val startside = FindSidstePunktMedSide (ps, alpha)
296     in

```

```
297     if startside <> 0 then
298         let
299             val (pa, pb) = KlassificerPunkter (startside, hd (rev ps), ps, alpha)
300         in
301             (* Undgå at lave ikke-polygoner *)
302             (if length pa > 2 then SOME (Polygon (farve, pa)) else NONE,
303              NONE,
304              if length pb > 2 then SOME (Polygon (farve, pb)) else NONE)
305         end
306     else
307         (NONE, SOME(p), NONE)
308 end
```

FIG 3.2-generator (FIG.sml)

```
1 (*
2  * FIG 3.2 generator
3  *
4  * @version: $Revision: 1.11 $
5  *)
6
7 (**
8  * En FIG-fil
9  *)
10 datatype figfil = Fig of string
11
12 signature FIG =
13     sig
14         val Start: figfil
15
16         val toString: figfil -> string
17         val TilfoejPolygon: figfil * polygon * real -> figfil
18     end
19
20 structure FIG :> FIG =
21     struct
22         (*
23          * Overgang fra cm til FIG-koordinater, hvor enheden er 1/1200 tomme.
24          *)
25         val fak = 1200.0 / 2.54
26
27         (**
28          * FIG-filens header, en gyldig start-verdi for en fig-variabel
29          *)
30         val Start = Fig ("#FIG 3.2\nPortrait\nCenter\nMetric\nA4\n100.00\nSingle\n-2\n# K
31                          -Opgave\n1200 2\n\n\n");
32
33         (**
34          * @type figfil -> string
35          *)
36         fun toString (Fig (streng)) = streng
37
38         (**
39          * Laver et heltal om til en streng der passer med hvad FIG forventer
40          *)
41         fun IntToString n =
42             if n >= 0 then Int.toString n else "-" ^ Int.toString (~n)
43
44         (**
45          * Regner om et givet mål i cm til FIG-koordinater
46          *
47          * @type real -> int
48          *)
49         fun FigMaal k = Real.round (k * fak)
50
51         (**
52          * Resten af programmet arbejder med positiv y = op, så vi skal lige spejle y-
53          koordinaten
```

```

52     *
53     * @type vektor3d -> string
54     * (dog strengt taget real * real * 'a -> string)
55     *)
56     fun KoordinatTilTekst (x, y, _) =
57       IntToString (FigMaal x) ^ " " ^ IntToString (FigMaal (~y));
58
59     (**
60     * Omsætter en række punkter til en polylinie.
61     *
62     * @param p1 Det første punkt i listen, fåes typisk ved
63     *           "hd (rev liste)"
64     *
65     * @type vektor3d list * string * vektor3d -> string
66     *)
67     fun PunkterTilPolylinie ( [], linie, p1 ) = linie ^ KoordinatTilTekst p1
68       | PunkterTilPolylinie ( p::ps, linie, p1 ) =
69         PunkterTilPolylinie( ps, linie ^ KoordinatTilTekst p ^ " ", p1)
70
71     (**
72     * Tæl antal skifter i koordinatværdier, da FIG arbejder ved at
73     * kollapse overlappende punkter hvis de kommer efter hinanden.
74     *
75     * @param fx      X-koordinat for det forrige punkt
76     * @param fy      Y-koordinat for det forrige punkt
77     * @param p :: ps Ræsten af listen
78     *
79     * @type vektor3d * vektor3d list -> int
80     * (dog strengt taget (real * real * 'a) * (real * real * 'a) list -> int)
81     *)
82     fun TelSkifter ((fx, fy, _), (p as (x, y, _)) :: ps) =
83       (if (FigMaal fx <> FigMaal x) orelse (FigMaal fy <> FigMaal y)
84        then 1 else 0) + TelSkifter (p, ps)
85     | TelSkifter (_, []) = 0
86
87     (**
88     * Ved afrunding kan meget små polygoner blive til punkter,
89     * og det brokker xfig sig over. Vi ønsker derfor at udfiltrere
90     * disse inden de havner i filen.
91     *
92     * @type polygon -> bool
93     *)
94     fun VerificerPolygon (Polygon (_, ps)) = TelSkifter (hd (rev ps), ps) > 2
95
96     (**
97     * Tilføj det givne polygon
98     *
99     * @type figfil * polygon * real -> figfil
100    * @param lys Belysning, 0.0 for mørke, 1.0 for fult lys, 2.0 for
101    * fullt overlys (hvid)
102    *)
103    fun TilfoejPolygon (fig, Polygon (_, []), _) = fig
104      | TilfoejPolygon (figfil as Fig (fig), p as Polygon ((kant, fyld), ps), lys) =
105        let
106          val linjestil = 1
107          val linjetykkelse = 1 (* i 1/80 tommer *)
108          (*val udfyldningskode = 20 "den ræne farve" *)
109          val laedstil = 0 (* 0-2 *)
110          val endepunktstil = 0 (* 0-2 *)
111          val hjoerneradius = ~1 (* i 1/80 tommer *)
112          val foroverpil = false
113          val bagoverpil = false
114          val udfyldningskode = if lys <= 0.0 then 0
115                                else if lys >= 2.0 then 40
116                                else Real.round (lys * 20.0)
117        in
118          if VerificerPolygon p then
119            Fig (

```

```

120         fig ^
121         "2 3 " ^
122         IntToString linjestil ^ " " ^
123         IntToString linjetykkelse ^ " " ^
124         IntToString kant ^ " " ^
125         IntToString fyld ^ " " ^
126         "50" (* IntToString dybde: Ikke påkrævet *) ^ " " ^
127         "0 " ^ (* pen_style: ubrugt *)
128         IntToString udfyldningskode ^ " " ^
129         "0.000 " ^ (* Streg vs. prik-længde forhold *)
130         IntToString laedstil ^ " " ^
131         IntToString endepunktstil ^ " " ^
132         IntToString hjoerneradius ^ " " ^
133         (if foroverpil then "1" else "0") ^ " " ^
134         (if bagoverpil then "1" else "0") ^ " " ^
135         IntToString (List.length ps + 1) ^ "\n" ^
136         " " ^ PunkterTilPolylinie (ps, "", hd ps) ^ "\n"
137     else figfil
138 end
139 end

```

BSP-træ (BSPTrae.sml)

```

1 (*
2  * Grundlaeggende typer og funktioner til at arbejde med
3  * BSP-træer.
4  *
5  * @version: $Revision: 1.10 $
6  *)
7
8 (**
9  * Et deltræ
10 *
11 * (Grund til engelsk navn: jf. sektion 6, pkt. 2 i opgaveteksten)
12 *)
13 datatype BSPtree = Knude of BSPtree * BSPtree * plan * polygon list | TomtTrae;
14
15 signature BSPTrae =
16   sig
17     val toString: BSPtree -> string
18     val BygTrae: polygon list -> BSPtree
19     val FoldMaler: (polygon * 'a -> 'a) -> 'a -> BSPtree -> vektor3d -> 'a
20
21     val navn: string
22     val afproevninger: (string * (unit -> bool)) list
23   end
24
25 structure BSPTrae :> BSPTrae =
26   struct
27     (**
28      * @type string * polygon list -> string
29      *)
30     fun PolygonListToString (_, []) = ""
31       | PolygonListToString (indent, (p :: ps)) =
32         indent ^ Polygon.toString p ^ "\n" ^
33         PolygonListToString (indent, ps)
34
35     (**
36      * @type string * BSPtree -> string
37      *)
38     fun IndentedToString (indent, TomtTrae) = "%\n"
39       | IndentedToString (indent, Knude (a, b, plan, ps)) =
40         indent ^ "{\n" ^
41         indent ^ "  plan = " ^ Plan.toString plan ^ "\n" ^
42         indent ^ "  polygoner = {\n" ^
43         PolygonListToString (indent ^ "    ", ps) ^
44         indent ^ "  }\n" ^
45         indent ^ "  vænstre undertræ = " ^

```

```

46         IndentedToString (indent ^ " ", a) ^
47         indent ^ " højre undertræ = " ^
48         IndentedToString (indent ^ " ", b) ^
49         indent ^ "}\n"
50
51     (**
52     * @type BSPtree -> string
53     *)
54     fun toString u = IndentedToString ("", u)
55
56     (**
57     * Opdeler den givne liste af polygoner i overensstemmelse med
58     * afsnittet om BSP-træ-opbygning
59     *
60     * @type polygon list * plan * polygon list * polygon list * polygon list ->
61     *       polygon list * polygon list * polygon list
62     *)
63     fun OpdelPolygoner ([], plan, a, b, c) = (a, b, c)
64     | OpdelPolygoner (p :: ps, plan, a, b, c) =
65         let
66             val (pa, pb, pc) = Polygon.SplitPolygon (p, plan)
67         in
68             OpdelPolygoner (ps, plan,
69                             case pa of SOME(q) => q :: a | NONE => a,
70                             case pb of SOME(q) => q :: b | NONE => b,
71                             case pc of SOME(q) => q :: c | NONE => c)
69         end
72
73     (**
74     * Opbygger et BSP-træ ud fra en liste af polygoner.
75     *
76     * @type polygon list -> BSPtree
77     *)
78     fun BygTrae [] = TomtTrae
79     | BygTrae (p :: ps) =
80         let
81             val plan = Polygon.PolygonPlan p
82             val (a, b, c) = OpdelPolygoner (ps, plan, [], [], [])
83         in
84             Knude (BygTrae a, BygTrae c, plan, p :: b)
85         end
86
87     (**
88     * En udgave af fold, der traverserer et BSPtree i
89     * malerrækkefølge
90     *
91     * @type (polygon * 'a -> 'a) -> 'a -> BSPtree -> vektor3d -> 'a
92     *)
93     fun FoldMaler f b TomtTrae oejepunkt = b
94     | FoldMaler f b (Knude (vTrae, hTrae, plan, polygoner)) oejepunkt =
95         let
96             val oejeside = Plan.Sammenlign (plan, oejepunkt)
97             val b2 = FoldMaler f b (if oejeside > 0 then vTrae else hTrae) oejepunkt
98             val b3 = foldr f b2 polygoner
99         in
100             FoldMaler f b3 (if oejeside > 0 then hTrae else vTrae) oejepunkt
101         end
102
103     (**
104     * Afprøvningstilfælde
105     *)
106     val navn = "BSPTrae"
107     val afproevninger = []
108
109     end

```

Projicering (CentralProjektor.sml)

1 (*

```

2  * Grundlaeggende typer og funktioner til at arbejde med tredimensionale vektorer.
3  *
4  * @author: Jørgen H. Seland
5  * @version: $Revision: 1.12 $
6  *)
7  use "Matrix3x3.sml";
8
9  (**
10 * Definerer en projektion, med komponenterne øjepunkt, opvektor
11 * og billedplan
12 *)
13 type projektion = vektor3d * vektor3d * plan
14
15 signature CentralProjektor =
16   sig
17     val Projicer: projektion -> polygon -> polygon option
18     val MaalProjektion: vektor3d * vektor3d * vektor3d * real -> projektion
19
20     val navn: string
21     val afproevninger: (string * (unit -> bool)) list
22   end
23
24 structure CentralProjektor :> CentralProjektor =
25   struct
26     (**
27      * Projicer punkter
28      *
29      * @type vektor3d * plan * matrix * vektor3d list -> vektor3d list
30      *)
31     fun ProjicerPunkter (., ., ., []) = []
32       | ProjicerPunkter (oeje, alpha, m, p::ps) =
33         let
34           (* Projektion *)
35           val oeje_til_p = Vektor3D.Sub (p, oeje);
36           val projiceret_p = Plan.Skaeringspunkt (alpha, oeje, oeje_til_p)
37
38           (* Flytning af origo *)
39           val alpha_origo = Plan.NaermestePunkt (alpha, oeje)
40           val flyttet_p = Vektor3D.Sub (projiceret_p, alpha_origo)
41
42           (* Rotation så akserne stemmer *)
43           val transformeret_p = Matrix3x3.Mul(m, flyttet_p)
44         in
45           transformeret_p :: ProjicerPunkter (oeje, alpha, m, ps)
46         end
47
48     (**
49      * Opstiller en transformationsmatrix ud fra det givne plan og
50      * op-vektor
51      *
52      * @type plan * vektor3d -> matrix
53      *)
54     fun Transformation ((a, b, c, d), vop) =
55       let
56         val vz = (a, b, c)
57         val vx = Vektor3D.Kryds(vz, vop)
58         val vy = Vektor3D.Kryds(vx, vz)
59       in
60         Matrix3x3.InvTransformation (Vektor3D.Norm vx, Vektor3D.Norm vy, Vektor3D
61           .Norm vz)
62       end
63
64     (**
65      * Klipper væk den del, der er på hitsiden af billedplanet
66      *
67      * @type polygon * plan * vektor3d -> polygon option
68      *)
69     fun FjernHitsiden (poly, alpha, oeje) =

```

```

69     let
70     val (pa, pb, pc) = Polygon.SplitPolygon (poly, alpha)
71     in
72     case pb of
73     p as SOME(-) => p
74     (*
75     * Tag den del der er på den modsatte side af alpha i
76     * forh. til øjepunktet
77     *)
78     | NONE => if Plan.Sammenlign (alpha, oeje) > 0 then pa else pc
79     end
80
81     (**
82     * Projicerer en given polygon og klipper den mod billedplanet
83     *
84     * @type projektion -> polygon -> polygon option
85     *)
86     fun Projicer (oeje, vop, alpha) poly =
87     case FjernHitsiden (poly, alpha, oeje) of
88     SOME(Polygon (farve, ps)) =>
89     let
90     val m = Transformation (alpha, vop)
91     in
92     SOME (Polygon (farve, ProjicerPunkter (oeje, alpha, m, ps)))
93     end
94     | NONE => NONE
95
96     (**
97     * Laver projektion ud fra et øjepunkt, en opvektor, et sigtepunkt
98     * og afstand fra øjepunktet til billedplanet
99     *
100    * Dette er for nemmere at kunne placere kameraet frit for at
101    * granske en scene, og er kun tiltænkt at bruges under udvikling
102    *
103    * @type vektor3d * vektor3d * vektor3d * real -> projektion
104    *)
105    fun MaalProjektion (oeje, vop, maal, afstand) =
106    let
107    (* Normaliserer normalvektoren sådan at N * afstand bliver
108    * afstand lang *)
109    val alpha_normal = Vektor3D.Norm (Vektor3D.Sub (maal, oeje))
110    val alpha_punkt = Vektor3D.Add (oeje, Vektor3D.Skalar (afstand,
111    alpha_normal))
112    val alpha = Plan.PlanFraPunkt (alpha_normal, alpha_punkt)
113    in
114    (oeje, vop, alpha)
115    end
116
117    (**
118    * Afprøvningstilfælde
119    *)
120    val navn = "CentralProjektor"
121    val afproevninger =
122    [
123    ("Opstilling af transformation",
124    fn () => Transformation ((0.0, 0.0, ~1.0, 0.0), (0.0, ~1.0, 0.0)) =
125    ((~1.0, 0.0, 0.0),
126    (0.0, ~1.0, 0.0),
127    (0.0, 0.0, ~1.0))),
128    ("Projicering af diverse punkter",
129    fn () => ProjicerPunkter ((0.0, 0.0, ~10.0), Plan.xy, Matrix3x3.id,
130    [(0.0, 0.0, 10.0), (* I sigtelinjen mod origo *)
131    (4.0, 6.0, 10.0)]) = (* Ekvidistant med øjet i
132    forh. til alpha *)
133    [(0.0, 0.0, 0.0),
134    (2.0, 3.0, 0.0)]),
135    ("Ugyldige projiceringe: I \"øjeplanet\"",
136    fn () => Afproevning.SkalFejle (ProjicerPunkter,

```

```

135                                     ((0.0, 0.0, ~10.0), Plan.xy, Matrix3x3.id,
136                                     [(1.0, 1.0, ~10.0)])),
137     ("Klipping mod xy-planet",
138     fn () => FjernHitsiden (Polygon((1, 1),
139                                     (* Et triangel udstrakt i xz-planet *)
140                                     [(10.0, 0.0, 5.0),
141                                     (~10.0, 0.0, 5.0),
142                                     (0.0, 0.0, ~5.0)]),
143                                     Plan.xy, (0.0, 0.0, ~10.0)) =
144     SOME (Polygon((1, 1),
145                  [(5.0, 0.0, 0.0),
146                  (10.0, 0.0, 5.0),
147                  (~10.0, 0.0, 5.0),
148                  (~5.0, 0.0, 0.0)])))
149     ]
150     end;

```

Afprøvningsvektøj (Afprøevning.sml)

```

1 (*
2  * Rammeværk for modulafprøvning
3  *
4  * Dette modul er udenfor datastrømmen, og indeholder værktøjer til
5  * nemt at lave intern afprøvning af de forskellige moduler.
6  *
7  * Den centrale funktion, er Afproevning.Afproev, der kører en liste af
8  * prædikater, og returnerer nummer og beskrivelse på de, der fejlede.
9  *
10 * Tanken er, at de forskellige moduler skal have en liste af
11 * afprøvningstilfælde, og at disse så køres ved hjælp af dette modul.
12 *
13 * De fordele dette giver, er:
14 *
15 * - Lokalisering: Afprøvningstilfælderne der fejler er meget nemmere at
16 *   genfinde end ved den typiske 'grep "false"-løsning
17 *
18 * - Strukturering: Både kildekodefilerne og skrivningen af
19 *   afprøvningstilfælde bliver automatisk mer struktureret.
20 *
21 * - Afprøvninger køres ikke uanset: Man ønsker som regel ikke at
22 *   afprøvningsne skal køres i de situationer hvor man kun skal
23 *   anvende modulen da det bl.a. lægger støj ind i programmets
24 *   uddata.
25 *
26 * - Dokumentation af afprøvningstilfælde er ikke mulig at udelade
27 *
28 * - Potentiale for udvidelse: Denne metode har potentiale for at
29 *   udvides med de korrækte midler til at omfatte fx metoder
30 *   for at vise hvad der var forventet sammen med de faktiske
31 *   uddata o.l.
32 *
33 * @author: Jørgen H. Seland
34 * @version: $Revision: 1.5 $
35 *)
36 signature Afproevning =
37     sig
38         val TilnaermetLig: real * real -> bool
39         val SkalFejle: (('a -> 'b) * 'a) -> bool
40         val Afproev: int * (string * (unit -> bool)) list -> (int * string) list
41     end
42
43 structure Afproevning :> Afproevning =
44     struct
45         (* TODO: Få dette til at virke
46
47         type afpr = ('a -> 'b) * 'a * 'b;
48
49         signature Afproevbar =

```

```
50         sig
51         val navn: string;
52         val afproevninger: afpr list;
53     end;
54
55     fun AfproevModul (modul : Afproevbar) =
56         Afproev (modul.navn, 0, modul.afproevninger);
57     *)
58
59     (**
60     * Kører den givne liste af afprøvninger, og returnerer en liste der indeholder
61     * nummeret på dem, der fejlede.
62     *
63     * Hvis et afprøvningstilfælde rejser en undtagelse, regnes det som en fejl.
64     * Brug 'SkalFejle' for de funktioner der skal rejse en undtagelse.
65     *
66     * @type int * (string * (unit -> bool)) list -> (int * string) list
67     *)
68     fun Afproev (_, []) = []
69     | Afproev (n, (navn, f) :: fs) =
70         if f () then Afproev(n + 1, fs) handle Fail e => (n, navn) :: Afproev(n + 1,
71             fs)
72         else (n, navn) :: Afproev(n + 1, fs)
73
74     (**
75     * @return true hvis a er lig b på 0.00001 nær
76     *)
77     fun TilnaermetLig (a, b) = abs(a - b) < 0.00001;
78
79     (**
80     * "Konvertering" for funktioner der skal rejse undtagelser under bestemte
81     * forhold.
82     *
83     * TODO: Send undtagelsen videre hvis den ikke er en "Fail".
84     *
85     * @type (('a -> 'b) * 'a) -> bool
86     * @return true hvis den givne funktion rejser en undtagelse
87     *)
88     fun SkalFejle (f, a) = (f a; false) handle _ => true;
89 end;
```

Program for at køre den interne afprøvning (intern-afproevning.sml)

```
1 (*
2  * Program for at køre afprøvning på alle modulene
3  *
4  * Mere information i modulen "Afproevning"
5  *
6  * @author: Jørgen H. Seland
7  * @version: $Revision: 1.3 $
8  *)
9 load "Math";
10 load "Real";
11 load "Int";
12
13 use "Analyse.sml";
14 use "Afproevning.sml";
15 use "Vektor3D.sml";
16 use "Plan.sml";
17 use "Matrix3x3.sml";
18 use "Polygon.sml";
19 use "CentralProjektor.sml";
20 use "BSPTrae.sml";
21 use "FIG.sml";
22
23 val moduler =
24     [
25         (Vektor3D.navn, Vektor3D.afproevninger),
```

```

26     (Plan.navn, Plan.afproevninger),
27     (Matrix3x3.navn, Matrix3x3.afproevninger),
28     (Polygon.navn, Polygon.afproevninger),
29     (CentralProjektor.navn, CentralProjektor.afproevninger),
30     (BSPTrae.navn, BSPTrae.afproevninger)
31
32     (* Ingen interne tester for FIG, da de ville blive så komplicerede
33      * at vi ville bruge en masse tid på at debugge
34      * afprøvningstilfælderne... *)
35
36 ];
37
38 (**
39  * Laver en liste i tekstform ud ifra den givne liste af fejlnavn
40  * og -nummer
41  *
42  * @type (int * string) list -> string
43  *)
44 fun FejlListeTilStreng [] = "OK"
45   | FejlListeTilStreng [(n, navn)] = Int.toString(n) ^ " (" ^ navn ^ ")"
46   | FejlListeTilStreng ((n, navn) :: ns) = Int.toString(n) ^ " (" ^ navn ^ "), " ^
      FejlListeTilStreng ns;
47
48 (**
49  * Kører afprøvningstilfælderne i den givne liste
50  *
51  * @type string * (unit -> bool) list -> (int * string) list
52  *)
53 fun AfproevModuler [] = ""
54   | AfproevModuler ((navn, afproevninger) :: ms) =
55     let
56       val fejl = Afproevning.Afproev (1, afproevninger);
57     in
58       (if fejl <> [] then navn ^ ": " ^ FejlListeTilStreng fejl ^ "\n" else "") ^
        AfproevModuler ms
59     end;
60
61 print ("Fejlede afproevninger:\n" ^ AfproevModuler moduler ^ "Ferdig.\n");

```

C Afprøvningsscener

gyldig/1-a.sud

```

1 ; Markørscene med forskellige farver der peger i de positive akserætningerne
2 (scene
3   ; Indikerer origo, sort
4   (para (vekt -0.5 -0.5 -0.5) (vekt 1 0 0) (vekt 0 1 0) (vekt 0 0 1) (farve 7 5))
5   ; Peger mod positiv X, rød
6   (tetra (vekt 5.5 0 0) (vekt -4 -0.5 0) (vekt -4 0.5 -0.5) (vekt -4 0.5 0.5) (farve 0 4))
7   ; Peger mod positiv Y, blå
8   (tetra (vekt 0 5.5 0) (vekt 0 -4 -0.5) (vekt -0.5 -4 0.5) (vekt 0.5 -4 0.5) (farve 0 2))
9   ; Peger mod positiv Z, grøn
10  (tetra (vekt 0 0 5.5) (vekt 0 -0.5 -4) (vekt -0.5 0.5 -4) (vekt 0.5 0.5 -4) (farve 0 1))
11 )

```

gyldig/tetra.sud

```

1 (scene
2   (tetra (vekt 0 -1 0) (vekt -1 2 -1) (vekt 1 2 -1) (vekt 0 2 1) (farve 0 6)))

```

gyldig/tom.sud

```

1 (scene)

```

gyldig/skaering-linje.sud

```
1 (scene
2 (para (vekt -1 -1 -1) (vekt 2 0 0) (vekt 0 2 0) (vekt 0 0 2) (farve 0 4))
3 (para (vekt 0 0 -1) (vekt 2 0 0) (vekt 0 2 0) (vekt 0 0 2) (farve 0 2)))
```

gyldig/to-markoer.sud

```
1 (scene
2 (tetra (vekt 0 -1 0) (vekt -1 2 -1) (vekt 1 2 -1) (vekt 0 2 1) (farve 0 4))
3 (tetra (vekt 5 -1 0) (vekt -1 2 -1) (vekt 1 2 -1) (vekt 0 2 1) (farve 0 2)))
```

gyldig/cyclisk.sud

```
1 (scene
2 (para (vekt -5 0 1) (vekt 10 5 0) (vekt 0 1 0) (vekt 0 0 2) (farve 0 4))
3 (para (vekt -5 5 -3) (vekt 10 -5 0) (vekt 0 1 0) (vekt 0 0 2) (farve 0 2))
4 (para (vekt -3 5 5) (vekt 0 -5 -10) (vekt 0 1 0) (vekt 2 0 0) (farve 0 1))
5 (para (vekt 1 0 5) (vekt 0 5 -10) (vekt 0 1 0) (vekt 2 0 0) (farve 0 6))
6 )
```

gyldig/para.sud

```
1 (scene
2 (para (vekt -1 -1 -1) (vekt 2 0 0) (vekt 0 2 0) (vekt 0 0 2) (farve 0 6)))
```

gyldig/skaering-punkt.sud

```
1 (scene
2 (para (vekt -1 -1 -1) (vekt 2 0 0) (vekt 0 2 0) (vekt 0 0 2) (farve 0 4))
3 (para (vekt 0 -1 0) (vekt 2 0 0) (vekt 0 2 0) (vekt 0 0 2) (farve 0 2))
4 (para (vekt 0 0 -1) (vekt 2 0 0) (vekt 0 2 0) (vekt 0 0 2) (farve 0 1)))
```

gyldig/klippemarkoer.sud

```
1 (scene
2 (para (vekt -2 -2 -2) (vekt 4 0 0) (vekt 0 4 0) (vekt 0 0 4) (farve 0 4))
3 (tetra (vekt 0 -1 0) (vekt -1 2 -1) (vekt 1 2 -1) (vekt 0 2 1) (farve 0 5)))
```

ugyldig/2-a.sud

```
1 (scene (ukendt_udtryk))
```

ugyldig/2-b-i.sud

```
1 (scene (tetra ukendt))
```

ugyldig/2-b-ii.sud

```
1 (scene (tetra (vekt 2.0 3 4.0) (vekt 2.0 2 3.0) (vekt 2.0 2 2.0)))
```

ugyldig/2-b-iii.sud

```
1 (scene
2 (tetra (vekt 3.0 2.0) (vekt A B C) (vekt 3.3 3.3. 3.3) (vekt 3.3 23 43) (farve 2 2)))
```

ugyldig/2-c-i.sud

```
1 (scene
2 (para (vekt 1 1 1) (vekt 2 3 2) (vekt 5 3 4) (vekt 1 1 4) (farve 1)))
```

ugyldig/2-c-ii.sud

```
1 (scene
2 (para (vekt 1 1 1) (vekt 2 3 2) (vekt 5 3 4) (vekt 1 1 4) (farve 1 2.5)))
```

ugyldig/2-c-iii.sud

```
1 (scene
2 (para (vekt 1 1 1) (vekt 2 3 2) (vekt 5 3 4) (vekt 1 1 4) (farve 544 2)))
```

ugyldig/2-c-iv.sud

```
1 (scene
2 (para (vekt 1 1 1) (vekt 2 3 2) (vekt 5 3 4) (vekt 1 1 4) (farve -2 2)))
```

ugyldig/2-d-i.sud

```
1 (scene
2 (tetra (vekt 3.0 2.0 4.0) (vekt 1.4 2.3 2.3) (vekt 3.3 3.3 3.3) (vekt 3.3 43) (farve
2 2)))
```

ugyldig/2-d-ii.sud

```
1 (scene
2 (tetra (vekt 3.0 2.0 4.0) (vekt 3.2 3.2 1.1) (vekt 3.3 3.3 3.3) (vekt 3.3 A 43) (farve
2 2)))
```

ugyldig/3-a-i.sud

```
1 (scene
2 (tetra (vekt 0 0 0) (vekt 0 0 0) (vekt 1 0 0) (vekt 0 1 0) (farve 1 2)))
```

ugyldig/3-a-ii.sud

```
1 (scene
2 (tetra (vekt 0 0 0) (vekt 1 0 0) (vekt 1 0 0) (vekt 0 1 0) (farve 1 2)))
```

ugyldig/3-b-i.sud

```
1 (scene
2 (para (vekt 0 0 0) (vekt 0 0 0) (vekt 1 0 0) (vekt 0 1 0) (farve 1 2)))
```

ugyldig/3-b-ii.sud

```
1 (scene
2 (para (vekt 0 0 0) (vekt 1 0 0) (vekt 1 0 0) (vekt 0 1 0) (farve 1 2)))
```