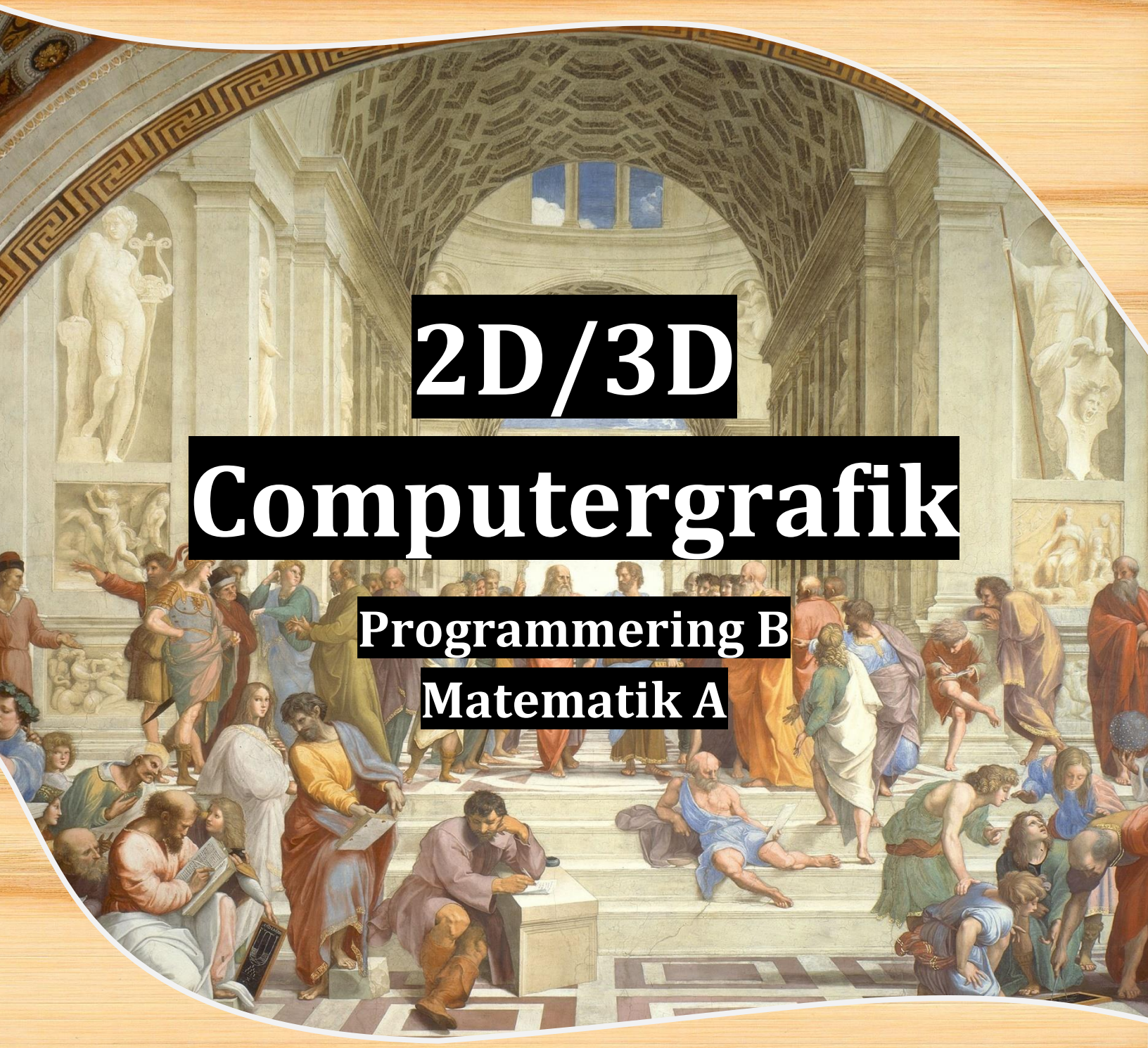
Studieområde

3D Computergrafik

Rybners
Tekniske
Gymnasie



2D/3D Computergrafik

Programmering B
Matematik A

AF ESBEN DAMKJÆR SØRENSEN

STUDIEOMRÅDEPROJEKT

PROG. B - VEJLEDER: [REDACTED]

MAT. A - VEJLEDER: [REDACTED]

PROJEKTPERIODE:
6/12/2019 - 20/12/2019

Resumé

Med udgangspunkt i emnet 3D-/2D-grafik undersøger dette projekt, hvordan matriceregning vha. Falks Skema kan anvendes i 3D-grafik til at give illusionen af tre dimensioner på en todimensionel skærm.

På baggrund af teori gennemgår projektet et matematisk eksempel, der ved anvendelse af perspektivmatricen transformerer en pyramides vertexdata til en visuel todimensionel repræsentation af objektet. Herudover gennemgår projektet en softwareimplementation af selvsamme eksempel vha. grafikbiblioteket OpenGL.

Projektet konkluderer, at perspektivmatricen anvendes til at simulere et kamera, men ikke alene er nok til at give illusionen af 3D. Det er først ved den perspektivgivende division, at transformationen fra 3D til 2D er fuldendt.

Ved anvendelse af OpenGL, kan perspektivmatricen programmeres ind i den såkaldte ”rendering pipeline”, hvorved alle vertexdataene, en model er opbygget af, vil blive transformeret, så de kan repræsenteres i 2D.

OpenGL giver en stor kontrol over computerens ressourcer. Det kan være en fordel, hvis produktet er meget ydelseskrævende, men kræver samtidigt en stor investering af tid for selv at rendere noget simpelt. Hvis formålet f.eks. er at udvikle et computerspil, kan en spilmotor som Unity være et bedre valg, da denne tilbyder prædefinerede værktøjer til udvikling af spil, der så bygger på OpenGL eller lign.

Indhold

1 Indledning

3D computergrafik har stor betydning for os, hvor det bliver anvendt til et utal af applikationer fra udvikling af aerodynamiske fly til molekylærbiologi og computerspil; men vi er så vandt til selv at se 3D-objekter i perspektiv, at vi ikke tænker over, hvor perspektivet egentligt kommer fra, og at vi faktisk ser i 2D. Netop derfor vil dette projekt beskæftige sig med emnet 2D/3D computergrafik, og i det følgende afsnit præsenteres projektets opgaveformulering.

2 Opgaveformulering

Man ønsker ofte at fremstille 3D-objekter på en 2D computerskærm. Til dette formål anvendes en såkaldt perspektivmatrix.

Redegør for de grundlæggende matematiske og programmatiske principper i forbindelse med perspektivmatricen og dens anvendelse. Herunder inddrages:

- Introduktion til matricer og grundlæggende matrixregning (f.eks. matrice multiplikation)
- Redegør og forklar perspektivmatricens opbygning og funktion. Brug til dine forklaringer et passende eksempel.
- Forklar processen hvorved man går fra vertexdata til repræsentationen af en 3D-figur på en todimensionel skærm, samt de bagvedliggende processer for 3D-grafik generelt.
- Producér et eksempel på implementation af 3D-grafik ved hjælp af OpenGL og vurder OpenGL som værktøj til 3D-grafik.

Perspektiver din teoretiske viden omkring perspektivmatricen og dens softwareimplementering m.h.t. den praktiske anvendelse af grafisk fremstilling af 3D-objekter på en 2D computerskærm.

Metoderne, der benyttes til at besvare ovenstående opgaveformulering, præsenteres i næste afsnit.

3 Metode

Projektet vil med udgangspunkt i emnet søge information gennem faglitterære kilder primært bestående af bøger. Den erhvervede viden vil blive anvendt til at besvare opgaveformuleringen.

Gennem en matematisk udledning vha. Falks Skema vil projektet redegøre for matricens opbygning og anvende denne gennem et beregningseksempel for at sammenligne det med en softwareimplementation.

I forbindelse med projektet vil en softwareimplementation, der bygger på viden fra kilderne, blive udviklet og undervejs testet gennem en metodisk og struktureret tilgang vha. Agile Unified Process for at mindske spildtid. Softwareimplementationen vil anvendes til at binde den matematiske og programmeringsfaglige viden sammen.

Teorien, der danner baggrund for projektets undersøgelser, præsenteres i de følgende afsnit.

4 Teori

4.1 Matrixregning

Selvom 3D computergrafik er forholdsvist nyt, har matematikken bag været kendt i flere hundrede år. Matricer er en måde at gruppere talværdier på, og de er en vigtig del af computergrafik. De anvendes bl.a. til at lagre information om et objekts transformationer, herunder rotation, skalering og forskydning.

Matricer navngives med store bogstaver som f.eks. A. En matrix af $m \times n$ størrelse er en matrix med m rækker og n kolonner. Hvis A er en $m \times n$ matrix, betegnes indgangen til i'te række og j'te kolonne i A som det lille bogstav af matricens navn efterfulgt af række- og kolonnenummer a_{ij}, og kaldes for (i,j)-indgangen (1 s. 94). F.eks. er indgang $(4,3)$ tallet a_{43}, der ligger på fjerde række i kolonne tre.
Hver kolonne er en liste med reelle tal, og disse kolonner kan betegnes $a_1\ a_2 \dots a_{n-1}\ a_n$, og dermed kan matricen A skrives $A = [a_1 \quad a_2 \quad \cdots \quad a_{n-1} \quad a_n]$. Disse kolonner kan også identificeres som vektorer af m dimensioner. Matrix notation kan ses i figur 1.

$$\begin{matrix} & & & \text{Kolonne } j & & \\ & [a_1 & \dots & a_j & \dots & a_n] \\ & \downarrow & & \downarrow & & \downarrow \\ A = & \begin{bmatrix} a_{11} & \dots & a_{1j} & \dots & a_{1n} \\ \vdots & & \vdots & & \vdots \\ a_{i1} & \dots & a_{ij} & \dots & a_{in} \\ \vdots & & \vdots & & \vdots \\ a_{m1} & \dots & a_{mj} & \dots & a_{mn} \end{bmatrix} & & \text{Række } i \end{matrix}$$

figur 1: Matrix notation

To matricer er lige store, hvis de har samme antal kolonner og rækker. Hvis begge matricer også har lige store kolonner, vil det sige, at alle indgange parvis fra hhv. den ene og den anden matrix indeholder ens værdier. Dette svarer også til at sige, at begge matricer har "tilsvarende lige store indgange" (1).

4.1.1 Matrixaddition

Addition af to $m \times n$ matricer svarer til at lægge alle tilsvarende kolonner sammen parvis, som var det vektoraddition, og sætte i en resulterende matrice af $m \times n$ størrelse. Når to matricer A og B af $m \times n$ størrelse lægges sammen, sker det per indgang. Hver indgang fra A lægges til den tilsvarende indgang i B og kan skrives som ligning (1), hvor C er summen af matricerne A og B.

$$A + B = C \Rightarrow a_{ij} + b_{ij} = c_{ij} \quad \text{Ref: (2)} \tag{1}$$

Af den grund er summen af to matricer kun defineret, hvis begge matricer er lige store, da der ellers for nogle indgange ikke er en tilsvarende indgang i den ene addend.

4.1.2 Skalar-matrix multiplikation

Hvis r er en skalar, og A er er en matrix, ganges skalaren ind på hver kolonne, hvor samme regler gælder, som var det en vektor, der blev multipliceret med skalaren. Skalaren bliver multipliceret med hver indgang i matricen, der ses af ligning (2).

$$rA = C \Rightarrow c_{ij} = r \cdot a_{ij} \quad \text{Ref: (2)} \tag{2}$$

4.1.3 Matrixsubtraktion

Ligesom med vektorer, er $-A$ det samme som $-1 \cdot A$, og matrixsubtraktion kan dermed skrives som ligning (3).

$$A - B = A + (-1)B \tag{3}$$

4.1.4 Matrix-vektor multiplikation

Hvis A er en $m \times n$ matrix, hvor kolonnerne betegnes $a_1 \dots a_n$, og v er en vektor af n dimensioner $v = \begin{bmatrix} v_1 \\ \vdots \\ v_n \end{bmatrix}$, kan disse multipliceres som i ligning (4). Dette giver en ny vektor Av, hvor hver af vektor v's koordinater er ganget ind i hver kolonne ved skalarmultiplikation.

$$Av = v_1 a_1 + v_2 a_2 \ldots + v_n a_n \tag{4}$$

Hvis ikke vektoren har lige så mange koordinater, som vektoren har kolonner, kan dette ikke beregnes, og resultatet er ikke defineret.

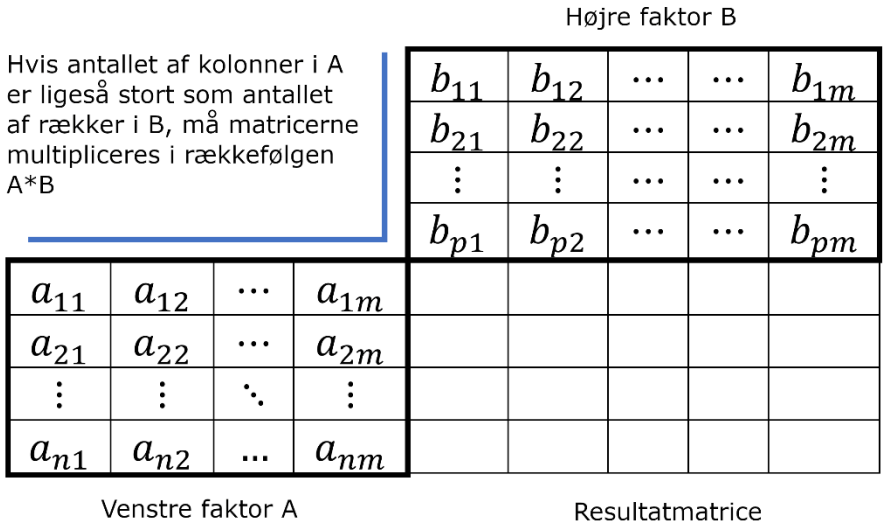
4.1.5 Matrix-matrix multiplikation

Ligesom en skalar multipliceres en matrix ind på hver kolonne af en anden matrix, hvorefter samme regler for matrix-vektor multiplikation er gældende. Hvis A er en matrice af $m \times n$ størrelse, og B er en matrice af $n \times p$ størrelse, hvis kolonner betegnes $b_1 \ldots b_p$, kan de multipliceres som i ligning (5).

$$A \cdot B = [Ab_1 \quad \ldots \quad Ab_p] \tag{5}$$

Som følge af matrix-vektor multiplikation er det nødvendigt, at den første faktor har lige så mange kolonner, som den anden faktor har rækker, da et resultat ellers ikke er defineret. Derfor er det heller ikke uden betydning, hvilken rækkefølge matricer multipliceres i.

Den tyske matematiker Sigurd Falk udviklede en metode kaldet "Falkches Schema" (3) (Da. Falks Skema), der let kan benyttes til at finde ud af, hvilken rækkefølge to faktorer kan placeres i for at være gyldig, og derefter beregne den resulterende matrice. Falks Skema er vist i figur 2, hvor faktorerne placeres som i tabellen. Den venstre faktors højde og den højre faktors bredde danner tilsammen resultatmatricen, hvor den resulterende matrice kan beregnes (3).



figur 2: Falks Skema

Når resultatmatricen skal beregnes, kigges på indgangens tilsvarende række i A og kolonne i B. Hvis rækken fra A rejses vertikalt som en vektor, er indgangen i den resulterende matrix lig skalarproduktet af kolonnen i B og rækken i A (3), som det ses i ligning (6) og figur 3.

$$(AB)_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \cdots + a_{in}b_{nj} \tag{6}$$

				b_{11}	b_{12}	$\cdots$	$\cdots$	b_{1m}
				b_{21}	b_{22}	$\cdots$	$\cdots$	b_{2m}
				$\vdots$	$\vdots$	$\cdots$	$\cdots$	$\vdots$
				b_{p1}	b_{p2}	$\cdots$	$\cdots$	b_{pm}
a_{11}	a_{12}	$\cdots$	a_{1m}	$a_{11} \cdot b_{11} + a_{12} \cdot b_{21} + \cdots + a_{1m} \cdot b_{p1}$	$a_{11} \cdot b_{12} + a_{12} \cdot b_{22} + \cdots + a_{1m} \cdot b_{p2}$	$\cdots$	$\cdots$	$a_{11} \cdot b_{1m} + a_{12} \cdot b_{2m} + \cdots + a_{1m} \cdot b_{pm}$
a_{21}	a_{22}	$\cdots$	a_{2m}	$a_{21} \cdot b_{11} + a_{22} \cdot b_{21} + \cdots + a_{2m} \cdot b_{p1}$	$a_{21} \cdot b_{12} + a_{22} \cdot b_{22} + \cdots + a_{2m} \cdot b_{p2}$	$\cdots$	$\cdots$	$a_{21} \cdot b_{2m} + a_{22} \cdot b_{2m} + \cdots + a_{2m} \cdot b_{pm}$
$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
a_{n1}	a_{n2}	$\cdots$	a_{nm}	$a_{n1} \cdot b_{11} + a_{n2} \cdot b_{21} + \cdots + a_{nm} \cdot b_{p1}$	$a_{n1} \cdot b_{12} + a_{n2} \cdot b_{22} + \cdots + a_{nm} \cdot b_{p2}$	$\cdots$	$\cdots$	$a_{n1} \cdot b_{1m} + a_{n2} \cdot b_{2m} + \cdots + a_{nm} \cdot b_{pm}$

figur 3: Matrixmultiplikation vha. Falks skema

4.1.6 Identitetsmatricer

En kvadratisk matrice A af $n \times n$ størrelse har hoveddiagonal ned gennem a_{ii}. Hvis disse indgange har værdien 1, mens de resterende værdier har værdien 0, kaldes denne matrice for en identitetsmatrice. En identitetsmatrix har den egenskab, at identitetsmatricen multipliceret med en anden matrix giver denne matrix (1). Hvis en identitetsmatrix I af $n \times n$ størrelse multipliceres med en matrix B af $n \times m$ størrelse gælder ligning (7), som kan ses i Falks skema i figur 4.

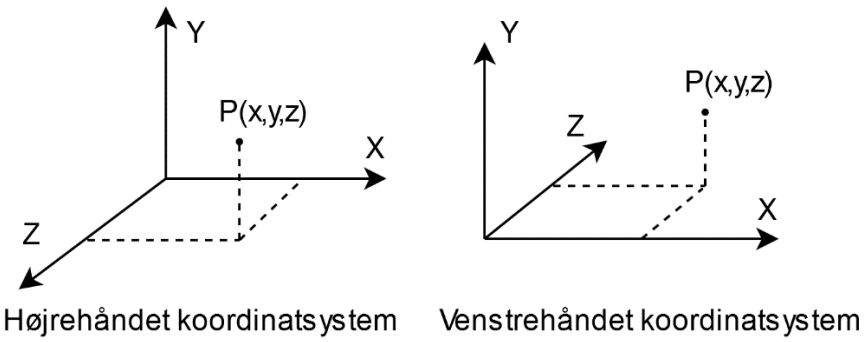
$$I \cdot B = B \tag{7}$$

				b_{11}	b_{12}	$\cdots$	$\cdots$	b_{1m}
				b_{21}	b_{22}	$\cdots$	$\cdots$	b_{2m}
				$\vdots$	$\vdots$	$\cdots$	$\cdots$	$\vdots$
				b_{p1}	b_{p2}	$\cdots$	$\cdots$	b_{pm}
1	0	$\cdots$	0	$1 \cdot b_{11} + 0 \cdot b_{21} + \cdots + 0 \cdot b_{p1} = b_{11}$	$1 \cdot b_{12} + 0 \cdot b_{22} + \cdots + 0 \cdot b_{p2} = b_{12}$	$\cdots$	$\cdots$	$1 \cdot b_{1m} + 0 \cdot b_{2m} + \cdots + 0 \cdot b_{pm} = b_{1m}$
0	1	$\ddots$	$\vdots$	$0 \cdot b_{11} + 1 \cdot b_{21} + \cdots + 0 \cdot b_{p1} = b_{21}$	$0 \cdot b_{12} + 1 \cdot b_{22} + \cdots + 0 \cdot b_{p2} = b_{22}$	$\cdots$	$\cdots$	$0 \cdot b_{2m} + 1 \cdot b_{2m} + \cdots + 0 \cdot b_{pm} = b_{2m}$
$\vdots$	$\ddots$	$\ddots$	0	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
0	$\cdots$	0	1	$0 \cdot b_{11} + 0 \cdot b_{21} + \cdots + 1 \cdot b_{p1} = b_{p1}$	$0 \cdot b_{12} + 0 \cdot b_{22} + \cdots + 1 \cdot b_{p2} = b_{p2}$	$\cdots$	$\cdots$	$0 \cdot b_{1m} + 0 \cdot b_{2m} + \cdots + 1 \cdot b_{pm} = b_{pm}$

figur 4: Matrixmultiplikation af $I \cdot B$

4.1.7 Koordinatsystemet

Det kartesiske koordinatsystem blev i 1600-tallet opfundet af den franske filosof, fysiker og matematiker René Descartes. Dette er det traditionelle anerkendte retvinklede koordinatsystem (4). Den tredimensionelle udgave af det kartesiske koordinatsystem findes i to udgaver; det venstre- og højrehåndede koordinatsystem, hvor Z-aksen er spejlet over X-aksen (figur 5).



figur 5: 3D koordinatsystemer

I størstedelen af tilfældene er det det højrehåndede koordinatsystem, der bliver anvendt i OpenGL (5). Et punkt i et højrehåndet koordinatsystem kan let overføres til et venstrehåndet koordinatsystem ved at negere z-koordinatet.

Det tredimensionelle kartesiske koordinatsystems tre akser kan beskrives med de tre enhedsvektorer, der ses i ligning (8).

$$i = \begin{bmatrix}1\\0\\0\end{bmatrix};\ j = \begin{bmatrix}0\\1\\0\end{bmatrix};\ k = \begin{bmatrix}0\\0\\1\end{bmatrix} \tag{8}$$

4.1.7.1 Koordinater

I det kartesiske koordinatsystem opskrives et punkt P typisk $P(x, y, z)$. Dette kan også opskrives som en stedvektor, der går fra origo til punktet P; $\vec{v} = \begin{bmatrix}P_x\\P_y\\P_z\end{bmatrix}$, hvilket også kan opstilles som summen af enhedsvektorerne *i*, *j* og *k* der hhv. er skaleret med P_x, P_y og P_z, som set i ligning (8).

$$\vec{v} = P_x \cdot i + P_y \cdot j + P_z \cdot k = P_x \cdot \begin{bmatrix}1\\0\\0\end{bmatrix} + P_y \cdot \begin{bmatrix}0\\1\\0\end{bmatrix} + P_z \cdot \begin{bmatrix}0\\0\\1\end{bmatrix} = \begin{bmatrix}P_x\\P_y\\P_z\end{bmatrix} \quad (9)$$

Det kan fra det tidligere afsnit om matrix-vektor multiplikation (0) genkendes, at ligning (9) ligner ligning (4), og ligning (9) kan derfor også skrives som set i ligning (10).

$$\vec{v} = [i \quad j \quad k] \cdot \begin{bmatrix}P_x\\P_y\\P_z\end{bmatrix} = \begin{bmatrix}1 & 0 & 0\\0 & 1 & 0\\0 & 0 & 1\end{bmatrix} \cdot \begin{bmatrix}P_x\\P_y\\P_z\end{bmatrix} = \begin{bmatrix}P_x\\P_y\\P_z\end{bmatrix} \quad (10)$$

Det kan observeres af ligning (10), at der er tale om multiplikation med en 3×3 identitetsmatrix, der er opbygget af enhedsvektorerne *i*, *j* og *k* og punktet P i vektornotation. Denne måde at benytte enhedsvektorerne *i*, *j* og *k*, viser sig meget praktisk, når et punkt skal transformeres for eksempel ved rotation eller skalering af koordinatsystemets akser.

4.1.8 Skaleringsmatricen

Skaleringsmatricen kan bruges til at skalere koordinatsystemet, et punkt er placeret i. Hver af koordinatsystemets akser kan skaleres individuelt ved at multiplicere med en skalar på enhedsvektorerne *i*, *j* og *k* (6). Denne matrice kan dermed bruges til at skalere et punkt (figur 6) eller hele objekter opbygget af en serie af punkter.

$$M_S = [S_x \cdot i \quad S_y \cdot j \quad S_z \cdot k] = \begin{bmatrix}S_x & 0 & 0\\0 & S_y & 0\\0 & 0 & S_z\end{bmatrix} \quad (11)$$

			x
			y
			z
S_x	0	0	$S_x \cdot x + 0 \cdot y + 0 \cdot z = S_x \cdot x$
0	S_y	0	$0 \cdot x + S_y \cdot y + 0 \cdot z = S_y \cdot y$
0	0	S_z	$0 \cdot x + 0 \cdot y + S_z \cdot z = S_z \cdot z$

figur 6: Skalering vha. Falks Skema

4.1.9 Rotationsmatricer

I det tredimensionelle højrehåndskoordinatsystem kan den positive omdrejningsretning bestemmes ved at tage fat med den højre hånd omkring aksen, der roteres omkring med tommelfingeren i aksens positive retning. Den positive omdrejningsretning er dermed retningen, hvormed de øvrige fingre griber omkring aksen (1).

Når et punkt eller et objekt udgjort af en serie af punkter skal roteres omkring en akse i den positive omdrejningsretning, kan dette gøres ved brug af en rotationsmatrice. For rotation i tredimensionelt rum findes der tre rotationsmatricer; én til hver akse.

De tre rotationsmatricer, der anvendes til at rotere et punkt omkring hhv. x-, y- og z-aksen, kan ses i figur 7, og beviset for disse findes under afsnit 8.1 i bilag.

$$R_x(\alpha) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\alpha) & -\sin(\alpha) \\ 0 & \sin(\alpha) & \cos(\alpha) \end{bmatrix}$$

$$R_y(\alpha) = \begin{bmatrix} \cos(\alpha) & 0 & \sin(\alpha) \\ 0 & 1 & 0 \\ -\sin(\alpha) & 0 & \cos(\alpha) \end{bmatrix}$$

$$R_z(\alpha) = \begin{bmatrix} \cos(\alpha) & -\sin(\alpha) & 0 \\ \sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

figur 7: Rotationsmatricer til rotation omkring hhv. x-, y- og z-aksen

4.1.10 Homogene koordinater

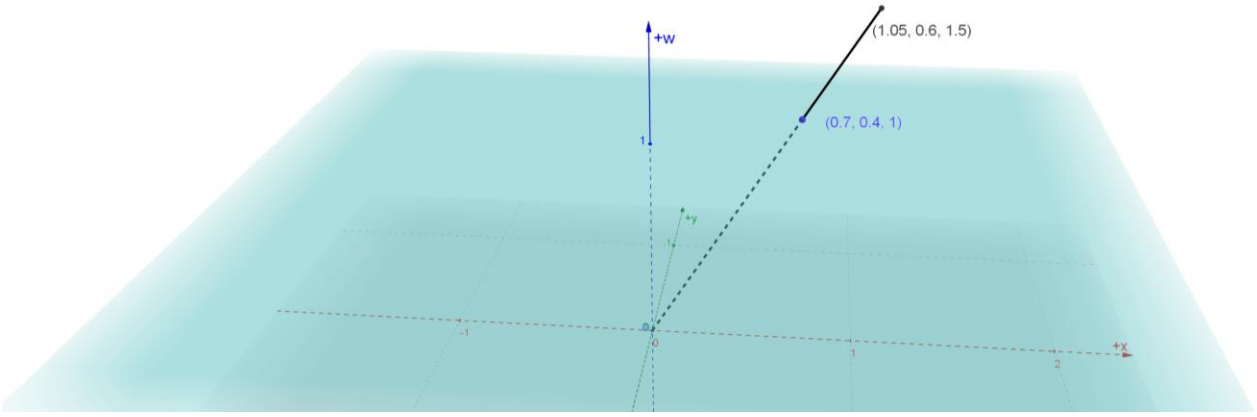
Når der er tale om homogene koordinater introduceres en ekstra dimension, der kaldes w (ligning (12)) (4).

$$P = \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} \tag{12}$$

Homogene koordinater siges at blive normaliseret, når hvert koordinat divideres med w-koordinatet, hvorved de projiceres ind i et tredimensionelt rum (4), som det ses af ligning (13).

$$P = \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} \Leftrightarrow P = \begin{bmatrix} x/w \\ y/w \\ z/w \\ 1 \end{bmatrix} \tag{13}$$

Dette princip kan bedre illustreres med homogene 3D-koordinater, der ved normalisering projiceres på planet $w = 1$.



figur 8: Normalisering - homogent koordinat projiceres på planet w=1 i 2D.

4.1.11 Forskydningsmatricen

Homogene koordinater finder praktisk anvendelse, når det kommer til transformationsmatricer, der spiller en stor rolle i 3D-grafik, men disse kan ikke multipliceres med transformationsmatricerne, da antallet af kolonner ikke længere stemmer overens med antallet af rækker som følge af det fjerde w-koordinat.

Hvis det antages, et punkt altid er normaliseret, således at $w = 1$, og et 3D-punkt (x, y, z) repræsenteres i 4D som $(x, y, z, 1)$, kan hvilken som helst 3×3 transformationsmatrice skrives som i ligning (14) (4), hvilket følger samme tilgang som ligning (10), men for et firedimensionelt koordinatsystem.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \Rightarrow \begin{bmatrix} a_{11} & a_{12} & a_{13} & 0 \\ a_{21} & a_{22} & a_{23} & 0 \\ a_{31} & a_{32} & a_{33} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{14}$$

Det kan ses i figur 9, at den eneste forskel er w-koordinatet, der er lig 1, og den nye række og kolonne på transformationsmatricen ellers ikke har betydning for de resterende koordinater, da der multipliceres med 0.

				x
				y
				z
				1
a_{11}	a_{12}	a_{13}	0	$a_{11} \cdot x + a_{12} \cdot y + a_{13} \cdot z + 0 \cdot 1 = a_{11}x + a_{12}y + a_{13}z$
a_{21}	a_{22}	a_{23}	0	$a_{21} \cdot x + a_{22} \cdot y + a_{23} \cdot z + 0 \cdot 1 = a_{21}x + a_{22}y + a_{23}z$
a_{31}	a_{32}	a_{33}	0	$a_{31} \cdot x + a_{32} \cdot y + a_{33} \cdot z + 0 \cdot 1 = a_{31}x + a_{32}y + a_{33}z$
0	0	0	1	$0 \cdot x + 0 \cdot y + 0 \cdot z + 1 \cdot 1 = 1$

figur 9: 4×4 tranformationsmatrix multipliceret med homogent 4D punkt

Denne måde at opskrive en transformationsmatrice gør det muligt at forskyde et punkt ved brug af matrixmultiplikation uden at gøre brug af matrixaddition. Dette foregår vha. kolonne a_4 i figur 9. Ved matrixmultiplikation multipliceres dennes indgange med 1 fra punktets w-koordinat og lægges til x-, y- og z-koordinaterne af punktet (4).

Forskydning af et punkt kan ses i figur 10, hvor 3×3 transformationsmatricen er en identitetsmatrice, hvilket betyder, at faktoren foran koordinaterne til punktet, der forskydes, er 1, hvormed resultatet er et punkt (x, y, z) forskudt med $(\Delta x, \Delta y, \Delta z)$.

				x
				y
				z
				1
1	0	0	Δx	$1 \cdot x + 0 \cdot y + 0 \cdot z + \Delta x \cdot 1 = x + \Delta x$
0	1	0	Δy	$0 \cdot x + 1 \cdot y + 0 \cdot z + \Delta y \cdot 1 = y + \Delta y$
0	0	1	Δz	$0 \cdot x + 0 \cdot y + 1 \cdot z + \Delta z \cdot 1 = z + \Delta z$
0	0	0	1	$0 \cdot x + 0 \cdot y + 0 \cdot z + 1 \cdot 1 = 1$

figur 10: Forskydning af punkt vha. forskydningsmatricen

Forskydningsmatricen kan derfor opskrives som vist i ligning (15).

$$T\begin{pmatrix}\Delta x\\ \Delta y\\ \Delta Z\end{pmatrix} = \begin{bmatrix}1 & 0 & 0 & \Delta x\\ 0 & 1 & 0 & \Delta y\\ 0 & 0 & 1 & \Delta z\\ 0 & 0 & 0 & 1\end{bmatrix} \quad (15)$$

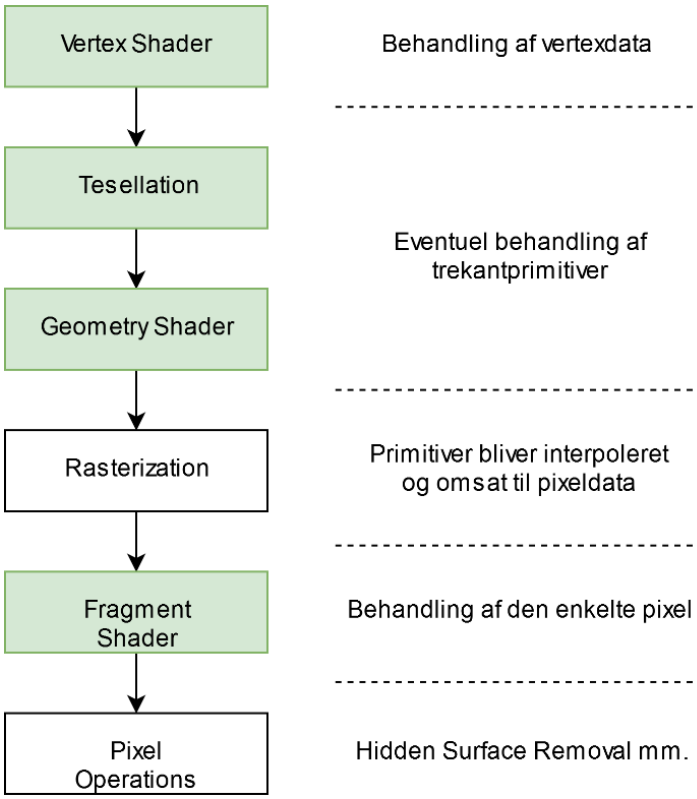
4.2 OpenGL

Navnet ”OpenGL” er en forkortelse for ”Open Graphics Library”. OpenGL er en 2D- og 3D-grafik API, der både omfatter software og hardware. Det betyder, at for at gøre brug af OpenGL, kræver det et grafikkort, der understøtter en version,

der er tilsvarende den eller højere, end den OpenGL version, et vilkårligt stykke software gør brug af, for at kunne eksekvere på den specifikke computer og grafikkort (5).

OpenGL er multiplatform, hvilket vil sige, at den kan køre på tværs at operativsystemer, hvis softwaren kompileres til den vilkårlige platform. Gennem de omkring 27 år, OpenGL har eksisteret, har sættet af funktioner ændret sig meget (7), og i denne opgave vil der blive brugt OpenGL version 4.5. API'en udvikles af Khronos Group, der også er udvikler på efterfølgeren til OpenGL kaldet Vulkan (8).

Når OpenGL skal rendere et objekt på en skærm, gennemgår den på hardware-siden en fastsat sekvens af trin, der betegnes som OpenGLs "rendering pipeline", der ses i figur 11. Nogle af disse trin markeret med grønt er programmerbare gennem et shadingsprog der forkortes GLSL, hvilket kommer af "OpenGL Shading Language" (5). Dette projekt vil kun kigge på to af de programmerbare trin; vertex shader og fragment shader.



figur 11: Den såkaldte OpenGL pipeline

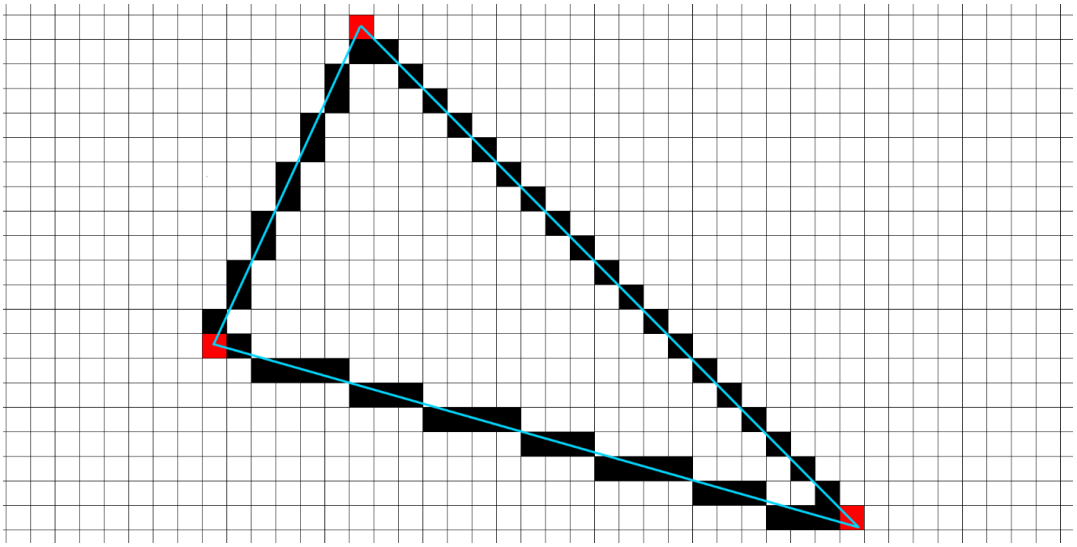
På softwaresiden er selve OpenGL API'en, der er skrevet i programmeringssproget C, og er derfor også direkte kompatibelt med programmeringssproget C++, men ikke umiddelbart med andre programmeringssprog. Til dette formål findes der såkaldte "wrappers" og "bindings", der kan bruges til at kalde metoder i OpenGL API'en fra et andet programmeringssprog end det, API'en er skrevet i (5).

I dette projekt anvendes der objektorienterede programmeringssprog Java og JOGL[1], der er en Java-binding til OpenGL. En fordel ved JOGL er, at det kan bruges sammen med Javas Swing framework hvorved et GLCanvas, der er JOGL's tegneflade, kan tilføjes til en JFrame eller anden container. Derudover anvendes biblioteket Graphicslib3D[2] til matrixoperationer, der ikke foretages af GPU'en.

4.3 Rasterisering

OpenGL er kun i stand til at tegne simple ting som punkter, linjer eller trekanter, hvilket bliver kaldt for primitiver (5). Dette er også årsagen til, at de fleste 3D-modeller er bygget op af mange primitiver, som oftest er trekanter. Primitiver er opbygget af hjørnepunkter (engelsk: "vertices"), og disse kan blive læst fra en fil, som består af en lang række vertexdata, der definerer de mange vertices, der definerer trekanterne, en 3D-model består af.

Når de mange primitiver, som en 3D-figur består af skal vises på en todimensionel skærm, der er bygget op af en såkaldt "raster". En raster er et todimensionelt eller rektangulært array af pixels. Når primitiverne, som oftest er trekanter, skal omdannes til pixels, sker dette ved rasterisering, som OpenGL tager sig af vha. interpolering (5).



figur 12: Illustration af rasterisering

Som standard tegner OpenGL udfyldte trekanter, men det kan tegne gittermodeller ved at kalde følgende metode på det GL4-objekt, der tilhører JOGL's tegneflade:

$$glPolygonMode(GL4.GL\_FRONT\_AND\_BACK, GL4.GL\_LINE)$$

[1] https://jogamp.org/
[2] https://imada.sdu.dk/~rolf/Edu/DM567/E19/graphicslib3D/

GL_LINE angiver at trekanterne ikke skal udfyldes, og $GL\_FRONT\_AND\_BACK$ angiver, at både forside og bagside af 3D-modellen skal tegnes (5).

4.4 Shaders

Shaders er programmer skrevet i det C-lignende sprog GLSL, der eksekveres på computerens grafikprocessor. GLSL-kode bliver kompileret ved kørselstid af den pågældende applikation. Årsagen til dette er, at processorarkitekturen i grafikkort varierer fra grafikkort til grafikkort. Derfor vil GLSL-koden ikke kunne køre på en stor del af grafikkort, hvis det kompileres til én specifik arkitektur (5).

4.4.1 Vertex og fragment shader

Alle vertexdata vil passere vertex shaderen ét punkt ad gangen, hvor koden i vertex shaderen vil eksekvere for hver vertex parallelt med hinanden, og det er her, alle transformationerne på dataene foregår vha. matrixberegninger.

Efter pipelinens rasteriseringsproces gennemgår hver eneste pixel en fragment shader, en primitiv dækker over. Hvis flere primitiver dækker over samme pixel, vil denne pixel gennemgå fragment shaderen for hver primitiv. Formålet med fragment shaderen er at farvelægge den pågældende pixel, og det er bl.a. her, eventuelle lysberegninger foregår (5).

4.5 JOGL og OpenGL-kald

For at anvende OpenGL gennem JOGL er det nødvendigt at implementere interfacet GLEventListener, der indeholder metoderne, der kan ses i figur 13.

Metode	**Beskrivelse**
$\boldsymbol{void\ display(GLAutoDrawable\ drawable)}$	$display$ bliver kaldt hver gang, tegnefladen GLCanvas bliver omtegnet.
$\boldsymbol{void\ dispose(GLAutoDrawable\ drawable)}$	$dispose$ bliver kaldt, når programmet afslutter.
$\boldsymbol{void\ init(GLAutoDrawable\ drawable)}$	$init$ bliver kaldt én gang før første kald til display-metoden.

$\boldsymbol{void\ reshape(GLAutoDrawable, int\ x, int\ y, int\ width, int\ width, int\ height)}$	$reshape$ bliver kaldt ved første omtegning af tegnefladen, efter størrelsen af det anvendte GLCanvas ændres.

figur 13: Metoderne i GLEventListener interfacet

I JOGL foregår alle OpenGL 4 kaldene gennem et GL4-objekt, der kan hentes ved at kalde den statiske metode $getCurrentGL()$ i klassen $GLContext$. Udviklerne bag JOGL har forsøgt at gøre Java-bindingen stort set lig C API'en, hvilket betyder, at metoderne kan slås op i den officielle dokumentation fra Khronos (5).

4.5.1 Oprettelse af et shader program

Når et shader program bestående af en vertex og fragment shader skal oprettes, foregår dette vha. metoderne, der ses i tabel 1. Først begyndes med et kald til $glCreateShader$, der opretter et shader objekt i OpenGL. Når shaderen skal bindes sammen med sin indlæste kildekode, foregår dette vha. metoden $glShaderSource$ efterfulgt af $glCompileShader$. Dette udføres for både vertex og fragment shaderen.
Selve programmet oprettes ved et kald til $glCreateProgram$, der ligesom $glCreateShader$, returnerer et ID til OpenGL-objektet. Herefter bliver de to vertex og fragment shaders fastgjort til programmet ved brug af $glAttachShader$. Når de to shaders er fastgjort til programmet, kan disse slettes vha. $glDeleteShader$. De to shaders skal sammenkædes vha. $glLinkProgram$, så de anvendes i forlængelse af hinanden af GPU'en.
Til sidst kan det oprettede program-objekt anvendes ved at kalde $glUseProgram$ (5).

Metode	Beskrivelse
$\boldsymbol{int\ glCreateShader(int\ type)}$	Denne metode tager imod et heltal, der definerer, hvilken slags shader, der er tale om. Disse typer er defineret som statiske konstanter i GL4-klassen; GL_VERTEX_SHADER og $GL\_FRAGMENT\_SHADER$. Metoden returnerer også et heltal, der er et ID, oprettet af OpenGL, hvilket refererer til shader-objektet.
$\boldsymbol{void\ glShaderSource(int\ shader, int\ count, String[]\ source, IntBuffer\ length)}$	Denne metode tager imod fire parametre. ID'et fra $glCreateShader$, antallet af linjer, shaderens kildekode består af, et array der indeholder kildekoden linje for linje og en sidste parameter, der ikke anvendes i dette projekt, og derfor er lig $null$.

$\boldsymbol{void\ glCompileShader(int\ shader)}$	Denne metode kompilerer shaderen med det givne ID, så den kan eksekveres på GPU'en.
$\boldsymbol{void\ glAttachShader(int\ program, int\ shader)}$	Denne metode fastgør en shader til et program, der angives ved brug af de tidligere fundne ID'er, i de to parametre.
$\boldsymbol{void\ glDeleteShader(int\ shader)}$	Denne metode frigør den allokerede hukommelse dedikeret til shader-objektet, hvis ID er den eneste parameter.
$\boldsymbol{void\ glLinkProgram(int\ program)}$	Denne metode sammenkæder de forbundne shaders til program-objektet.
$\boldsymbol{void\ glUseProgram(int\ program)}$	Denne metode fortæller OpenGL, at GPU'en skal anvende dette program til behandling af vertexdataene osv.

tabel 1: Metoder til oprettelse af et shader program

4.5.2 Vertex-attributter og buffere

For at et objekt kan blive tegnet, skal dets vertexdata sendes til vertex shaderen. Vertexdata sendes fra computerens processor CPU'en til GPU'en gennem en buffer på Java-siden og skal associeres med en såkaldt vertex attribut i shaderen. I OpenGL kaldes denne buffer for et "Vertex Buffer Objekt" eller VBO og bliver erklæret og instantieret i JOGL-applikationen. I mange tilfælde kræver en scene, at der bliver oprettet flere VBO'er, hvilket sædvanligvis foregår i $init$-metoden, så de er til rådighed, når programmet skal tegne en eller flere af dem.

Når $glDrawArrays$ bliver kaldt, vil alle dataene i bufferen blive sekventielt overført til shaderen. Shaderen vil som sagt blive eksekveret for hver vertex, hvilket vil sige hver tredje værdi i bufferen, da denne er et array af alle vertexkoordinaterne, og én vertex er ét 3D-koordinat.

I forbindelse med VBO'er introduceres også "Vertex Array Objekt" eller VAO, der anvendes til at organisere flere VBO'er, og OpenGL kræver derfor, at der bliver oprettet mindst én VAO. Hvis formålet er at tegne to objekter, kan dette gøres ved at erklære en enkelt VAO og to VBO'er; én for hvert objekt.

Når VAO'en og VBO'erne skal oprettes, foregår det vha. metoderne i tabel 2. Her har udviklerne af JOGL valgt, at metoderne "returnerer" ID'er gennem de givne int arrays. Dette er med henblik på at opnå lighed med OpenGL C API'en. Her er anvendt såkaldt int pointer argumenter, der refererer til et sted i hukommelsen (9). I Java findes pointers

ikke, men i stedet er alle objekter en reference til hukommelsen, hvor objektet findes. Dog er primitive typer som int ikke et objekt og kan derfor ikke refereres, hvorfor der her anvendes int arrays, der er objekter (10).

Metode	Beskrivelse
$\boldsymbol{void\ glGenVertexArrays(int\ n, int[]\ arrays, int\ offset)}$	Denne metode anvendes til at erklære et n antal VAO'er og indsætter deres ID'er i det andet givne parameter eventuelt med et offset.
$\boldsymbol{void\ glBindVertexArray(int\ array)}$	Denne metode anvendes til at "aktivere" en VAO, så de senere oprettede VBO'er vil blive forbundet til denne.
$\boldsymbol{void\ glGenBuffers(int\ n, int[]\ buffers, int\ offset)}$	Denne metode anvendes på samme måde som $glGenVertexArrays$ og bruges til at erklære et n antal VBO'er.

tabel 2: Metoder til erklæring af VAO'er og VBO'er

For at en buffer kan overføre data til GPU'en, skal en tilsvarende vertex attribut oprettes i shaderen, der ser ud som følgende:

$$layout\ (location = 0)\ in\ vec3\ position;$$

Nøgleordet in betyder "input" og indikerer, at en vertex attribut vil modtage data fra en buffer. $vec3$ betyder, at hver kørsel af shaderen vil indlæse tre værdier, x, y og z, fra bufferen, eftersom vec3 er en tredimensionel stedvektor. $vec3$ er derfor variabeltypen, mens $position$ er selve variablens navn. $layout\ (location = 0)$ kaldes for en "layout qualifier", og denne anvendes til at forbinde vertex attributten med bufferen gennem JOGL-applikationen. I linjen her er den såkaldte "identifier" lig 0, hvilket kan anvendes til at referere til attributten.

Når vertexdataene skal overføres fra hukommelsen til grafikhukommelsen, foregår det vha. metoderne i tabel 3. Først kaldes $glBindBuffer$ for at "aktivere" VBO'en, hvorefter den kan associeres med en Java-buffer gennem et kald til $glBufferData$.

Metode	Beskrivelse
$\boldsymbol{void\ glBindBuffer(int\ target, int\ buffer)}$	Denne metode anvendes til at "aktivere" en VBO, hvor $target$ er dens formål, hvilket i dette projekt kun er GL_ARRAY_BUFFER, der indikerer, at det skal sendes til vertex shaderen

	som en vertex attribut. $buffer$ er VBO'ens reference ID.
$\boldsymbol{void\ glBufferData(int\ target, int\ size, Buffer\ data, int\ usage)}$	Denne metode anvendes til at forbinde en OpenGL buffer til sin korresponderende Java-buffer. Igen bruges samme $target$ som til $glBindBuffer$. $size$ angiver hvor meget data, der skal overføres i bytes, hvorefter et Java bufferobjekt angives, hvilket er selve bufferen, der skal stå for dataoverførslen af vertexdataene. Til sidst angives, hvordan denne data vil blive tilgået. I dette projekt arbejdes kun med statisk data, der ikke vil blive ændret senere; hvorfor denne vil være GL_STATIC_DRAW.

tabel 3: Metoder til associering af data med buffer

Når vertexdataene skal sendes fra VBO'en til shaderen, foregår det ved et kald til $glBindBuffer$, der "aktiverer" bufferen efterfulgt af kald til de to metoder i tabel 4.

Metode	**Beskrivelse**
$\boldsymbol{void\ glVertexAttribPointer(int\ index, int\ size,}$ $\boldsymbol{int\ type, boolean\ normalized, int\ stride, long\ offset)}$	Denne metode anvendes til at forbinde den aktive buffer med attributten, der defineres af $index$, hvilket er den føromtalte "identifier". $size$ angiver hvor mange værdier, der skal hentes ind ad gangen. Da der arbejdes med vec3, er denne lig 3. type angiver hvilken datatype, der bliver overført gennem bufferen, i tilfælde af $float$, vil dette være GL_FLOAT. $stride$ og $offset$ er to typer offset henholdsvis målt i bytes og den anden målt i antal vertices. Disse er ofte lig 0.
$\boldsymbol{void\ glEnableVertexAttribPointer(int\ index)}$	Denne metode anvendes til at aktivere attributten defineret af $index$. Bufferens indhold vil blive overført til vertex attributten ved næste kald til $glDrawArrays$, hvorved objektet vil blive tegnet på det pågældende GLCanvas.

tabel 4: Metoder til associering af buffer med shader attribut

4.5.3 Uniforme variabler

Uniforme variabler er en type variabel, hvor indholdet er ens for alle shader iterationer. Disse kan bl.a. indeholde en models matrixtransformationer, der ikke ændrer sig fra vertex til vertex. Uniforme variabler erklæres med nøgleordet "uniform" efterfulgt af variabeltypen og navnet. I tilfælde af en transformationsmatrice er typen $mat4$, hvilket er en 4×4-matrice.

$$uniform\ mat4\ transform;$$

De metoder, der anvendes i dette projekt, kan ses i tabel 5, der kaldes i samme rækkefølge, som de fremgår af tabellen. Først hentes et reference-ID til den uniforme variabel, hvorefter de opgældende data sendes til variablen i shaderprogrammet fra JOGL-applikationen.

Metode	Beskrivelse
$\boldsymbol{int\ glGetUniformLocation(int\ program, String\ name)}$	Denne metode anvendes til at hente et reference-ID til en uniform variabel i shaderprogrammet. $program$ er selve programmet ID, mens $name$ er variablens navn.
$\boldsymbol{void\ glUniformMatrix4fv(int\ location, int\ count, boolean\ transpose, float[]\ values, int\ offset)}$	Denne metode anvendes til at overføre værdier fra et $float$ array til den uniforme variabel. $location$ er variablens ID. $count$ er antallet af matricer, der ændres, hvilket er én, når der ikke er tale om et array af matricer. $transpose$ er en matrixoperation, der ikke er relevant for dette projekt, og er derfor lig $false$. $values$ er et sekventielt array af matricens værdier, der skal overføres, og $offset$ er et eventuelt offset i arrayet.

tabel 5: Metoder til at overføre 4×4-matrix-transformationer til et shaderprogram

4.6 Hidden Surface Removal

Hvis ét objekt er foran et andet, forventes det også, at det foranliggende tegnes øverst, mens det bagvedliggende ikke er synligt. Teknikken til at opnå dette kaldes ”hidden surface removal” (HSR) (5). OpenGL tager sig af dybdetesten, hvis dette er aktiveret. Dybdetesten fungerer ved, at OpenGL sammenligner alle de interpolerede z-værdier for et fragment. Alt efter hvilken dybdefunktion, der er er valgt, vil OpenGL bestemme hvilket fragment, der vil blive vist i den pågældende pixel på skærmen (5).

4.7 Filformatet Wavefront (.obj)

3D-modelleringsværktøjer som f.eks. Blender[3] kan anvendes til at fremstille 3D-modeller. Blender kan eksportere vertexdataene, en model er bygget af til forskellige filformater, hvor Wavefront er f.eks. et letlæseligt format.

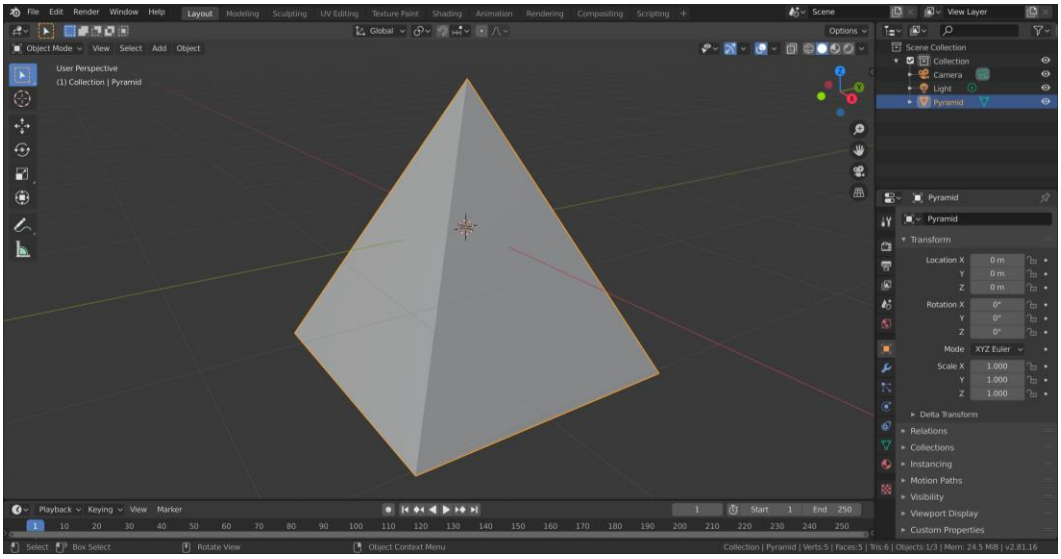
En Wavefront-fil er skrevet i klartekst og opbygget linjevis, hvor hver linje starter med et fortegn, der indikerer, hvilken type data, der findes på linjen. På hver linje er fortegn og dataelementer separeret af et mellemrum, hvilket gør den let at læse (5). Wavefront-filtypen understøtter adskillige typer data, hvoraf denne opgave kun vil kigge på datatyperne i tabel 6.

Datatype	**Fortegn**	**Syntaks**
Geometrisk data (vertexdata)	v	$v\ x\ y\ z$
Fladedata (Referencer til de vertexdata, modellens trekanter er opbygget af)	f	$f\ v_1\ v_2\ v_3$

tabel 6: Wavefront datatyper (5) anvendt i dette projekt

Hvis en pyramide, som det ses i figur 14, fremstilles i Blender, vil det ved eksport til Wavefront-filformatet give dataene, der er fremvist i figur 15.

[3] https://www.blender.org/



figur 14: Pyramide fremstillet i Blender

De to datatyper, geometrisk data og fladedata, er farvelagt i figur 15 hhv. i farverne blå og grøn. I figur 15 bemærkes også fortegnene #, *o* og *s*. # indikerer, at der er tale om en kommentar, og *o* er modellens navn (5). Da disse ikke er interessante, ignoreres de. *s* ignoreres også, da denne angiver, om modellen skal udglattes (5) vha. en metode, der ikke beskrives i dette projekt.

```
# Blender v2.81 (sub 16) OBJ File: 'Pyramid.blend'
# www.blender.org
o Pyramid
v 1.000000 -1.000000 -1.000000
v 1.000000 -1.000000 1.000000
v 0.000000 1.000000 0.000000
v -1.000000 -1.000000 -1.000000
v -1.000000 -1.000000 1.000000
s off
f 2 5 3
f 5 4 3
f 5 1 4
f 1 2 3
f 4 1 3
f 5 2 1
```

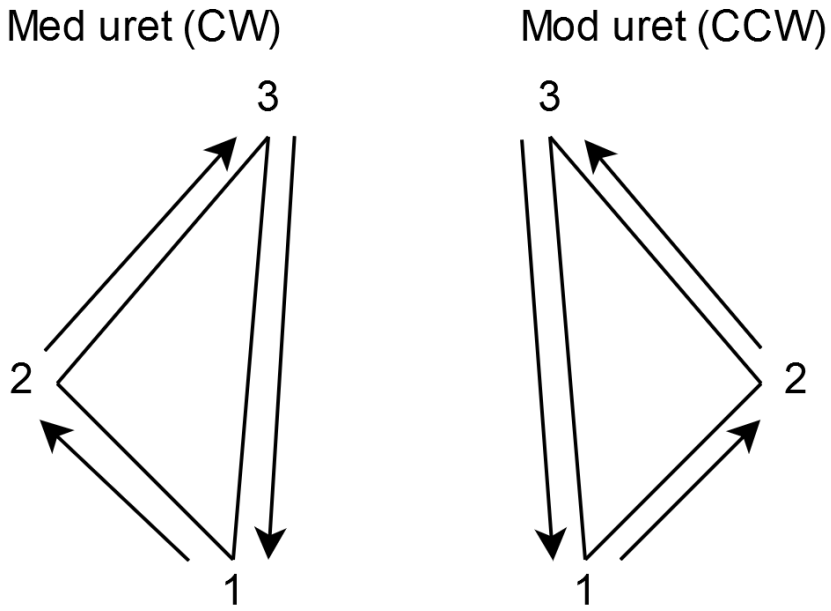
figur 15: Pyramiden eksporteret i Wavefront filformatet

Hver linje, der har fortegnet f, definerer en trekant i modellen. En pyramide består af seks trekanter. De fire sider er hver én trekant, og den kvadratiske grundflade består af to trekanter. Syntaksen følger den anviste i tabel 6, hvor x, y og

z er vertexkoordinater, og v_1, v_2 og v_3 er referencer til de tre vertices, den pågældende trekant består af. Referencerne er indekseret fra 1. Dermed får den første linje med fortegnet v referencen 1 og den næste 2 osv. (5).

4.8 Back-face culling

Når et objekt skal tegnes på en skærm, forventes det også, at siden, der kigges ind på, bliver vist, mens bagsiden skjules. Dette foretages allerede af HSR, men dybdetesten kan aflastes ved først at foretage det, der kaldes ”back-face culling”. Back-face culling er en teknik, der anvendes til at bestemme, om en flade vender mod kameraet eller væk fra kameraet. Alt efter, hvordan en model eksporteres, kan rækkefølgen, vertexdataene angives i, bruges til at vurdere hvilke flader, der vender mod eller væk fra kameraet. Denne rækkefølge kaldes ”winding order”, og der findes to typer; ”clockwise” og ”counterclockwise”, hvilket ses i figur 16.



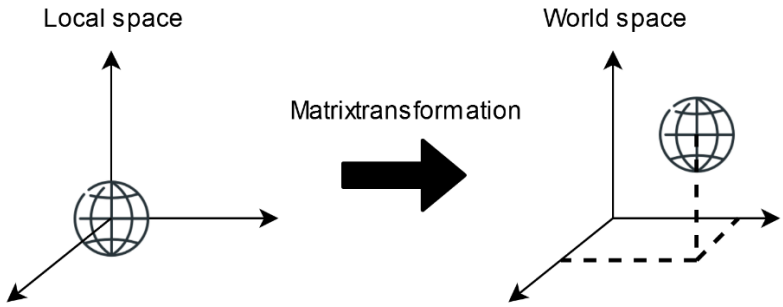
figur 16: tv: clockwise winding order; th: counterclockwise winding order

Hvis en model f.eks. angives med CW-winding, vil forsiden blive til en CCW-winding, hvis den roteres 180° om en akse, hvormed siden ikke vil blive tegnet.

4.9 Local space og world space

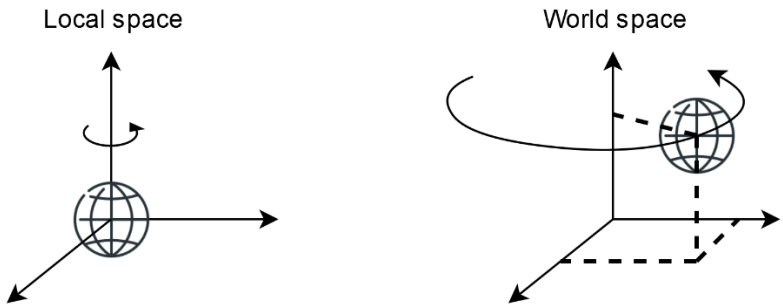
Når en 3D-model modelleres, placeres den passende i et koordinatsystem, så den på praktisk vis kan transformeres senere. F.eks. er det oplagt at placere en kugle, så den har centrum i origo, da kuglens centrum derved ikke vil flyttes, når kuglen skaleres.

Hvis kuglen ønskes placeret på et specifikt punkt i verden, kan den forskydes til dette punkt, og det er efter denne forskydning, at modellen siges at være i world space. Før forskydningen siges modellen at være i local space (5).



figur 17: Transformation fra local space til model space

Denne måde at fremstille en model gør det muligt at rotere en model omkring sig selv i local space, hvorefter den roterede model kan flyttes ud i world space. Hvis modellen først roteres i world space, vil den ikke rotere om sig selv, men omkring den pågældende akse, som sad modellen for enden af en pind på aksen med en længde tilsvarende den forskudte afstand (figur 18).

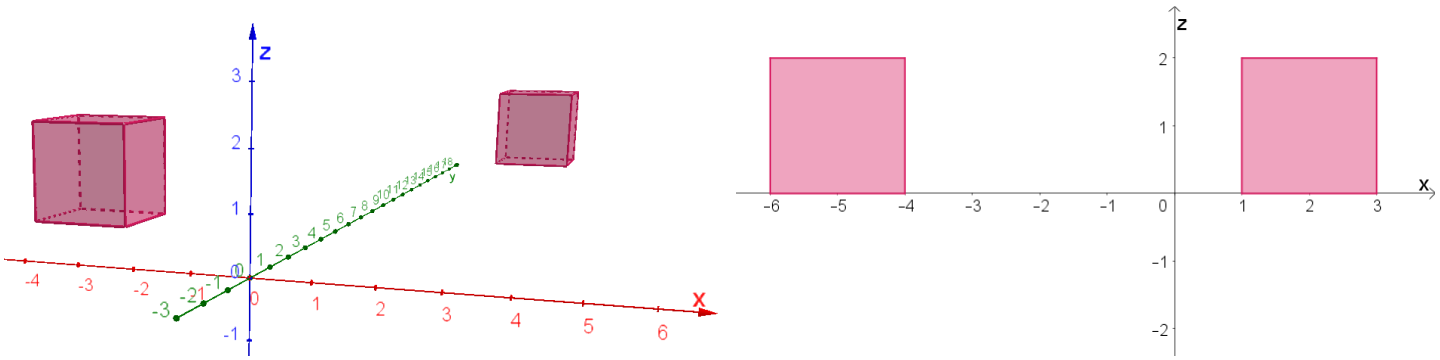


figur 18: Rotation i hhv. local space og world space

En matrix der indeholder en models transformationer inklusive forskydningen til world space, kaldes for en modelmatrix (5).

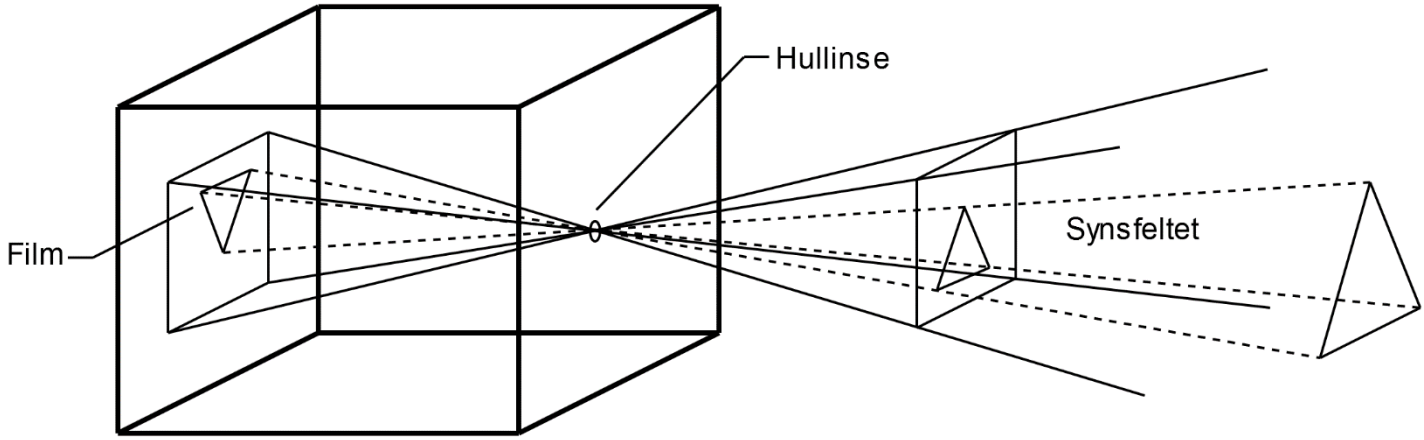
4.10 Perspektiv

Perspektivet giver et billede dybde. Jo længere noget er væk, des mindre fremstår det, og det ser ud til at forsvinde i et såkaldt forsvindingspunkt. Uden perspektiv vil to ens objekter fremstå lige store, selvom de er observeret på forskellig afstand i forhold til hinanden (figur 19). Perspektivet kan bruges til at give illusionen af 3D på et todimensionelt medie som en computerskærm.



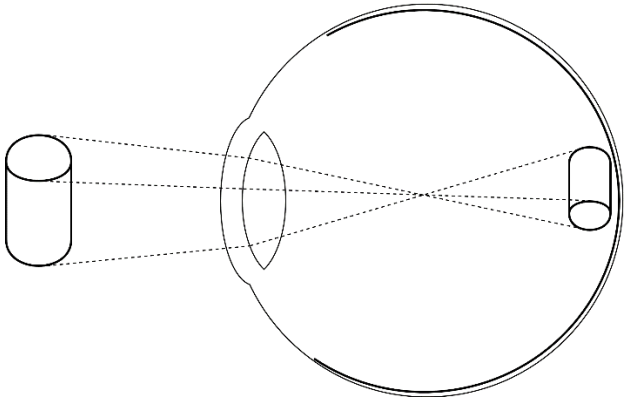
figur 19: To lige store kasser med forskellig afstand til xz-planet (tv.), fremstår lige store, når de projiceres på xz-planet (th.)

Hullinsen er et af de simpleste og første måder at fotografere på, hvor lyset gennem et meget lille hul projekteres på et lysfølsomt film, hvor billedet bliver optaget (11).



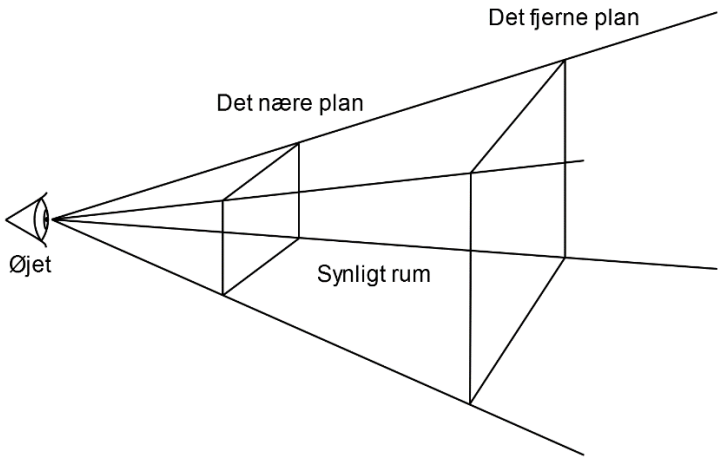
figur 20: Projektion vha. en hullinse

I essensen er det på samme måde, projektion i øjet foregår, hvilket er, hvad der ønskes efterlignet. Tilsvarende hullinsens film, bliver lyset projekteret på øjets lysfølsomme nethinde vha. en linse, der forkorter brændvidden, som det ses i figur 21.



figur 21: Projektion i øjet på nethinden

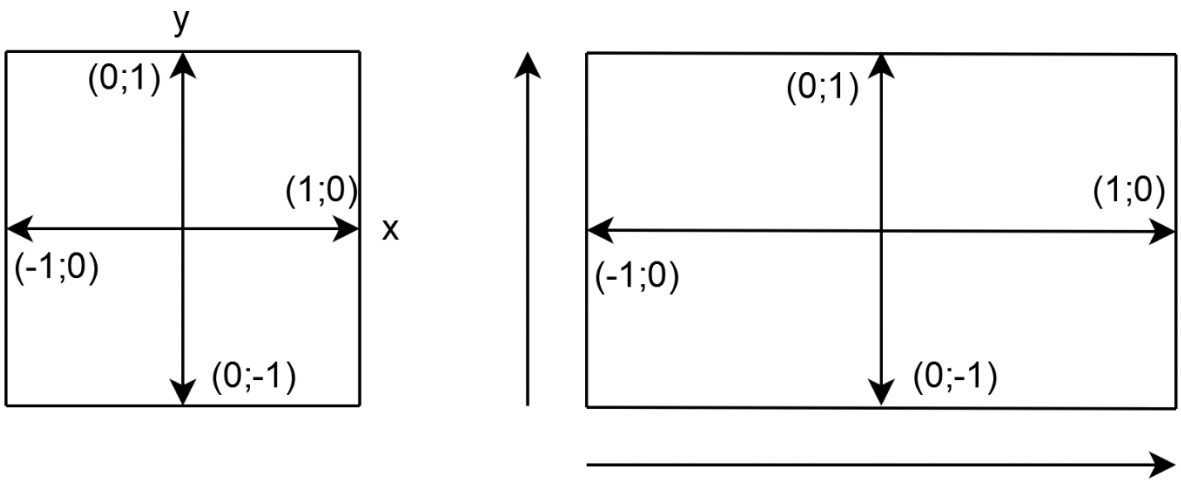
Hullinsen kan også simuleres ved at placere filmen foran kameraet i samme afstand, som da det var bagved hullinsen. Dertil tilføjes også et fjernt plan. Det fjerne plan definerer, hvor langt væk kameraet er i stand til at se. Årsagen til at anvende et fjernt plan er, at det ikke er nødvendigt at rendere objekter, der er så langt væk, at de knap udfylder en pixel. Kameraets pyramideformede synsfelt afgrænses altså af et nært plan og et fjernt plan, hvilket resulterer i en pyramidestub, der definerer det synlige rum også kaldet eye space, hvilket kan ses i figur 22 (11).



figur 22: Simulering af hullinse

Som standard inkluderer OpenGL et kamera, der er fast placeret i origo (0;0;0), og kigger langs den negative z-akse. Kameraet er orienteret således, at y-aksen er vertikal med den positive retning opadgående, og x-aksen er horisontal med den positive akse gående mod højre (5).

Uanset et vindues størrelse vil dets hjørner altid have samme koordinater i OpenGL. Koordinatsystemet vil dermed altid have origo i midten af vinduet, og den vinkelrette afstand til siderne fra origo er altid 1. Dette gælder uanset vinduets størrelse i pixels. Hvis vinduet f.eks. er bredere, end det er højt, vil koordinatsystemet blive udstrakt, som det ses i figur 23, hvilket OpenGL gør i baggrunden vha. en viewport transformation (12).



figur 23: Koordinatsystemet i OpenGL (Skærmkoordinater)

5 Opgavebesvarelse

Med udgangspunkt i teorien og metoderne præsenteret i de foregående afsnit, undersøges perspektivmatricens opbygning gennem en matematisk udledning. Derudover undersøges processen, hvorved man går fra vertexdata til repræsentation af en 3D-figur på en todimensionel flade. Dette gøres med henblik på senere at producere en softwareimplementation af dette samt vurdere det anvendte værktøj OpenGL.

5.1 Udledning af perspektivmatricen

Når der multipliceres med perspektivmatricen efterfulgt af den perspektivgivende division, er der tale om en afbildning af pyramidestubben, der definerer det synlige rum til den såkaldte NDC-terning eller NDC space (Normalized Device Coodinates) (11), som det ses i figur 24. Når punkterne i det synlige rum er tilordnet NDC-terningen, vil en retvinklet projicering på xy-koordinatsystemet give den todimensionelle repræsentation af de tredimensionelle figurer i det synlige rum. Dermed er punkterne i NDC-terningen egentligt allerede todimensionelle. Z-koordinatet er punkternes indbyrdes relative afstand og bruges af OpenGL til at vurdere, hvad der ligger forrest og skal tegnes, samt hvad der ligger bagerst, og dermed ikke skal tegnes, da det ikke kan ses.

NDC-terningen i OpenGL er af design placeret i et venstrehåndskoordinatsystem, hvorved punkter tættere på kameraet får et mindre z-koordinat end punkter længere fra kameraet, når de tilordnes NDC space. Dette er også årsagen til, at der i dette projekt anvendes dybdefunktionen GL_LEQUAL (mindre eller lig), hvormed OpenGL for hver pixel vil tegne fragmenterne med de mindste interpolerede z-værdier.

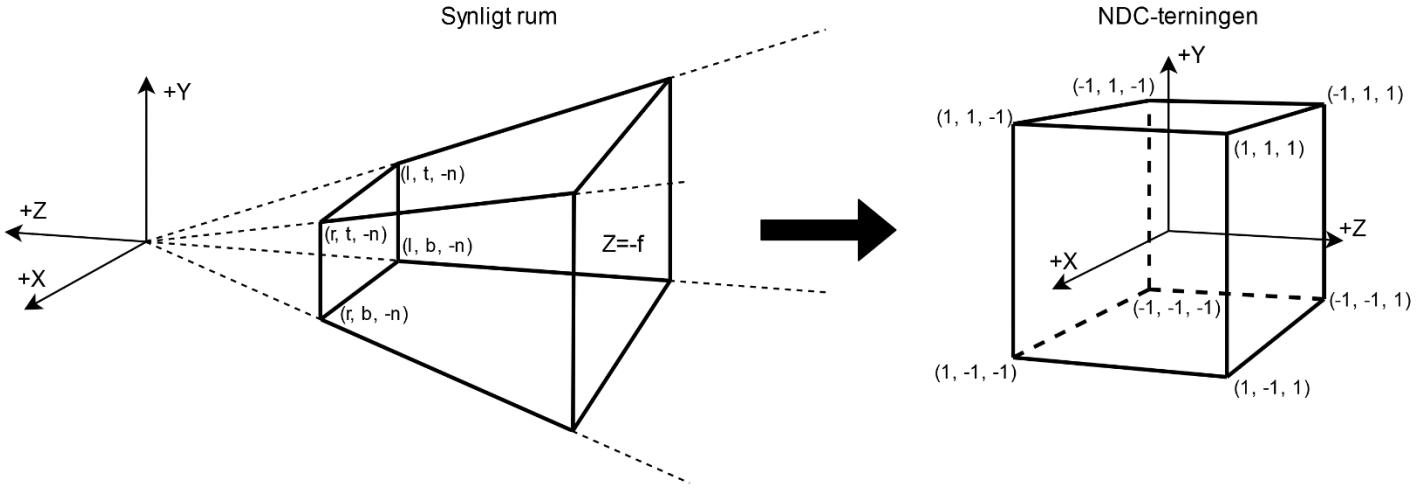
Til udledningen af perspektivmatricen er det nødvendigt at antage en række kendte værdier, matricen kan fremstilles af. Disse variabler fremgår af tabel 7 og ses grafisk placeret i figur 24.

Variabel	Beskrivelse
l	x-koordinatet af det nære plans venstre side
r	x-koordinatet af det nære plans højre side
t	y-koordinatet af det nære plans top
b	y-koordinatet af det nære plans bund
n	z-koordinatet af det nære plan
f	z-koordinatet af det fjerne plan

tabel 7: Pyramidestubbens antaget kendte værdier

Det er generelt anvendt, at nærplanet og fjernplanet kan opgives i positive værdier, hvorfor Z-koordinaterne til det synlige rums nær- og fjernplan er negeret (figur 24).

Perspektivprojicering



figur 24: Afbildning af det synligt rum til NDC space

Perspektiv-dybdematricen, som den ser ud til brug med OpenGL, ses defineret i ligning (16).

$$M_p = \begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{-(f+n)}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix} \tag{16}$$

Perspektivmatricen i sig selv tilordner ikke 3D-figuren NDC space, når denne multipliceres på punkterne. I stedet bliver punkterne tilordnet det såkaldte clipping space (ligning (17)). Dette kaldes clipping space, da det er her, OpenGL i baggrunden fjerner alt, der ligger udenfor det synlige felt, og tilpasser primitiver, der skærer det synlige felts sider.

$$M_p \cdot \begin{bmatrix} eye_x \\ eye_y \\ eye_z \\ 1 \end{bmatrix} = \begin{bmatrix} clip_x \\ clip_y \\ clip_z \\ clip_w \end{bmatrix} \tag{17}$$

Det er ved den homogene normalisering, som er beskrevet i afsnit 4.1.10, at vertexdataene får tilført sit respektive perspektiv. Divisionen med w-koordinatet er det sidste skridt, der foretages for at opnå NDC-koordinater (ligning (18)), og kaldes ”den perspektivgivende division”.

$$\begin{bmatrix} clip_x/clip_w \\ clip_y/clip_w \\ clip_z/clip_w \\ clip_w/clip_w \end{bmatrix} = \begin{bmatrix} NDC_x \\ NDC_y \\ NDC_z \\ 1 \end{bmatrix} \tag{18}$$

Multiplikationen med perspektivmatricen og det individuelle punkt kan opstilles som vist i ligning (19).

$$\begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{-(f+n)}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix} \cdot \begin{bmatrix} eye_x \\ eye_y \\ eye_z \\ 1 \end{bmatrix} = \begin{bmatrix} clip_x \\ clip_y \\ clip_z \\ clip_w \end{bmatrix} \tag{19}$$

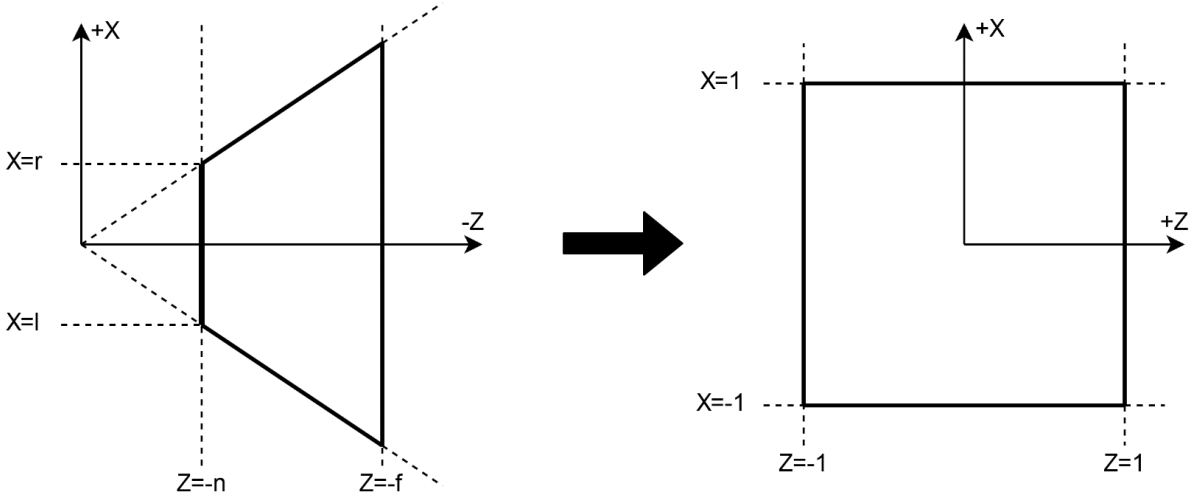
Ligning (19) kan opdeles i tre individuelle ligninger for hvert clip-koordinat. Selve multiplikationen ses udført i Falks skema i figur 25.

				eye_x
				eye_y
				eye_z
				1
$\frac{2n}{r-l}$	0	$\frac{r+l}{r-l}$	0	$\frac{2n}{r-l} \cdot eye_x + 0 \cdot eye_y + \frac{r+l}{r-l} \cdot eye_z = \frac{2n}{r-l} \cdot eye_x + \frac{r+l}{r-l} \cdot eye_z$
0	$\frac{2n}{t-b}$	$\frac{t+b}{t-b}$	0	$0 \cdot eye_x + \frac{2n}{t-b} \cdot eye_y + \frac{t+b}{t-b} \cdot eye_z + 1 \cdot 0 = \frac{2n}{t-b} \cdot eye_y + \frac{t+b}{t-b} \cdot eye_z$
0	0	$\frac{-(f+n)}{f-n}$	$\frac{-2fn}{f-n}$	$0 \cdot eye_x + 0 \cdot eye_y + \frac{-(f+n)}{f-n} \cdot eye_z + \frac{-2fn}{f-n} \cdot 1 = \frac{-(f+n)}{f-n} \cdot eye_z + \frac{-2fn}{f-n}$
0	0	-1	0	$0 \cdot eye_x + 0 \cdot eye_z - 1 \cdot eye_z + 0 \cdot 1 = -eye_z$

figur 25: Perspektivmatricen anvendes på et punkt vha. Falks skema.

For at nå frem til en ligning for hvert NDC-koordinat divideres der med w-koordinatet (ligning (20)), der er lig $-eye_z$, som vist i figur 25.

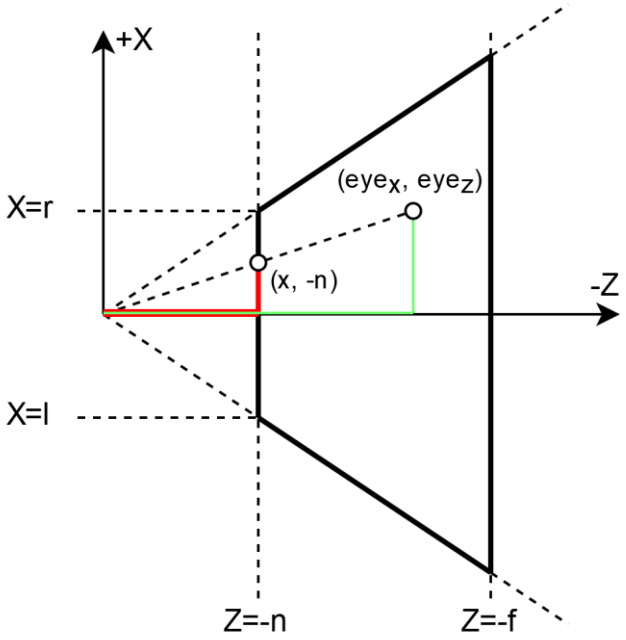
$$\begin{bmatrix} NDC_x \\ NDC_y \\ NDC_z \\ 1 \end{bmatrix} = \begin{bmatrix} clip_x/clip_w \\ clip_y/clip_w \\ clip_z/clip_w \\ clip_w/clip_w \end{bmatrix} = \begin{bmatrix} \dfrac{\frac{2n}{r-l}\cdot eye_x + \frac{r+l}{r-l}\cdot eye_z}{-eye_z} \\ \dfrac{\frac{2n}{t-b}\cdot eye_y + \frac{t+b}{t-b}\cdot eye_z}{-eye_z} \\ \dfrac{\frac{-(f+n)}{f-n}\cdot eye_z + \frac{-2fn}{f-n}}{-eye_z} \\ 1 \end{bmatrix} \quad (20)$$



figur 26: Transformation fra synligt rum til NDC-rum

5.1.1 Udledning af X- og Y-koordinatet

Ved at projicere punktet på nærplanet, hvis forhold til NDC-terningen kendes, som det ses i figur 27, kan dets tilsvarende x-koordinat i NDC-terningen let bestemmes.



figur 27: Tværsnit af det synlige rum, xz-plan

Da den røde og grønne trekant i figur 27 er kongruente, kan forholdet i ligning (21) opstilles, hvorefter x-koordinatet kan isoleres.

$$\frac{x}{-n} = \frac{eye_x}{eye_z} \tag{21}$$

⇳ *Ligningen løses for x vha. CAS-værktøjet WordMat.*

$$x = \frac{n \cdot eye_x}{-eye_z} \tag{22}$$

Afstanden mellem koordinatet x og l kan opstilles som en procentdel af nærplanets bredde, hvilket er en del af intervallet $[0; 1]$, som ligning (23) viser.

$$\frac{x - l}{r - l} \in [0; 1] \tag{23}$$

Et tilsvarende x-koordinat i NDC-terningen kan nu tilordnes (ligning (25)), da dennes interval kendes (ligning (24)).

$$NDC_x \in [-1; 1] \tag{24}$$

$$NDC_x = \frac{x - l}{r - l} \cdot 2 - 1 \tag{25}$$

Udtrykket fundet for x i ligning (22) kan nu indsættes i udtrykket for x-koordinatet i NDC-terningen (ligning (25)).

$$NDC_x = \frac{\frac{n \cdot eye_x}{-eye_z} - l}{r - l} \cdot 2 - 1$$

$$\frac{\frac{2 \cdot n \cdot eye_x}{-eye_z}}{r - l} - \frac{2l}{r - l} - 1 = \frac{2 \cdot n \cdot eye_x}{(r - l) \cdot (-eye_z)} - \frac{2l}{r - l} - \frac{r - l}{r - l} = \frac{2 \cdot n \cdot eye_x}{(r - l) \cdot (-eye_z)} - \frac{r + l}{r - l}$$

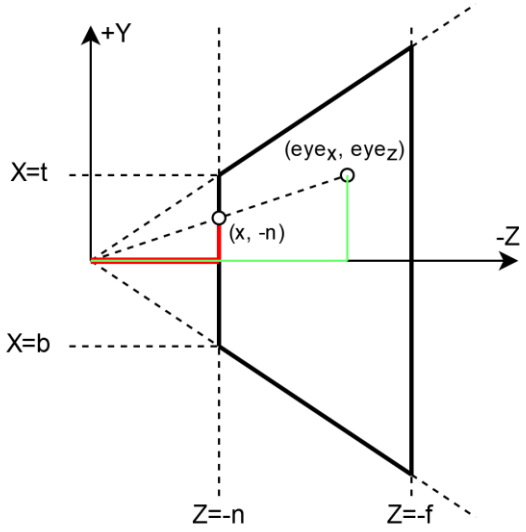
$$= \frac{2 \cdot n \cdot eye_x}{(r - l) \cdot (-eye_z)} + \frac{(r + l) \cdot eye_z}{(r - l) \cdot (-eye_z)} = \frac{2 \cdot n \cdot eye_x + (r + l) \cdot eye_z}{(r - l) \cdot (-eye_z)} = \frac{\frac{2n}{r - l} \cdot eye_x + \frac{r + l}{r - l} \cdot eye_z}{-eye_z}$$

Dette giver udtrykket i ligning (26), hvilket er det, som ses i x-koordinatet i ligning (20), der var ønsket redegjort.

$$NDC_x = \frac{\frac{2n}{r - l} \cdot eye_x + \frac{r + l}{r - l} \cdot eye_z}{-eye_z} \tag{26}$$

På samme måde kan udtrykket for NDC_y findes til udtrykket, der ses i ligning (27), ud fra tværsnittet af det synlige rum i figur 28.

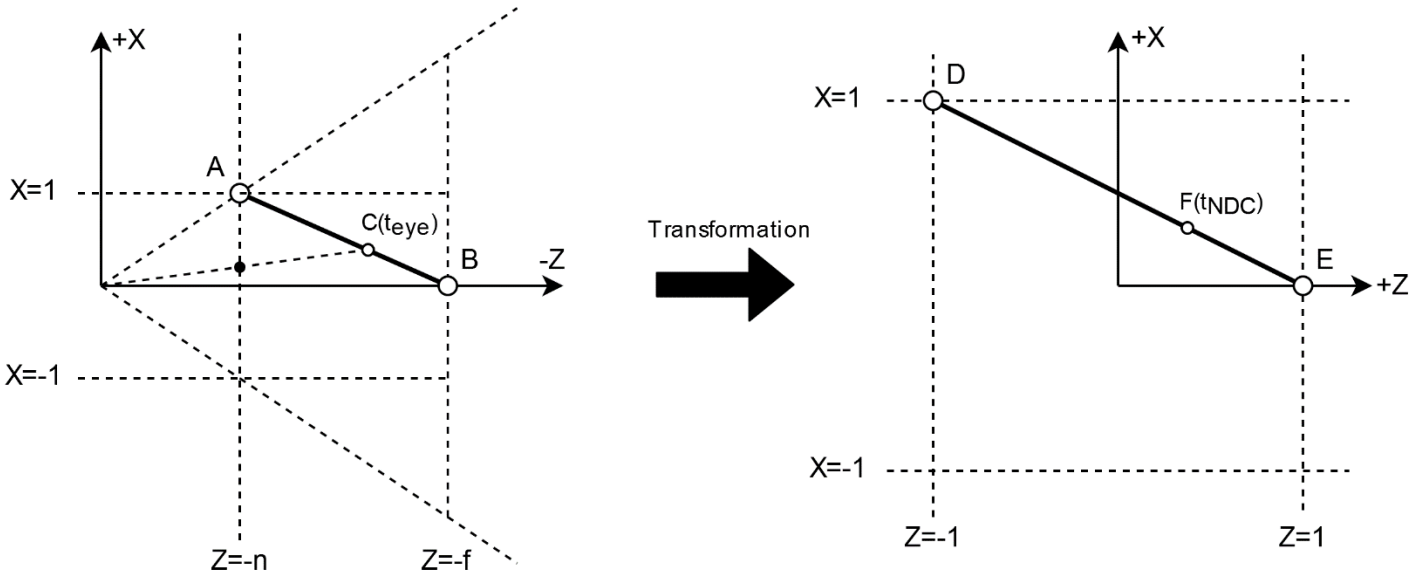
$$NDC_y = \frac{\frac{2n}{t - b} \cdot eye_y + \frac{t + b}{t - b} \cdot eye_z}{-eye_z} \tag{27}$$



figur 28: Tværsnit af det synlige rum, yz-plan

5.1.2 Udledning af Z-koordinatet

Det er mere komplekst at redegøre for ligningen for z-koordinatet. Koordinatet kan udledes ved at indlægge en linje, hvis endepunkters transformation er kendt. Linjen indlægges i et synligt rum, hvor x- og y-koordinatet allerede er tilordnet $[1; -1]$. Et vilkårligt punkt kan indlægges på linjen, som vist i figur 29. Det ønskede resultat er at finde en F_z som funktion af C_z, $F_z(C_z)$.



figur 29: z-koordinatets transformation fra synligt rum til NDC space.

For at overskueliggøre de kendte værdier i figur 29, er disse indlagt i tabel 8.

Eye space (Synligt rum)		**NDC Space**	
$\boldsymbol{A_x = 1}$	$B_x = 0$	$D_x = 1$	$E_x = 0$
$\boldsymbol{A_z = -n}$	$B_z = -f$	$D_z = -1$	$E_z = 1$
$\boldsymbol{C(t_{eye}) = A + t_{eye} \cdot (B - A)}$		$F(t_{NDC}) = D + t_{NDC} \cdot (E - D)$	

tabel 8: Kendte værdier og indlagt linjes parameterfremstilling for hhv. synligt rum og NDC space

Det tilordnede x-koordinat af det vilkårlige punkt kan opskrives som det tidligere fundne udtryk i ligning (22) fra udledningen af NDC_x. Dette ses i ligning (28), hvor det vilkårlige punkt i det synlige rum her er $C(t_{eye})$, og x er x-koordinat til det vilkårlige punkt på den transformerede linje.

$$F_x(t_{NDC}) = \frac{n \cdot C_x(t_{eye})}{-C_z(t_{eye})} \tag{28}$$

Både $F(t_{NDC})$ og $C(t_{eye})$ kan substitueres med de fulde ligninger (ligning (17)).

$$D_x + t_{NDC} \cdot (E_x - D_x) = \frac{n \cdot \left(A_x + t_{eye} \cdot (B_x - A_x)\right)}{-\left(A_z + t_{eye} \cdot (B_z - A_z)\right)} \tag{29}$$

Udtrykket i ligning (29) kan forsimples ved at substituere de kendte variabler med sine værdier fra tabel 8, hvilket ses i ligning (30). Et udtryk for t_{NDC} kan hermed findes ved at isolere denne (ligning (31)).

$$1 + t_{NDC} \cdot (0 - 1) = \frac{n \cdot \left(1 + t_{eye} \cdot (0 - 1)\right)}{-\left(-n + t_{eye} \cdot \left(-f - (-n)\right)\right)} \tag{30}$$

⇳

$$1 - t_{NDC} = \frac{n - t_{eye} \cdot n}{n - t_{eye} \cdot (n - f)}$$

⇕ *Ligningen løses for t_{NDC} vha. CAS-værktøjet WordMat.*

$$t_{NDC} = \frac{f \cdot t_{eye}}{n - t_{eye} \cdot (n - f)} \tag{31}$$

Dette udtryk for t_{NDC} kan indsættes i parameterfremstillingen for linjen i NDC space, hvormed det vilkårlige punkt i NDC space bliver en funktion af t_{eye}, der er den eneste ukendte, som det ses af ligning (32).

$$F_z(t_{eye}) = D_z + \frac{f \cdot t_{eye}}{n - t_{eye} \cdot (n - f)} \cdot (E_z - D_z) \tag{32}$$

Udtrykket for $F_z(t_{eye})$ i ligning (32) kan forsimples ved igen at substituere de kendte variabler med sine værdier fra tabel 8, hvilket ses i ligning (33).

$$F_z(t_{eye}) = -1 + \frac{f \cdot t_{eye}}{n - t_{eye} \cdot (n - f)} \cdot \big(1 - (-1)\big) \tag{33}$$

Gennem en omskrivning, der ses i ligning (34), findes frem til udtrykket for $F_z(t_{eye})$ i ligning (35).

$$\begin{aligned} F_z(t_{eye}) &= \frac{-\big(n - t_{eye} \cdot (n - f)\big)}{n - t_{eye} \cdot (n - f)} + \frac{f \cdot t_{eye}}{n - t_{eye} \cdot (n - f)} \cdot 2 \\ &= \frac{-n + t_{eye} \cdot (n - f)}{n - t_{eye} \cdot (n - f)} + \frac{2 \cdot f \cdot t_{eye}}{n - t_{eye} \cdot (n - f)} = \frac{2 \cdot f \cdot t_{eye} - n + t_{eye} \cdot (n - f)}{n - t_{eye} \cdot (n - f)} \\ &= \frac{2 \cdot f \cdot t_{eye} - n + t_{eye} \cdot n - t_{eye} \cdot f}{n - t_{eye} \cdot (n - f)} = \frac{f \cdot t_{eye} + t_{eye} \cdot n - n}{n - t_{eye} \cdot (n - f)} = \frac{t_{eye} \cdot (f + n) - n}{n - t_{eye} \cdot (n - f)} \\ &= \frac{t_{eye} \cdot (f + n) - n}{n + t_{eye} \cdot (f - n)} \end{aligned} \tag{34}$$

$$F_z(t_{eye}) = \frac{t_{eye} \cdot (f+n) - n}{t_{eye} \cdot (f-n) + n} \tag{35}$$

Ved at isolere t_{eye} i parameterfremstillingen af linjen i det synlige rum (ligning (36)), kan en ønsket funktion for F_z som funktion af C_z bestemmes ved at indsætte udtrykket for t_{eye}, hvor C_z er den eneste ukendte i ligning (35).

$$C_z = A_z + t_{eye} \cdot (B_z - A_z) \tag{36}$$

⇳ *Ligningen løses for t_{eye} vha. CAS-værktøjet WordMat.*

$$t_{eye} = \frac{C_z - A_z}{B_z - A_z} \tag{37}$$

$$F_z\left(\frac{C_z - A_z}{B_z - A_z}\right) = \frac{\frac{C_z - A_z}{B_z - A_z} \cdot (f+n) - n}{\frac{C_z - A_z}{B_z - A_z} \cdot (f-n) + n}$$

Ved at substituere de kendte værdier, kan ligningen forsimples, hvilket ses i ligning (38).

$$F_z(C_z) = \frac{\frac{C_z - (-n)}{-f - (-n)} \cdot (f+n) - n}{\frac{C_z - (-n)}{-f - (-n)} \cdot (f-n) + n} = \frac{\frac{C_z + n}{n - f} \cdot (f+n) - n}{\frac{C_z + n}{n - f} \cdot (f-n) + n} \tag{38}$$

Efter en omskrivning af ligningen findes et udtryk, der ligner det ønskede:

$$F_z(C_z) = \frac{\frac{C_z \cdot f + C_z \cdot n + n \cdot f + n^2}{n-f} - n}{\frac{C_z + n}{n-f} \cdot (f-n) + n} = \frac{\frac{C_z \cdot f + C_z \cdot n + n \cdot f + n^2}{n-f} - \frac{n \cdot (n-f)}{n-f}}{\frac{C_z + n}{n-f} \cdot (f-n) + n}$$

$$= \frac{\frac{C_z \cdot f + C_z \cdot n + n \cdot f + n^2}{n-f} - \frac{n^2 - n \cdot f}{n-f}}{\frac{C_z + n}{n-f} \cdot (f-n) + n} = \frac{\frac{C_z \cdot f + C_z \cdot n + n \cdot f + n^2 - n^2 + n \cdot f}{n-f}}{\frac{C_z + n}{n-f} \cdot (f-n) + n}$$

$$= \frac{\frac{C_z \cdot f + C_z \cdot n + 2 \cdot n \cdot f}{n-f}}{\frac{C_z + n}{n-f} \cdot (f-n) + n} = \frac{\frac{C_z \cdot f + C_z \cdot n + 2 \cdot n \cdot f}{n-f}}{\frac{C_z \cdot f - C_z \cdot n + n \cdot f - n^2}{n-f} + n} = \frac{\frac{C_z \cdot f + C_z \cdot n + 2 \cdot n \cdot f}{n-f}}{\frac{C_z \cdot f - C_z \cdot n + n \cdot f - n^2}{n-f} + n}$$

$$= \frac{\frac{C_z \cdot f + C_z \cdot n + 2 \cdot n \cdot f}{n-f}}{\frac{C_z \cdot f - C_z \cdot n + n \cdot f - n^2}{n-f} + \frac{n \cdot (n-f)}{n-f}} = \frac{\frac{C_z \cdot f + C_z \cdot n + 2 \cdot n \cdot f}{n-f}}{\frac{C_z \cdot f - C_z \cdot n + n \cdot f - n^2}{n-f} + \frac{n^2 - n \cdot f}{n-f}}$$

$$= \frac{\frac{C_z \cdot f + C_z \cdot n + 2 \cdot n \cdot f}{n-f}}{\frac{C_z \cdot f - C_z \cdot n + n \cdot f - n^2}{n-f} + \frac{n^2 - n \cdot f}{n-f}} = \frac{\frac{C_z \cdot f + C_z \cdot n + 2 \cdot n \cdot f}{n-f}}{\frac{C_z \cdot f - C_z \cdot n + n \cdot f - n^2 + n^2 - n \cdot f}{n-f}}$$

$$= \frac{\frac{C_z \cdot f + C_z \cdot n + 2 \cdot n \cdot f}{n-f}}{\frac{C_z \cdot f - C_z \cdot n}{n-f}} = \frac{\frac{C_z \cdot f + C_z \cdot n + 2 \cdot n \cdot f}{n-f}}{\frac{C_z \cdot (f-n)}{n-f}} = \frac{C_z \cdot f + C_z \cdot n + 2 \cdot n \cdot f}{C_z \cdot (f-n)}$$

$$= \frac{C_z \cdot (f+n) + 2 \cdot n \cdot f}{f-n} = \frac{\frac{f+n}{f-n} \cdot C_z + \frac{2fn}{f-n}}{C_z}$$

Ved at multiplicere med det komplicerede ét-tal $\frac{-1}{-1}$ opnås det ønskede udtryk, der ses i ligning (39).

$$\frac{\frac{f+n}{f-n} \cdot C_z + \frac{2fn}{f-n}}{C_z} \cdot \frac{-1}{-1} = \frac{-\frac{f+n}{f-n} \cdot C_z - \frac{2fn}{f-n}}{-C_z}$$

$$F_z(C_z) = \frac{-\frac{f+n}{f-n} \cdot C_z - \frac{2fn}{f-n}}{-C_z} \tag{39}$$

Det skal huskes, C_z er koordinatet i det synlige rum, og at F_z er koordinatet C_z tilordnet NDC space, hvorfor ligning (39) kan skrives som ligning (40), hvilket er det der ses for z-koordinatet i ligning (20), der var ønsket udledt.

$$NDC_z = \frac{-\frac{f+n}{f-n} \cdot eye_z - \frac{2fn}{f-n}}{-eye_z} \tag{40}$$

Hermed er udledningen af perspektivmatricen udført, da de tre ønskede ligninger er fundet.

$$\begin{bmatrix} NDC_x \\ NDC_y \\ NDC_z \\ 1 \end{bmatrix} = \begin{bmatrix} clip_x/clip_w \\ clip_y/clip_w \\ clip_z/clip_w \\ clip_w/clip_w \end{bmatrix} = \begin{bmatrix} \frac{\frac{2n}{r-l} \cdot eye_x + \frac{r+l}{r-l} \cdot eye_z}{-eye_z} \\ \frac{\frac{2n}{t-b} \cdot eye_y + \frac{t+b}{t-b} \cdot eye_z}{-eye_z} \\ \frac{\frac{-(f+n)}{f-n} \cdot eye_z + \frac{-2fn}{f-n}}{-eye_z} \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{2n}{r-l} \cdot \frac{eye_x}{-eye_z} + \frac{r+l}{r-l} \\ \frac{2n}{t-b} \cdot \frac{eye_y}{-eye_z} + \frac{t+b}{t-b} \\ \frac{-2fn}{f-n} \cdot \frac{1}{-eye_z} + \frac{f+n}{f-n} \\ 1 \end{bmatrix} \tag{41}$$

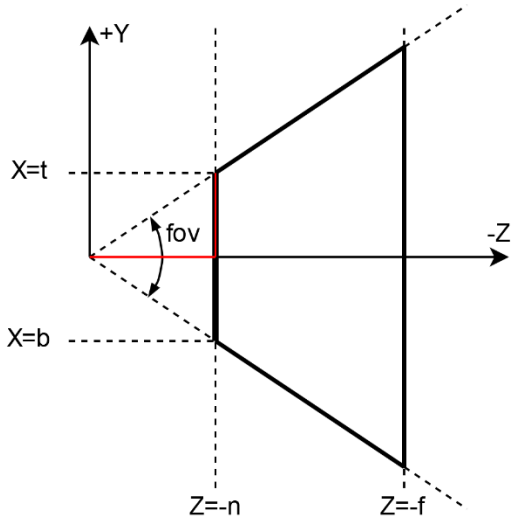
Når eye_z tilnærmer sig ∞, altså når et objekt er infinitesimalt langt væk fra kameraet, vil første led af hver række i ligning (41) tilnærme sig 0 som følge af divisionen med $-eye_z$. Dette efterlader konstante værdier for x, y og z. Dermed gælder, at forsvindingspunktet ligger i $\left(\frac{r+l}{r-l}; \frac{t+b}{t-b}\right)$. NDC-terningen kan altså forskydes ved at ændre på værdierne i indgang $(1;3)$, $(2;3)$ og $(3;3)$ af perspektivmatricen. Udtrykkene i indgang $(1;1)$, $(2;2)$ projicerer punkterne i eye space ned på det nære plan samt skalerer denne til NDC-terningen, ligesom indgang $(3;4)$ skalerer punkternes z-værdi. Indgang $(4;3)$ er grunden til, at den perspektivgivende division med $-eye_z$ sker ved normalisering, da denne efterlader $-eye_z$ i w-koordinatet efter multiplikation med en vertex.

$$M_p = \begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{-(f+n)}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

Oftest ligger det synlige felt således, at y-aksen og x-aksen er dets symmetriakser. Dette giver, at $r = -l$ og $t = -b$, hvormed indgang (1; 3) og $(2;3)$ begge giver 0 som følge af additionen i tælleren.

$$\begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{-(f+n)}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix} \Rightarrow \begin{bmatrix} \frac{2n}{r-l} & 0 & 0 & 0 \\ 0 & \frac{2n}{t-b} & 0 & 0 \\ 0 & 0 & \frac{-(f+n)}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

Når en perspektivmatrice oprettes, er det ofte anvendt kun at opgive en vertikal synsvidde, afstanden til det nære og fjerne plan samt et billedformat (Eng. aspect ratio). Billedformatet er forholdet mellem programvinduets bredde i pixels og dets højde i pixels. For at muliggøre dette skal perspektivmatricens indgang $(1;1)$ og $(2;2)$ omskrives.



figur 30: Tværsnit af det synlige rum med indlagt synsvidde

Variablen n i udtrykket fra perspektivmatricens indgang $(2;2)$, der ses i ligning (42), kan omskrives vha. tangens, der ses defineret i ligning (43) ved at tage udgangspunkt i den røde trekant i figur 30.

$$\frac{2n}{t-b} \tag{42}$$

$$\tan(\alpha) = \frac{modstående\ katete}{hosliggende\ katete} \tag{43}$$

I den røde trekant, hvor vinklen er tilsvarende den halve vertikale synsvidde, er den modstående katete tilsvarende den halve højde af firkanten, der definerer det nære plan. Den hosliggende katete er tilsvarende den vinkelrette afstand fra origo til det nære plan, hvilket er n. Derudfra kan ligning (44) opstilles, hvorefter n kan isoleres.

$$\tan\left(\frac{fov}{2}\right) = \frac{\frac{1}{2}\cdot(t-b)}{n} \tag{44}$$

⇕ *Ligningen løses for n vha. CAS-værktøjet WordMat.*

$$n = \frac{t-b}{2\cdot\tan\left(\frac{fov}{2}\right)} \tag{45}$$

Det fundne udtryk for n fra ligning (45) kan nu indsættes i udtrykket for indgang $(2;2)$, som det ses i ligning (46).

$$\frac{2n}{t-b} = \frac{2\cdot\left(\frac{t-b}{2\cdot\tan\left(\frac{fov}{2}\right)}\right)}{t-b} \tag{46}$$

Ligning (46) kan videre reduceres til det endelige udtryk opnås, hvilket ses i ligning (47).

$$\frac{2\cdot\left(\frac{t-b}{2\cdot\tan\left(\frac{fov}{2}\right)}\right)}{t-b} = \frac{\frac{t-b}{\tan\left(\frac{fov}{2}\right)}}{t-b} = \frac{\left(\frac{t-b}{t-b}\right)}{\tan\left(\frac{fov}{2}\right)} = \frac{1}{\tan\left(\frac{fov}{2}\right)} \tag{47}$$

Variablen n i udtrykket fra indgang $(1;1)$, der ses i ligning (48), kan på samme måde substitueres med udtrykket for n, der blev udledt i ligning (45).

$$\frac{2n}{r-l} \tag{48}$$

Efter substitueringen i ligning (49) kan udtrykket videre reduceres, som det fremgår af samme ligning.

$$\frac{2 \cdot \left(\frac{t-b}{2 \cdot \tan\left(\frac{fov}{2}\right)} \right)}{r-l} = \frac{\frac{t-b}{\tan\left(\frac{fov}{2}\right)}}{r-l} = \frac{\frac{t-b}{r-l}}{\tan\left(\frac{fov}{2}\right)} \tag{49}$$

I tælleren af det endelige udtryk i ligning (49) bemærkes det, at det er intet andet end den reciprokke værdi af billedformatet, der ses i ligning (50).

$$aspectRatio = \frac{h}{b} = \frac{r-l}{t-b} \tag{50}$$

Dermed kan tælleren i ligning (49) substitueres med udtrykket i ligning (51), hvilket giver et endeligt udtryk for indgang $(1; 1)$ i ligning (52).

$$\frac{t-b}{r-l} = \frac{1}{aspectRatio} \tag{51}$$

$$\frac{\frac{1}{aspectRatio}}{\tan\left(\frac{fov}{2}\right)} = \frac{1}{\tan\left(\frac{fov}{2}\right) \cdot aspectRatio} \tag{52}$$

Således udledes perspektivmatricen i ligning (53), der vil blive anvendt i følgende beregningseksempel.

$$\begin{bmatrix} \frac{2n}{r-l} & 0 & 0 & 0 \\ 0 & \frac{2n}{t-b} & 0 & 0 \\ 0 & 0 & \frac{-(f+n)}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix} \Rightarrow \begin{bmatrix} \frac{1}{\tan\left(\frac{fov}{2}\right) \cdot aspectRatio} & 0 & 0 & 0 \\ 0 & \frac{1}{\tan\left(\frac{fov}{2}\right)} & 0 & 0 \\ 0 & 0 & \frac{-(f+n)}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix} \quad (53)$$

5.2 Matematisk anvendelse af perspektivmatricen

Perspektivmatricens anvendelse kan vises gennem et matematisk eksempel, hvor der regnes på en pyramide, der i sidste ende vil give NDC-koordinater, hvoraf en 2D-repræsentation af pyramiden bør kunne indtegnes i GeoGebra.

5.2.1 Eksemplets forudsætninger

Før eksemplet kan beregnes, er det nødvendigt at oplyse visse værdier, der skal lægge til grund for eksemplet. De anvendte værdier fremgår af tabel 9.

Værdi	**Beskrivelse**
$\boldsymbol{fov = 60°}$	Den vertikale synsvidde
$\boldsymbol{n = 0,1}$	Det nære plan
$\boldsymbol{f = 1000}$	Det fjerne plan
$\boldsymbol{P_{Pyr} = (0; 2; 0)}$	Pyramidens placering
$\boldsymbol{\alpha_y = -25°}$	Pyramidens rotation omkring y-aksen
$\boldsymbol{P_{Kam} = (2; 0; 8)}$	Kameraets placering

$\boldsymbol{aspectRatio = 1}$	Forholdet mellem det nære plans højde og bredde

tabel 9: Anvendte værdier i beregningseksemplet

5.2.2 Beregning af perspektivmatricen

Ved at substituere de ukendte værdier i perspektivmatricen fra ligning (53) kan en perspektivmatrice for de anvendte parametre beregnes, som vist i ligning (54).

$$PM = \begin{bmatrix} \frac{1}{\tan\left(\frac{60}{2}\right) \cdot 1} & 0 & 0 & 0 \\ 0 & \frac{1}{\tan\left(\frac{60}{2}\right)} & 0 & 0 \\ 0 & 0 & \frac{-(1000+0{,}1)}{1000-0{,}1} & \frac{-2 \cdot 1000 \cdot 0{,}1}{1000-0{,}1} \\ 0 & 0 & -1 & 0 \end{bmatrix} \approx \begin{bmatrix} \sqrt{3} & 0 & 0 & 0 \\ 0 & \sqrt{3} & 0 & 0 \\ 0 & 0 & -1 & -0{,}2 \\ 0 & 0 & -1 & 0 \end{bmatrix} \quad (54)$$

5.2.3 Beregning af modelmatrix

En modelmatrix kan fremstilles ved at multiplicere forskydningsmatricen (ligning (55)) med rotationsmatricen (ligning (56)), hvormed et udtryk for modellens rotation og forskydning til world space beregnes.

$$T(P) = \begin{bmatrix} 1 & 0 & 0 & P_x \\ 0 & 1 & 0 & P_y \\ 0 & 0 & 1 & P_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (55)$$

$$R_y(\alpha) = \begin{bmatrix} \cos(\alpha) & 0 & \sin(\alpha) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\alpha) & 0 & \cos(\alpha) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (56)$$

Beregningen af pyramidens modelmatrix ses opstillet i ligning (57).

$$MM = T(P_{Pyr}) \cdot R_y(\alpha_y) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 2 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \cos(-25) & 0 & \sin(-25) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(-25) & 0 & \cos(-25) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \approx \begin{bmatrix} 0{,}9063 & 0 & -0{,}4226 & 0 \\ 0 & 1 & 0 & 2 \\ 0{,}4226 & 0 & 0{,}9063 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (57)$$

Fordi det i OpenGL er fastsat, at kameraet kigger i den negative retning af z-aksen placeret i origo, er det ikke muligt at forskyde kameraet til punktet P_{Kam}. Derimod er det muligt at simulere kameraets placering ved at flytte verdenen, kameraet er placeret i. Hvis kameraet f.eks. ønskes flyttet to enheder op, kan dette opnås ved at flytte alle objekter i verden to enheder ned, hvilket giver illusionen af, at kameraet har flyttet sig. Matricen, der indeholder kameraets transformationer, eller nærmere verdens transformationer i forhold til kameraet, kaldes en viewmatrix (5). Denne matrix kan multipliceres med modelmatricen for at opnå illusionen af kamerabevægelse.

$$VM(P) = \begin{bmatrix} 1 & 0 & 0 & -P_x \\ 0 & 1 & 0 & -P_y \\ 0 & 0 & 1 & -P_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (58)$$

Viewmatricen kan beregnes ved at forskyde med de negerede kamerakoordinater, som det ses i ligning (58), hvis variabler er substitueret med de anvendte værdier fra tabel 9 i ligning (59).

$$VM(P_{Kam}) = \begin{bmatrix} 1 & 0 & 0 & -2 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -8 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (59)$$

En MVP-matrice kan beregnes ved at multiplicere perspektivmatricen med hhv. viewmatricen og modelmatricen, hvilket ses i ligning (60).

$$MVP = PM \cdot VM \cdot MM = \begin{bmatrix} \sqrt{3} & 0 & 0 & 0 \\ 0 & \sqrt{3} & 0 & 0 \\ 0 & 0 & -1 & -0{,}2 \\ 0 & 0 & -1 & 0 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 & -2 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -8 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 0{,}9063 & 0 & -0{,}4226 & 0 \\ 0 & 1 & 0 & 2 \\ 0{,}4226 & 0 & 0{,}9063 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \approx \begin{bmatrix} 1{,}57 & 0 & -0{,}732 & -3{,}46 \\ 0 & 1{,}73 & 0 & 3{,}46 \\ -0{,}423 & 0 & -0{,}906 & 7{,}8 \\ -0{,}423 & 0 & -0{,}906 & 8 \end{bmatrix} \quad (60)$$

5.2.4 Transformation af vertices

Alle pyramidens vertexdata kan opstilles i kolonner i en matrix, som det ses i ligning (61), hvor hver tredje kolonne angiver starten på en ny trekant i modellen. Vertexdataene kommer fra figur 15 i teoriafsnittet på side 24.

$$Pyr_{eye} = \begin{bmatrix} 1 & -1 & 0 & -1 & -1 & 0 & -1 & 1 & -1 & 1 & 1 & 0 & -1 & 1 & 0 & -1 & 1 & 1 \\ -1 & -1 & 1 & -1 & -1 & 1 & -1 & -1 & -1 & -1 & -1 & 1 & -1 & -1 & 1 & -1 & -1 & -1 \\ 1 & 1 & 0 & 1 & -1 & 0 & 1 & -1 & -1 & -1 & 1 & 0 & -1 & -1 & 0 & 1 & 1 & -1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \quad (61)$$

MVP-matricen indeholder alle de transformationer, der er nødvendige for at rotere, forskyde og overføre pyramiden til clipping space, og denne kan derfor bare multipliceres på alle pyramidens vertexdata, som det ses i ligning (62). Det er denne multiplikation, der foregår på GPU'en vha. en vertex shader for hver enkelte vertex individuelt. I eksemplet her er vertexdataene grupperet i én matrice, da dette er mere belejligt, men selve multiplikationen foregår på samme måde, som var det gjort for den individuelle vertex, da MVP-matricen multipliceres ind på hver kolonne.

$$Pyr_{clip} = MVP \cdot Pyr_{eye} \quad (62)$$

$$Pyr_{clip} = MVP \cdot Pyr_{eye} = \begin{bmatrix} 1{,}57 & 0 & -0{,}732 & -3{,}46 \\ 0 & 1{,}73 & 0 & 3{,}46 \\ -0{,}423 & 0 & -0{,}906 & 7{,}8 \\ -0{,}423 & 0 & -0{,}906 & 8 \end{bmatrix} \cdot \begin{bmatrix} 1 & -1 & 0 & -1 & -1 & 0 & -1 & 1 & -1 & 1 & 1 & 0 & -1 & 1 & 0 & -1 & 1 & 1 \\ -1 & -1 & 1 & -1 & -1 & 1 & -1 & -1 & -1 & -1 & -1 & 1 & -1 & -1 & 1 & -1 & -1 & -1 \\ 1 & 1 & 0 & 1 & -1 & 0 & 1 & -1 & -1 & -1 & 1 & 0 & -1 & -1 & 0 & 1 & 1 & -1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} -2{,}626 & -5{,}766 & -3{,}464 & -5{,}766 & -4{,}302 & -3{,}464 & -5{,}766 & -1{,}162 & -4{,}302 & -1{,}162 & -2{,}626 & -3{,}464 & -4{,}302 & -1{,}162 & -3{,}464 & -5{,}766 & -2{,}626 & -1{,}162 \\ 1{,}732 & 1{,}732 & 5{,}196 & 1{,}732 & 1{,}732 & 5{,}196 & 1{,}732 & 1{,}732 & 1{,}732 & 1{,}732 & 1{,}732 & 5{,}196 & 1{,}732 & 1{,}732 & 5{,}196 & 1{,}732 & 1{,}732 & 1{,}732 \\ 6{,}472 & 7{,}318 & 7{,}80 & 7{,}318 & 9{,}131 & 7{,}802 & 7{,}318 & 8{,}285 & 9{,}131 & 8{,}285 & 6{,}472 & 7{,}801 & 9{,}131 & 8{,}285 & 7{,}802 & 7{,}318 & 6{,}472 & 8{,}285 \\ 6{,}671 & 7{,}516 & 8 & 7{,}516 & 9{,}329 & 8 & 7{,}516 & 8{,}484 & 9{,}329 & 8{,}484 & 6{,}671 & 8 & 9{,}329 & 8{,}484 & 8 & 7{,}516 & 6{,}671 & 8{,}484 \end{bmatrix}$$

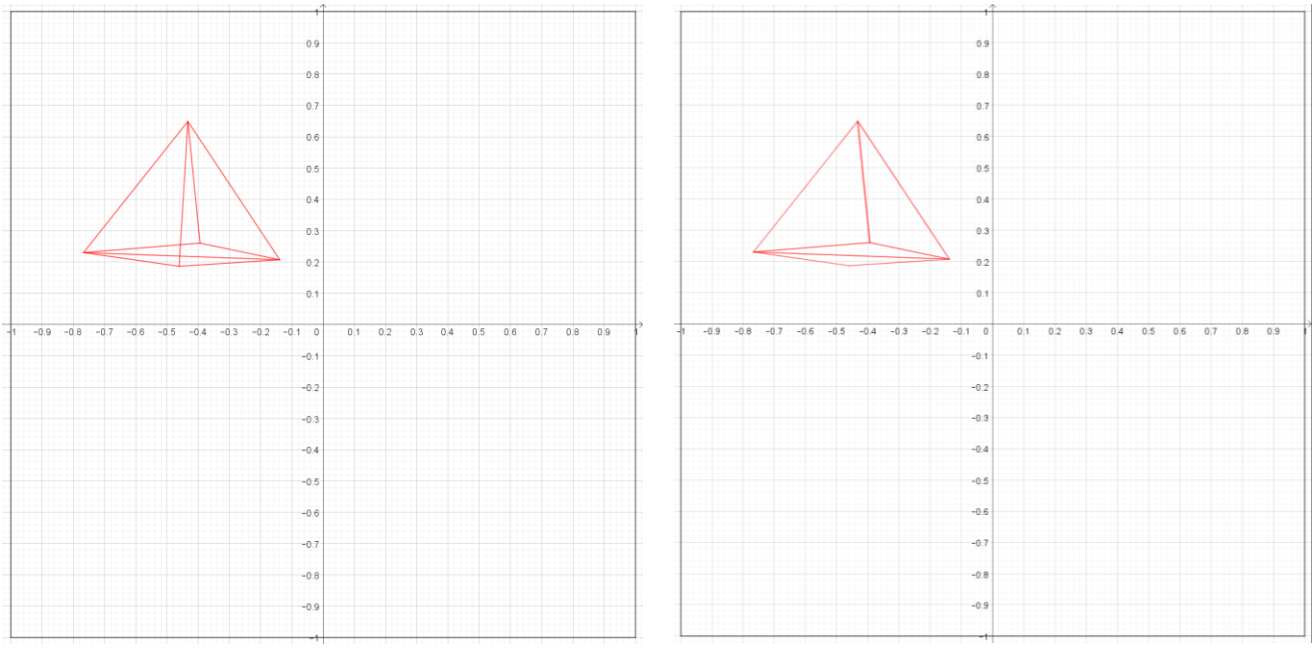
Ved at dividere med w-koordinatet overgår alle vertexdataene fra clipping space til NDC space, der kan indtegnes på en todimensionel tegneflade. Resultatet af vertexdataene efter den perspektivgivende division ses i ligning (63)

$$Pyr_{NDC}$$

$$= \begin{bmatrix} -0{,}394 & -0{,}767 & -0{,}433 & -0{,}767 & -0{,}461 & -0{,}433 & -0{,}767 & -0{,}139 & -0{,}461 & -0{,}139 & -0{,}394 & -0{,}433 & -0{,}461 & -0{,}139 & -0{,}433 & -0{,}767 & -0{,}394 & -0{,}139 \\ 0{,}26 & 0{,}23 & 0{,}649 & 0{,}23 & 0{,}186 & 0{,}649 & 0{,}23 & 0{,}207 & 0{,}186 & 0{,}207 & 0{,}26 & 0{,}649 & 0{,}186 & 0{,}207 & 0{,}649 & 0{,}23 & 0{,}26 & 0{,}207 \\ 0{,}97 & 0{,}974 & 0{,}975 & 0{,}974 & 0{,}979 & 0{,}975 & 0{,}974 & 0{,}988 & 0{,}979 & 0{,}988 & 0{,}97 & 0{,}975 & 0{,}979 & 0{,}988 & 0{,}975 & 0{,}974 & 0{,}97 & 0{,}988 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \quad (63)$$

5.2.5 Grafisk afbildning

Vertexdataene kan simpelt projiceres vinkelret på xy-planet ved at ignorere z-koordinatet ved indtegning i GeoGebra, hvilket kan ses i figur 31. Det ene billede er indtegnet, hvor de bagvedliggende flader fjernes vha. manuel udførelse af culling, mens modellens bagvedliggende flader ikke er indtegnet på det andet billede.



figur 31: Vertexdataene indtegnet I GeoGebra: tv. uden culling; th. med culling i urets retning

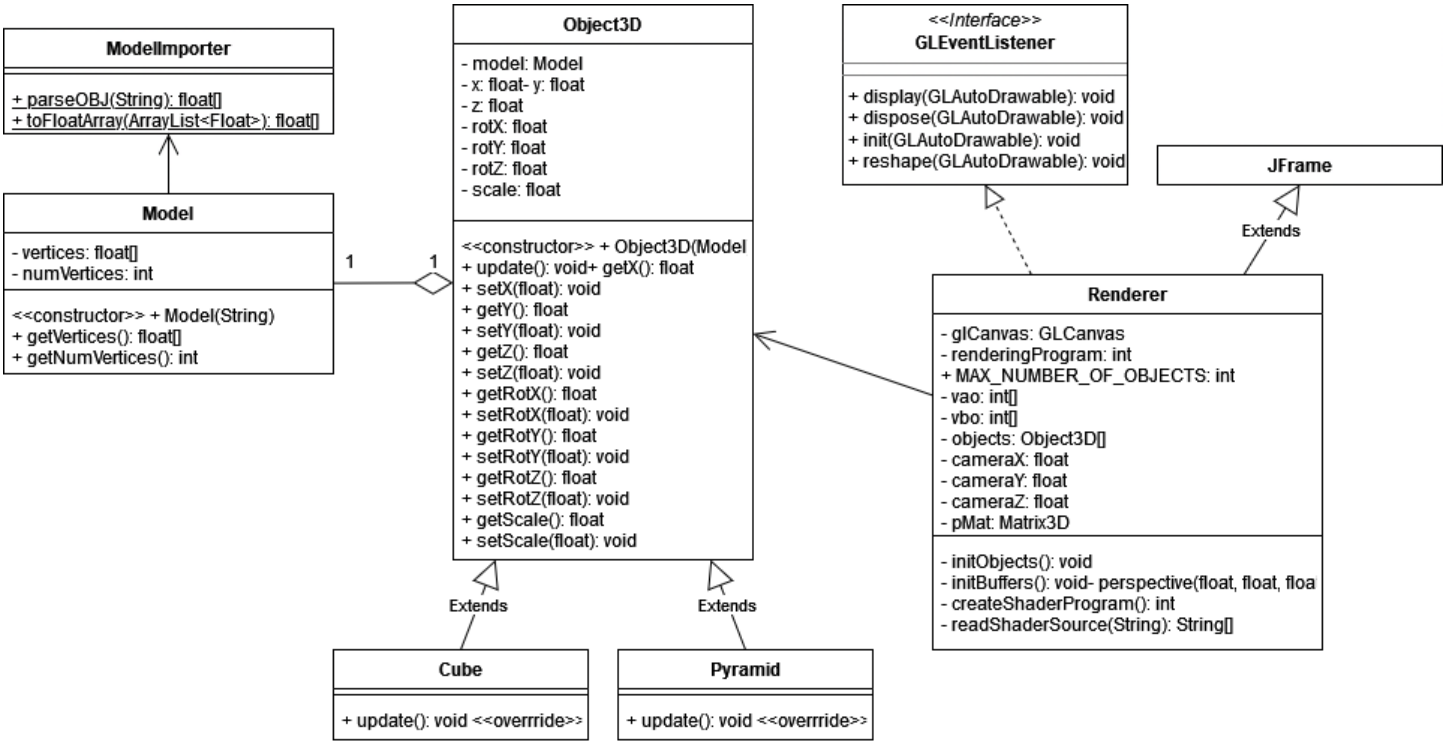
Det visuelle resultat i figur 31 virker realistisk. Det kan f.eks. ses, at pyramiden er blevet roteret, ved, at den mest synlige pyramidesides grundlinje ikke er parallel med x-aksen. Derudover er pyramidens bund også synlig som resultat af, at pyramiden er placeret højere end kameraet. Dernæst fremkommer pyramiden i venstre side af billedet, hvilket er som følge af kamerapositionen, der er forskudt i den positive x-retning.

På baggrund af dette resultat foretages nu en softwareimplementation af perspektivmatricen, og derefter vurderes det anvendte værktøj OpenGL.

5.3 Softwareimplementation

5.3.1 Programmets opbygning

I forbindelse med projektet er et simpelt eksempel på perspektivmatricens implementation gennem software blevet udarbejdet ved anvendelse af den kendte teori om perspektivmatricen og OpenGL. Programmet er udviklet ved brug af bibliotekerne JOGL og Graphicslib3D, og dets struktur kan ses i klassediagrammet i figur 32.



figur 32: Klassediagram af softwareimplementationen

Programmet består af seks klasser, hvoraf den vigtigste er ”Renderer”, der står for alle OpenGL-kald. Renderer arver fra JFrame, der kommer fra Javas Swing framework, og er selve vinduet. GLCanvas er selve tegnefladen. GLCanvas arver fra Swing’s component, hvilket betyder, at den kan føjes til en JFrame container, hvor den vil blive vist. Samtidigt med at Renderer agerer vindue, står den også for at implementere metoderne fra interfacet GLEventListener.

5.3.2 Udsnit af kildekoden

5.3.2.1 Programmets Initialisering

Det første, der sker, efter Renderer er instantieret, er, at $init$, der ses i figur 33, bliver kaldt. Herefter oprettes shaderprogrammet, og OpenGL indstilels til bl.a. at aktivere culing og z-algoritmen. Til slut beregnes perspektivmatricen, og programmets visuelle objekter instantieres, hvorefter det samme sker for deres buffere.

```
public void init(GLAutoDrawable drawable) {
    GL4 gl = (GL4) GLContext.getCurrentGL();

    renderingProgram = createShaderProgram();
    gl.glUseProgram(renderingProgram);

    gl.glPolygonMode(GL4.GL_FRONT_AND_BACK, GL4.GL_LINE);

    gl.glEnable(GL4.GL_CULL_FACE);
    gl.glFrontFace(GL4.GL_CW);

    gl.glEnable(GL4.GL_DEPTH_TEST);
    gl.glDepthFunc(GL4.GL_LEQUAL);

    cameraX = 2.0f; cameraY = 0.0f; cameraZ = 8.0f;

    float aspect = (float) glCanvas.getWidth() / (float) glCanvas.getHeight();
    pMat = perspective(60.0f, aspect, 0.1f, 1000.0f);

    initObjects();
    initBuffers();
}
```

figur 33: Metoden init

5.3.2.2 Indlæsning af shaders

Inde i $init$ bliver $createShaderProgram$ kaldt. Denne metode anvender metoder i afsnit 4.5.1 til at oprette et shaderprogram ud fra de to shader-filer, der bliver indlæst ved et kald til $readShaderSource$, der ses i figur 34.

```
private String[] readShaderSource(String filename) {

    ArrayList<String> lines = new ArrayList<String>();

    try {
        BufferedReader br = new BufferedReader(new InputStreamReader(Renderer.class.getResourceAsStream(filename)));

        String line;
        while ((line = br.readLine()) != null) {
            lines.add(line);
        }

        br.close();

    } catch (IOException e) {
        e.printStackTrace();
    }

    String[] program = new String[lines.size()];

    for(int i = 0; i < lines.size(); i++) {
        program[i] = (String) lines.get(i) + "\n";
    }

    return program;
}
```

figur 34: Metoden readShaderSource

Til indlæsning af filerne anvendes en BufferedReader og bygger på en InputStreamReader. Modsat en InputStreamReader kan en BufferedReader læse større blokke af data ad gangen gennem en buffer (10) og har tilmed en $readLine$ metode, der kan læse en hel linje ad gangen. En datastrøm til shaderen hentes ved brug af metoden $getResourceAsStream$, der henter en fil ud fra en relativ sti i forhold til $Renderer$-klassen. Fordi der er en risiko for, at filen ikke eksisterer, foregår instantieringen i en try/catch-sætning, der fanger eventuelle IO-fejl (10). Inde i metoden $readShaderSource$ læses fra filen, indtil der ikke er flere linjer og $readLine$ returnerer $null$. Til sidst omdannes ArrayListen til et array, og der lægges en newline-termintor på enden af hver linje, da dette kræves af OpenGL.

5.3.2.3 Indlæsning af Wavefront-filer

$initObjects$ instantiere alle objekterne, før deres VBO'er erklæres vha. $initBuffers$. Alle objekter er instanser af $Object3D$, der egentligt fungerer som dataholder. Bl.a. har den variablen $model$ af typen $Model$, der ved instantiering indlæser en Wavefront-fil vha. den statiske metode $parseOBJ$, der ses i figur 35 i $ModelImporter$.

```
public static float[] parseOBJ(String filename) throws IOException {
    ArrayList<Float> triangleVerts = new ArrayList<Float>();
    ArrayList<Float> vertCoords = new ArrayList<Float>();
    BufferedReader br = new BufferedReader(new InputStreamReader(ModelImporter.class.getResourceAsStream(filename)));

    String line;
    while ((line = br.readLine()) != null) {
        if (line.startsWith("v ")) {
            for (String s : (line.substring(2)).split(" ")) {
                vertCoords.add(Float.valueOf(s));
            }
        } else if (line.startsWith("f ")) {

            for (String s : (line.substring(2).split(" "))) {
                // Wavefront-index 0-indekseres og multipliceres med 3, da vertexdataene nu lagres sekventielt i vertCoords
                int vertRef = (Integer.valueOf(s)-1)*3;

                triangleVerts.add(vertCoords.get(vertRef));
                triangleVerts.add(vertCoords.get(vertRef+1));
                triangleVerts.add(vertCoords.get(vertRef+2));
            }
        }
    }
    br.close();
    return toFloatArray(triangleVerts);
}
```

figur 35: Metoden parseOBJ

$parseOBJ$ kan indlæse en Wavefront-fil, hvis den følger den basiske udgave beskrevet i afsnit 4.7. For hver linje kigger på fortegnet vha. en if-sætning. Hvis denne begynder med v, opsplitter den de efterfølgende elementer og tilføjer dem sekventielt til $vertCoords$. Hvis linjen begynder med f, opsplittes de efterfølgende elementer, der er referencer til vertexdataene. Herefter føjes hver vertex sekventielt til $triangleVerts$, der sekventielt indeholder alle vertexkoordinater til modellens primitiver.

5.3.2.4 Erklæring af buffers (VBO)

I $initBuffers$, der ses i figur 36, erklæres VAO'en og alle objekternes VBO'er, der bliver forbundet med objekternes indlæste vertexdata. Dette foregår vha. en direct buffer, der giver direkte adgang til hukommelsen udenom Javas JVM, hvilket giver højere ydeevne, når der skal indlæses tusindvis af vertexdata fra større 3D-modeller (5). I $glBufferData$-kaldets anden parameter multipliceres antallet af vertexkoordinater, der hentes med $limit$-metoden med fire, da dette er bytestørrelsen af variabeltypen $float$.

```
private void initBuffers() {

    GL4 gl = (GL4) GLContext.getCurrentGL();
    gl.glGenVertexArrays(vao.length, vao, 0);
    gl.glBindVertexArray(vao[0]);
    gl.glGenBuffers(vbo.length, vbo, 0);

    for (int i = 0; i < objects.length; i++) {
        if (objects[i] == null) continue;

        float[] vertValues = objects[i].getModel().getVertices();

        gl.glBindBuffer(GL4.GL_ARRAY_BUFFER, vbo[i]);
        FloatBuffer vertDataBuffer = Buffers.newDirectFloatBuffer(vertValues);
        gl.glBufferData(GL4.GL_ARRAY_BUFFER, vertDataBuffer.limit()*4, vertDataBuffer, GL4.GL_STATIC_DRAW);
    }

}
```

figur 36: Metoden initBuffers

5.3.2.5 Rendering

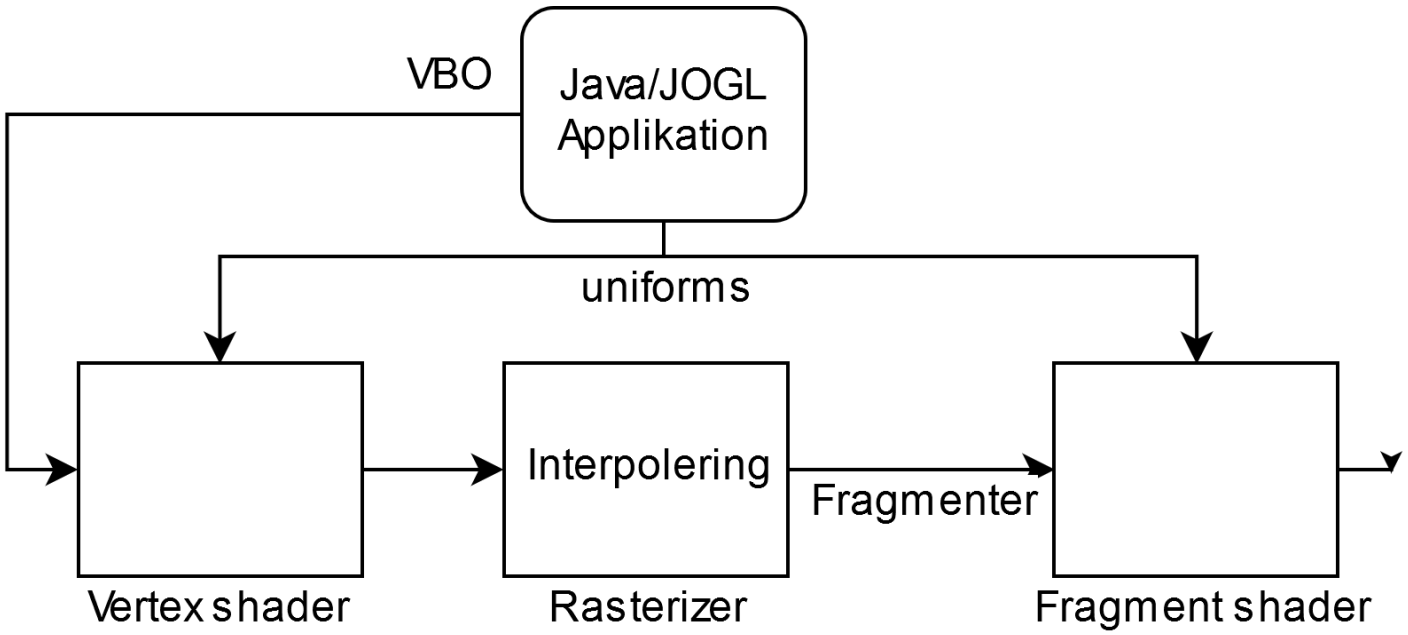
Hver gang GLCanvas gentegnes, kaldes $display$-metoden, der ses i figur 37. $display$ afgør, hvad der skal tegnes, hvilket er alle de instantierede objekter. Først kaldes update-kommandoen på alle objekterne, hvorved eventuel bevægelse kan foretages. Herefter nulstilles dybdebufferen ved et kald til $glClear$, så den kan anvendes til dybdetesten. Herefter renses vinduet for alt tidligere tegnet ved at give den farven fra arrayet bkg, der består af tre værdier, der er en del af intervallet $[0; 1]$. Værdierne angiver rød, grøn, blå samt gennemsigtighed. Efterfølgende kommer den del af processen, der er tilsvarende den gennemgåede i afsnit 0.

```
public void display(GLAutoDrawable drawable) {
    for (Object3D object : objects) {
        if (object == null) continue;
        object.update();
    }

    GL4 gl = (GL4) GLContext.getCurrentGL();
    gl.glClear(GL4.GL_DEPTH_BUFFER_BIT);

    float bkg[] = {1.0f, 1.0f, 1.0f, 1.0f};
    FloatBuffer bkgBuffer = Buffers.newDirectFloatBuffer(bkg);
    gl.glClearBufferfv(GL4.GL_COLOR, 0, bkgBuffer);

    Matrix3D vMat = new Matrix3D();
    vMat.translate(-cameraX, -cameraY, -cameraZ);

    for (int i = 0; i < objects.length; i++) {
        if (objects[i] == null) continue;

        Matrix3D mMat = new Matrix3D();
        mMat.translate(objects[i].getX(), objects[i].getY(), objects[i].getZ());
        mMat.rotate(objects[i].getRotX(), objects[i].getRotY(), objects[i].getRotZ());
        mMat.scale(objects[i].getScale(), objects[i].getScale(), objects[i].getScale());

        Matrix3D mvpMat = new Matrix3D();
        mvpMat.concatenate(pMat);
        mvpMat.concatenate(vMat);
        mvpMat.concatenate(mMat);

        int mvp_loc = gl.glGetUniformLocation(renderingProgram, "mvpMatrix");
        gl.glUniformMatrix4fv(mvp_loc, 1, false, mvpMat.getFloatValues(), 0);

        gl.glBindBuffer(GL4.GL_ARRAY_BUFFER, vbo[i]);
        gl.glVertexAttribPointer(0, 3, GL4.GL_FLOAT, false, 0, 0);
        gl.glEnableVertexAttribArray(0);

        int numVerts = objects[i].getModel().getNumVertices();

        gl.glDrawArrays(GL4.GL_TRIANGLES, 0, numVerts);
    }
}
```

figur 37: Metoden display

5.3.2.5.1 Programmets data flow

Først oprettes en viewmatrix, hvorefter for-lykke gennemgår hvert eneste objekt, og beregner dets modelmatrix. Af viewmatricen, modelmatricen og den tidligere beregnede perspektivmatrix beregnes en $mvpMatrix$, der sendes til shaderen som en uniform-variabel, samtidigt med, at det pågældende objekts vertexdata sendes til den tilsvarende vertex-attribut. Til slut kaldes $glDrawArrays$, der fortæller OpenGL, at den pågældende model skal renderes. $GL\_TRIANGLES$ fortæller metoden, at der er tale om trekanter og f.eks. ikke punkter, mens den næste parameter er et eventuel offset i den aktive vertex-attribut, og det sidste angiver, hvor mange vertices, der er tale om. Selve programmets data flow ses visualiseret i figur 38.



figur 38: Programmets data flow

De to shaders GLSL-kode ses i figur 39. Den første linje i begge shaders angiver hvilken version af OpenGL, der er tale om. Begge shaders har også en $main$-funktion, der blive kaldt, når shaderen eksekveres. Udenfor funktionen erklæres de anvendte variabler. I vertexshaderen er $gl\_Position$ en prædefineret variabel, der anvendes til at give en vertex' sit koordinatsæt i tredimensionelt rum, der bliver sendt videre i OpenGL's pipeline. $gl\_Position$ sættes lig MVP-matricen multipliceret med den pågældende vertex fra bufferen, der også får tilføjet et fjerde koordinat lig 1.
I dette projekt anvendes en meget simpel fragment shader, der udelukkende anvendes til at farve objekterne røde gennem en outputvariabel, der kaldes $color$. Ligesom da vinduets baggrundsfarve blev initialiseret, består denne også af fire værdier; her repræsenteret af en firedimensional vektor $vec4$.

```
#version 430

layout (location=0) in vec3 position;

uniform mat4 mvpMatrix;

void main(void) {
    gl_Position = mvpMatrix * vec4(position, 1.0);
}
```

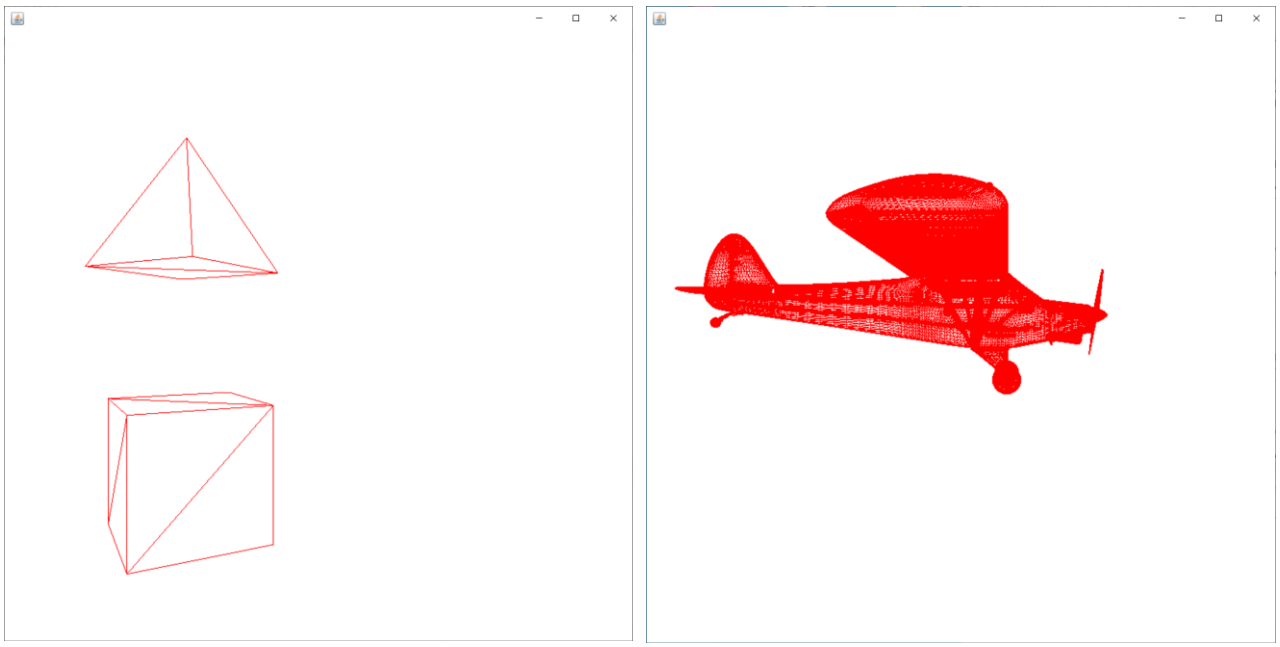
```
#version 430

out vec4 color;

uniform mat4 mvpMatrix;

void main(void) {
    color = vec4(1.0, 0.0, 0.0, 1.0);
}
```

figur 39: De to shaders: tv. vertexshaderen; th. fragmentshaderen

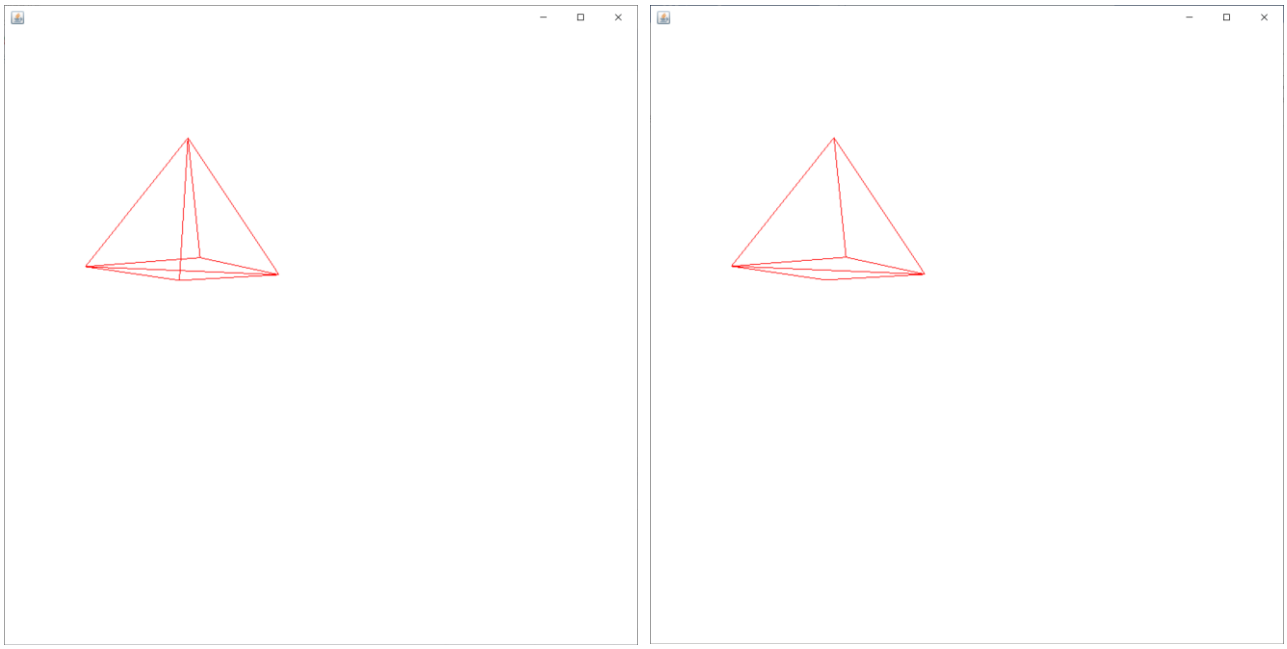
5.4 Det grafiske resultat

Resultatet, når programmet eksekveres, vil variere alt efter, hvordan og hvilke objekter, der instantieres. I figur 40 ses en flymodel, der består af flere hundredetusinde trekanter renderet. ModelImporter-klassen har gjort det let at indlæse

komplekse modelfiler. I figur 41 ses samme eksempel, der før blev beregnet matematisk renderet ved hjælp af programmet. Det kan ses, at resultatet viser det samme, som tegnet i GeoGebra. Hele programmets kildekode er afleveret i en zip-fil som ekstramateriale.



figur 40: tv. rendering af to objekter; tv. rendering af en flymodel



figur 41: Pyramiden af softwareimplementation af perspektivmatricen vha. JOGL: tv. uden culling; th. med culling i urets retning

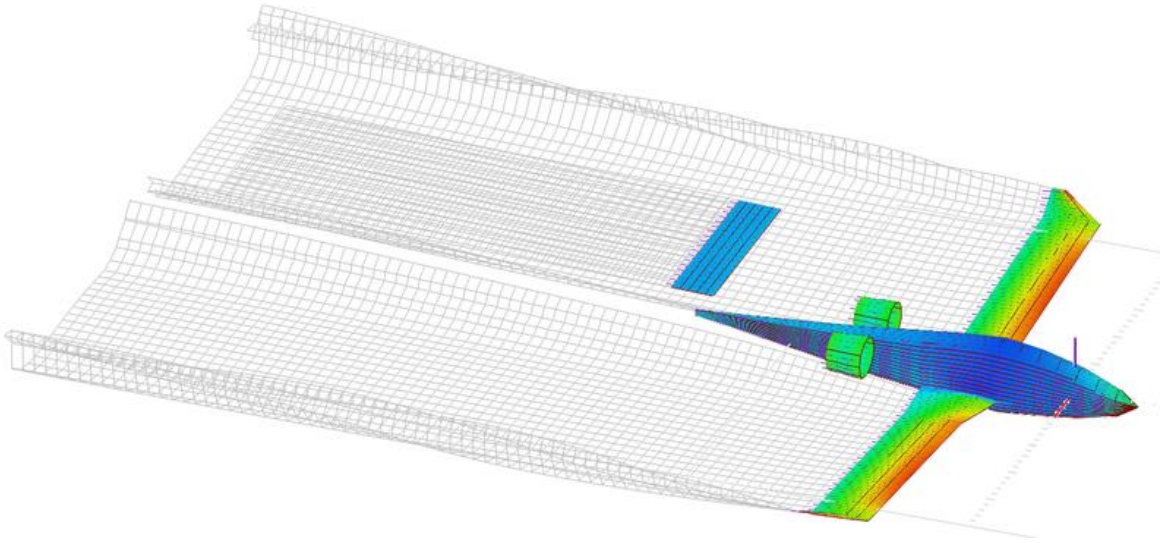
Perspektivmatricen sammen med den perspektivgivende division giver illusionen af 3D på en todimensionel flade. I praksis lægges der mange flere elementer til, der skal forstærke illusionen af 3D. Herunder er blandt andet lys og skygger, der medvirker til at fremme modellens form, ved at nuancere udfyldningsfarven af trekanterne, en model består af. Hertil kommer mange flere effekter, der skaber realisme og forstærker den tredimensionelle illusion såsom genspejling, lysspredning, blankhed mm. (5).

6 Perspektivering

3D-grafik samt anvendelsen af perspektiv til at repræsentere tredimensionelle objekter på en computerskærm finder stor anvendelse i en lang række brancher. Bl.a. bliver det anvendt inden for molekylærbiologi til at visualisere og simulere komplekse molekyler, der ikke er todimensionelle strukturer, som deres strukturformler ofte fremstilles. 3D-grafik finder også anvendelse inden for byggebranchen, hvor bl.a. terrænet og byggeriet kan konstrueres virtuelt inden opførelse.

Indenfor aerodynamik spiller 3D-grafik og matrixregning en stor rolle. F.eks. anvender virksomheden Boeing Phantom Works 3D-modellering og numeriske strømningsberegninger (CFD) (Eng. Computational Fluid Dynamics), der er læren om væskers bevægelse (1). 3D-grafik kan bruges til at visualisere turbulens og luftens opførsel omkring en flymodel. En fragment shader kan anvendes til at farvelægge en model af et fly, så flader der påvirkes af et højt tryk, får én farve, mens lavere tryk får en anden, som det ses i figur 42.

De bagvedlæggende beregninger kan endda foretages parallelt på en GPU ved brug af shadere for at opnå en højere ydeevne. Ved brug af Lattice Boltzmann-metoder kan luftens strømninger omkring modellen beregnes parallelt, hvilket gør det oplagt at anvende GPU'en til disse udregninger. I en rapport fra 2011 undersøger tre forskere den ydelsesmæssige fordel ved at benytte en GPU fremfor en CPU til at køre en række forskellige Lattice Boltzmann algoritmer, hvorved de fandt frem til, at beregningerne kan foretages 10 til 25 gange hurtigere på en GPU (13).



figur 42: "Simulation of a flying airplane in Open VOGEL" af GuillermoHazebrouck, udgivet under CC BY-SA 4.0[4]

De beregnede data kan omsættes til en 3D-model, der kan pålægges perspektiv vha. perspektivmatricen og tegnes på en todimensionel skræm, som den i figur 42. Ingeniørerne kan dreje og flytte på modellen og observere, hvordan luften opfører sig, når det bevæger sig hen over modellen, hvilket de kan bruge til at optimere flyet ud fra. Teknikkerne er ikke begrænset til flyudvikling, men kan også bruges til udvikling af biler, pumper, vindmøller, og ventilation mm. (1).

OpenGL er ikke det eneste eksisterende grafikbibliotek. OpenGL er sammen med Vulkan og Microsofts proprietære bibliotek DirectX de tre mest populære og udbredte low-level grafikbiblioteker, og disse danner grundlaget for langt størstedelen af alle 3D-programmer (8) (14) (12).

Valget af grafikbibliotek afhænger af produktets behov. Hvis det f.eks. er et produkt, der kræver stor beregningskraft og ydeevne, samt specialiserede shaders som f.eks. et CFD-værktøj, kan det være givende at benytte et low-level værktøj som f.eks. OpenGL. OpenGL giver udvikleren meget stor kontrol over hardwaren i computeren. Dette projekt har kun dækket en minimal del af OpenGL's funktionaliteter. Jo større kontrol over computeren, udvikleren har, des bedre forudsætninger er der for at skrive optimeret software. Det er i dette tilfælde måske endda et bedre valg at anvende Vulkan eller DirectX 12, der tilbyder endnu større kontrol over computerens hardware end den nyeste version af OpenGL, der er ved at være udateret i forhold til moderne grafikprocessorer (8).

Det er også nødvendigt at tage platformen i betragtning. Selvom OpenGL og Vulkan er multiplatformbiblioteker, findes der lukkede systemer som det Microsoft-ejede Xbox, der udelukkende understøtter DirectX (14). Derudover er Apple også ved at omstille til kun at understøtte sit eget proprietære bibliotek Metal (15).

Hvis formålet f.eks. er at udvikle et spil, kan det være en bedre idé at anvende en såkaldt spilmotor som Unity eller lign. Unity indeholder allerede mange værktøjer til fremstilling af 3D-spil. Bl.a. er det ikke nødvendigt at skrive en 3D-renderer, hvilket allerede er en del af Unity, mens det samtidigt er muligt at vælge, hvilken low-level API som OpenGL,

[4] https://creativecommons.org/licenses/by-sa/4.0

Unity skal anvende i baggrunden. Unity indeholder også mange prædefinerede shaders, der gennem en grafisk brugergrænseflade kan sammensættes til at give et materiale det rette udseende, herunder lysets opførsel, når det rammer materialets overflade (16). At skulle skrive kode, der opfylder alle de funktioner, en anerkendt spilmotor indeholder, vil være meget tidskrævende og være som at genopfinde den dybe tallerken.

I visse tilfælde, hvor der udvikles krævende spil, kan det være en fordel at skrive en spilmotor og rendere vha. et low-level grafikbibliotek, der fokuserer på det individuelle spils funktioner. Det er dog nødvendigt med erfarne udviklere, der ved, hvordan systemet kan udvikles optimalt ved at gøre brug af den øgede kontrol over hardwaren, et low-level bibliotek tilbyder. Det er ikke udelukkende værktøjet, der sætter grænser for et systems ydeevne, men også udvikleren bag systemet.

Når det kommer til perspektivmatricen, bygger alle grafikbibliotekerne på samme grundlag. Der kan dog være enkelte detaljer, der er til forskel. Bl.a. bruger DirectX venstrehåndskoordinatsystemer, og NDC-terningen ligger i z-intervallet $[0; 1]$ (14), hvilket er den halve OpenGL NDC-terning.

7 Konklusion

Matrix-matrix multiplikation er den mest anvendte matrixoperation i 3D-grafik og kan opsummeres gennem ligning (64). Hver $(i; j)$-indgang i den resulterende matrix er lig summen af de parvise produkter af elementerne i matrice A's kolonner og matrice B's rækker. Den resulterende matrice har samme antal rækker som A og samme antal kolonner som B.

$$(AB)_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \cdots + a_{in}b_{nj} \tag{64}$$

Perspektivmatricen er grundlæggende for ethvert 3D-program, der anvender perspektiv. Perspektivmatricen er et eksempel på en matrix, der finder sin anvendelse gennem matrixmultiplikation. Multipliceret på en models vertexdata efterfulgt af den perspektivgivende division, anvendes den til at transformere et objekt fra 3D til 2D, så det kan tegnes på en todimensionel computerskærm, men give illusionen af 3D. Dette foregår ved at transformere det synlige felt til en terning, hvor vertexdataene ligger i 2D, men bibeholder et z-koordinat, der angiver deres relative afstand til hinanden, så objekter tættere på kameraet tegnes ovenpå bagvedliggende objekter.

Perspektivmatricen efterfulgt af den perspektivgivende division er til for at simulere et kamera. Matricen er opbygget, så den ved anvendelse sammen med den efterfølgende division, tilordner alle punkterne i det synlige rum til en terning med sidelængden 2 og centrum i origo. Når vertexdataene har gennemgået denne transformation, repræsenterer de modellen i 2D. Det tilordnede Z-koordinat angiver punkternes indbyrdes relative afstand, hvilket anvendes af OpenGL til "Hidden Surface Removal".

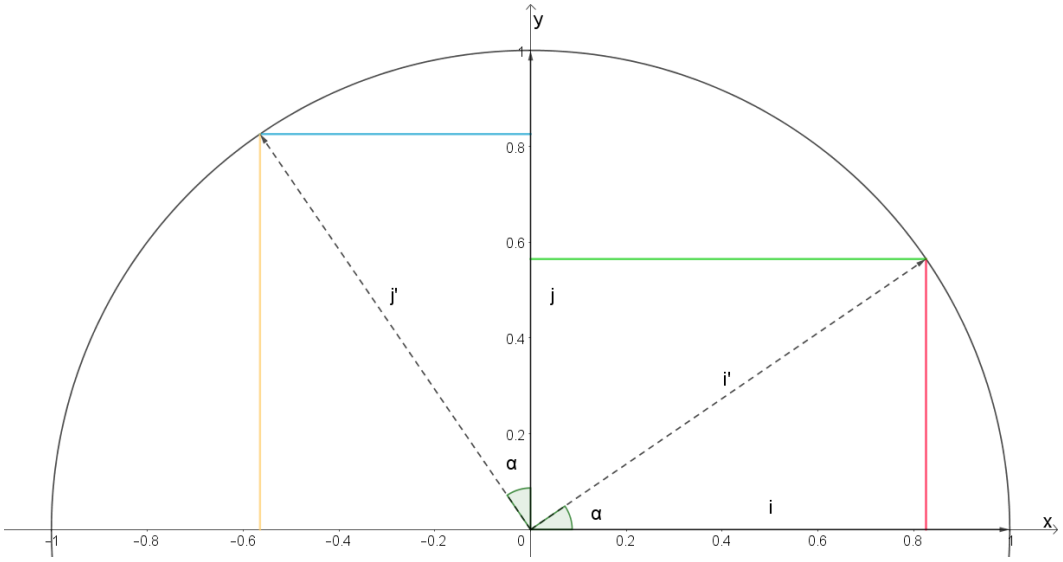
OpenGL følger den såkaldte ”rendering pipeline”, der er en prædefineret sekvens af trin. Ved brug af OpenGL’s API og sproget GLSL kan denne rendering pipeline programmeres til at anvende perspektivmatricen på alle vertexdataene, en model består af.

OpenGL tilbyder en meget stor kontrol over computerens GPU gennem denne pipeline, hvilket gør udvikleren i stand til at bruge GPU’ens ressourcer mest effektivt. Dette er ikke nødvendigvis en fordel, da det selv for et simpelt program kræver, at der bliver investeret meget tid i udvikling modsat f.eks. Unity, der indeholder prædefinerede værktøjerne til hurtigt at udvikle et simpelt spil.

8 Bilag

8.1 Udledning af rotationsmatricerne

Som vist i ligning (10) kan et punkt defineres ud fra enhedsvektorerne i og j. Når disse enhedsvektorer transformeres, vil punktet transformeres med og holde samme forhold til punktet, da disse er bundet ved en lineær sammenhæng i ligning (10). Derved kan et punkt roteres omkring origo ved at rotere enhedsvektorerne i og j.



figur 43: Rotation omkring origo i positiv omdrejningsretning placeret i enhedscirkel

Eftersom enhedsvektorerne har en længde på 1, kan der tages udgangspunkt i enhedscirklen.

Når i roteres omkring origo, kan den nye x-værdi bestemmes ved brug af cosinus, hvor den hosliggende katete er markeret med grønt i figur 43.

$$\cos(\alpha) = \frac{hosliggende\ katete}{hypotenuse}$$

$$\cos(\alpha) = \frac{hosliggende\ katete}{1} = hosliggende\ katete$$

$$i'_x = \cos(\alpha)$$

$$i'_y = \sin(\alpha)$$

Dermed findes den roterede vektor i' til ligning (65).

$$i' = \begin{bmatrix} \cos(\alpha) \\ \sin(\alpha) \end{bmatrix} \tag{65}$$

Det kan ses i figur 43, at $j's$ afstand til y-aksen kan beregnes ved brug af sinus, da dette er den modstående katete til vinklen α, men da ændringen er i den negative retning ad y-aksen, er det den negerede sinus.

$$j'_x = -\sin(\alpha) \tag{66}$$

Trekantens hosliggende katete svarende til $j's$ y-koordinat, kan beregnes ved brug af cosinus.

$$j'_y = \cos(\alpha)$$

Dermed findes den roterede vektor i' til ligning (67).

$$j' = \begin{bmatrix} -\sin(\alpha) \\ \cos(\alpha) \end{bmatrix} \tag{67}$$

$$R(\alpha) = [i' \quad j'] = \begin{bmatrix} \cos(\alpha) & -\sin(\alpha) \\ \sin(\alpha) & \cos(\alpha) \end{bmatrix} \tag{68}$$

Denne todimensionelle rotationsmatrice kan også blive anvendt i tre dimensioner, hvor der dermed foregår en rotation omkring z-aksen, da kun x- og y-koordinaterne transformeres og ikke z-koordinatet (ligning (69)).

$$R_z(\alpha) = [i' \quad j' \quad k] = \begin{bmatrix} \cos(\alpha) & -\sin(\alpha) & 0 \\ \sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{69}$$

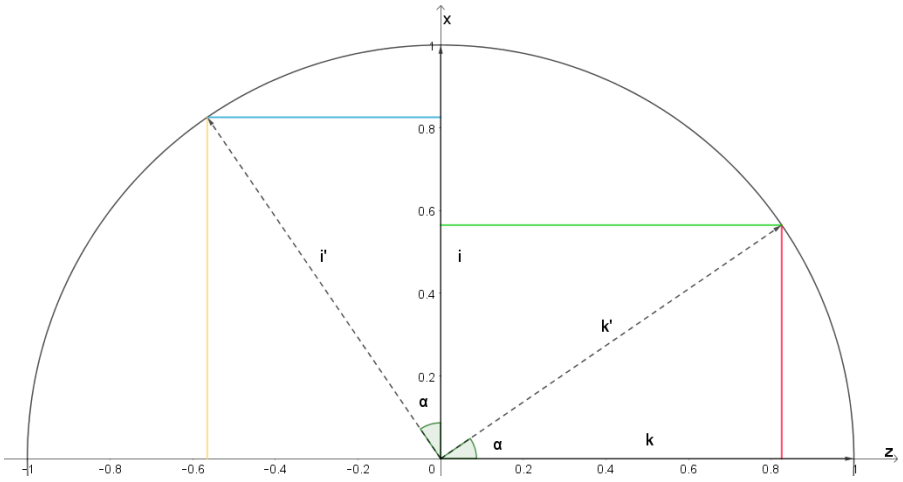
På samme måde rotationsmatricen omkring z-aksen blev fremstillet ved at kigge på et xy-koordinatsystem, kan to øvrige rotationsmatricer fremstilles omkring x- og y- aksen, ved at kigge på hhv. yz-, xz-koordinatsystemer og det viste xy-koordinatsystem.

$$R_x(\alpha) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\alpha) & -\sin(\alpha) \\ 0 & \sin(\alpha) & \cos(\alpha) \end{bmatrix}$$

$$R_y(\alpha) = \begin{bmatrix} \cos(\alpha) & 0 & \sin(\alpha) \\ 0 & 1 & 0 \\ -\sin(\alpha) & 0 & \cos(\alpha) \end{bmatrix}$$

$$R_z(\alpha) = \begin{bmatrix} \cos(\alpha) & -\sin(\alpha) & 0 \\ \sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Måden, hvorpå disse findes, kan illustreres ved at tegne et xz-koordinatsystem, hvor y-aksen peger mod læseren, hvor der her er tale om enhedsvektorerne k og k', der repræsenterer transformation af z-aksen. Samtidigt er koordinatsystemet orienteret således at x- og z-aksernes positive retninger hhv. er opad og mod højre (figur 44).



figur 44: Rotation omkring y-aksen

Her kan k's z-koordinat svarende til afstanden markeret med et grønt linjestykke beregnes ved brug af cosinus.

$$k'_z = \cos(\alpha)$$

Vektorens x-koordinat kan beregnes ved brug af sinus, der på tegningen er markeret med det røde linjestykke.

$$k'_x = \sin(\alpha)$$

Disse to koordinater, kan nu opstilles på vektorform, hvor y-koordinatet fortsat er 0.

$$k \Rightarrow k' = \begin{bmatrix} \sin(\alpha) \\ 0 \\ \cos(\alpha) \end{bmatrix}$$

På samme måde, kan i' findes ved brug af de trigonometriske definitioner.

For i's z-koordinat gælder det samme som beskrevet for ligning (66).

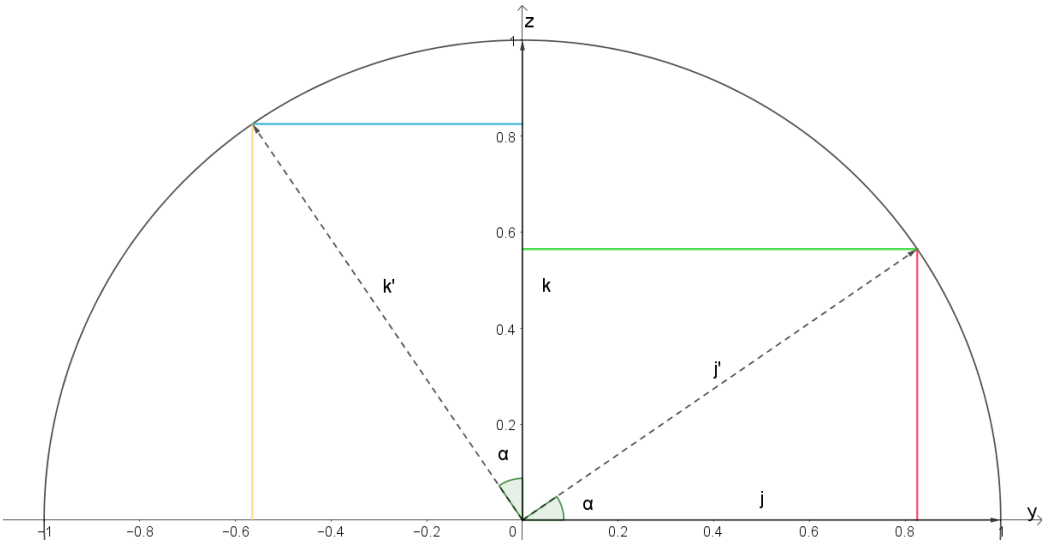
$$i'_z = -\sin(\alpha)$$

$$i'_x = \cos(\alpha)$$

$$i' = \begin{bmatrix} \cos(\alpha) \\ 0 \\ -\sin(\alpha) \end{bmatrix}$$

Disse vektorer kan nu indsættes som vist i ligning (10), hvorved en rotationsmatrice for rotation omkring y-aksen findes.

$$R_y(\alpha) = [i' \quad j \quad k'] = \begin{bmatrix} \cos(\alpha) & 0 & \sin(\alpha) \\ 0 & 1 & 0 \\ -\sin(\alpha) & 0 & \cos(\alpha) \end{bmatrix}$$



figur 45: Rotation omkring x-aksen

Igen kan den transformerede enhedsvektors $j's$ koordinater findes ved brug af de trigonometriske definitioner, som set ved tidligere rotationsmatricer.

$$j'_y = \cos(\alpha)$$

$$j'_z = \sin(\alpha)$$

Disse bliver fortsat opsat i vektorform.

$$j' = \begin{bmatrix} 0 \\ \cos(\alpha) \\ \sin(\alpha) \end{bmatrix}$$

Det samme gælder igen $k's$ koordinater.

$$k'_y = -\sin(\alpha)$$

$$k'_z = \cos(\alpha)$$

$$k' = \begin{bmatrix} 0 \\ -\sin(\alpha) \\ \cos(\alpha) \end{bmatrix}$$

Til sidst indsættes disse i en matrice, så en rotationsvektor findes.

$$R_x(\alpha) = [i \quad j' \quad k'] = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\alpha) & -\sin(\alpha) \\ 0 & \sin(\alpha) & \cos(\alpha) \end{bmatrix}$$

9 Referencer

1. **Lay, David C., Lay, Steven R. og McDonald, Judi J.** *Linear Algebra and Its Applications.* 5. Boston : Pearson Education, 2016. ISBN 978-0-321-98238-4.

2. **Golub, Gene H. og Loan, Charles F. Van.** *Matrix Computations.* 4. Baltimore : The Johns Hopkins University Press, 2013. ISBN 978-1-4214-0859-0.

3. **Matthäus, Wolf-Gert og Matthäus, Heidrun.** *Mathematik für BWL-Bachelor.* 4. Wiesbaden : Springer Fachmedien, 2015. ISBN 978-3-658-06206-4.

4. **Dunn, Fletcher og Parberry, Ian.** *3D Math Primer for Graphics and Game Development.* Boca Raton : Taylor & Francis Group, 2015. ISBN 978-1-4987-5989-2.

5. **Gordon, V. Scott og Clevenger, John.** *Computer Graphics Programming in OpenGL with Java.* Virginia : Mercury Learning and Information, 2017. ISBN 978-1-683920-4.

6. **Hughes, John F., et al.** *Computer Graphics: Principles and Practive.* Boston : Addison-Wesley, 2014. ISBN 978-0-321-39952-6.

7. **Madhav, Sanjay.** *Game Programming in C++: Creating 3D Games (Game Design).* Boston : Addison-Wesley, 2018. ISBN 978-0-13-459720-1.

8. **Singh, Parminder.** *Learning Vulkan.* Birmingham : Packt Publishing, 2016. ISBN 978-1-78646-980-9.

9. **Oualline, Steve.** *Practical C Programming.* Sebastopol : O'Reilly Media, 2011. 9781565923065.

10. **Nordfalk, Jacob.** *Objektorienteret programmering i Java.* Nærum : Globe A/S, 2013. 978-87-7900-811-3.

11. **Pharr, Matt, Jakob, Wenzel og Humphreys, Greg.** *Physically Based Rendering: From Theory to Implementation.* 3. Cambridge : Elsevier, 2017. ISBN 978-0-12-800645-0.

12. **Bejarano, Antonio Hernández.** 3D Game Development with LWJGL 3. *Frederick Institute of Technology.* [Online] 2017. http://staff.fit.ac.cy/eng.ap/FALL2017/3d-game-development-with-lwjgl.pdf.

13. **Vanka, S. Pratap, Shinn, Aaron F. og Sahu, Kirti C.** *Computational Fluid Dynamics Using Graphics Processing Units: Challenges and Opportunities.* Denver : ASME, 2011. doi: 10.1115/IMECE2011-65260.

14. **Luna, Frank D.** *3D Game Programming With DirectX 12.* Dulles : Mercury Learning and Information, 2016. ISBN 978-1-942270-06-5.

15. **Axon, Samuel.** Ars Technica. *The end of OpenGL support, plus other updates Apple didn't share at the keynote.* [Online] WIRED Media Group, 6. juni 2018. [Citeret: 13. december 2019.] https://arstechnica.com.

16. **Hocking, Joseph.** *Unity in Action: Multiplatform game development in C#.* Shelter Island : Manning Publications Co., 2018. ISBN 9781617294969.