

3D Render ProjektEt billede, der indeholder tekst

Automatisk genereret beskrivelseStudie Område Projekt i Matematik og Programmering

Alexander Hillerup Tuff  
3d2

9 December 2022

TEC H.C Ørsted Gymnasium Lyngby

Indholdsfortegnelse

Forside 2

[Resume 4](#_Toc120828042)

[Indledning 4](#_Toc120828043)

[Problemformulering delt op 5](#_Toc120828044)

[Hvordan virker en 3D render? 5](#_Toc120828045)

[Matematikken bag en 3D render2 6](#_Toc120828046)

[3 dimensional transformations matrix 8](#_Toc120828047)

[Transformation i 3D render 9](#_Toc120828048)

[Hvad er acceptable FPS? 9](#_Toc120828049)

[Ray-tracing og AI upscaling 9](#_Toc120828050)

[Undersøgelse af en 3D render 11](#_Toc120828051)

[Min 3D render 11](#_Toc120828052)

[Begrænsninger og Overvejelser 11](#_Toc120828053)

[Forsøg og resultater 11](#_Toc120828054)

[Konklusion 11](#_Toc120828055)

[Diskussion og Perspektivering 12](#_Toc120828056)

[Kilder 13](#_Toc120828057)

[Bilag 13](#_Toc120828058)

# Resume

Opgaven blabla…

# Indledning

Jeg har fået problemformuleringen: **Hvordan kan ray-tracing og AI upscaling hjælpe moderne spil og 3D renders til at give højere opløsning, med mere komplekse 3D figurer, uden uacceptable lave FPS?**  
 Jeg har i opgaven tænkt mig først at Redegøre for vigtige begreber og matematiske metoder til en 3D render, hvorefter jeg laver en 3D render i Java, da al min programmerings erfaring er i Java. Med mit program vil jeg undersøge hvor høj FPS jeg kan få på forskelligt hardware, for at vurdere og forklare udviklingen af moderne hardware. Til sidst i min opgave vil jeg med udgangspunkt af mine forsøgsresultater komme med en forklaring på grafikkortets udvikling og vigtigheden i dette, udover det vil jeg komme med et gæt på fremtidige grafikkorts udvikling og hvilke udfordringer der forekommer ved fremstilling af moderne hardware og hvorfor andre teknikker som AI upscaling er nødvendige til 3D grafik.

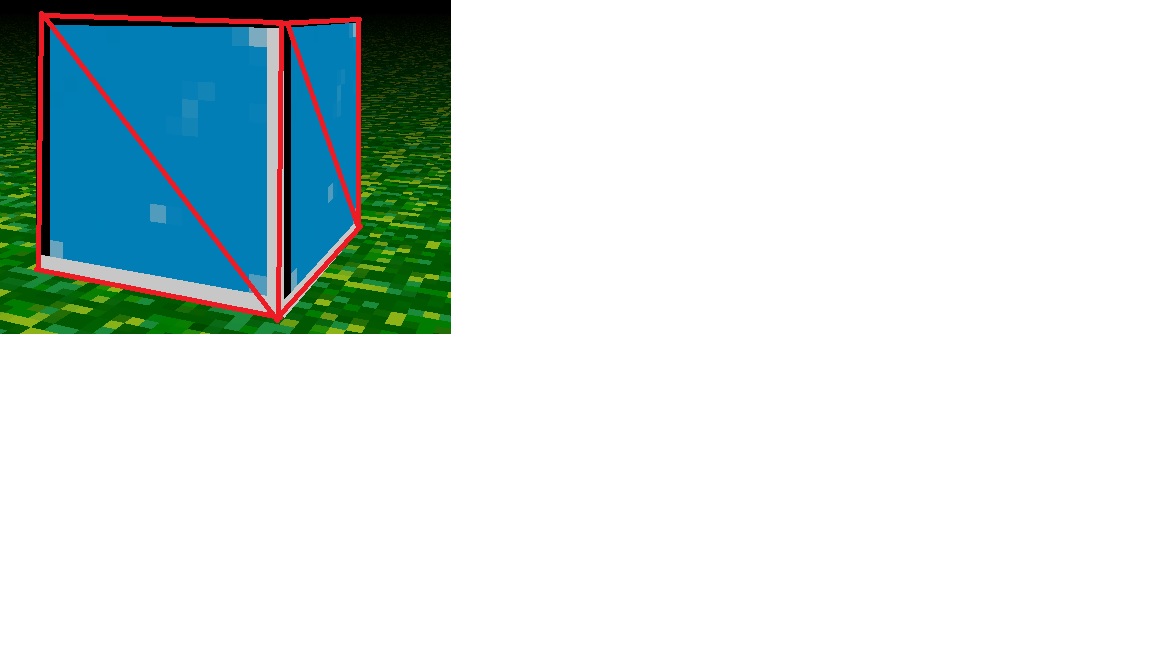
Ret til når du har skrevet opgaven

# Problemformulering delt op

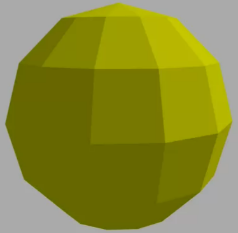
Som nævnt var min problemformulering: Hvordan kan ray-tracing og AI upscaling hjælpe moderne spil og 3D renders til at give højere opløsning, med mere komplekse 3D figurer, uden uacceptable lave FPS? - Jeg har delt den op i nogle mindre bider for bedre at forklare emnet.

## Hvordan virker en 3D render?

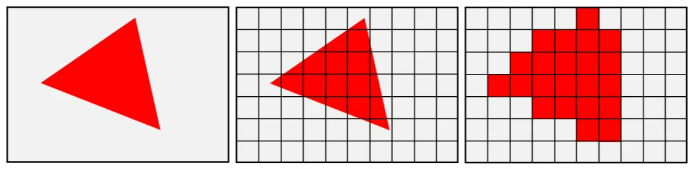
Alle 3D renders starter et ”sted”, det sted hvor alt har default værdier, ofte i midten af det 3D rum man udregner, altså i (0,0,0) i et 3 dimensionalt kartesisk koordinatsystem, Det kaldes for ”frame zero”, og ofte i spil og andre 3D renders vises det ikke men bliver bare regnet ud. Det vigtigste til at fremvise 3D objekter er en vertex eller verticies i flertal, her bruger man engelske begreber da de ikke eksistere på dansk. En vertex er et hjørne i en trekant, som bruges til at beskrive former, hvis man har en hel flad væg i sin 3D render bruges der 4 vertecies, men hvis man derimod har en kube bruges der 8 verticies, som sammen er 12 trekanter, alle 3D programmer bruger verticies til at udregne objekters positioner, både i programmet, men også i forhold til transformationer og kameraets position.3.1

   
figur 1: et eksempel på 6 verticies (og 4 trekanter) fra mit program

En kube er en meget simpel figur, sammenlignet med en kugle. Du kan måske forestille dig at det er svært at lave en kugle med trekanter, men det er så her hvor man skal vælge det rigtige antal verticies, for at få en flot kugle, men også for ikke at gøre programmet langsomt kig evt. figur 2. på figur 2 kan der tydeligt ses en forskel, både på grund af lyset på figuren, men også fordi antallet af verticies er meget højere i den ene.3.1 Hvis man så vil have figurer der skal ligne virkeligheden rigtig meget, så kan det være svært at køre programmet, med en god FPS. dog kan moderne computere fremvise spil med flere millioner af verticies, så for det meste er det ikke antallet af objekter og verticies, som kvæler en 3D renders hastighed.

  
figur 2: forskellen på antal af vericies vist på kugler.3.1

Når man har sin 3D verden så skal den selvfølgelig vises på skærmen, det gøres den ved at udregne hver enkelt pixels farve. Den laver de udregner ud fra hvor mange objekter der er i spillet, om de objekter er på skærmen, hvor langt væk de er (i mit tilfælde) og om der er en tekstur knyttet til det objekt. Det programmet så gør er den regner en RGB-værdi til hver pixel.3.1 I mit program har jeg lavet lyskilden til kameraet, så når objekter kommer væk fra kameraet, bliver de mørkere, indtil de forsvinder helt. Figur 3 visualiserer farvevalget på pixelsne. Det kaldes rasterisation når man laver 3D verdenen til 2D.

  
figur 3 viser hvordan pixels bliver farvet3.1

Hvis man gerne vil arbejde med 3D skal man normalt ikke skrive alt koden selv, mange spil bruger biblioteker, som laver alle de her steps, så man ikke selv skal holde styr på millioner af vertices, men bare de objekter man tilføjer. Når det handler om spil bruger man ofte Direct3D, OpenGL eller Vulkan, som alle er biblioteker, som laver alt grafik-render delen, man vælger bare selv hvad der skal være.3.1 Det smarte ved de biblioteker er at de benytter GPUen altså Grafikkortet, hvor mit program laver alle de her steps som der er i en 3D render, men gør det med CPUen.

## Matematikken bag en 3D render2

En 3D render bruger en masse matematik, men det mest interessante og det der står for hvor objekter er og bevæger sig, er transformationsmatrix. En matrix er i mit tilfælde en samling af vektorer, en vektor har en retning og en længde, og skrives på måden . X er den længde på x-aksen og y længden på y-aksen.

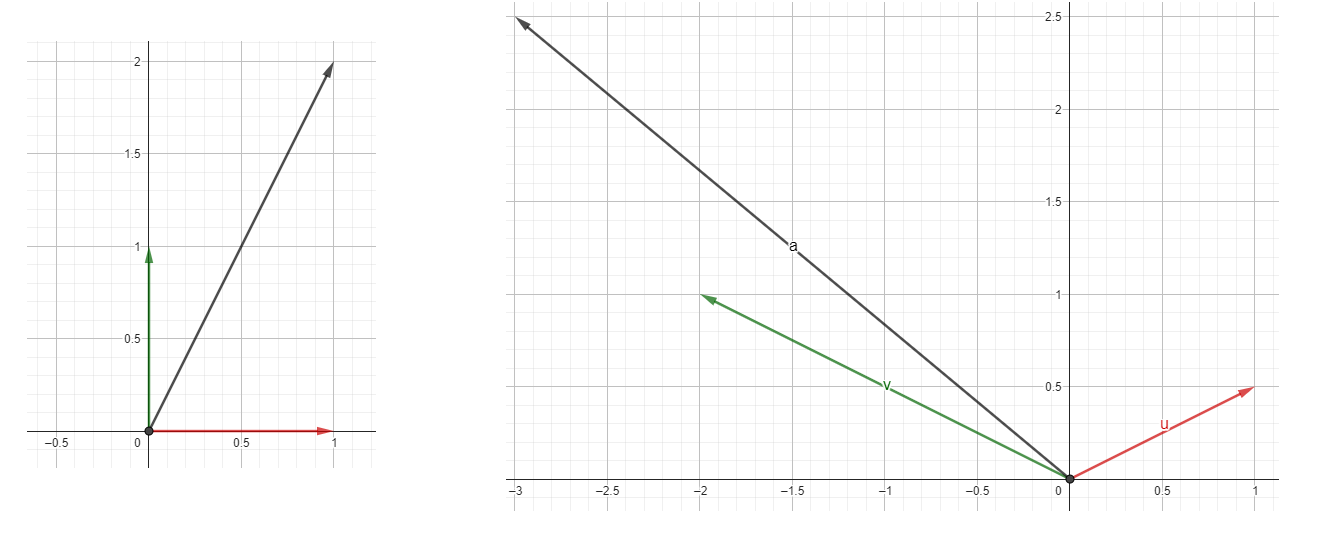
Før jeg kommer om den transformation matrix der sker i en 3D render skal man først forstå en lineær transformation i 2D. Transformation betyder simpelt set at vi giver et input og for et output, lidt ligesom en funktion, dog er det forskelligt fordi når det er en transformation er der en bevægelse knyttet til, så det man kigger efter, er ikke bare outputtet, men også den bevægelse hen til outputtet. Når det er en lineær transformation, er det også vigtigt at origo ikke ændre sig, og at alle linjer forbliver linjer uden at kurve, evt. se figur 4.1  
Et billede, der indeholder side om side

Automatisk genereret beskrivelseFigur 4: lineær transformation (blå linjer og fremhævede hvide) med originalt koordinatsystem i baggrunden

Måden hvorpå man regner en lineærtransformationen er ud fra basisvektor, som er enhedsvektorer som følger x-aksen og y-aksen

Man kan så efter en transformation aflæse deres position i det originale koordinatsystem og ud fra det se transformationen. Her er transfomrationen

Man kan ud fra de 2 basisvektorer udregne alle andre vektorer ved at skrive dem op som udtryk af basisvektor.

  
Figur 6: en transformation hvor og og en vektor som efter transformationen er i det originale koordinatsystem.

Man kan skrive og ind i en matrix ud fra deres koordinater, og matrixen er ikke andet end en måde at samle de to vektorer denne 2x2 matrix beskriver vores todimensionelle lineære transformation, den første kolonne beskriver hvor ender og den anden hvor .1

For eksempel

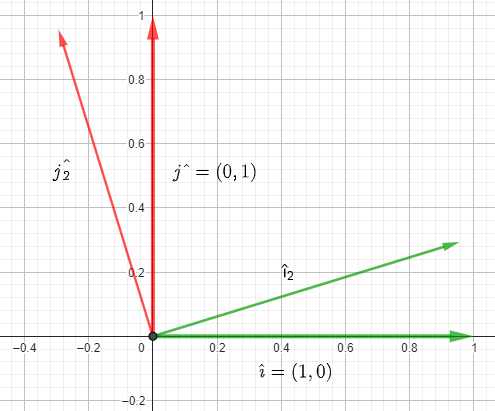
Nogle af de matricer som ofte forekommer, er identitet matrixen altså de originale værdier

Skalerings matrix, hvor er den skalering der sker på enten x- eller y-aksen

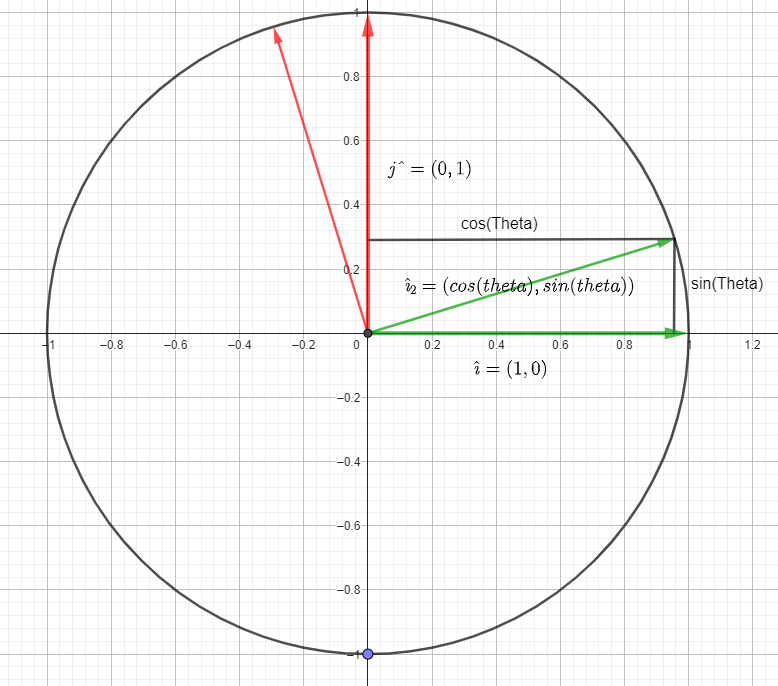
Rotations matrix, som bruger trigonometri fordi de to basisvektorer følger enhedscirklen, derfor vil skrives som , hvor er vinklen mellem aksen og vektoren. for er det

Rotationsmatrixen er

### Bevis for rotationsmatrix

Formlen for rotationsmatrixen virker måske lidt tilfældig, men det er den ikke, hvis man tegner de 2 basisvektorer op i et koordinatsystem og derefter deres rotation i samme koordinatsystem så kan man måske gennemskue rotationen.  
   
figur 7: rotationsmatrix i originalt koordinatsystem

Rotationen følger nemlig enhedscirklen altså trigonometri, hvis man så skriver værdierne ind for er det tydeligt hvorfor rotationsmatrixen ser ud som den gør.



Figur 8: enhedscirkel med basisvektor og deres rotation

Man kan se at for nemt kan beskrives ud fra sinus og cosinus, da den på x-aksen følger cosinus til vinklen (Theta). , man kan også tydeligt se at det samme sker for , dog er det omvendte akser og sinus går i negativ på x-aksen derfor kan den beskrives som

De to udtryk for vektorerne kan samles i en matrix som er rotationsmatrixen.

Translations matrix hvor man flytter koordinatsystem, hvilket gør den ikke er lineær, men fordi man bare tilføjer en vektor som beskriver hvor det nye nulpunkt er, så er det stadigvæk en lineærtransformation. Den vektor hedder og den er lig . Man har så matrixen der beskriver transformationen og en vektor som beskriver translationen. Man kan så sætte det sammen til en enkelt matrix

den her matrix har en ekstra række.

Det vil sige når man regner med en translation skal vektoren også augmenteres altså få en række mere. Med værdi 1 eksempelvis .

Hvis man så laver to transformationer, så skal man gange de 2 matricer sammen.

Matrix multiplikation er faktisk meget simpelt, fordi man læser det bare som en matrix ganget med vektorer, for eksempel hvis vi har en transformeret matrix så skrives den nye matrix op som hvor er den første kolonne i produktet af de 2 matricer. Den næste som er den anden kolonne

Derfor er

Et krav til multiplikation af matricer er at der skal være lige mange kolonner i den venstre matrix som rækker i den højre matrix. Der er også altid lige mange rækker i den første matrix som i den resulterende matrix, og der er lige mange kolonner i den resulterende som i den anden matrix. Rækkefølgen man ganger matricerne I er også vigtig. Den transformation som bliver lavet først, er den man ganger på.

### 3 dimensional transformations matrix

Den her lineære transformation er som nævnt kun i 2 dimensioner, hvis man skal lave det til 3D er det meget det samme.  
transformationen bliver til en 3x3 matrix som ser således ud

Hvor er basisvektor på z aksen1

Hvis man vil lave en translation, ser den således ud:

Igen kommer der en række mere på i bunden.

Identitetsmatrixen vil se således ud

Skalering matrix:

Rotationsmatrixen kræver en matrix for hver akse, det vil sige der er 3 matricer.

når man roterer i 2D er det z-aksen man roterer, derfor minder den om den fra 2 dimensioner.2

### Transformation i 3D render

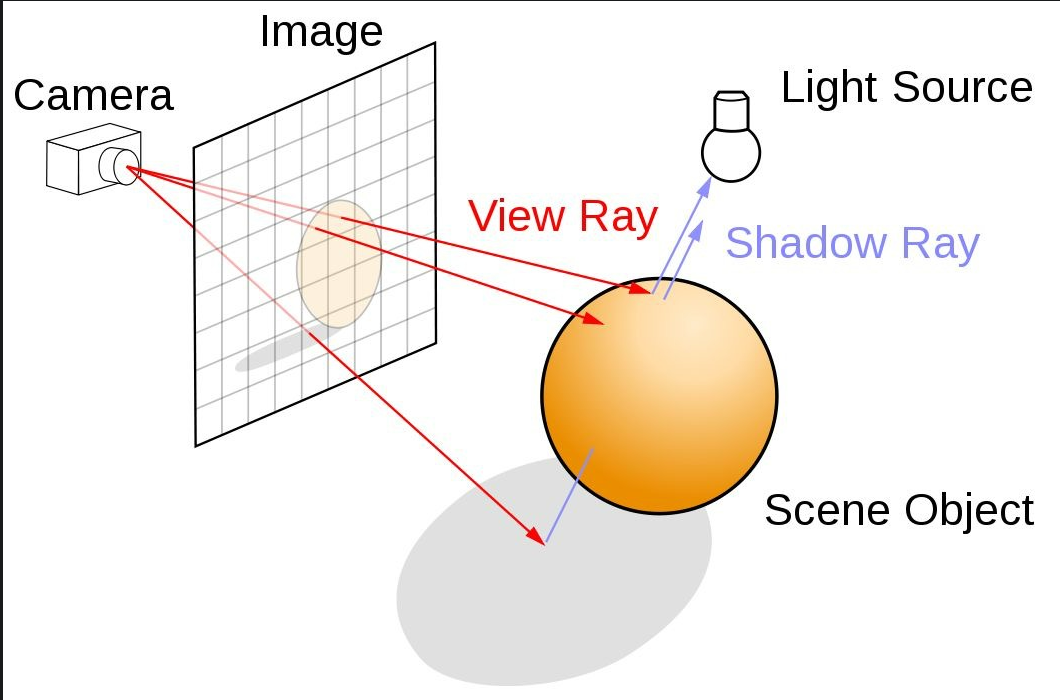
De her transformationer og udregninger bliver lavet af computeren ved hver frame, for eksempel i mit program så laver den de her udregner for min renderWall funktion, fordi væggen skal blive på sin position, imens kameraet flytter sig. Funktionen virker rimelig uoverskuelig både på grund af de mange udregninger, syntaksen, men også fordi der lægges ekstra værdier til da bevægelse i spillet også ændre kameraets position (i y-aksen) en smule. Jorden og Loftet (funktionen floor) i spillet bliver der også regnet en transformation, her er det primært en rotation om y-aksen fordi man kan dreje kameraet til højre og venstre.

## Hvad er acceptable FPS?

For at svare på problemformuleringen skal jeg definere hvad uacceptable FPS (Frames Per Second) er, der er mange forskellige holdninger til FPS, og det kommer også an på i hvilken sammenhæng. Ofte ved film og videoer, er 24FPS acceptabel, hvor i computerspil ville man ofte sige omkring 30-60 FPS, og da jeg mest har fokus på computerspil og selv godt kan lide høje FPS vil jeg vurdere acceptabel FPS som alt over 60, det vil sige at uacceptabel FPS er 59 og ned.

## Ray-tracing og AI upscaling

Ray-tracing er en metode, man bruger i stedet for rasterisation, forskellen er bare at man som navnet siger følger de forskellige lysstråler, det vil sige man laver en stråle fra kameraet igennem hver pixel hen mod lyskilden, og man følger så den stråle og tjekker om den rammer nogle trekanter. Ray-tracing giver derfor et meget mere realistisk billede (pga. lys og refleksioner), men det tager også meget mere computerkraft og tid, at gøre.4



Figur 5: viser 3 stråler, og deres vej til lys kilden aka ray-tracing4

Som det kan ses på figur 5, så går den ene stråle igennem objektet på scenen, hvilket vil gøre at jorden under den vil have noget af den gule farve og fremvise dens form (hvis jorden under er refleksiv). Den vil som det kan ses også vise skyggen4

Ray-tracing har ikke rigtig været i brug (i computerspil) fordi det er så langsomt, men nvidea har udviklet nye kerner (tensor-cores) i grafikkort som er specialiseret til at lave ray tracing, derfor er det blevet populært i moderne spil.

En anden ny teknologi, som nvidea har udviklet er DLSS (deep learning super sampling), som er en måde grafikkortet kan producere flere billeder (øge FPS), ved forskellige AI.  
Den første AI, som blev brugt, genererer pixels. Det vil sige at hvis man fx kørte sit spil i 1080p (fuld hd), så ville DLSS, kører spillet i 720p og lave de sidste pixels ved hjælp af AI, det vil gøre at computeren skal lave mange færre udregner hvilket giver beder FPS. den nyeste generation af DLSS (DLSS 3) kan gøre det endnu bedre, fordi den gør både det første step ved at render spillet i lavere opløsning og skalere op, men også fremstille 100% AI genererede billeder, hvilket ca. fordobler FPS, fordi hvert andet billede er lavet af AI, derfor kan computeren klare sig med kun at render hvert andet billede.5 Dog har det så en konsekvens, hvis man bruger denne ”Frame Generation” så vil man i nogle tilfælde kunne mærke at spillet ikke er ligeså flydende fordi ens bevægelse (med mus eller tastatur) ikke sker med det samme.

# Undersøgelse af en 3D render

Jeg har lavet en 3D render, som jeg vil undersøge på forskelligt hardware for at få en ide om hardware krav for en simpel 3D render, og for at understrege vigtigheden af et grafikkort til at lave udregningerne som bruges til fremvisning af grafik. For at teste forskelligt hardware har jeg skrevet et en benchmark funktion ind i mit program, når man starter programmet, kører den med det samme, den gemmer FPS hvert sekund, og den går rundt på en bane jeg har lavet. Programmet stopper efter et minut og skriver gennemsnits- og 10%lows FPS.

## Min 3D render

Jeg har skrevet min 3D render i Java 19 og har skrevet render koden selv, jeg har brugt nogle standartbiblioteker som er i Java. Programmet er skrevet efter de principper nævnt til en 3D render, jeg har også lavet nogle flowcharts til at visualisere programmet (idk bilag). Mit program er meget inspireret af en YouTube tutorial (kilde 6). jeg har lavet et flowchart for mit gameloop og for min ”render()” funktion, som bliver kørt inde i game loopet.

## Begrænsninger og Overvejelser

En af de problemer, som er ved computere, er decimaltal, hvis man bruger floats har man en begrænset mængde decimaltal, fordi floats ikke er præcise. En float har 32 bit hvor den mest til venstre (mest signifikante) styrer fortegn, de næste 8 (fra venstre af) bestemmer eksponenten, de sidste 23 bit er ”mantissa” som repræsenterer de stigende negative kræfter af 2. et eksempel fra teksten i kilde 7: “*if we assume that the mantissa is 1110000000000000000000, the value of this mantissa is calculated as follows:* *.*”. Det gør som sagt at ikke alle decimaltal kan repræsenteres, dog i mit program bruger jeg ”double”, som i bund og grund er det samme, dog er den lavet af 64 bit (doblet så mange altså navnet double), hvilket gør den mere præcis, men stadig vil dens værdi afvige. Dog vil det ikke betyde det store fordi der bliver lavet så mange billeder i sekundet at en enkelt pixels afvigelse ikke kan ses.7

en lille ting jeg har gjort for at forbedre performance, var jeg skiftede /255 (division med 255) ud med bitwise operatøren >> (shift right), som flytter bitene så langt til høje som man skriver efter for eksempel *x >> 8;* når man flytter en værdi 8 bit til højre virker det som division med 255, men det kører meget hurtigere og da jeg skrev det ind i koden gik min FPS op med 30%. denne division med 255 var til udregning af RGB-værdier for hver enkelt pixel.

## Forsøg og resultater

Jeg har undersøgt programmet på 4 forskellige computere, jeg har kørt programmet et par gange på hver for at se om resultaterne afviger eller om de er nogenlunde ens. De forskellige hardwareplatforme er her:

|  |  |  |  |
| --- | --- | --- | --- |
| Enhed \ Specifikationer | CPU | GPU | RAM |
| Computer 1 | Ryzen 7 5800x (3,8GHz) | Nvidea RTX 3060 12GB | 16GB 3200MHz |
| Computer 2 (laptop) | intel I5-11300H (3,1GHz) | Nvidea GTX 1650 4GB | 8GB 3200MHz |
| Computer 3 | Ryzen 5 3600  (3,6GHz) | Nvidea GTX 1060 2GB | 8GB 3200MHz |
| Computer 4 (laptop) | Intel i3-3227U (1,9GHz) | Intel HD Graphics 4000  (CPU) | 4GB ????MHz |

Selvom Grafikkort ikke bliver brugt direkte i min 3D render, så bruger computeren stadig grafikkortet til at fremvise billedet på skærmen.

## Konklusion

# Diskussion og Perspektivering

# Kilder

1. YouTube, two blue one Brown, Linear transformation and matricies <https://www.youtube.com/watch?v=kYB8IZa5AuE&t>   
2. YouTube, FloatyMonkey, Matricies and Transformations - Math for Gamedev  
<https://www.youtube.com/watch?v=HgQzOmnBGCo&t=2s>

3. Techspot, Nick Evanson, 3D game Rendering 101:   
 3.1 <https://www.techspot.com/article/1851-3d-game-rendering-explained/>   
 3.2 <https://www.techspot.com/article/1857-how-to-3d-rendering-vertex-processing/>   
 3.3 <https://www.techspot.com/article/1888-how-to-3d-rendering-rasterization-ray-tracing/>

4. nvidea, nvidea, Ray Tracing:  
<https://developer.nvidia.com/discover/ray-tracing>

5. nvidea, Henry C Lin & Andrew Burnes, nvidia dlls3: AI-Powered performance…  
<https://www.nvidia.com/en-us/geforce/news/dlss3-ai-powered-neural-graphics-innovations/>

6. YouTube, The Cherno, 3D game programming in Java: <https://www.youtube.com/playlist?list=PL656DADE0DA25ADBB>

7. ScienceDirect, Dogan Ibrahim, Floating Point Numbers  
<https://www.sciencedirect.com/topics/engineering/floating-point-number>

# Bilag