Ray Tracing i Realtid: Vil Moore's Law blive Indhentet af Machine Learning?

Forfatter:
Mads Evander Jensen

Vejledere:
Bo Larsen & Niels Hansen Aarøe

Studieretningsprojekt i Matematik og Informationsteknologi
Anslag: ~34700 Tilsvarende sider: ~14.5
H.C. Ørsted Gymnasiet Ballerup
Klasse: 3C. Matematik A & IT B
Deadline: 20. december, 2018

Abstract

Ray tracing is a method of rendering 3d scenes, but it’s special because it doesn’t use any programming tricks or effects. Ray tracing is mathematically modeled after how light behaves in real life. This is why, all modern 3d graphics are rendered with this technique, except real time applications. Real time applications require that the scene is rendered at least 24 times a second. Because of this, all modern real time applications are forced to use rasterization or any other similar method. We are about to break this barrier though and finally begin using ray tracing in real time applications. Not because the processors and graphics cards are becoming faster, but because we are using neural networks to help render ray tracing faster. I want to compare these methods of rendering and explain why ray tracing is such an amazing technology. I will show how ray tracing is done with some examples of vector calculation. All calculations in this study will be done in Maple 2018. One important comparison I will make between the two rendering methods is their hidden object removal. This comparison is interesting because ray tracing in and of itself is a hidden object removal method. I will of course also look at the difference in reflection quality, since this effect has been very hard to replicate with cheap tricks. I will conclude that, with nVidia pushing machine learning GPUs, the progress is faster than ever and that we will definitely see some real time ray tracing in the coming years. Early examples of machine learning rendered ray traced scenes are already beginning to appear and these immature technologies can only become better with time. Though we will probably not see real time ray tracing done with CPUs before engineers get quantum processors to work.

Indholdsfortegnelse

Indledning

Ray tracing findes rigtig mange steder, men man tager det tit for givet at der bag al 3d grafik er forskellige metoder til at rendere et billede. To af de mest brugte metoder til at rendere 3d billeder er ray tracing og rasterization. Ray tracing er den ældste og modelleret efter hvordan lys bevæger sig i virkeligheden. Fotoner i virkeligheden bevæger sig fra en lyskilde og ind i øjet, dette princip har man duplikeret i ray tracing for at skabe billeder der ligner virkelighed. Rasterization derimod er ikke ligeså pænt og ligger på et grundlag af en masse tricks og grafik snyd, men man bruger det stadig fordi det er mange gange hurtigere. Så hurtigt at man bruger det til videospil, hvorimod ray tracing bliver brugt til film og reklamer. Disse to typer har været standarden i mange år og først nu er vi ved at se de første applikationer der bruger ray tracing i realtid. Men hvordan virker dette ray tracing matematisk, hvilke forskelle er der egentligt i forhold til rasterization og hvordan har vi opnået at få de første ray tracing applikationer til at køre i realtid?

Problemformulering

Ray tracing har alle set på et eller andet tidspunkt, men sikkert uden at vide det. Disse metoder til at rendere billeder bliver taget for givet.

I dette projekt vil jeg udregne nogle af de forskellige metoder brugt til ray tracing, derudover vil jeg sammenligne ray tracing med rasterization der er den anden mest populære måde at rendere billeder, men af helt andre årsager. Til sidst vil jeg analysere den moderne ray tracing der skal kunne implementeres i real tid.

Krav:

- Redegør for emnet Ray tracing og kom kort ind på brugen af matematiske modeller, som vektor og matriceberegninger.

- Sammenlign Ray tracing med en anden form for 3D til 2D projicering og analyserer fordele og ulemper ved begge metoder.

- Udfør en ’hidden surface removal’ proces, på et selvvalgt 3D-legeme med begge algoritmer.

- Diskuter og vurder med baggrund i dine analyser, fordele og ulemper ved de førnævnte algoritmer.

Introduktion til Ray tracing

Ray tracing er en metode til at rendere billeder i 3D. Metoden går ud på at man fra sit virtuelle kamera sender en masse 'rays' ud der rammer de forskellige objekter i scenen. Denne information kan man så bruge til at vise hvor lys og hvilken farve en pixel skal have. Dette bliver blandt andet brugt i spillefilm(CGI), Disneys tegnefilm og 25% af IKEA's reklame kataloger. Grunden til at man bruger ray tracing er fordi det er en fysisk korrekt måde at rendere billeder, dette gør så billederne kan blive ultra realistiske. Kilde[1, 2]

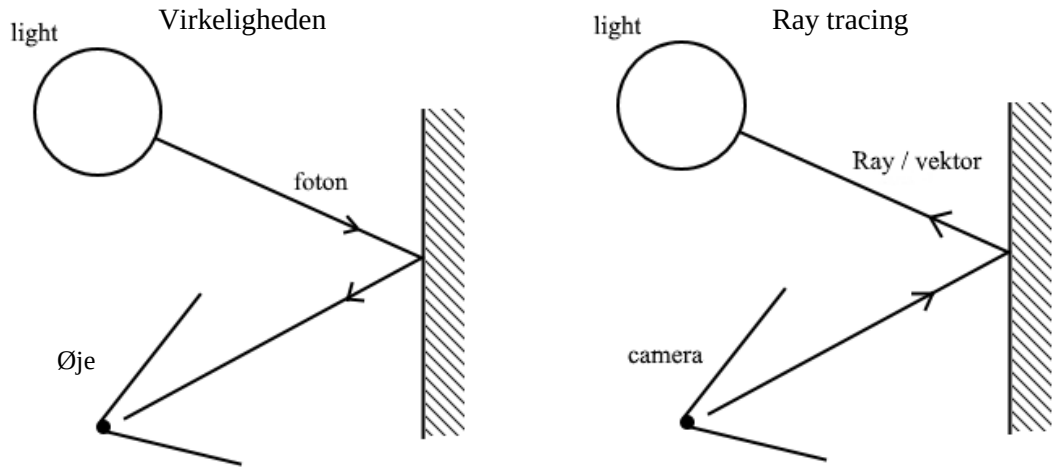
(fig. 1. Ray Tracet billede fra IKEA-reklame katalog)

Grunden til at ray tracing er fysisk korrekt, er fordi det modellerer fotoner fra virkeligheden, idet at fotoner også bliver affyret i en retning og rammer forskellige objekter inden det remmer vores nethinde og vi ser hvad fotonerne sidst ramte ind i. I matematikkens verden er disse rays bygget op af vektorer. Inden for vektorregning er der mange forskellige udregninger man skal lave for at kunne opbygge en rigtig ray tracing engine.

1) 25% of IKEA catalogues will have CAD renders instead of real products
https://blog.grabcad.com/blog/2012/08/25/25-of-ikea-catalogues-will-have-cad-renders-instead-of-real-products/
2) Disney explains why its 3d animation looks so realistic
https://www.engadget.com/2015/08/02/disney-explains-hyperion-renderer/

Vektorer til ray tracing

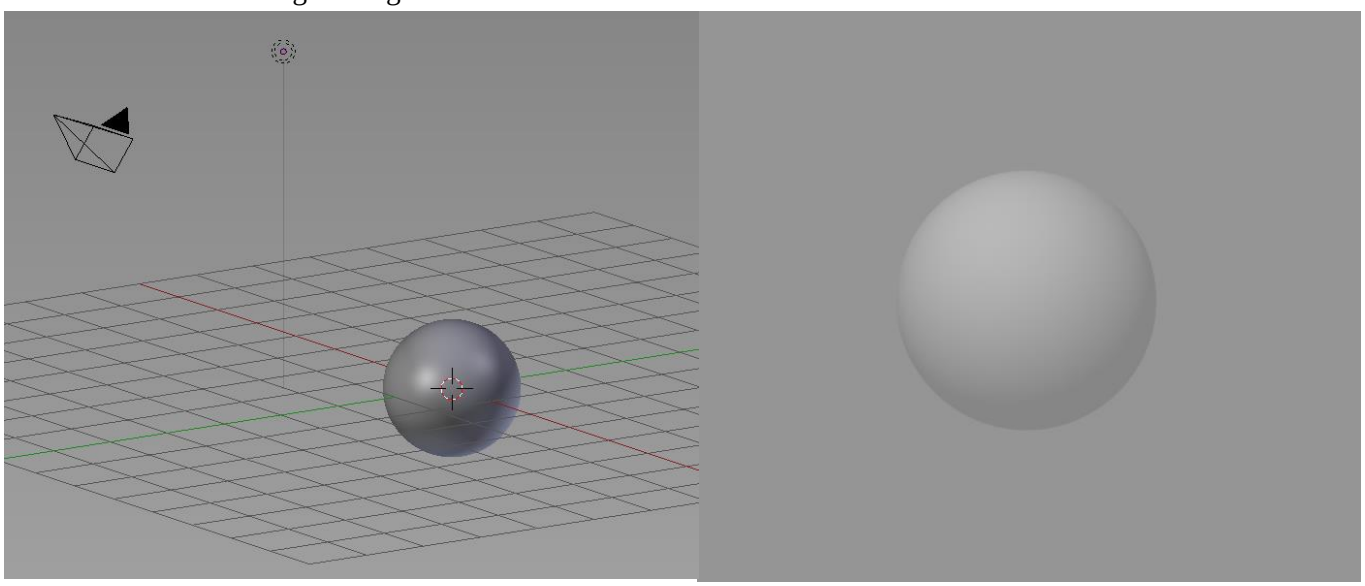
En vektor beskriver en retning og en længde. Dette er en vigtig egenskab siden at vi så med vektorer kan beskrive fotoners bevægelse. En foton bliver ’skudt’ ud fra et lys, det rammer et objekt og bliver reflekteret. Afstanden fra lyset og hen til objektet kan man matematisk beskrive som en vektor.



(fig. 2. Diagram af fotoner og rays bevægelse, egen kreation[Paint.Net])

I virkeligheden, bliver der kastet utallige fotoner fra lyskilder. Men i ray tracing gør man det omvendt, man ’kaster’ rays eller vektorer fra kameraet ind i scenen og så finder man der hvor de rammer noget lys. Dette gør så man ikke kaster millioner af rays ud fra lyskilder der aldrig vil ramme ens kamera. F.eks. kunne vi i en enkelt virtuel scene have et kamera, en kugle og et lys. Hvis man rendere billedet vil der blive skudt mange rays ud fra hver pixel i ens billede, disse rays rammer måske forbi kuglen og ud i ingenting, andre rays rammer kuglen og bliver reflekteret. Nogle af de reflekterede rays rammer en lyskilde og den information bruger vi til at vise at kuglen bliver lyst op.

Billede af scene og endelige billede:



(fig. [3, 4]. 3D-software og rendering, egen kreation[Blender 3d])

Den vigtigste vektor udregning der bliver brugt i ray tracing er refleksionen. Det giver sig selv at det er når vi kaster en ray og den så rammer et objekt der kaster den i en ny retning. Formlen for refleksioner er:

Vektor refleksioner

i_e er indfaldsvektorens enhedsvektor
n_e er overfaldets normalvektors enhedsvektor
r er den reflekterede vektor

$$r := i_e - 2\left(i_e \bullet n_e\right)n_e$$

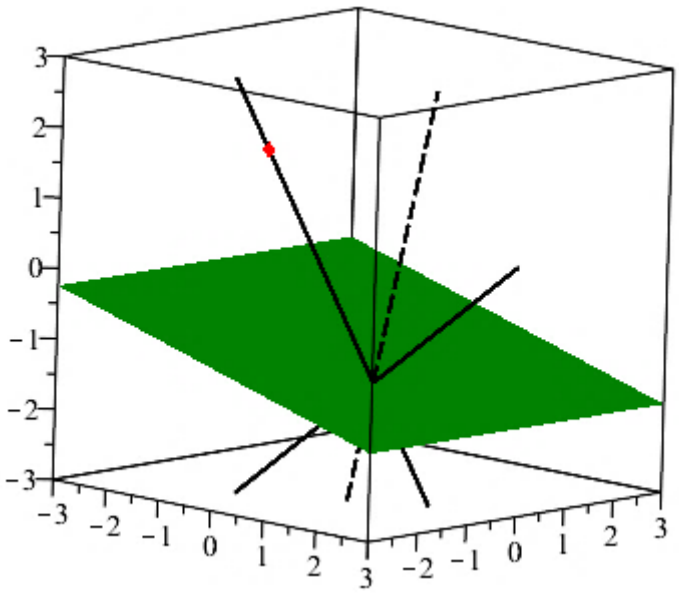
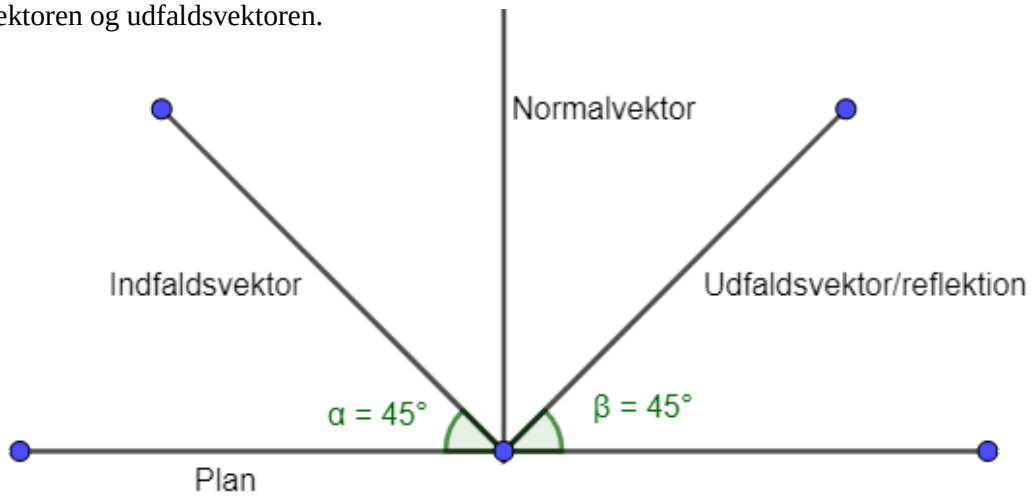


fig. 5. Maple afbildning af refleksion, egen kreation[Maple 2018]) Animation:
https://drive.google.com/file/d/1WJl_ZS3zRRDC5j6sKyh92RFLPy-YQW0K/view?usp=sharing

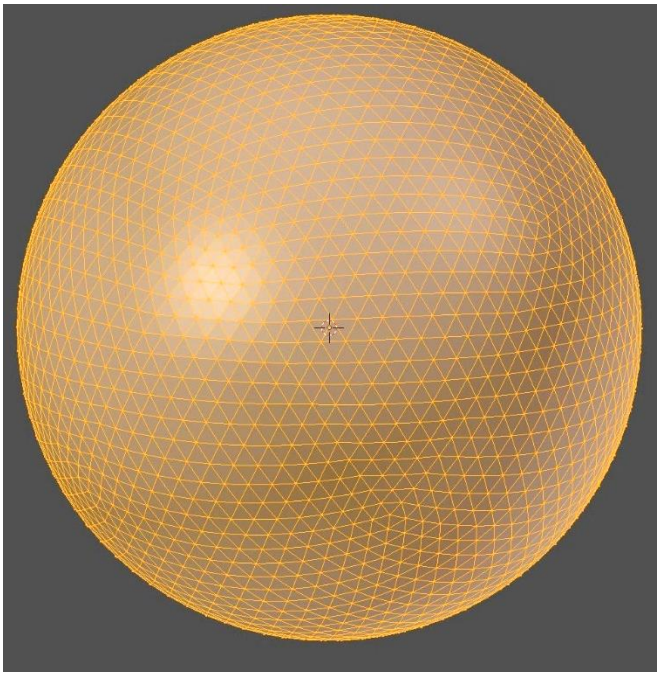
På billedet er den røde prik vores virtuelle kameras position. Kameraet kaster mange vektorer, og jeg vælger at kigge på en af de mange vektorer. Den stiplede linje er normal vektoren af planet, og den sidste linje er udfalds refleksions linjen, eller 'r's linje. Til denne udregning har jeg brugt formlen for vektor refleksioner oven over billedet, og selvom at formelen ser lidt avanceret ud, har man brugt nogle simple principper til at få den. For en refleksion er bare en betegnelse for når en indfaldsvinkel rammer noget og bliver kastet tilbage med samme udfaldsvinkel. Det er derfor man bruger planets normal vektor, siden at den også vil være indfaldsvektoren og udfaldsvektorens halveringslinje. Altså skal vinklen mellem indfaldsvektoren og normalvektoren være lige så stor som vinklen mellem normalvektoren og udfaldsvektoren.



(fig. 6, Diagram over refleksionsprincippet, egen kreation[GeoGebra])

Linjer og Planets Parameterfremstilling

Det er vigtigt her at forstå at alle 3D objekter er ombygget af små flade overflader. Selv kuglen i de tidligere billeder er opbygget af små trekanter:



(fig. 7, En kugles vertex, egen kreation[Blender 3d])

Alle disse trekanter er selvfølgeligt opbygget af tre punkter. I et stort objekt kalder vi punkterne for vertex og hvis objektet ikke er en defineret figur, så som en kugle, kalder vi dem for polygoner.

Tre vertex laver en enkelt overflade som vi kan behandle matematisk. For hvis man har tre punkter i et tredimensionelt koordinat system kan man konstruere et plan og kun et plan. F.eks. i mit tidligere eksempel hvor jeg udregner en refleksion fig. 5. Starter jeg kun med tre punkter og et kamera. Altså alt den information man også ville have i et rigtigt 3d software.

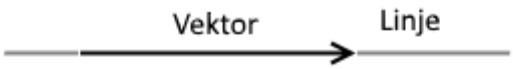
Igennem mine udregninger senere vil jeg flere gange bruge et plan konstrueret af 3 punkter. Dette er for at simulere hvilke værdier en programmør i virkeligheden ville have mulighed for at få:

Vi starter med tre vertex i en scene

$$P_1 := \langle 1, 0, -1 \rangle :$$
$$P_2 := \langle 0, 1, -1 \rangle :$$
$$P_3 := \langle 0, 0, -1 \rangle :$$

Disse tre vertex skal behandles så vi har et plan vi kan arbejde med. Vektorer kan bruges til at bygge både linjer i 3d og planer. Det er her formålet med vektorer kommer ind, for dette er nemlig ikke muligt med normal geometrisk konstruktion at konstruere linjer og planer i 3d, selvom det er muligt i 2d. Disse konstruktioner med vektorer kalder man for parameterfremstilling, fordi de bruger parametre til at skalere vektorer i en given retning. F.eks. skalerer man en vektor med en variabel for at få en linje, her er længden af vektoren irrelevant, men retningen er essentiel. Det samme kan siges om parameterfremstilling af et plan. Her bruges bar to variable og to retningsvektorer.

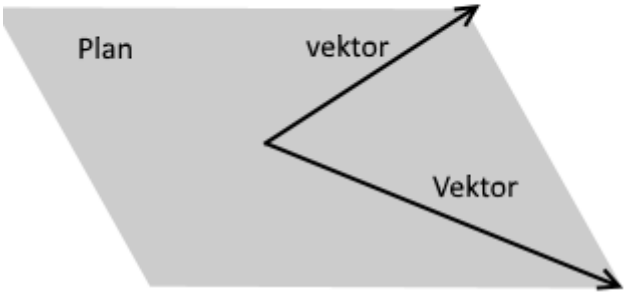
Et plot af dette princip kunne se sådan ud:
En parameterfremstilling af en linje kunne være dette:



$$Vektor := \langle 1, 1, 1 \rangle :$$
$$Linje := Vektor \cdot t :$$

Her har vi en vektor, og variablen 't'. Fordi vektoren bliver skaleret uendeligt i begge retninger, bliver resultatet en linje.

En tilsvarende parameterfremstilling af et plan ville være:



(fig. 8, Skitsering af parameterfremstilling, egen kreation(Paint.Net)

$$Vektor_1 := \langle 1, 0, 0 \rangle :$$
$$Vektor_2 := \langle 0, 1, 0 \rangle :$$
$$Plan := Vektor_1 \cdot t + Vektor_2 + s :$$

Her er der 2 vektorer der hver bliver skaleret.

I virkeligheden beskriver en parameterfremstilling en masse punkter som vektorerne kan ramme. Og selvfølgeligt kan vi også rykke rundt på linjerne og planet med stedvektorer. Men hvordan laver vi så et plan kun med 3 vertex.

$$P_1 := \langle 1, 0, -1 \rangle :$$
$$P_2 := \langle 0, 1, -1 \rangle :$$
$$P_3 := \langle 0, 0, -1 \rangle :$$

Imellem tre punkter kan man altid skabe et plan.

$$n_{plan} := P_1 + (P_2 - P_1) \cdot t + (P_3 - P_1) \cdot u :$$

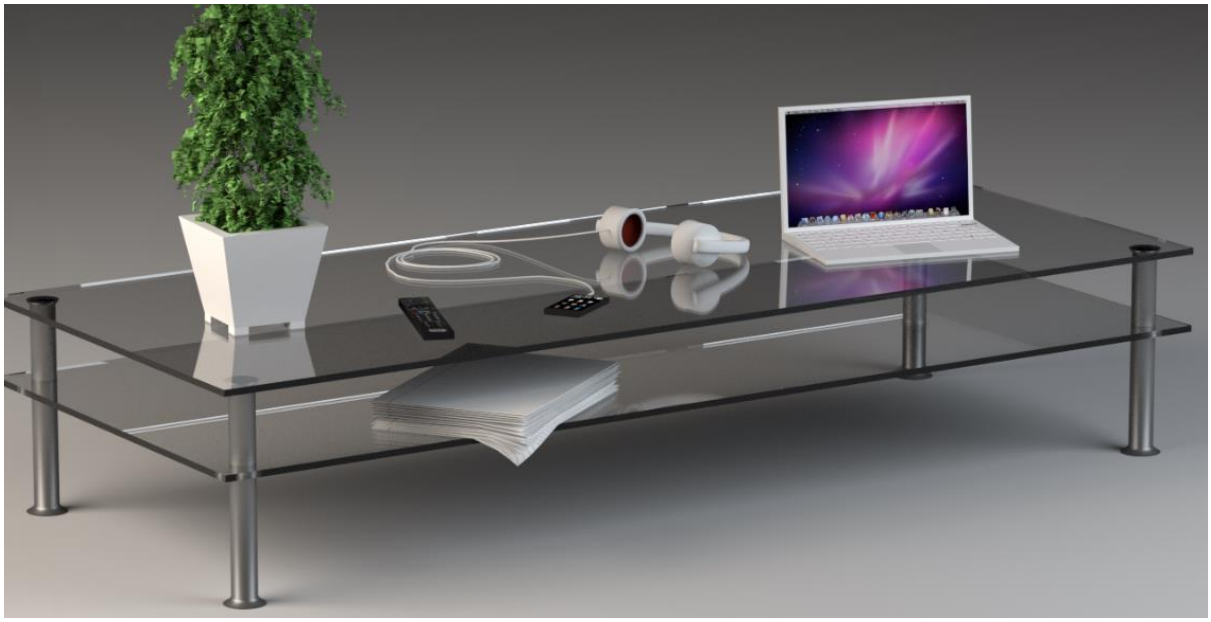
Her bruger jeg 2 af punkterne som retningsvektorer og et som en stedvektor.

- P_1 er stedvektor så den skal ikke ganges med nogen variabel og står alene.
- P_2 er den første retningsvektor, så jeg parallelforskyder den i stedvektorens retning og ganger resultatet med den første variabel
- P_3 er min anden retningsvektor i planet. Jeg parallelforskyder igen med stedvektoren og ganger med min anden variabel.

Dette skaber planet: $$n_{plan} := \begin{bmatrix} 1 - t - u \\ t \\ -1 \end{bmatrix}$$

Alle punkter ligger i planet, så planets udregning er korrekt. Denne måde at regne planer ud fra vertex skal gøres for hver overflade, det er også derfor der tit i gammel grafik blev sparet på hvor høj opløsning kugler var, eller hvor mange vertex der var i en kugle og at de derfor tit så 'kantede' ud.

Sammenligning af ray tracing og rasterization



(fig. 9. 3D grafik, egen kreation[Blender 3d])

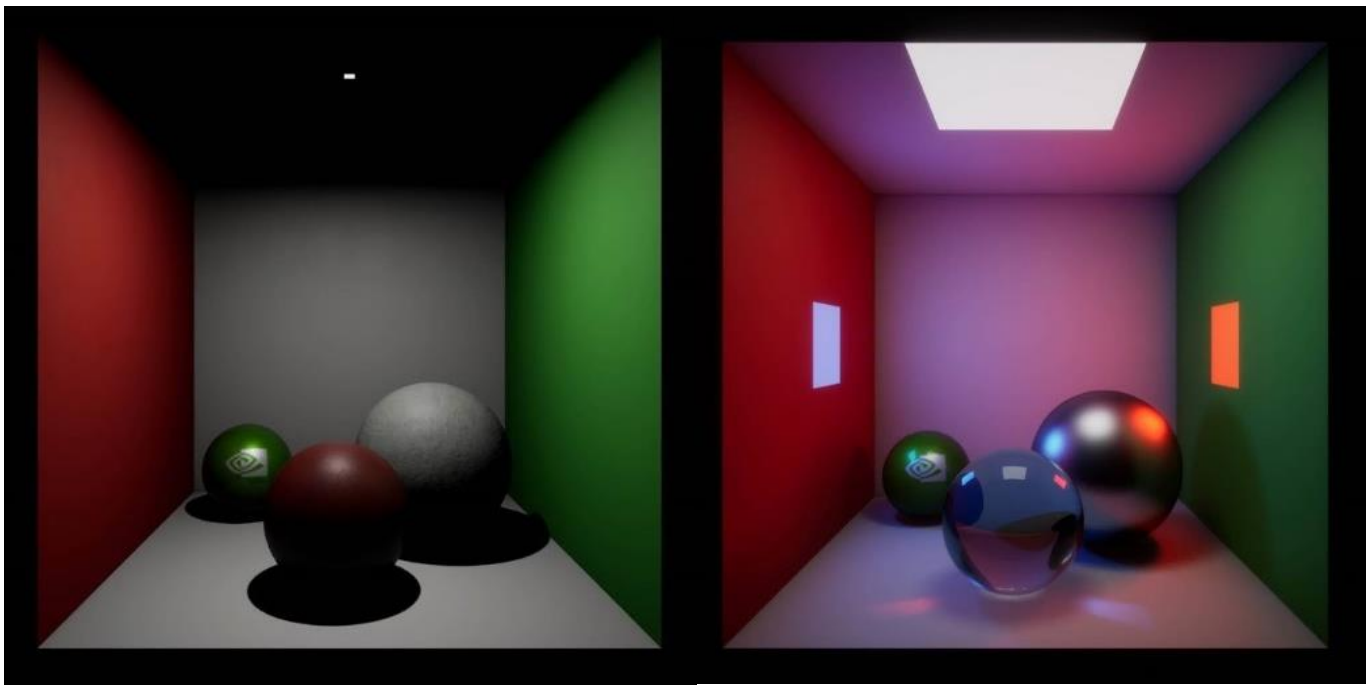
Læg mærke til bordets ben og bordets overflade, her bliver skygger og refleksioner vist fysisk korrekt. Dette er fordi billedet er renderet med ray-tracing, en teknik der bruger utrolig meget computerkraft men levere realistiske billeder. Det er af denne årsag at man bruger ray tracing til at rendere 3d film, og effekter til film som f.eks. robotterne i Transformers. Rasterized Graphics er den moderne metode til at rendere 3d grafik i realtid, det er dog overhovedet ikke fysisk korrekt, siden at den bruger mange smarte tricks til at vise refleksioner og skyer. En sådan effekt kunne være SSAO, der laver falske skygger i en 3d scene ved at udregne længden mellem overflader og gøre de tætteste overflader mørkere, dette gør at hjørner, små render, græs, mm. bliver renderet mere realistisk.

Sammenligning mellem SSAO og uden SSAO: (læg mærke til bunden af tønden der ligger ned)

(fig. 10. Sammenligning af AO-metoder, eget skærmbilleder[Battlefield 1])

I en sammenligning kan man se at effekten gør billedet pænere, dog er dette, som tidligere nævnt, ikke fysisk korrekt. Det er lignende effekter der gør refleksioner mulige i raster graphics, men igen er de ikke fysiske korrekte. Det er af disse grunde at man gerne vil rykke væk fra rasterized graphics og bruge ray tracing på en større skala, f.eks. i computerspil.

Direkte sammenligning af de to metoder:



(fig. 11. nVIDIA's egene renderet billede der sammenligner rasterization og ray tracing, Gamescom

Selvom at dette billede er lavet af en kommerciel virksomhed, designet til at vise det ray tracede billede bedre end det egentligt er. Er det stadig en god sammenligning der kan bruges til at forstå forskellene mellem de to renderings metoder. Billedet til venstre, dvs. billedet der er blevet renderet uden ray tracing, er fladt, uden refleksioner og med hårde skygger. Derimod er billedet med ray tracing mere farverigt. Lyset bliver kastet korrekt rundt i rummet, og refleksioner og refraktioner er fysisk korrekte. Grunden til at man bruger rasterized graphics er simpelthen fordi det er hurtigere at køre på en computer, faktisk mange gange hurtigere, især når scenen man vil rendere bliver mere kompleks. Hvis man f.eks. vil rendere et billede af en hel by, skal man rendere gade lys, billygter, neon lys, måske skinner solen, alle disse lys bliver reflekteret af bygninger, folk, andre biler, osv. Alt dette skal ens computer udregne.

Pixar film bliver renderet på en supercomputer med over 2000 meget kraftige processorer. Hele denne supercomputer tager 29 timer om at rendere **et enkelt** billede. Hvis en enkelt processor skulle rendere hele filmen ville det tage over 10.000 år og med deres supercomputer tog det stadig et par år. Så der skal rigtig meget kraft til for at rendere billeder med ray tracing (i hvert fald hvis vores standard er Pixar). Det er af denne grund at man bruger rasterization til rendering af alt der er i realtid, det er altså alt fra Google Earth til videospil. kilde[3]

Målet fremover er altså primært at gøre hastigheden af renderingen hurtigere så film kan renderes i højere kvalitet og spil kan bruge ray tracing i realtid. Tænk hvis IKEA's reklame blade var virtuelle og at man selv kunne udforske deres produkter virtuelt i realtid. Dette er et eksempel hvor ray tracing kunne blive brugt i realtid, men vi har bare ikke computerkraft til at rendere en sådan applikation.

3) How Pixar made Monsters University, its latest technological marvel
https://venturebeat.com/2013/04/24/the-making-of-pixars-latest-technological-marvel-monsters-university/

Hidden surface removal

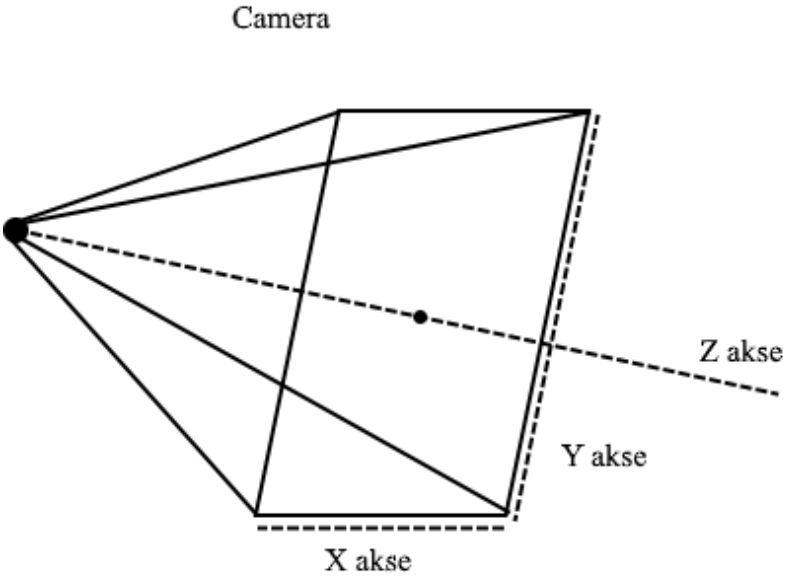
Ray tracing er som sagt en fysisk korrekt måde at rendere billeder, og der bruges mange forskellige vektor udregninger der gør processen utrolig langsom, men rasterization kan også opnå flotte billeder, dog med noget snyd og restriktioner og alligevel når man ikke det samme niveau af realisme. Eksempler hvor det er tydeligt at ray tracing er mere realistisk er altid eksempler af refleksioner eller lignende, men et problem som ray tracing løser bare på grund af den måde ray tracing er bygget op på er 'Hidden surface removal'.

'Hidden surface removal' er et problem der skal løses i alt rasterized grafik. Problemet går ud på at finde ud af om et objekt eller dele af et objekt kan blive set fra det virtuelle kamera. F.eks. hvis vi har to objekter i en scene der overlapper hinanden, hvilke dele af objektet skal man så vise. Det samme kan man sige om objekter der ikke er inde for kameraets synsvinkel, hvornår ved man om et objekt skal vises eller ej. En forkert måde man kunne løse problemet, ville være kun at rendere objekter der har deres center inde for kameraets synsvinkel. Men hvad så hvis man har et kæmpestort bjerg i baggrunden man vil rendere og man kigger på bjergets side, så centeret er ude for synsvinklen. Så forsvinder bjerget magisk og her opstår problemet man skal løse med 'hidden surface removal'. Hvis to objekter overlapper og man ikke har nogen 'hidden surface removal' algoritme til at fjerne overflader man ikke burde kunne se, vil det se ud som om objekterne er gennemsigtige. Selv et enkelt objekt som en kugle har en bagside man ikke burde kunne se. Til alle disse typer objekter skal man have nogle algoritmer der udregner hvad der er indenfor kameraets synsvinkel. Det er af grunde som denne at ray tracing er så fremragende. Metoden er ligeglad med objekter kameraet ikke kan observere og derfor opstår problemer med 'hidden surface removal' aldrig.

Der er mange populære og vidt forskellige metoder til at fjerne gemte overflader, men den mest populære er med Z-Buffer. Z-Bufferen er faktisk så populær at alle moderne computere har en indbygget hardware chip der udregner Z-Bufferen. Selv mobiltelefoner har denne teknologi indbygget i dens hardware. Kilde[4]

Hvis vi siger at kameraet er en flade man kigger igennem, så vil længden fra den ene side til den anden være X og fra bund til top Y, derfor er Z aksen, ud fra kameraet og ind i scenen.

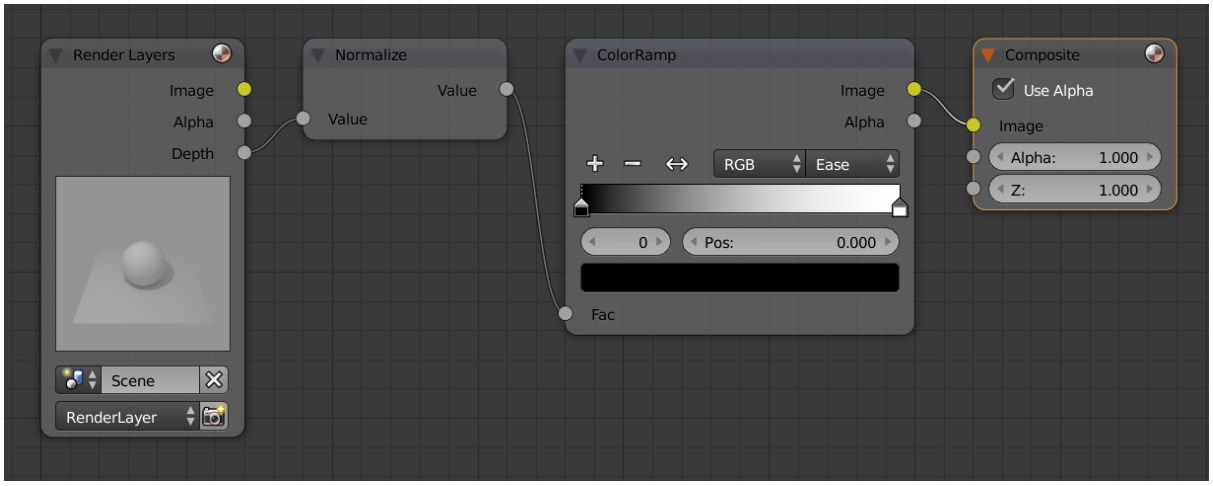
Hver eneste pixel som kameraet skal vise har dens egen z-akse, eller retning. Z-Buffering eller Z-Culling virker på den måde at vi for hver pixel finder afstanden fra kameraet til objekterne igennem Z-aksen. Dette gør os i stand til kun at vise de objekter i scenen der har punkter tættest på kameraet. Dette kan man vise ved at rendere Z-Bufferen med en gråskala, med afstanden til kameraet som parameter.



(fig. 12. Forklarings diagram af et virtuelt kameras akser, egen kreation[Paint.Net])

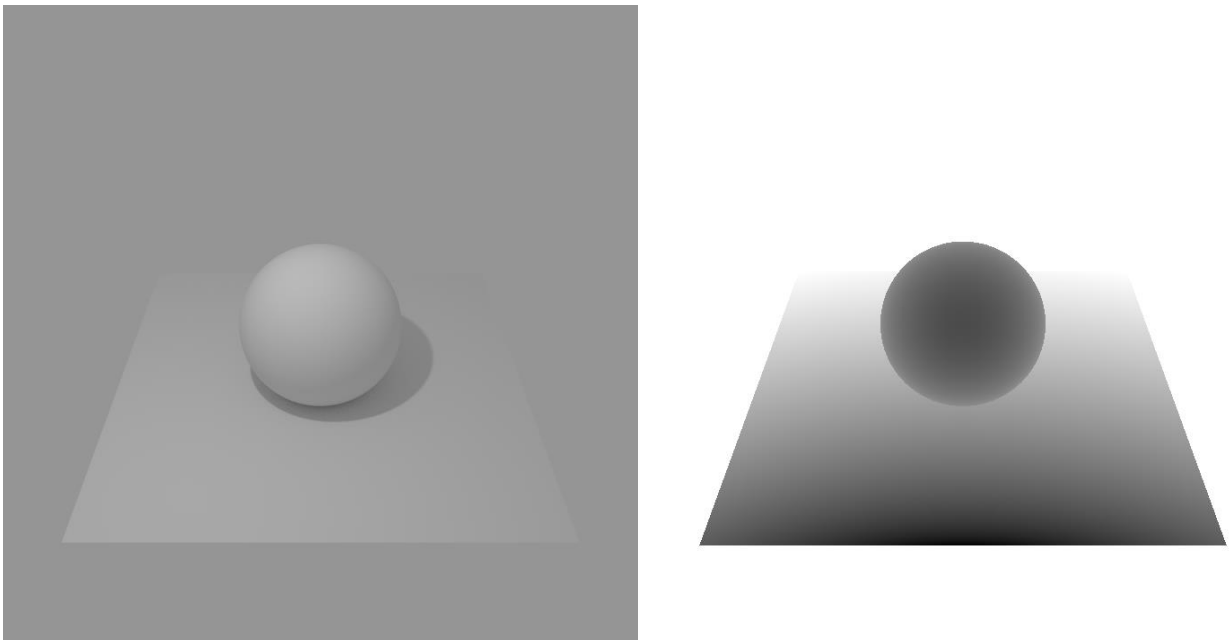
4) Visibility in Computer Graphics (3.2.1)
https://www.cg.tuwien.ac.at/research/publications/2003/Bittner-2003-VCG/TR-186-2-03-03Paper.pdf

Her kan man se mit setup for at vise Z-Bufferen, i blender kalder de den bare ’Depth’ for brugervenlighedens skyld.



(fig. 13. Blender nodes indstillinger for at rendere Z-bufferen, egen kreation[Blender 3d])

Output rendere Z-Buffer



(fig. 14. Rendering og Z-bufferen, egen kreation[Blender 3d])

Hidden object removal video demonstration:
https://i.kinja-img.com/gawker-media/image/upload/ucoln8kedwfglsrlxvm5.mp4

Hele denne proces skal man lave med rasterized grafik, men med ray tracing skyder man 3d vektorer fra sit kamera, og de ting det rammer bliver de reflekteret, så alt man skal regne, er vektorernes refleksioner. Dette gør at man ikke behøver nogen anden algoritme overhovedet. Objekter der er ude for kameraets vinkel, vil ikke blive ramt af nogen rays, objekter der overlapper, vil indersiden ikke bliver ramt af rays, så ray tracing virker bare uden nogen tricks. Dog kan man sige at ray tracing skal gennem gå nogle af de samme steps i udregningen, for når man ray tracer skal man selvfølgeligt også finde kollisioner med de afskudte rays.

Refleksioner

En af de sværeste effekter i rasterization er refleksionerne. I tidlige spil som Doom(1993), der var rasterized spil i realtid, havde de implementeret, deres egne refleksioner. Deres måde at skabe refleksioner var at kopiere hele rummet ind i spejlets plan, så det ligner lidt en refleksion. Dette er næsten komisk at kigge tilbage på, men også en dejlig simpel måde at lave refleksioner, dog laver dette selvfølgeligt en masse ekstra arbejde at rendere et rum to gange, men det virkede også kun fordi man ikke kan rykke kameraet op og ned, man kunne altså kun se direkte på spejlet og derfor virker effekten. Nyere metoder til at rendere refleksioner med rasterization er at lægge en tekstur oven på den overflade der skal reflektere noget. Dette kunne f.eks. være hvis der var et hus i en skov, og vinduerne skulle reflektere omverdenen. I dette eksempel ville man lægge en tekstur oven på vinduet der bevæger sig når spilleren også bevæger sig, denne tekstur ville måske bare have en masse træer på. Denne effekt er blevet bruget rigtig meget i mange ældre spil. Eksempelvis Battlefield 4 der kom ud i 2013, kan man i alle vinduer se at de bruger en samme tekstur lagt ind over vinduets normale tekstur. Disse effekter vil vi selvfølgelig helst væk fra, og det er derfor ray tracing er bedrer.

Her bevæger jeg ikke kameraet, men vender kun rotationen så man kan se hvad vinduet rigtigt burde reflektere. Læg også mærke til at spillerens karakter ikke bliver reflekteret selvom den brudte. Hvis man spiller med andre karakterer, ville dette være tydeligt siden at andre karakterer heller ikke bliver reflekteret.

(fig. [15, 16]. Urealistiske refleksioner lavet med ’cube-maps’, eget skærmbilleder[Battlefield 4])
Video demonstration: https://drive.google.com/file/d/1xjI9P808iUPytaPLAERYF5e_kQkterXU/view?usp=sharing

Til sammenligning kan vi kigge på Battlefield 5, der har implementeret ray tracede refleksioner. Her kan man se at refleksionerne viser præcis det de skal, fordi de bruger ray tracing.

(fig. 17. Realistiske refleksioner med ray tracing, eget skærmbilleder[Battlefield 5])

Man kan endda se sin egen spiller model i vinduer og refleksionen af det ’muzzle flash’ der kommer når man affyrere et våben på f.eks. en flyvemaskine.

(fig. 18. Promotionnelle billeder fra Battlefield 5 launch traileren, Battlefield 5 lauch)

Udregninger

Alle figurer under dette emne vil naturligvis være af egen kreation. Maple filer findes under bilag

Find refleksion af vektor på plan.

Vi starter med tre vertex i en scene

$$P_1 := \langle 4, 1, -2 \rangle :$$
$$P_2 := \langle 0, 1, -1 \rangle :$$
$$P_3 := \langle 0, 0, -1 \rangle :$$

Imellem tre punkter kan man altid skabe et plan.

$$n_{plan} := P_1 + \left(P_2 - P_1\right) \cdot t + \left(P_3 - P_1\right) \cdot u :$$

Ud fra planet kan vi også skabe planets normalvektor.

$$n := \left(P_2 - P_1\right) \times \left(P_3 - P_1\right) :$$

Planet bliver konstrueret med parameterfremstilling. Normalvektoren bliver konstrueret med parameterfremstillingens to retnings vektorer.

Så kan vi lave et kamera, jeg kalder kameraets værdier for ’i’ fordi de repræsentere indfaldsvinklen. Et rigtigt ray tracing software vil have et meget mere avanceret kamera der skyder mange rays ud med eksakte vinkler for hver pixel, men det er ikke så vigtig for demonstrationen af ray tracing princippet.

$$i_{dir} := \langle var, 1, -3 \rangle :$$
$$i_{pos} := \langle 0, -2, 2 \rangle :$$
$$i := i_{dir} \cdot s + i_{pos} :$$

(grunden til at jeg kalder indfaldsvinklens første værdi for ’var’ er så jeg kan animere kameraets retning senere. [’dir’ = retning, pos = ’position’])

For at vi kan bruge formlen for refleksion skal vi bruge de normaliserede vektorer. Altså enhedsvektorerne.

Enhedsvektor for normalvektoren

$$n_e := \frac{n}{\mathrm{sqrt}\left(n[1]^2 + n[2]^2 + n[3]^2\right)} :$$

Enhedsvektor for indfaldsvektoren

$$i_e := \frac{i_{dir}}{\mathrm{sqrt}\left(i_{dir}[1]^2 + i_{dir}[2]^2 + i_{dir}[3]^2\right)} :$$

Nu har vi endelig mulighed for at udregne refleksionsvektoren ved brug af de to enhedsvektorer.

$$r := i_e - 2\left(i_e \cdot n_e\right) n_e :$$

Det er sådan set alt man skal igennem for at få refleksionsvektoren. Mit Maple dokument, med kode til animationen ligger i bilag. Animations link:
https://drive.google.com/file/d/1WJl_ZS3zRRDC5j6sKyh92RFLPy-YQW0K/view?usp=sharing

5) Reflections and Refractions in Ray Tracing
https://graphics.stanford.edu/courses/cs148-10-summer/docs/2006--degreve--reflection_refraction.pdf

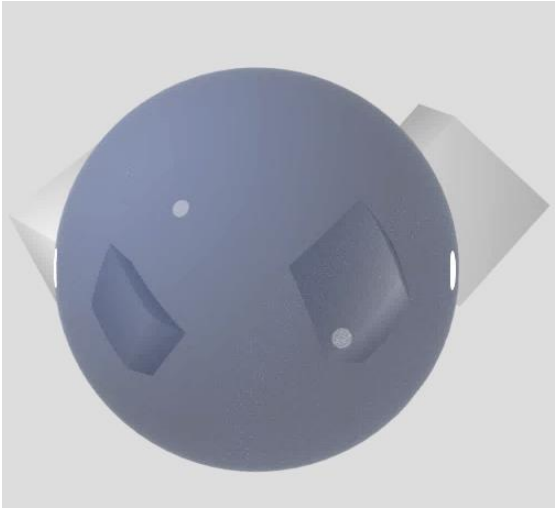
Find refraktion af vektor på plan.

Refraktioner er en anden måde lys bevæger sig på i virkeligheden. Dette fænomen sker når lys passerer igennem materialer der har forskellig tykkelse, dette kan altså være lys der går fra at være i luft til glas og tilbage til luft igen. Når dette sker ved vi godt at lyset bliver forvrænget. Dette fænomen forstås godt og kan derfor matematisk modelleres korrekt. I mit eksempel vil jeg udregne en vektor der går fra luft ind i vand. Målet er udfaldsvektoren. For at lave udregningerne bruger jeg Snell's lov, der siger at forholdet mellem sinus af indfaldsvinkel og sinus af udfaldsvinkel er det samme som forholdet mellem lysets hastighed i de to materialer.

$$\frac{\sin(\theta_1)}{\sin(\theta_2)} = \frac{v_1}{v_2}$$

Heldigvis behøver vi ikke at bruge lysets forskellige hastigheder i materialer siden at de er blevet lavet om til forhold siden da. De forhold går ud fra at vakuum er lig 1, og at alle andre materialer har en værdi der er forholdsvis den rette størrelse i forhold til den af et vakuum. Luft har et indeks på 1.000 og vand på 1.333.

Det kan være ret svært at forestille sig hvordan dette ville se ud, så jeg har renderet et billede af refraktion igennem en kugle. Her kan man se at de to firkanter bag kuglen bliver forvrænget fordi kuglen er refraktiv. Kuglens refraktive indeks er sat på 1.333, så det ville være det samme som vand i virkeligheden.



(fig. 19. Rending af refraktion igennem en kugle, egen kreation[Blender 3d])

Her er de refraktive indeks jeg vil bruge:

$$n_1 := 1:$$
$$n_2 := 1.333:$$

Derfor ser min formel nu sådan ud:

$$\frac{\sin(\theta_1)}{\sin(\theta_2)} = \frac{1.000}{1.333} \approx 0.75$$

Vi kender også allerede indfaldsvinklen. Derfor har vi en formel med en ubekendt:

$$\frac{\sin(\theta_1)}{\sin(\theta_2)} = \frac{n_1}{n_2}$$

$$\Leftrightarrow$$

$$n_1 \cdot \sin(\theta_1) = n_2 \cdot \sin(\theta_2)$$

$$\Leftrightarrow$$

$$\frac{n_1 \cdot \sin(\theta_1)}{n_2} = \sin(\theta_2)$$

Så kan vi finde udfaldsvinklen, hvis vi har indfaldsvinklen.

Jeg kan starte med at lave et eksempel i 2d med vektorer.

Mine variable er lavet således:

$plan := 1$:
$i_{dir} := \langle 1, -1 \rangle$:
$i_{pos} := \langle 0, 3 \rangle$:
$i_x := i_{pos}[1] + i_{dir}[1] \cdot t$:
$i_y := i_{pos}[2] + i_{dir}[2] \cdot t$:

Planet er en vandret linje med y-værdien 1.
I_{dir} er kameraets retning
I_{pos} er kameraets position
I_x og I_y er mine vektorfunktioner for indfaldsvektoren.

$n_1 := 1$:
$n_2 := 1.333$:

n_1 og n_2 er mine materialers refraktions indeks.

$\theta_1 := arcTan\left(\frac{i_{dir}[2]}{i_{dir}[1]}\right) + 90$

0_1 er indfaldsvinklen. Grunden til at jeg plusser med 90 er fordi vi skal lave vores udregninger i forhold til planets normalvektor, og planet er vandret i mit eksempel så derfor plusser jeg bare med 90 grader.

For at finde udfaldsvinklen bruger jeg Snell's lov (fordi vi finder vinklen)

$$angle := solve\left(\frac{n_1 \cdot Sin(\theta_1)}{n_2} = Sin(\theta_2), \theta_2\right) + 90 \xrightarrow{\text{evaluate at point}} 122.0367230$$

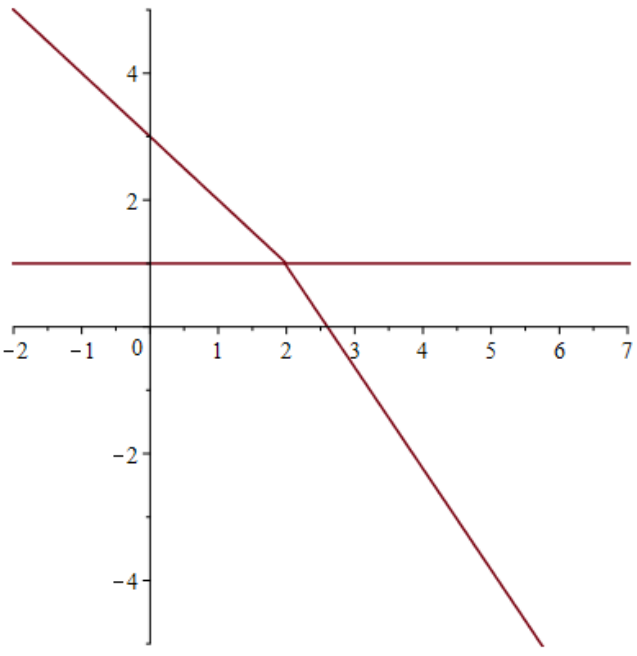
Udfaldsvektoren findes ved at tage cosinus og sinus af vinklen.

$$udfaldsvector := \langle Cos(angle), Sin(angle) \rangle = \begin{bmatrix} -0.5304627012 \\ 0.8477082768 \end{bmatrix}$$

Med udfaldsvektoren kan man opstille en vektorfunktion igen og derefter plotte linjerne:

$x(t) := udfaldsvector[1] \cdot t$:
$y(t) := udfaldsvector[2] \cdot t$:

Her kan man se at vektoren der bliver skudt ind bliver forvrænget og dermed ændre sin retning.



(fig. 20. Plot af 2d refraktion af lys fra luft til vand)

Men selvfølgeligt skal refraktionerne udregnes i 3d for at kunne bruges i ray tracing. Her bruger jeg det originale plan lavet med 3 punkter og også den samme indfaldsvektor. Forskellen på 2d og 3d er ikke så stor, men udregningerne bliver en del mere kompliceret. Begge løsninger bruger stadig Snell's lov og udregner vektorer fra vinkler og tilbage igen.

Jeg bruger denne formel til at finde refraktionen:

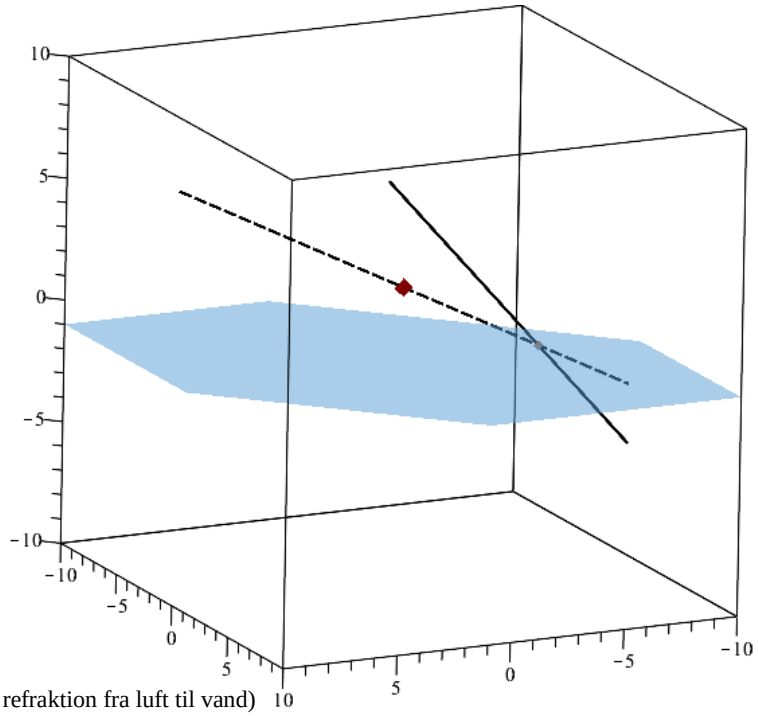
$$\hat{s}_2 = \frac{n_1}{n_2}\left[\hat{N} \times \left(-\hat{N} \times \hat{s}_1\right)\right] - \hat{N}\sqrt{1 - \left(\frac{n_1}{n_2}\right)^2 \left(\hat{N} \times \hat{s}_1\right) \cdot \left(\hat{N} \times \hat{s}_1\right)}$$

Og her er min Maple kode:

$$refraction := simplify\left(\left(\frac{n_1}{n_2}\right) \cdot i_e + \left(\left(\frac{n_1}{n_2}\right) \cdot \left(\left(-n_e\right) \bullet i_e\right) - \mathrm{sqrt}\left(1 - \left(\frac{n_1}{n_2}\right)^2 \cdot \left(1 - \left(\left(-n_e\right) \bullet i_e\right)^2\right)\right)\right) n_e\right) :$$

Her har jeg brugt n_1 og n_2 igen, men også normalvektorens enhedsvektor og indfaldsvektoren enhedsvektor. De er ens med dem fra mit eksempel af refleksion.

Når man plotter ser det sådan ud:



(fig. 21. 3d plot af refraktion fra luft til vand)

Animations link:
https://drive.google.com/file/d/1vTy-DNOvuwV13IlnYOJkOZwyFQy1dMFR/view?usp=sharing
https://drive.google.com/file/d/1jAj6EEi46TA7ZfEydBnayzxuRwgdgwrS/view?usp=sharing
(stiplede linje er indfaldsvinkel, linje er udfaldsvinkel, rødt punkt er kameraets position og det grå punkt er skæringspunktet mellem indfaldsvinklen og det nye materiale)

5) Reflections and Refractions in Ray Tracing
https://graphics.stanford.edu/courses/cs148-10-summer/docs/2006--degreve--reflection_refraction.pdf

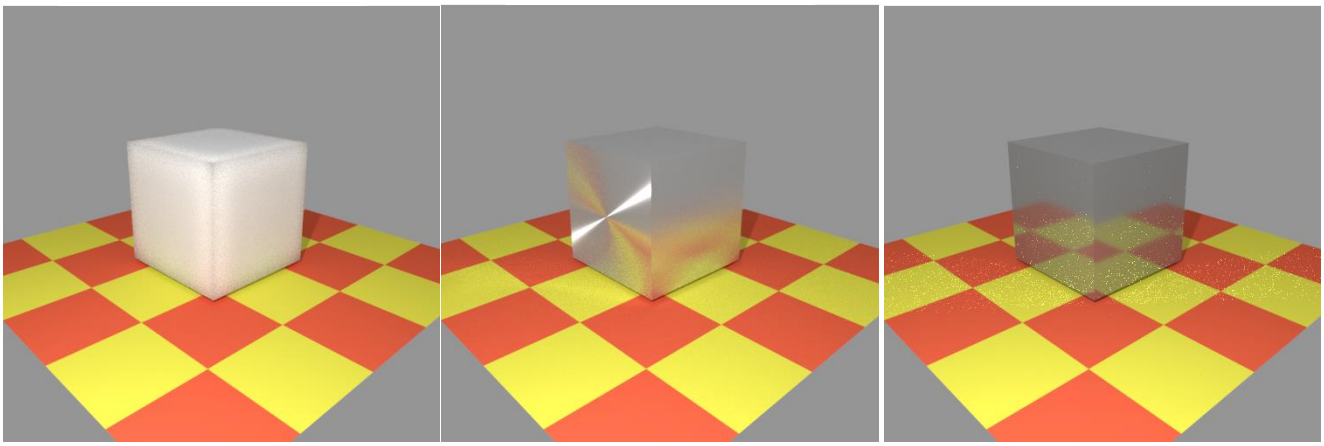
Andre typer udregninger

De to vigtigste renderinger af lys i ray tracing er af refleksion og refraktion, men ray tracing har en del andre typer rendering der også bruges. Her i blandt andet udregninger af:

- Diffuse reflection
- Specular reflection
- Glossy reflection
- Dielectric reflection
- Anisotropic reflection
- Subsurface scattering
- Sheen
- Clearcoat
- Clearcoat tint
- Roughness
- Mix Shaders
- Transparency
- Reflective caustics
- Refractive caustics
- Fresnel

Alle disse typer udregninger bliver brugt i ray tracing og det er derfor der skal så meget computerkraft til at rendere alt dette. En 3d engine ved dog godt at hvis der ikke bliver brugt nogle metaller skal der heller ikke udregnes nogle dielektriske refleksioner, så hvor hård udregningen er for computeren er også bestemt af hvilken scene man skal rendere. Det er også derfor at man i Battlefield 5 nu kan bruge ray tracing, for egentligt bruger man kun ray tracede refleksioner og overlader resten af renderingen til rasterization.

Subsurface scattering | Anisotropic reflection | Dieletric refelction



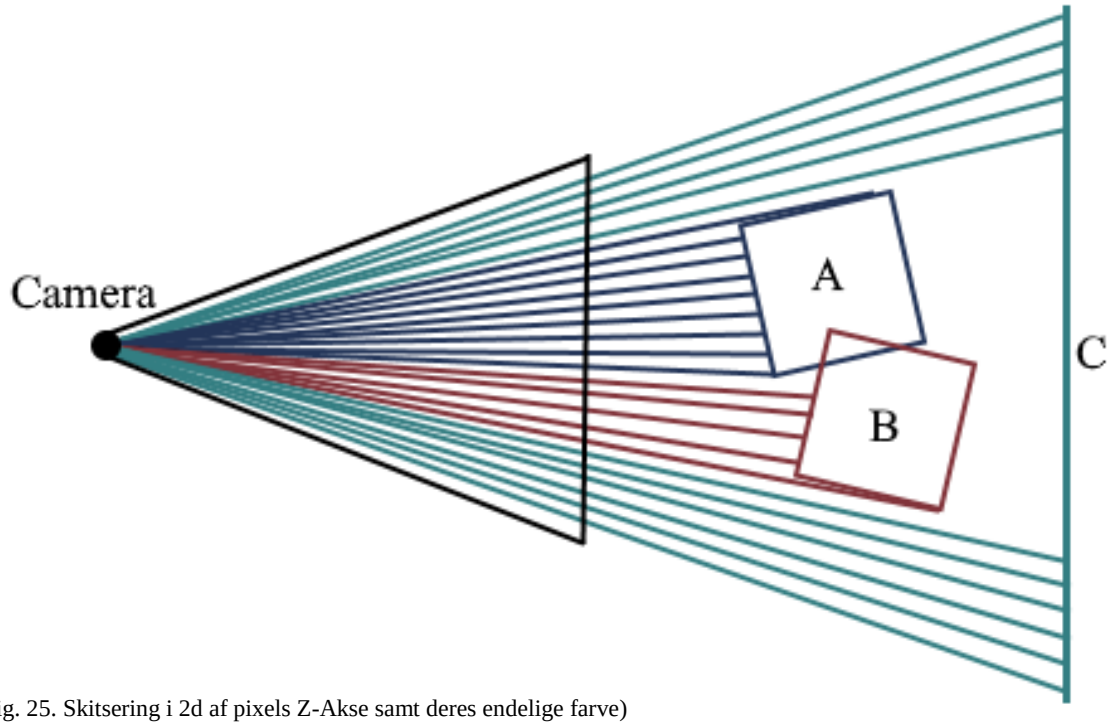
(fig. [22, 23, 24]. Sammenligning af andre typer rendering i ray tracing, egen kreation[Blender 3d])

Hidden Surface Removal

Hidden surface removal fjerner, som jeg skrev tidligere, objekter og dele af objekter, hvis de ikke skal kunne ses af kameraet. Jeg vil bruge et eksempel med en Z-Buffer metode fordi det er den mest brugte metode til hidden surface removal, det er endda implementeret som en del af moderne hardware.

Z-Bufferen tager hver pixel og finder afstanden til alle objekter der er i lige linje i z aksen ud af fra den pixel.

Et diagram over pixels z akse skitseret med farven som pixelen skal have kunne se sådan ud:



(fig. 25. Skitsering i 2d af pixels Z-Akse samt deres endelige farve)

Afstandene bliver målt med vektorer og afstandsformelen. Selve vektoren skal gå fra kameraet i en ret linje til det rammer nogle objekter, hvor mange objekter der er, er ikke vigtigt for vores udregning siden at vi kan sortere i dem med længderne af vektorerne. Vektorens længde fås med afstandsformlen. F.eks.

$A := \langle 10, 0, 0 \rangle$:

Længden af vektor A kan bestæmes med afstandsformlen:

$$Længde = \sqrt{A_1^2 + A_2^2 + A_3^2}$$

$$Længde := \text{sqrt}\left(A[1]^2 + A[2]^2 + A[3]^2\right) = 10$$

Vi kan altså finde længderne til objektet, men længderne skal vi selvfølgeligt også behandle. Her skal de organiseres så vi kun viser den forreste farve af et givent objekts kollision med en af pixelsen vektor fra z-aksen.

For at få de forskellige vektorer skal man også lave lidt udregning. Alle objekter er lavet af punkter der er forbundet til trekanter. Og trekanter er bare små planer i scenen der kan matematisk forklares. For at starte skal vi finde den vektor der bliver skudt fra kameraet (Det her er stadig rasterization). For at gøre dette starter vi med en stedvektor der peger på kameraet og derefter finder vi de vinkler der er til de forskellige pixels.

Som tidligere vælger jeg mit plan konstrueret af tre punkter.

Z-Buffer object detection

$restart : with(plots) : with(Gym) :$

Vi starter med tre vertex i en scene

$P_1 := \langle 1, 0, -1 \rangle :$

$P_2 := \langle 0, 1, -1 \rangle :$

$P_3 := \langle 0, 0, -1 \rangle :$

Imellem tre punkter kan man altid skabe et plan

$n_{plan} := P_1 + (P_2 - P_1) \cdot t + (P_3 - P_1) \cdot u :$

Nu kan vi skabe et virtuellt kamera

$i_{dir} := \langle 1, 0, -1 \rangle :$

$i_{pos} := \langle 0, 0, 2 \rangle :$

$i(s) := i_{dir} \cdot s + i_{pos} :$

Vi kan starte med at opstille vores ligning vi vil løse. Kameraets Z-akse er jo egentligt en punktmængde ligesom planet, det vi vil finde, er de punkter der er ens. Derfor kan vi opstille ligningen:

$$ligning := n_{plan} = i(s) \xrightarrow{\text{evaluate at point}} \begin{bmatrix} 1 - t - u \\ t \\ -1 \end{bmatrix} = \begin{bmatrix} s \\ 0 \\ -s + 2 \end{bmatrix}$$

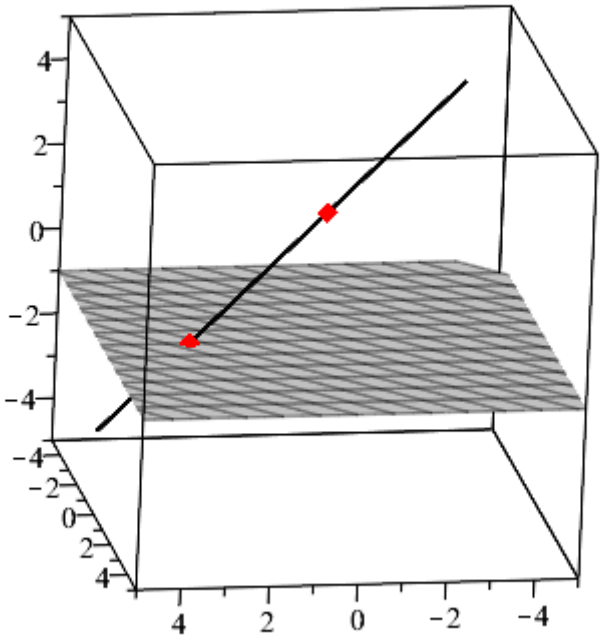
Denne ligning kan vi finde forskellige løsninger til, men der er nemmest at finde løsningen til s, siden at vi kun skal bruge en variabel til linjens parameterfremstilling.

$$løsning := vsolve(ligning)[1] = s = 3$$

$$afstand := subs(løsning, i(s)) = \begin{bmatrix} 3 \\ 0 \\ -1 \end{bmatrix}$$

Her finder vi forskellige løsninger og vælger den første, derefter indsætter vi denne løsning til s og for et punkt i 3d. Dette punkt er skæringspunktet mellem kameraets Z-akse linje og planet.

Her er et plot, de røde punkter der kameraets position og skæringen med planet:



(fig. 26. 3d plot af et enkelt Z-buffer kollision med et plan)

For at finde afstanden skal vi have en vektor der starter i origo. Derfor parallelforskyder jeg skæringspunktet med kameraets position. Vi har altså længden mellem kameraet og kollisionen der nu starter fra origo.

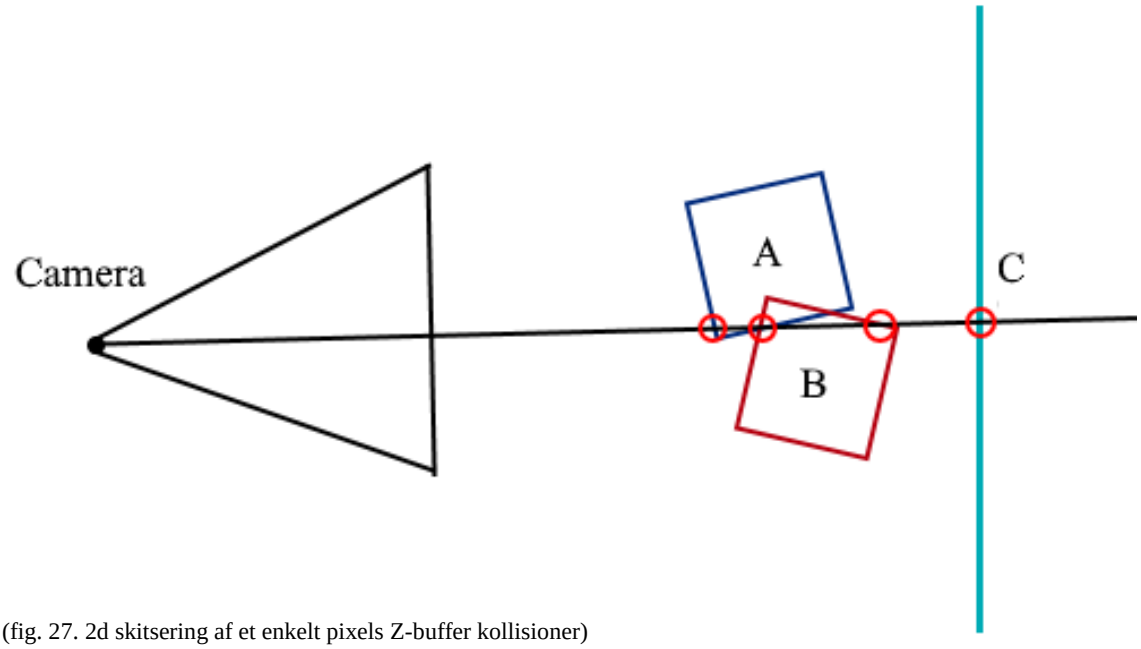
$$OA := afstand - i_{pos} = \begin{bmatrix} 3 \\ 0 \\ -3 \end{bmatrix}$$

Så mangler vi bare afstandsformlen:

$$afstand_1 := evalf\left(\text{sqrt}\left(OA[1]^2 + OA[2]^2 + OA[3]^2\right)\right) = 4.242640686$$

Og der er altså vores afstand mellem kameraet og dette plan lavet af 3 vertex. Disse udregninger er selvfølgeligt blevet optimeret på mange forskellige måder igennem computeres udvikling, men det er samme metode man stadig bruger. Nu har vi altså en enkelt afstand, men hvordan ryder vi så op i de forskellige afstande vi får som output?

Her er en enkelt pixels linje. Jeg har fremhævet kollisionerne med objekter i scenen.



(fig. 27. 2d skitsering af et enkelt pixels Z-buffer kollisioner)

Hvis vi har fundet disse kollisioners afstande, lad os for eksemplets skyld vælge nogle tilfældige tal. Kollisionerne har afstanden fra kameraet: [2, 4, 6, 1, 9]. Siden at vi bruger programmering kan vi sortere i et sådant array og dermed finde det den mindste afstand mellem et givent objekt og kameraet. Et stykke 'pseudocode' kunne se sådan ud:

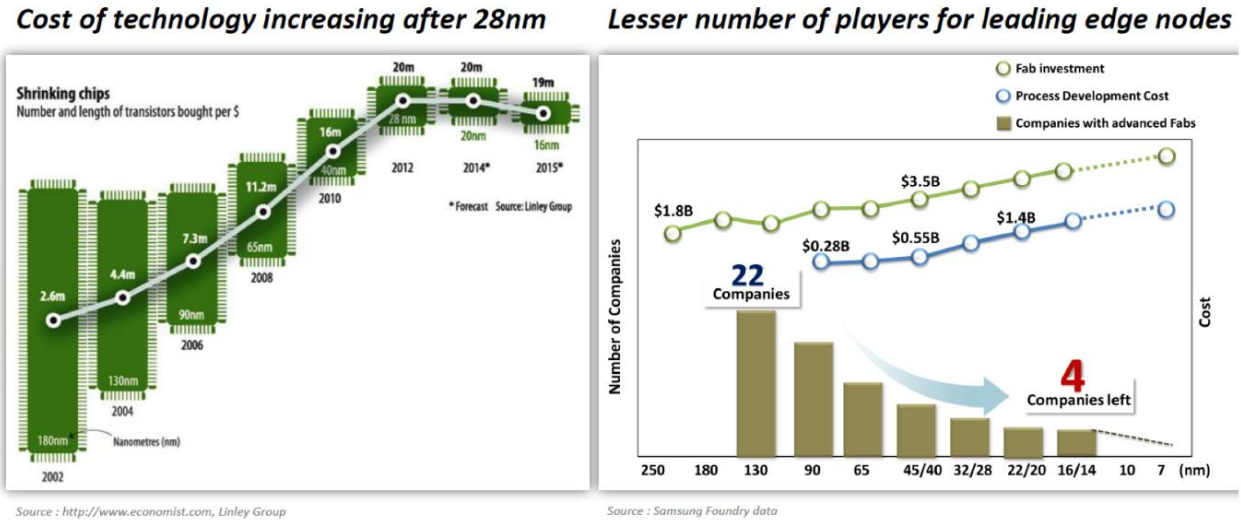
```
Array Afstande[] = [2, 4, 6, 1, 9];
Int Tal = Afstande[0];

For (i = 0; i < Afstande[].Lenght; i = i++;){
   If (Tal < Afstande[i]){
      Pixel_Farve = Afstand[i].Object.Color;
   }
}
```

Her ville programmet så gennemgå de forskellige afstande for en bestemt pixel og vælge den mindste afstand ligesom vi ville have. Rækkefølgen er altså lige meget så længe vi ender med at vælge den mindste afstand fra kameraet. I eksemplet ville det selvfølgeligt vælge tallet '1' i arrayet og vil dermed også vælge at pixelens farve skal være blå, siden at A på figuren er tættest på kameraet og har farven blå.

Hvornår kan vi bruge ray tracing i realtid

Processor kraften man skal bruge for at rendere ray tracing i realtid er alt for høj med vores nuværende teknologi. Så kan man måske spørge; ”hvornår vil vores teknologi være god nok”, men det spørgsmål er sværere at svare på end tidligere. Indtil de sidste par år har vi fulgt moore’s lov, der siger at størrelsen af transistorer vil formindske med 2x hvert andet år. Sagt med andre ord; computerkraften vil fordoble hvert andet år. Denne proces af at formindske transistorer har vi fulgt i de sidste 50 år, men vi et løbet ind i problemer. Intel der er den største producent af PC-processoren, er løbet ind i problemer med deres 10nm proces node. kilde[6]

At skalere transistorer er ikke nemt, jo mindre vi bygger vores komponenter jo mere præcision skal der være og dermed er der også mindre plads til fejl. Selvom processorerne bliver mindre, og trækker mindre kraft og at vi dermed kan køre dem hurtigere, bliver det også dyrere og dyrere at lave mindre chips.



(fig. 28. Side fra Samsung præsentation om processor størrelse, Semicon West(2016))

Processoren bliver altså dyrere og dyrere ved de nye transistorstørrelser. Udover dette har Intel længe lovet at deres 10nm processorer ville komme ud. Deres 14nm processorer lancerede i 2014, hvor de forberedte folk på at deres 10nm procore ville lancere i Q4 2016 – Q1 2017, men vi har stadig ikke set nogle 10nm processorer, nu er det planlagt at de skal lancere sent i 2019-2020. kilde[7, 8, 9]

Hvilke problemer de har med 10nm ved vi ikke, men forsinket er de i hvert fald, jeg kan ikke forestille mig at 7nm ikke også vil være forsinkede. Dog når vi nok de mindre transistorstørrelser på et tidspunkt, men man har allerede lavet forskning på hvor lille en transistor man muligt kan lave. Og man har fundet at 7nm og 5nm nok vil være mulige, men at vi begynder at ramme grænsen før at fysikkens love stopper os. Der kommer nemlig et punkt, hvor vi ikke kan lave mindre kredsløb siden at de forskellige lag i transistoren ikke vil kunne stoppe den strøm der vil igennem. kilder[7]

6) Happy birthday x86
https://www.computerworld.com/article/2535037/computer-hardware-happy-birthday-x86-an-industry-standard-turns-30.html#tk.drr_mlt
7) End of moore’s law: it’s not just about physics
https://www.cnet.com/news/end-of-moores-law-its-not-just-about-physics/
8) 28nm was the last node of moore’s law
https://www.eetimes.com/author.asp?section_id=36&doc_id=1330366
9) Intel’s 10nm problems have implications far beyond what the market is seeing
https://seekingalpha.com/article/4201847-intels-10nm-problems-implications-far-beyond-market-seeing

Hvorfor er 5nm det teoretisk mindste en transistor kan blive?

Siden at transistoren afhænger af at der er et elektrisk felt imellem dets positive og negative lag, så vil der være en størrelse, hvor det lag er for småt og der vil løbe strøm igennem lige meget hvad. Derudover vil det også være dyrere at producere disse chips så det ikke kan betale sig længere. kilde[10]

"A second reason for the slowdown is that it's simply getting harder to design, inspect and test chips at advanced nodes. Physical effects such as heat, electrostatic discharge and electromagnetic interference are more pronounced at 7nm than at 28nm. It also takes more power to drive signals through skinny wires, and circuits are more sensitive to test and inspection, as well as to thermal migration across a chip. All of that needs to be accounted for and simulated using multi-physics simulation, emulation and prototyping." – Ed Sperling, Redaktør for Semiconductor Engineering. Kilde[10]

Men hvad gør vi så i stedet? Indtil videre har vi kigget på om 'bruteforcing' af ray tracing med CPU er en mulighed. Det ser dog ikke sådan ud idet at der skal så meget mere computerkraft til for at rendere ray tracing i realtid. Computerkraften kunne komme fra kvantecomputere, men almindelig brug af denne teknologi er hypotetisk og i bedste fald meget langt ude i fremtiden.

NVIDIA

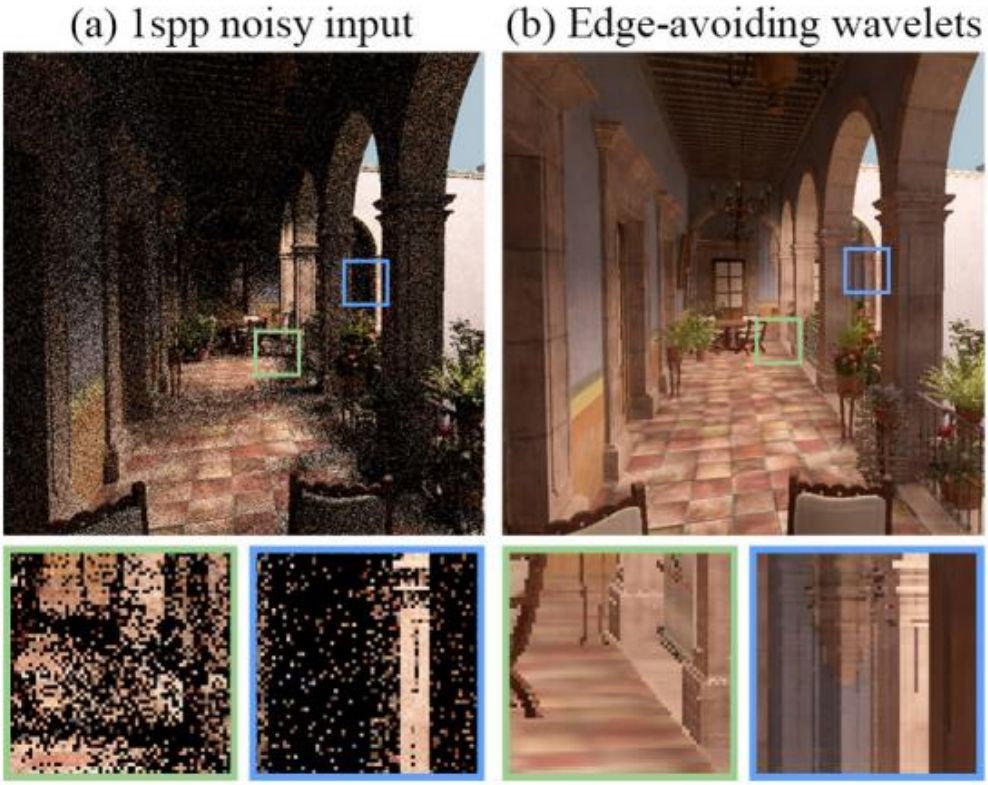
Normale processorers udvikling går simpelthen for langsomt og er ved at stoppe helt. Derfor har producenter som nVIDIA valgt at arbejde på at få andre typer af processorer til at rendere ray tracing. Deres løsning og lige nu den eneste der er kommerciel, er at bruge 'machine learning' til at gætte sig til hvordan et ray tracet billede ville se ud.

En vigtig faktor i hvor hurtig en computer renderer ray tracing er hvor mange rays der bliver sendt ud fra kameraet, for dette er noget brugeren vælger og ud fra dette ændres kvaliteten og hastigheden af rendering. Samples er et udtryk der beskriver hvor mange rays pr. Pixel man sender ud. Der vil derfor i et 'full hd' billede med 32 samples bliver sendt: $1920 \cdot 1080 \cdot 32 = 66355200$ rays.

Dette antal rays ville faktisk renderer relativt hurtigt, men ville også være meget grynet, siden at farven af en enkelt pixel bliver bestemt af kun 32 rays. Hvor mange samples en Disney eller Pixar film bruger er forretningshemmeligheder, men de fleste steder bliver omkring 1500 – 2000 samples brugt. En storproduktions film fra Disney bruger sikkert 10.000 samples. Men nVIDIA's løsning kræver kun en enkel sample pr. frame. Denne frame sender de igennem et neuralt netværk, der prøver at 'fixe' billedet som om der blev brugt mange flere samples. Derfor skal de også rendere en masse billeder fra 3d applikationen de vil implementere ray tracing i realtid i og så bruge de billeder som et datasæt til deres neurale netværk. Konstruktionen af sådan et enkelt billede med 1 sample ray tracing har man længe kunne rendere i realtid, men det er et ubrugeligt billede man får ud, fordi det er så grynet. Det er dette billede og noget mere information om scenen de sender igennem deres netværk for at konstruere hvad netværket vurdere til at være tættest på et billede med mange flere samples. Kilde[11]

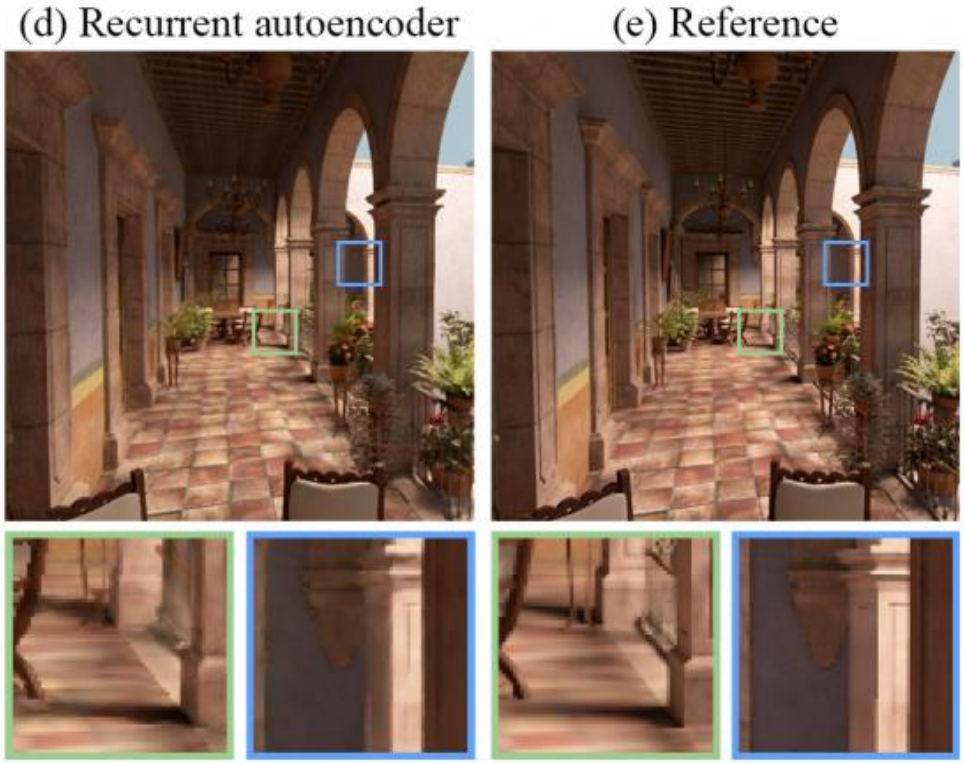
10) Is 7nm the last major node?
https://semiengineering.com/7nm-last-major-node/
11) Interactive Reconstruction of Monte Carlo Image Sequences using a Recurrent Denoising Autoencoder
https://research.nvidia.com/sites/default/files/publications/dnn_denoise_author.pdf

1 Sample rendering og det første step i 'rengøringen'



(fig. 29. Kilde[11])

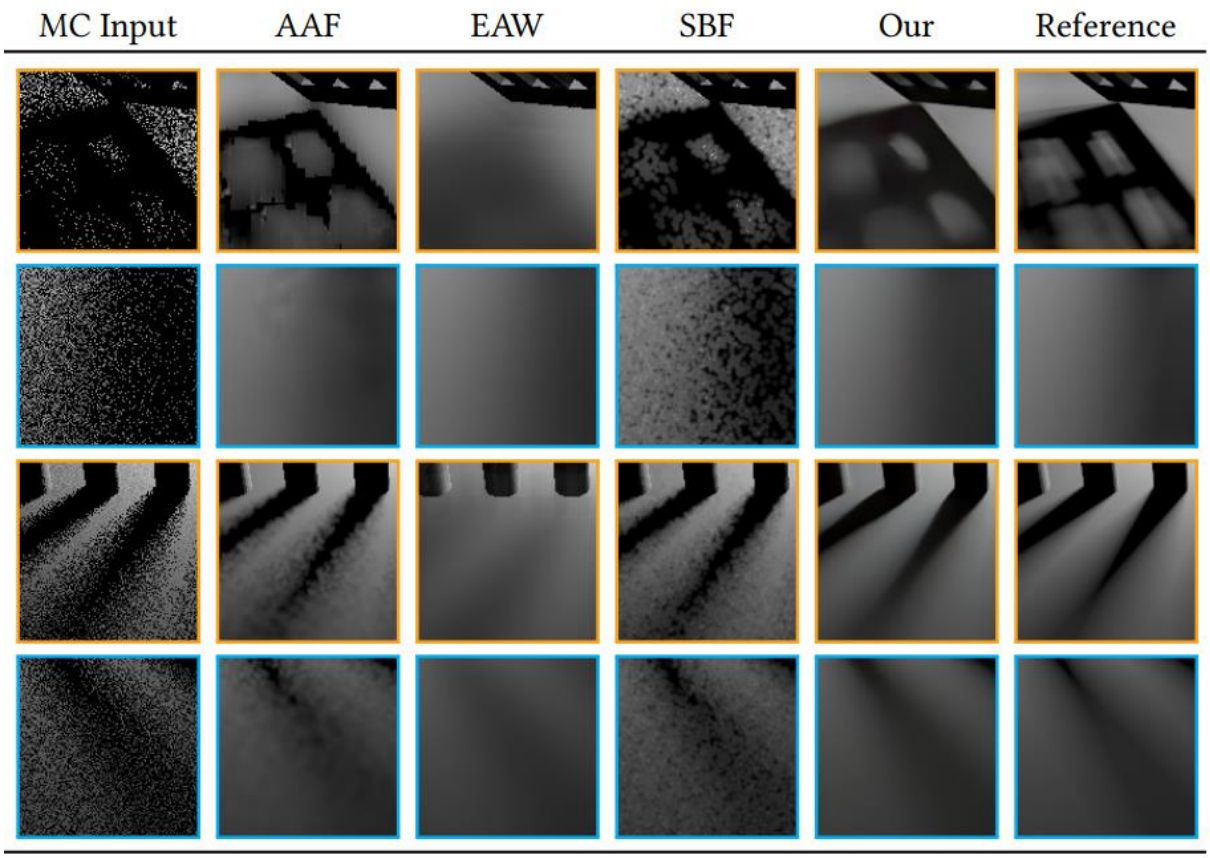
Endelige output i sammenligning med højere sample rendering



(fig. 30. Kilde[11])

Til at opnå dette resultat bruger de selvfølgeligt ikke bare det grynede RGB-output, men en række af output, heri blandt andet Z-bufferens output til at bestemme hvilke objekter der hænger sammen.

11) Interactive Reconstruction of Monte Carlo Image Sequences using a Recurrent Denoising Autoencoder
https://research.nvidia.com/sites/default/files/publications/dnn_denoise_author.pdf



(fig. 31. Sammenligning af ’noise reduction’ metoder, Kilde[11])

Her kan man se deres netværk blive sammenlignet med andre mere traditionelle metoder til at ’rengøre’ et grynet ray tracet billede. For at opnå et datasæt som deres netværk kan arbejde med, bliver de nødt til at rendere et datasæt selv, så når denne metode bliver implementeret i virkelige applikationer skal der først renderes et datasæt som programmet kan sammenligne med.

Denne metode til at rendere ray tracede billeder i realtid er kun implementeret et enkelt sted indtil videre, siden at teknologien er så ny, men det er allerede bekræftet at det bliver implementeret andre steder. For at bruge denne teknologi kræver det dog også at man har noget hardware der kan understøtte det. Det er derfor nVIDIA har lanceret deres nye RTX-linje af grafikkort der har indbygget hardware understøttelse af denne ray tracing rendering. Alt der skal beregnes, sker i et neuralt netværk derfor er hardwaredelen også bygget til at løse machine learning problemer. Så i fremtiden vil dette stykke indbyggede hardware sikkert også bliver brugt til meget andet. Jeg har selv fået fat i et sådan grafikkort, så jeg har muligheden for at arbejde videre med machine learning og ray tracing applikationer. Applikationen der udnytter disse nye teknologier, er som jeg tidligere viste spillet ’Battlefield V’, men det understøtter ikke fuld ray tracing, kun refleksioner, men dette er også en af de sværeste effekter at replikere i rasterization.

11) Interactive Reconstruction of Monte Carlo Image Sequences using a Recurrent Denoising Autoencoder
https://research.nvidia.com/sites/default/files/publications/dnn_denoise_author.pdf

Konklusion

At bruge ’brute force’ til at rendere ray tracing, vil sikkert blive ved med at være standarden i de næste mange år, men spil hvor kvaliteten af ray tracing ikke er ligeså vigtig bliver interessant. Ray tracing er nærmest en simulation af fotonerne i virkeligheden og man kan derfor rendere utrolig realistiske billeder. Spillet Battlefield 5, der er det eneste spil der udnytter ray tracing, dog kun til refleksioner, ser allerede lovende ud. Især i sammenligning med det tidligere Battlefield 4 der brugte ’cube maps’. Ray tracing vil erstatte rasterization i fremtiden, det handler bare om hvornår og hvordan. Lige nu ser det ud som om 1 sample ray tracing rengjort med et neuralt netværk er den bedste løsning, men teknologien er stadig så ny at det er svært at drage nogle konklusioner om hvordan problemet vil blive løst i fremtiden. Men man kan i hvert fald sige at implementeringen af ray tracing med machine learning er imponerende og teknologien virker lovende. Især efter at have undersøgt alt det der går ind i at rendere et enkelt ray tracet billede. Det bliver nok også nemmere for programøre i fremtiden at lave god grafik, siden at der ikke skal en masse tricks til for at grafikken virker. Ray tracing virker derimod bare på baggrund af at ray tracing er matematisk modelleret efter virkeligheden. Ray tracing vil altså blive brugt mere i fremtiden siden at vi endeligt har brudt barrieren for rendering i realtid og fordi at teknologien stadig er så umoden kan det kun blive bedre.

(fig. 32. Ray tracede refleksioner, egen skærmbillede[Battlefield 5])

Kilder

1) 25% of IKEA catalogues will have CAD renders instead of real products
JF Brandon, Anno: 2012, Lokaliseret: 22/11/18
blog.grabcad.com/blog/2012/08/25/25-of-ikea-catalogues-will-have-cad-renders-instead-of-real-products/

2) Disney explains why its 3d animation looks so realistic
Jon Fingas, Anno: 2015, Lokaliseret: 22/11/18
engadget.com/2015/08/02/disney-explains-hyperion-renderer/

3) How Pixar made Monsters University, its latest technological marvel
Dean Takahashi, Anno: 2013, Lokaliseret: 23/11/18
venturebeat.com/2013/04/24/the-making-of-pixars-latest-technological-marvel-monsters-university/

4) Visibility in Computer Graphics (3.2.1)
Jiri Bittner & Peter Wonka, Anno: 2003, Lokaliseret: 02/12/18
researchgate.net/publication/23541400_Visibility_in_Computer_Graphics
cg.tuwien.ac.at/research/publications/2003/Bittner-2003-VCG/TR-186-2-03-03Paper.pdf

5) Reflections and Refractions in Ray Tracing
Bram de Greve, Anno: 2006, Lokaliseret: 24/11/18
graphics.stanford.edu/courses/cs148-10-summer/docs/2006--degreve--reflection_refraction.pdf

Vektor matematik ikke dokumenteret i kilde[5] er dokumenteret i:
Mat A htx, Allan Bohnstedt, Bernt Hansen, Michael Jensen, Klaus Marthinus, Systime A/S, 2008

6) Happy birthday x86
Gary Anthes, Anno: 2008, Lokaliseret: 08/12/18
computerworld.com/article/2535037/computer-hardware-happy-birthday-x86-an-industry-standard-turns-30

7) End of moore's law: it's not just about physics
Brooke Crothers, Anno: 2013, Lokaliseret: 08/12/18
cnet.com/news/end-of-moores-law-its-not-just-about-physics/

8) 28nm was the last node of moore's law
Zvi Or-Bach, Anno: 2016, Lokaliseret: 08/12/18
eetimes.com/author.asp?section_id=36&doc_id=1330366

9) Intel's 10nm problems have implications far beyond what the market is seeing
Ed Sperling, Anno: 2003, Lokaliseret: 08/12/18
seekingalpha.com/article/4201847-intels-10nm-problems-implications-far-beyond-market-seeing

10) Is 7nm the last major node?
EnerTuition, Anno: 2018, Lokaliseret: 08/12/18
semiengineering.com/7nm-last-major-node/

11) Interactive Reconstruction of Monte Carlo Image Sequences using a Recurrent Denoising Autoencoder
Charkravarty R. Alla Chaitanya, Anton S. Kaplanyan, Christoph Schied, Marco Salvi, Aaron Lefohn,
Derek Nowrouzezahrai, Timo Aila, Anno: 2017, Lokaliseret: 15/12/18
research.nvidia.com/sites/default/files/publications/dnn_denoise_author.pdf

Figurliste

Selvkreerede diagrammer, skitser og skærmbilleder:

Fig. [2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 13, 14, 15, 16, 17, 19, 20, 21, 22, 23, 24, 25, 26, 27, 32]

Programmer brugt: [Blender 3d, Maple 2018, Paint.Net]
Skærmbilleder fra: [Battlefield 4, Battlefield 1, Battlefield 5]

Bilag

Mine egne billeder er renderet med Blender v2.79b.
Animationer og udregninger er lavet i Maple 2018.
Skitser er tegnet i Paint.Net.

Blender filer:

Simpel eksempel scene. fig[3, 4]
drive.google.com/file/d/1YyPf12xxNhSHcUTsE9NaZK1uqzZn_3JR/view?usp=sharing

Glasbord blender3d fil. fig[9]
drive.google.com/open?id=1NKxK-XZr4Ywn4GkatrOBP5SaceWNWKWJ

Refraktion eksempel. fig[19]
drive.google.com/file/d/1csM4WxwPvSmjbHjGaWSVUxwUYWd5uEee/view?usp=sharing

Z-Buffer eksempel. fig[13, 14]
drive.google.com/file/d/1cKcAJrERFJoNCfrNa3a1sUkjPkKCEVHt/view?usp=sharing

Subsurface scattering, Anisotropic reflection, Dielectric reflection. fig[22, 23, 24]
drive.google.com/drive/folders/1ssy8jpkcwLm2TF6l_tsMohIycEAED_5P?usp=sharing

Maple filer:

Filerne ligner tekstfiler, men de kan åbnes med Maple efter de er blevet downloadet

Refleksion og Refraktion
drive.google.com/file/d/1HzxlObfw6bj936uhaeKGd0emLOAGD5K3/view?usp=sharing

Refraktion 2D
drive.google.com/file/d/1pFtFX2xy9TFGEE8rioxExWk1CqMP2_j6/view?usp=sharing

Hidden Surface Removal
drive.google.com/file/d/1ucWJP8CuifVWdoTxUOB9EY5deuMmYXLa/view?usp=sharing

Animationer:

Refleksions animation. fig[5]
drive.google.com/file/d/1WJl_ZS3zRRDC5j6sKyh92RFLPy-YQW0K/view?usp=sharing

Refraktions animation. fig[21]
drive.google.com/file/d/1vTy-DNOvuwV13IlnYOJkOZwyFQy1dMFR/view?usp=sharing
drive.google.com/file/d/1jAj6EEi46TA7ZfEydBnayzxuRwgdgwrS/view?usp=sharing