

# PWS raytracing

Sytze Reitsma  
6 VWO

Met begeleiding van Dhr. Mulder, MSc  
Zaanlands Lyceum

4 januari 2026

# Inhoudsopgave

<b>1 Samenvatting</b>	<b>2</b>
<b>2 Inleiding</b>	<b>3</b>
<b>3 Doelstelling</b>	<b>4</b>
<b>4 Theoretisch kader</b>	<b>5</b>
4.1 Lichtransportvergelijking . . . . .	5
4.2 Monte Carlo Integratie . . . . .	6
4.3 Snijdingsdetectie . . . . .	7
4.3.1 Snijpunt bal en lijn . . . . .	7
4.3.2 Snijpunt face en lijn . . . . .	8
4.3.3 Snijpunt doos en lijn . . . . .	9
4.4 Emissie en absorptie . . . . .	10
4.5 BRDF: de weging van kleuren . . . . .	11
4.5.1 Spiegeling met GGX . . . . .	11
4.5.2 Diffuse verlichting . . . . .	12
4.6 De camera . . . . .	13
4.6.1 Camerarichting . . . . .	13
4.6.2 Besturing van de camera . . . . .	13
4.6.3 Focus . . . . .	13
<b>5 Methode</b>	<b>15</b>
5.1 main.py . . . . .	16
5.2 tracer.frag: de shader . . . . .	18
5.2.1 vec4 ballcollide(Line l, Ball c) . . . . .	18
5.2.2 CollisionT ballcollision(Line l, Ball c) . . . . .	18
5.2.3 collision_data collide(Line l, vec4 face[3]) . . . . .	18
5.2.4 hitInfo getHit(ray, balls, faces) . . . . .	18
5.2.5 brdf_data BRDF(hitInfo hit, Material mat, inout uint rngState) . . . . .	18
5.2.6 vec3 trace(ray, balls, faces) . . . . .	18
5.2.7 void main() . . . . .	18
<b>6 Proces</b>	<b>20</b>
6.1 AI . . . . .	20
6.2 Tijdsontwikkeling . . . . .	20
6.2.1 De ontwikkeling van de BRDF . . . . .	20
6.2.2 De ontwikkeling van objectondersteuning . . . . .	21
6.2.3 Problemen . . . . .	21
<b>7 Conclusie</b>	<b>23</b>
<b>8 Discussie</b>	<b>24</b>
<b>9 Bijlage</b>	<b>25</b>
9.1 Logboek . . . . .	25
9.2 Renders . . . . .	27

# **Hoofdstuk 1**

## **Samenvatting**

3D-rendering wordt vaak voor lief genomen in het creatieve proces, maar speelt een bijzonder grote rol. In dit profielwerkstuk wordt onderzocht hoe een ray-tracende 3D-renderer op basis van natuurkundige principes functioneert, en wordt er een geschreven met OpenGL. Eerst worden er eisen gesteld; daarna wordt er relevante theorie behandeld. Dan wordt het geschreven systeem beschreven, en wordt er gereflecteerd op de resultaten hiervan en het proces om het te maken. De voorafgestelde eisen zijn gehaald, en er zijn daarbovenop extra doelen bereikt om een beter resultaat te bieden.

# Hoofdstuk 2

## Inleiding

De overlap tussen kunst en wetenschap is een die vandaag de dag sterk ondergewaardeerd wordt. Kunst vormt zich als een uiting van het innerlijke van haar kunstenaar, maar de aanvullende rol die de wetenschap hierbij speelt, wordt vaak geheel weggelaten. Denk hierbij aan de wiskundige algoritmen achter de geluidseffecten die moderne muziekstromingen vormen, of, in ons geval, het systeem dat aan onder andere moderne 3d-animatie en videogamebelichting ten grondslag ligt: De ray-tracer.

De ray-tracer, of over het algemeen een 3D-renderer, kan worden gezien als de tussenweg tussen de visie van de kunstenaar en de informatie die de toeschouwer op een scherm ontvangt. Een scherm ontvangt een simpele tabel met zo'n één tot acht miljoen pixels, maar deze worden niet handmatig ingevuld door een ontwerper, maar berekend op basis van de ontvangen informatie: Een lijst van ontworpen 3d-objecten, met informatie over op welke plaats welke materiaaleigenschappen gelden, en misschien ergens een lichtbron. Een ray-tracer simuleert dan fysiek het licht met een statistisch model en voedt, nadat er genoeg willekeurige richtingen zijn nagebootst, dit terug als pixelabel.

Vaak wordt dit onderdeel als vanzelfsprekend beschouwd, maar de rol die ray-tracing speelt in de kunst is zeer significant. Een beter begrip van de logica achter dit systeem kan leiden tot nieuwe, creatieve toepassingen van ray-tracing en zo de kunstwereld verrijken en een directere uiting van het inzicht van de kunstenaar bieden. Een voorbeeld dat nu al in werking is, is de niet-fotorealistische renderstijl van films zoals *Spider-Man: Into the Spider-Verse*. Door een herschrijving van hun rendersysteem op een opzettelijk niet-fotorealistische manier hebben ze een visuele stijl gevormd die nergens daarvoor te zien was.



Figuur 2.1: Links: een promotiefoto voor *Toy Story 3*. Er is gebruik gemaakt van een rendersysteem gebaseerd op fysische principes. Rechts: een frame uit *Spider-Man: Into the Spider-Verse*. Het aangepaste rendersysteem zorgt voor een visuele stijl die lijkt op die van een stripboek.

Om een beter begrip van dit systeem te krijgen, heb ik in dit werkstuk besloten om vanuit zoveel mogelijk pure wiskunde een ray-tracer zelf te bouwen. De vraag die ik zal stellen, luidt:

*Hoe wordt een driedimensionale renderer op basis van fysische principes geschreven?*

# Hoofdstuk 3

## Doelstelling

Het doel van dit werkstuk is om een 3d-renderer te ontwerpen die gebruikmaakt van fysieke principes, zoals lichtreflectie, om een realistische scène weer te geven. Hierbij wil ik ray-tracing gebruiken met andere modellen om scènes zo natuurgetrouw mogelijk weer te geven. Snelheidsoptimalisaties zijn een tweede zaak, en hier wil ik vooral aan beginnen als ik voel dat ik al een goed visueel resultaat heb bereikt en als ik nog tijd over heb: een functioneel prototype is voor mij belangrijker. Hiervoor stel ik enige eisen:

1. (Minimaal:) Het systeem ondersteunt de invoer van bollen, met een door de gebruiker opgegeven kleur en emissiewaarde, en zal als uitvoer een beeld geven van het verkregen licht van een camera.
2. (Minimaal:) Het systeem ondersteunt het Lambertiaanse diffusie lichtmodel.
3. Het project moet door een gebruiker voldoende aangepast kunnen worden. Er moet een duidelijke grens zitten tussen de interne broncode van de renderer en het opstarten daarvan, en de invoer van de data naar dit systeem: een gebruiker moet geen broncode hoeven aan te passen om een bepaalde scène weer te geven en zal alleen code moeten schrijven om data in het systeem te voeren.
4. De gebruiker moet een afweging kunnen maken tussen ruishoeveelheid en rendertijd.
5. Het moet voor de gebruiker duidelijk zijn wat de renderer op ieder moment aan het doen is: informatie zoals het huidige renderframe moet zichtbaar zijn voor de gebruiker.

Modulariteit heeft voor mij een grote rol gespeeld in mijn keuze voor dit project. Zo heb ik ook een lijst met mogelijke toevoegingen voor het project, als er tijd voor is:

1. (vervuld) De gebruiker kan naast bollen ook driehoeken doorsturen naar de renderer als bouwsteen om complexere objecten te tonen.
2. (vervuld, vervangen met hieronder) De renderer ondersteunt lichtreflectie door middel van het Phong lichtmodel. [1]
3. (vervuld) Of: het lichtmodel werkt op basis van de fysische principes van de microfacettheorie. [2]
4. (vervuld) De gebruiker kan een texture-atlas en texture-coördinaten doorgeven aan de renderer om complexere materialen weer te geven.
5. (vervuld) Het systeem wordt versneld door gebruik te maken van verkregen bound-volumes.
6. Het systeem wordt verder versneld met recursieve bound-volume hierarchieën. [3]
7. (vervuld) De gebruiker kan tijdens de visuele modus van de renderer bewegingsinput invoeren om de camera te bewegen. (onderdeel van testcode)
8. De renderer houdt de afgelide van een texture-sample ten opzichte van schermverandering bij, zodat texture-detail optimaal kan worden opgevraagd. [4]
9. (vervuld) Er kan een focusafstand en focussterkte worden ingesteld voor de gesimuleerde camera.
10. (vervuld) Er is ondersteuning voor het invoeren van een skybox, die wordt gebruikt wanneer een lichtstraal niets raakt.

# Hoofdstuk 4

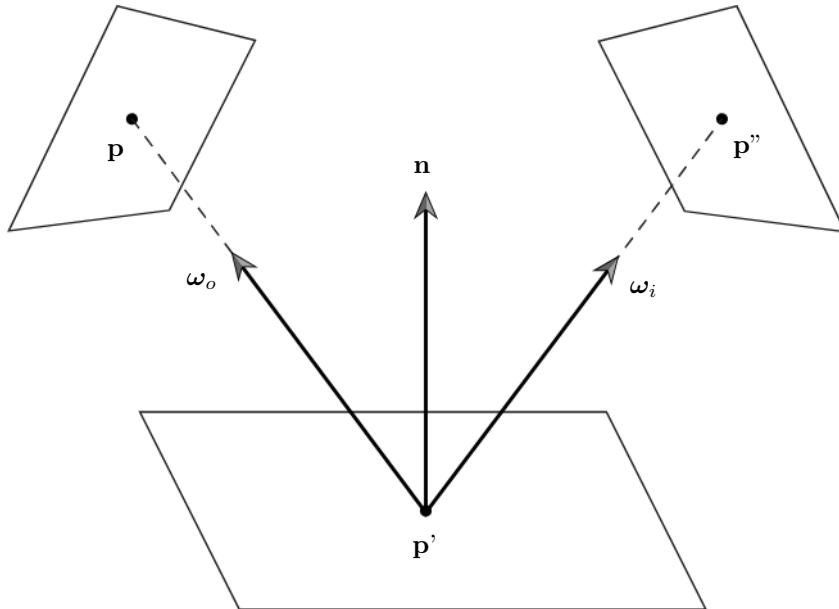
## Theoretisch kader

Hier zal een korte beschrijving en herleiding volgen van de wiskundige theorie waaruit dit werkstuk is opgebouwd.

### 4.1 Lichttransportvergelijking

De kernvergelijking voor physically-based rendering is de lichttransportvergelijking [5]:

$$L_o(\mathbf{p}, \omega_o) = L_e(\mathbf{p}, \omega_o) + L_r(\mathbf{p}, \omega_o)$$
$$L_o(\mathbf{p}, \omega_o) = L_e(\mathbf{p}, \omega_o) + \int_{S^2} f_r(\mathbf{p}, \omega_o, \omega_i) L_i(\mathbf{p}, \omega_i) |\cos\theta_i| d\omega_i$$



Figuur 4.1: Illustratie van lichttransport in een punt  $\mathbf{p}$ , PBRT 4th ed., Fig. 13.2 [6].

Hier geldt:

- $\mathbf{p}$  is het punt waarop reflectie plaatsvindt
- $\omega_o$  is de genormaliseerde richtingsvector waar het licht naartoe gaat
- $L_o$  is het licht dat in de richting  $\omega_o$  vanuit  $\mathbf{p}$  komt
- $L_e$  is het licht dat in de richting  $\omega_o$  wordt uitgestraald door  $\mathbf{p}$  (als lichtbron)
- $L_r$  is het licht dat vanuit andere bronnen in de richting  $\omega_o$  wordt gereflecteerd door  $\mathbf{p}$
- $\omega_i$  is een genormaliseerde richtingsvector waar het licht vandaan kan komen
- $L_i$  is het licht dat vanuit  $\omega_i$  op het materiaal schijnt
- $|\cos\theta_i| = \mathbf{n} \cdot \omega_i$  is de cosinusweging
- $f_r$  is de Bidirectional Reflectance Distribution Function, of BRDF
- $\mathbf{S^2}$  is de hemisfeer (halve bol) waar het licht vandaan komt

Wat hier eigenlijk staat, spreekt nogal voor zich: Het licht dat straalt naar een bepaalde hoek, is gelijk aan de som van de uitstraling in die richting en de (oneindige) som van al het licht dat in deze richting wordt gereflecteerd op basis van de BRDF-functie van het materiaal, vanuit elke hoek.

Het licht in  $\mathbf{p}$  is dus afhankelijk van het licht  $L_i$  in alle andere punten die via een  $\omega_i$  in  $\mathbf{S^2}$  wijzen, en die zijn dan weer afhankelijk van allemaal ander licht. Dit is een boom die zich op elke splitsing herhaaldelijk opsplitst in oneindig veel verschillende takken. Het is dus niet mogelijk om dit exact te berekenen. Algebraïsch kom je ook niet ver. Het doel van ray-tracing is dan ook om deze vergelijking te benaderen:

## 4.2 Monte Carlo Integratie

Monte Carlo integratie is een methode om een integraal (die hierboven, recursief) herhaaldelijk met willekeurige data uit te voeren en op deze wijze de integraal te benaderen. Hier geldt dat de meetfout proportioneel is aan  $\frac{1}{\sqrt{N}}$ , met  $N$  als het aantal samples. Binnen raytracing is dit het nemen van het 'gemiddelde' van meerdere willekeurige lichtstralen. De hoeveelheid samples die genomen worden, is hier de rays per pixel, of rpp.

## 4.3 Snijdingsdetectie

Om een lichtstraal te kunnen laten reflecteren, moeten we eerst weten waar een lichtstraal een object raakt. Dit kan worden gedaan zonder het licht als een bewegend object te simuleren. We kunnen met simpele snijdingsalgebra in een hogere dimensie de snijpunten vinden tussen lijnen en objecten. Een lichtstraal wordt in code ook altijd beschreven als een lijn in vectorvorm, en niet als een bewegend punt.

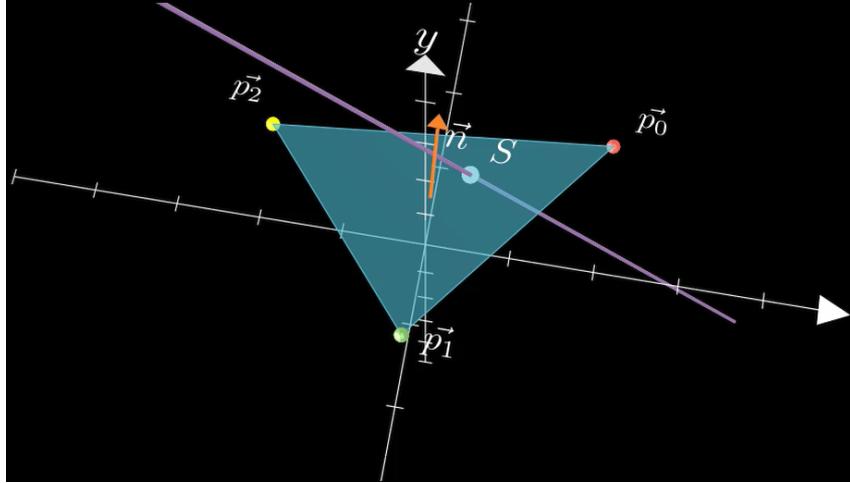
### 4.3.1 Snijpunt bal en lijn

Over het algemeen renderen 3D-renderers alleen met driehoeken. Om echter goed mijn logica te kunnen controleren, moet ik soms een frame in een erg korte tijd kunnen laden. Als ik de snijdingslogica voor bollen in mijn code apart schrijf, zal dit een stuk sneller laden dan wanneer ik deze zou opbouwen uit driehoeken. Neem een cirkel  $c$  met middelpunt  $\mathbf{P}$  en straal  $r$  en neem een lichtstraal met oorsprong  $\mathbf{o}$  en richting  $\mathbf{d}$ :

$$\begin{aligned} \begin{pmatrix} x \\ y \\ z \end{pmatrix} &= \mathbf{o} + \mathbf{dt} \\ r^2 &= \left\langle \begin{pmatrix} x \\ y \\ z \end{pmatrix} - \mathbf{P}, \begin{pmatrix} x \\ y \\ z \end{pmatrix} - \mathbf{P} \right\rangle \\ r^2 &= \langle \mathbf{o} + \mathbf{dt} - \mathbf{P}, \mathbf{o} + \mathbf{dt} - \mathbf{P} \rangle \\ r^2 &= |\mathbf{d}|^2 \cdot t^2 + 2 \langle \mathbf{d}, \mathbf{o} - \mathbf{P} \rangle \cdot t + |\mathbf{o} - \mathbf{P}|^2 \\ |\mathbf{d}|^2 \cdot t^2 + 2 \langle \mathbf{d}, \mathbf{o} - \mathbf{P} \rangle \cdot t + |\mathbf{o} - \mathbf{P}|^2 - r^2 &= 0 \\ a = |\mathbf{d}|^2, \quad b = 2 \langle \mathbf{d}, \mathbf{o} - \mathbf{P} \rangle, \quad c = |\mathbf{o} - \mathbf{P}|^2 - r^2 & \\ D = 4 \langle \mathbf{d}, (\mathbf{o} - \mathbf{P}) \rangle^2 - 4 \cdot |\mathbf{d}|^2 \cdot (|\mathbf{o} - \mathbf{P}|^2 - r^2) & \\ t_{1,2} = \frac{2 \langle \mathbf{d}, \mathbf{o} - \mathbf{P} \rangle \pm \sqrt{D}}{2 |\mathbf{d}|^2} & \end{aligned}$$

Het punt kan gevonden worden door  $t$  in te vullen, maar deze is niet nodig voor de detectie en zal pas later uitgerekend worden.

### 4.3.2 Snijpunt face en lijn



Figuur 4.2: Een snijding tussen een ray en een face.

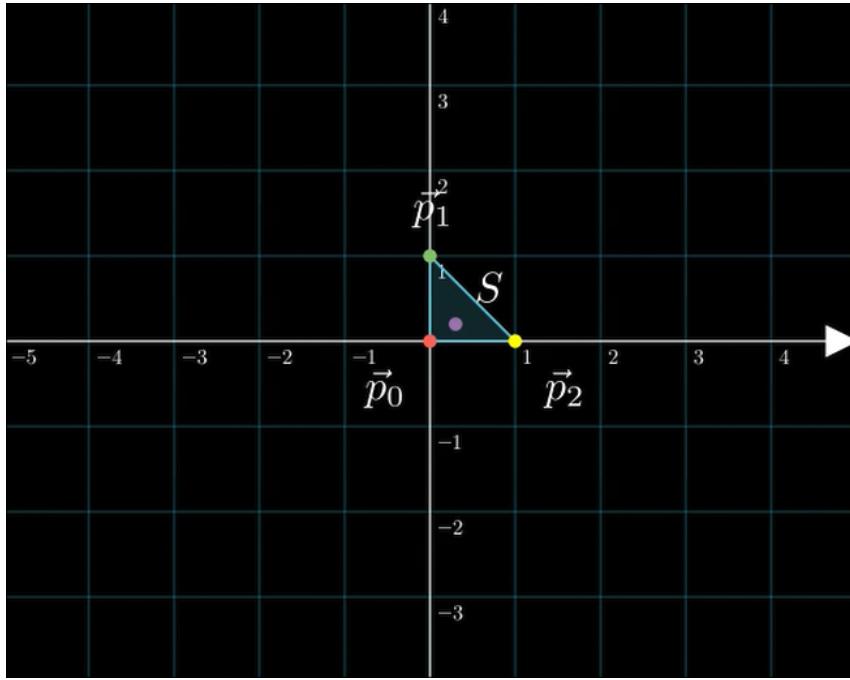
Ray-face-snijding is de snijding die het meest zal worden uitgevoerd: Een face, een driehoek tussen drie punten in 3d-ruimte, dient als de basisbouwsteen van elk 3d-model.

Eerst wordt het vlak berekend waarop deze driehoek ligt; dan wordt het snijpunt tussen de lijn en dit vlak berekend. Daarna wordt de vector van  $\mathbf{p}_0$  naar dit snijpunt genomen, en na een transformatie wordt bekeken of deze binnen de driehoek ligt.

$$\begin{aligned}
 \mathbf{d}_1 &= \mathbf{p}_2 - \mathbf{p}_0 \\
 \mathbf{d}_2 &= \mathbf{p}_1 - \mathbf{p}_0 \\
 \mathbf{n} &= \mathbf{d}_1 \times \mathbf{d}_2 \\
 V : \langle \mathbf{n}, \mathbf{s} \rangle + d &= 0 \\
 d &= -\langle \mathbf{n}, \mathbf{p}_0 \rangle \\
 V : \langle \mathbf{n}, \mathbf{s} \rangle - \langle \mathbf{n}, \mathbf{p}_0 \rangle &= 0 \\
 \mathbf{s} &= \mathbf{o} + \mathbf{dt} \\
 \langle \mathbf{n}, (\mathbf{o} + \mathbf{dt}) \rangle - \langle \mathbf{n}, \mathbf{p}_0 \rangle &= 0 \\
 t_h &= -\frac{\langle \mathbf{n}, \mathbf{o} \rangle - \langle \mathbf{n}, \mathbf{p}_0 \rangle}{\langle \mathbf{n}, \mathbf{d} \rangle} \\
 \mathbf{h} &= \mathbf{o} + \mathbf{dt}_h
 \end{aligned}$$

$$\begin{aligned}
 M &= (\mathbf{d}_1 \quad \mathbf{d}_2 \quad \mathbf{n}) \\
 \mathbf{h}' &= (\mathbf{h} - \mathbf{p}_0) \cdot M^{-1}
 \end{aligned}$$

Als laatste stap wordt de derde component van  $\mathbf{h}'$  verwijderd. Dit levert een barycentrisch coördinatensysteem op waar  $(0, 0)$   $\mathbf{p}_0$  represeneert,  $(0, 1)$  voor  $\mathbf{p}_2$  en  $(1, 0)$  voor  $\mathbf{p}_1$ . Hieruit volgt dat alle punten waarvoor geldt dat  $x + y < 1 \wedge x > 0 \wedge y > 0$  binnen de driehoek liggen. Deze waarden worden gebruikt om te vinden waar op de texture-map gelezen moet worden, via interpolatie over texturegegevens die in elk hoekpunt zijn opgeslagen.



Figuur 4.3: De 2D-transformatie van een snijding in het punt S.

### 4.3.3 Snijpunt doos en lijn

Complexe objecten bestaan vaak uit honderden driehoeken. Als elke lichtstraal de snijdingscontrole uitvoert met elke driehoek van elk object, zullen de renders zeer lang duren. Daarom wordt als tussenstap een bounding-box gebruikt: Een doos, met ribben parallel aan de x-, y- en z-as, wordt eerst berekend op de processor, waarvan gegeven is dat alle hoekpunten van alle driehoeken erin liggen, door de laagste en de hoogste x-, y- en z-waarde te vinden van alle punten, en dit te gebruiken als de twee hoekpunten van de doos. Nu zal eerst gekeken worden of een lichtstraal de doos snijdt, voordat alle driehoeken gecontroleerd worden. Dit gaat als volgt:

$$\mathbf{p}_{min} = \begin{pmatrix} x_{min} \\ y_{min} \\ z_{min} \end{pmatrix}, \mathbf{p}_{max} = \begin{pmatrix} x_{max} \\ y_{max} \\ z_{max} \end{pmatrix}$$

$$l : \mathbf{o} + \mathbf{dt}$$

$$t_{0,i} = \frac{\mathbf{p}_{min,i} - \mathbf{o}_i}{\mathbf{d}_i}$$

$$t_{1,i} = \frac{\mathbf{p}_{max,i} - \mathbf{o}_i}{\mathbf{d}_i}$$

$$t_{in} = \max(t_{0,x}, t_{0,y}, t_{0,z})$$

$$t_{uit} = \min(t_{1,x}, t_{1,y}, t_{1,z})$$

Als  $t_{in} < t_{uit} \wedge t_{in} > 0$ , wordt de doos gesneden.

## 4.4 Emissie en absorptie

Normaal begint licht van een bepaalde set golflengten uit een lichtbron en reflecteert het zich door de scène, waarbij het spectrum van het licht elke keer wordt vermenigvuldigd met het emissiespectrum van het materiaal. De mens kan echter maar drie groepen golflengten zien, en een computerscherm kan alleen de pieken van die groepen met LED uitzenden. Zo kan een emissiespectrum in bijna alle gevallen worden versimpeld tot een 3d-vector, waarbij x voor rood staat, y voor groen en z voor blauw. Voor een pad van één lichtbron na reflectie over  $n$  objecten:

$$\mathbf{L}_{i,\text{camera}} = \mathbf{L}_{e,\text{lichtbron}} \cdot \prod_n^{k-1} \mathbf{f}_n$$

$$\mathbf{f}_n = f_r(\mathbf{p}_n, \omega_o, \omega_i) \mathbf{L}_i(p, \omega_i) |\cos\theta_i| d\omega_i$$

en recursief:

$$\mathbf{U}_0 = \mathbf{L}_{e,\text{bron}}$$

$$\mathbf{U}_{n+1} = \mathbf{f}_n \cdot \mathbf{U}_n$$

$$\mathbf{f}_n = f_r(\mathbf{p}_n, \omega_o, \omega_i) L_i(p, \omega_i) |\cos\theta_i| d\omega_i$$

Zie ook 4.1.

Er wordt in raytracing licht getraceerd van de camera naar de bron, in plaats van andersom. Hierdoor moet onze methode iets anders worden, met een filter-vector en een throughput-vector. Het filter is het product van de reflectiekleur van alle materialen en dient als een filter voor welk licht er kan schijnen. De throughput is de som van alle bereikte lichtbronemissies, vermenigvuldigd met het filter op dat punt. Dit zal de uiteindelijke kleur zijn.

$$\mathbf{F}_0 = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$$

$$\mathbf{T}_0 = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$$

$$\mathbf{F}_{n+1} = \mathbf{F}_n \cdot \mathbf{f}_n$$

$$\mathbf{T}_{n+1} = \mathbf{T}_n + \mathbf{L}_{e,n} \cdot \mathbf{F}_n$$

Laten we dit controleren, met alweer één lichtbron na  $k$  objecten:

$$\mathbf{F}_{n+1} = \mathbf{F}_n \cdot \mathbf{f}_n$$

$$\mathbf{F}_k = \prod_n^{k-1} \mathbf{f}_n$$

$$\mathbf{T}_{n+1} = \mathbf{T}_n + \mathbf{L}_{e,n} \cdot \mathbf{F}_n$$

$$\mathbf{T}_k = \sum_n^{k-1} \mathbf{L}_{e,n} \cdot \mathbf{F}_n$$

$$\mathbf{T}_k = \mathbf{L}_{e,k} \cdot \mathbf{F}_k$$

$$\mathbf{T}_k = \mathbf{L}_{e,k} \cdot \prod_n^{k-1} \mathbf{f}_n$$

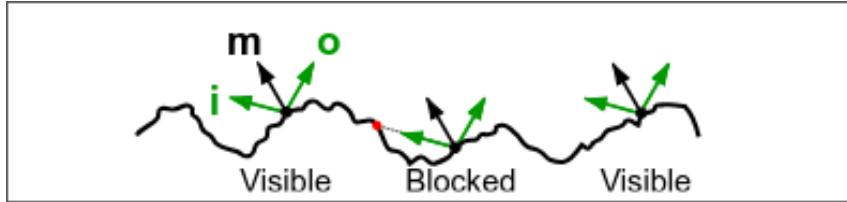
$$\mathbf{L}_{i,\text{camera}} = \mathbf{L}_{e,\text{lichtbron}} \cdot \prod_n^{k-1} \mathbf{f}_n$$

De som kan vereenvoudigd worden naar de laatste term, aangezien dit de enige straler is. Dit produceert dus dezelfde vergelijking als van bron naar camera.

## 4.5 BRDF: de weging van kleuren

Formeel is de BRDF een functie die lichtintensiteit beschrijft aan de hand van een inrichting, een uitrichting, en het materiaal. Gezien de volgorde waarin mijn tracer licht simuleert en de Monte Carlo methode die ik toepas, is het echter handiger om deze beide een nieuwe ray-richting en een lichtintensiteit te geven. De BRDF bestaat uit twee onderdelen; een spiegelend deel en een diffuus deel.

### 4.5.1 Spiegeling met GGX



Figuur 4.4: Drie voorbeelden van een reflectie op microniveau. [2]

Om een realistischer beeld te verkrijgen, moet een weerspiegelingsmodel worden gebruikt dat nauw gebaseerd is op bestaande natuurkundige principes. In werkelijkheid zijn de zichtbare eigenschappen van een materiaal op macroniveau te verklaren door microscopische onevenheden in het materiaal: microfacets. Licht reflecteert dan ook niet over de normaalvector van het gehele materiaal, maar over de normaalvectoren van deze microfacets. Gezien een computer er niet toe in staat is om dit te simuleren, wordt een statistisch model gebruikt om te voorspellen wat de kans is dat een reflectie een bepaalde kant op wijst.

De verdeling van deze microfacets ten opzichte van de normaalvector wordt beschreven door de functie  $D(\omega_m)$ , met  $\omega_m$  als normaalvector van een microfacet. Ook moet er gerekend worden met de fresnelvergelijkingen. Deze beschrijven de mate waarin materialen reflectiever worden wanneer ze onder een minder scherpe hoek worden bekeken. Hoeveel licht dit doet, wordt beschreven door de functie  $F(x)$ . Als laatste is er ook nog een kans dat gereflecteerd licht toch wordt geblokkeerd door het materiaal zelf, op een ander punt (fig. 4.4). Ook dit wordt beschreven met een functie:  $G(\omega_m)$ . Zo komen we uiteindelijk met de volgende functie:

$$f_r(\omega_i, \omega_o) = \frac{D(\omega_i) F(\omega_i \cdot \omega_o) G(\omega_i, \omega_o)}{4(\mathbf{n} \cdot \omega_i)(\mathbf{n} \cdot \omega_o)}$$

waar  $\omega_i$  zoals gewoonlijk de inrichting van het licht is, en  $\omega_o$  de uitrichting. Met het invullen van deze functies heb ik gekozen voor GGX, omdat dit een realistisch, maar ook niet al te traag systeem is voor lichtreflectie. GGX geeft dat:

$$D(\omega_m) = \frac{\alpha^2}{\pi((\mathbf{h} \cdot \omega_m)^2(\alpha^2 - 1) + 1)^2}$$

Dit is met  $\alpha$  als instelbare parameter die aangeeft hoe ruw een materiaal is: 0 zou een perfecte spiegel zijn, en 1 een materiaal dat absoluut niet reflecteert.  $\mathbf{h}$  is de vector voor de helft tussen de in- en uitvector:

$$\mathbf{h} = \frac{\omega_i + \omega_o}{|\omega_i + \omega_o|}$$

Voor  $F(x)$  gebruik ik een benadering van de fresnelvergelijking om rekentijd te besparen:

$$F(x) = F_0 + (1 - F_0)(1 - x)^5$$

$F_0$  is ook een instelbare parameter voor het materiaal en geldt waar  $\theta = 90^\circ$ . Ook geldt dat  $x = \cos \theta = \omega_i \cdot \omega_o$ . Dit volgt niet exact het behoud van energie, maar de benadering is goed genoeg om nog steeds een realistisch resultaat te krijgen. Als laatste  $G(x)$ : Bij GGX wordt meestal deze formule gebruikt:

$$G(\omega_i, \omega_o) = \frac{2(\mathbf{h} \cdot \omega_i)(\mathbf{h} \cdot \omega_o)}{(\mathbf{h} \cdot \omega_o)\sqrt{\alpha^2 + (1 - \alpha^2)(\mathbf{h} \cdot \omega_i)^2} + (\mathbf{h} \cdot \omega_i)\sqrt{\alpha^2 + (1 - \alpha^2)(\mathbf{h} \cdot \omega_o)^2}}$$

Het zou echter zeer inefficiënt zijn om elke reflectie een willekeurige kant op te laten gaan en de lichtintensiteit te vermenigvuldigen met de reflectiekans, aangezien een deel van deze functie voorspelbaar is. Ik maak gebruik van deze inverse cdf voor GGX:

$$\theta_m = \arccos \sqrt{\frac{1 - \xi_o}{\xi(\alpha^2 - 1) + 1}} \quad \phi = 2\pi\xi_1$$

waar  $\xi_0$  en  $\xi_1$  uniforme willekeurige getallen zijn tussen 0 en 1.  $\theta$  en  $\phi$  worden gebruikt als de twee hoeken die vervolgens de richtingsvector berekenen, in een ruimte waar geldt dat  $\mathbf{n} = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}$ . Dit versimpelt de oorspronkelijke formule naar

$$\frac{f(\omega_i, \omega_o)|\omega_i \cdot \mathbf{h}|}{p_i(\omega_m)} = \frac{F(\omega_i, \omega_m)G(\omega_i, \omega_o, \omega_m)|\omega_o \cdot \omega_m|}{|\omega_o \cdot \mathbf{h}||\omega_m \cdot \mathbf{h}|}$$

$p_i(\omega_m)$  is de kansverdeling van onze richtingsvectorgeneratie, dus kan genegeerd worden. Ook wordt cosinusweging hierin meegerekend. Dit betekent dat de rechterhelft van de vergelijking kan worden gebruikt als BRDF, en dat  $D(x)$  niet meer berekend hoeft te worden. Mijn implementatie van dit systeem is vooral gebaseerd op [deze blogpost](#).

#### 4.5.2 Diffuse verlichting

Hierbij wordt nog steeds vastgehouden aan het Lambertiaanse model. Ik genereer simpelweg een vector in een willekeurige richting en deel de intensiteit door  $\pi$  om ervoor te zorgen dat er niet meer licht uit de halve bol van richtingen kan komen dan dat erin gaat. Ten slotte wordt er willekeurig gekozen tussen spiegeling en diffusie, en wordt het resultaat ook nog gewogen op basis van de ingevulde  $F_0$ . Na een voldoende aantal samples zal dit leiden tot een realistisch eindresultaat.

## 4.6 De camera

Het instellen van een camerapositie bleek lastiger dan aanvankelijk aangenomen. Daarom volgt hier een korte beschrijving van de logica die ik heb toegepast.

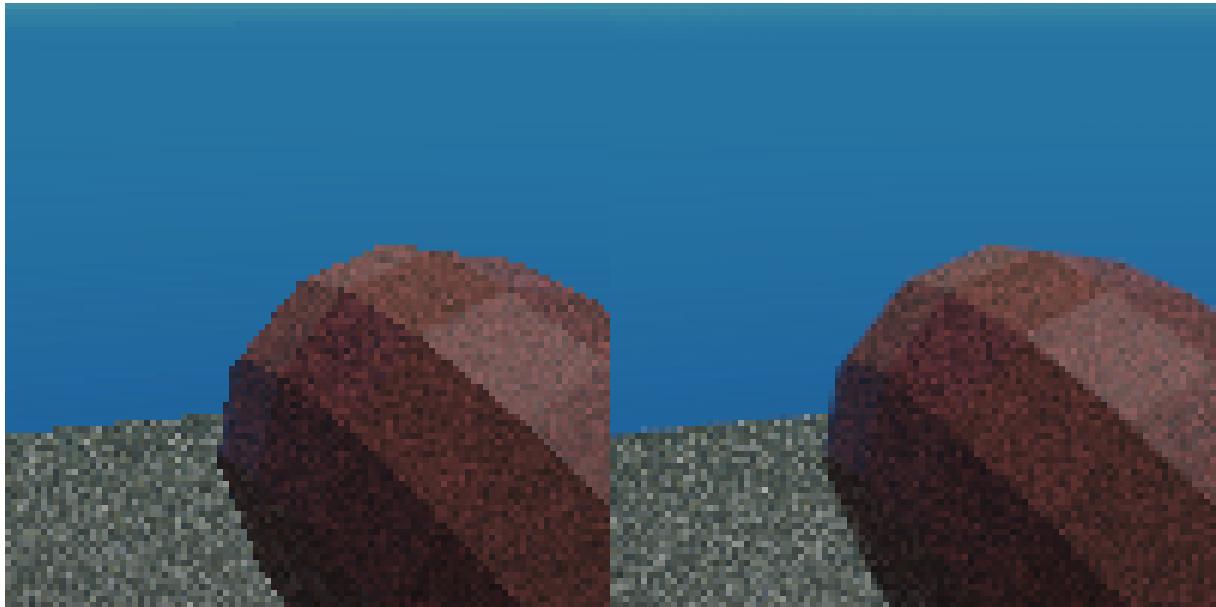
### 4.6.1 Camerarichting

Een camera heeft niet genoeg aan enkel een positie en een richtingsvector. Stel je een potlood voor dat recht omhoog wijst. Dit potlood kan, terwijl het de lucht in wijst, nog steeds draaien in een rollbeweging. Zo kan een camera ook naar bijvoorbeeld de positieve x-richting kijken, maar toch op zijn kop staan. Vandaar dat er extra ruimte nodig is. Dit kan door middel van quaternions, maar dit onderwerp was voor mij iets te complex om toe te passen. Daarom maak ik simpelweg gebruik van een 3x3 view-matrix, met als x- en y-richting datgene dat voor de kijker rechts en omhoog zou zijn, en als z-richting de echte kijkrichting. Al deze vectoren staan in rechte hoeken tot elkaar, en dit betekent dat er deels overbodige informatie wordt gestuurd. Ik doe dit echter toch omdat dit een eenvoudigere implementatie is. Vermenigvuldiging met deze view-matrix zal een vector relatief tot de camera dus relatief tot de wereld maken.

### 4.6.2 Besturing van de camera

Om goed te demonstreren dat het geen driedimensionaal is, heb ik een eenvoudig bewegingsprogramma gemaakt dat vanuit een inputthread beweging stuurt naar het camerasysteem. Voor de draairichting van de camera zou normaal in de context van 3d-games de positie van de muis gebruikt worden, maar ik heb gekozen voor de pijltjestoetsen, met dezelfde mapping. Hier geldt dat twee waarden,  $(\phi, \theta)$ , worden gebruikt en met elke input een bepaalde hoeveelheid graden verschoven worden. Hiermee wordt de camerarichting berekend met draaiingsmatrices.

### 4.6.3 Focus

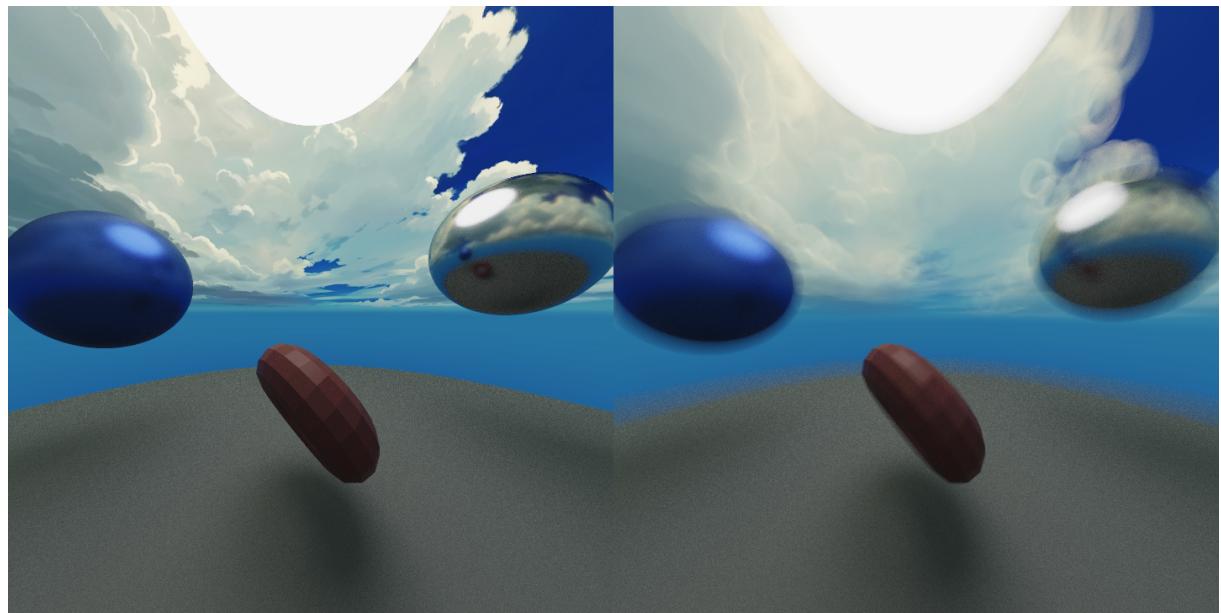


Figuur 4.5: Een vergelijking van Anti-aliasing. Links is deze niet gebruikt, rechts wel.

Er wordt ook random noise gebruikt om de lichtrichting te vinden. Als eerste wordt er een zeer kleine afwijking bij elke richtingsvector toegevoegd, wat dient als natuurlijke Anti-Aliasing: een manier om schuine lijnen op een computerscherm gedetailleerder te laten lijken, door discrete pixelstappen te blurren naar een mooiere lijn.

Daarbovenop kan een sterkere afwijking worden toegevoegd: Focussterkte. Nu wordt de startlocatie van de lichtvector in een willekeurige richting verplaatst, en hiervoor wordt gecorrigeerd door de richting

aan de andere kant te plaatsen. Dit zorgt ervoor dat alleen op een bepaalde gegeven focusafstand alle mogelijke afstanden van de pixel op hetzelfde punt terechtkomen, wat zich voordoet in een focus.



Figuur 4.6: Op de rechterhelft zijn alle dingen op een afstand van meer of minder dan twee meter onduidelijk.

# Hoofdstuk 5

## Methoden

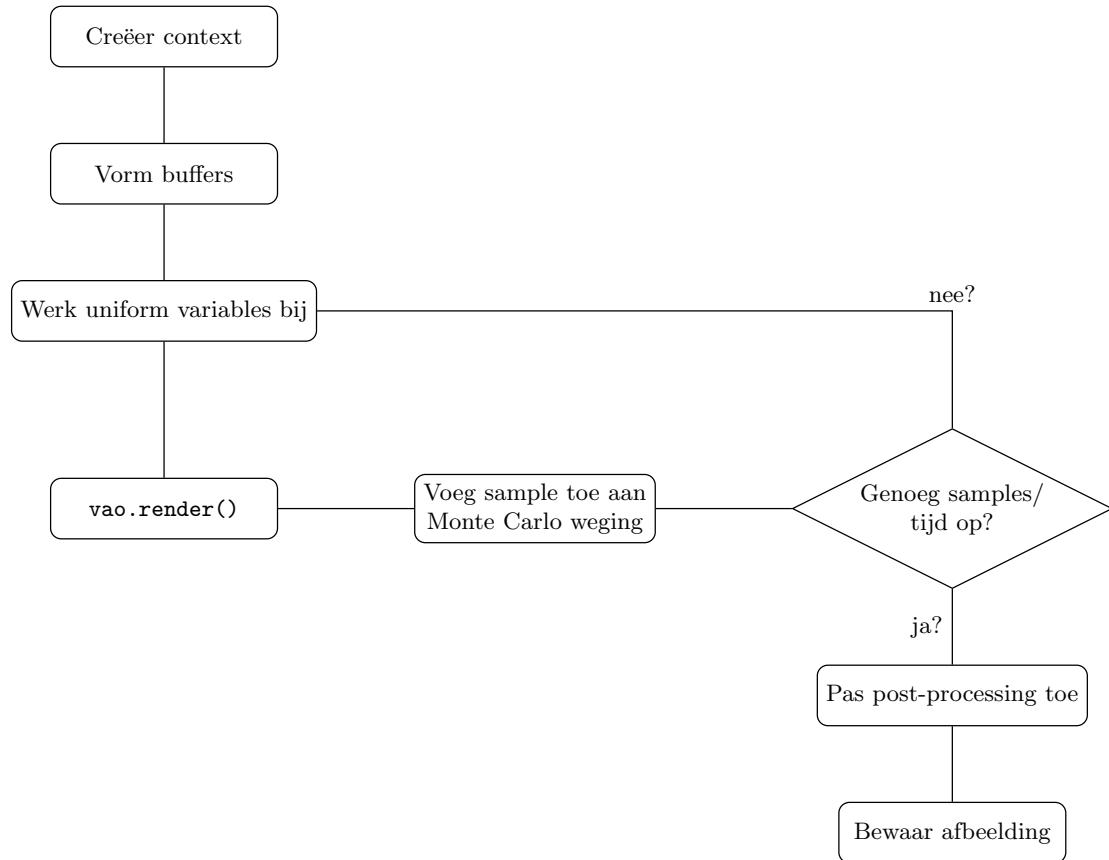
In de toepassing bestaat dit werkstuk uit drie kernbestanden:

1. `tracer.frag`: de raytracinglogica, in gpu-code
2. `main.py`: de processorcode die de gpu-code opstart en aanstuurt
3. `conf.py`: de code die een gebruiker zou schrijven om informatie door te sturen naar het project

Hierbij is het dus belangrijk dat mijn implementatie van `conf.py` en de andere hulpbestanden die dingen hiervoor klaarzetten, enkel dienen voor debugging, het laten zien van resultaten en als voorbeeldimplementatie: als auto zou `tracer.frag` de motor, `main.py` de andere interne onderdelen en `conf.py` de rest van de auto zijn, waarbij mijn project vooral zou gaan over motorontwerp. Ik heb dus los mijn eigen programma's gemaakt die

1. 3d-modellen maken en laten bewegen
2. textures mappen naar die modellen
3. de gebruiker de camera laten bewegen terwijl de render runt
4. verschillende testkleuren en -scenes maken om te laten zien dat mijn renderer werkt

## 5.1 main.py



Figuur 5.1: De loop van `main.py`

`main.py` bevat de kernloop van mijn programma en wordt uitgevoerd op de CPU. Hij stuurt alle ontvangen objectdata op nadat deze in het juiste formaat is omgezet, start de renderer en ontvangt de verkregen render. In mijn geval is efficiëntie voor deze code niet van het grootste belang, aangezien het knelpunt van mijn code altijd bij de videokaart zal liggen. Dit komt doordat alle complexe berekeningen uitgevoerd worden op de GPU, en het zeker is dat de CPU moet wachten totdat de GPU klaar is met de render.

Als eerste zal er voor het programma een window-context gecreëerd worden, wat ik overlaat aan `pygame` en `modernGL`. Hierna maak ik buffers aan die dienen als de tunnels waardoor de processor informatie over de vorm, locatie en tijd van de scène kan laden. Ook wordt er een plek aangemaakt waar de gerenderde texture bewaard kan worden in geheugen, en een plaats voor de Monte Carlo texture. De uv-textures van de materialen worden ook geladen. Hierbij geldt de mapping:

Buffer ID	Buffernaam
0	ping texture
1	pong texture
2	output tracer
3	albedo atlas
4	specular atlas
5	ambient occlusion atlas
6	displacement atlas
7	normal atlas
8	emission atlas
9	bal buffer
10	vertex buffer
11	bounding box buffer
12	bounding box naar vertex map buffer

Tabel 5.1: Mapping texturebuffers

Het is belangrijk dat de texturebuffers (0-8) een ander systeem gebruiken dan de SSBO buffers (9-12) en dat beide een getal tegelijk kunnen gebruiken. Toch heb ik ervoor gekozen om deze apart te houden, zodat er geen verwarring is over welk getal bij welke buffer hoort.

Ook worden de shaderbestanden geladen, met behulp van `pymoderngl`. Ik gebruik steeds één bestand genaamd `fs_quad.vert`, dat een  $1 \times 1$  fullscreen-quad laadt, waardoor alle scherminformatie direct wordt doorgegeven aan de fragment-shader, waar ik ray-tracing kan doen. Dit is effectief een bypass die het ingebouwde rendersysteem overslaat en direct pixelinformatie doorgeeft aan mijn shader. De twee fragmentshaders die ik heb, zijn `accumulator.frag` en `tracer.frag`. De accumulator telt de pixelinformatie van de Monte Carlo texture op bij de nieuw gegenereerde render met een gegeven `frameIndex`, die bepaalt hoe sterk de nieuwe texture meetelt in het gemiddelde.

Hierna wordt de loop gestart. De buffers worden bijgewerkt door middel van hun desbetreffende functie, die is geschreven in het configuratiebestand, om het systeem zo aanpasbaar mogelijk te maken. Hierna bereidt het de output-frame voor om in te schrijven en werkt het alle uniforme variabelen bij.

Nu, nog steeds binnen de loop, wordt `vao.render()` opgeroepen. Alle gegeven informatie wordt gebruikt om de shader te starten.

Dan wordt de accumulator gestart, die, nadat alle textures op hun plaats staan, de gegenereerde afbeelding optelt bij het gemiddelde. Hiervoor gebruik ik een ping-pongsysteem, waarbij ik steeds afwisselend een frame render van de ene plek en lees van de andere, en dan lees van de andere en render op de ene.

Hierna wordt de `frameIndex` geüpdatet, totdat het gewenste aantal iteraties is voltooid of de gewenste frametijd is bereikt.

## 5.2 tracer.frag: de shader

Dit is de kern van mijn profielwerkstuk. Na het ontvangen van alle data wordt er een lichtsimulatie gedaan op de verkregen objecten en wordt er een frame uitgezonden.

Als eerste definieer ik de variabelen die ik zal krijgen. Daarna stel ik enkele constanten in die op compile-tijd moeten worden ingesteld. Dan worden de objecten gedefinieerd:

- `Material(vec4 color, vec4 brdf)` materiaalinformatie
- `Ball(vec4 pos, mat)` een bal met een straal pos:  $(x, y, z, r)$
- `Line(vec3 pos, vec3 dir)` een lijn, in ons geval een lichtstraal, in de vorm  $\mathbf{O} + t\mathbf{D}$
- `CollisionT(float t1, float t2, float D)` snijdingsinformatie

### 5.2.1 `vec4 ballcollide(Line l, Ball c)`

Berekent het snijpunt tussen een bol en een lijn in 3d, aan de hand van de wiskunde van 4.3.1; geeft de discriminant.

### 5.2.2 `CollisionT ballcollision(Line l, Ball c)`

Gebruikt `ballcollide` om de discriminant te vinden, vult deze in in de abc-formule van 4.3.1, en controleert met  $D$  of er reële oplossingen zijn. Als er geen snijpunt is gevonden, wordt  $t$  naar het hoogst mogelijke getal gezet, om te zorgen dat alle andere snijdingen voorgaan.

### 5.2.3 `collision_data collide(Line l, vec4 face[3])`

controleert of een bepaalde face wordt gesneden door een lijn, aan de hand van 4.3.2.

### 5.2.4 `hitInfo getHit(ray, balls, faces)`

Voert de snijdingslogica uit voor elke vertex en bol in de scène. Als er niets wordt geraakt of de maximale reflecties zijn bereikt, past het de achtergrondverlichting toe met `getEnvironmentLight()`.

### 5.2.5 `brdf_data BRDF(hitInfo hit, Material mat, inout uint rngState)`

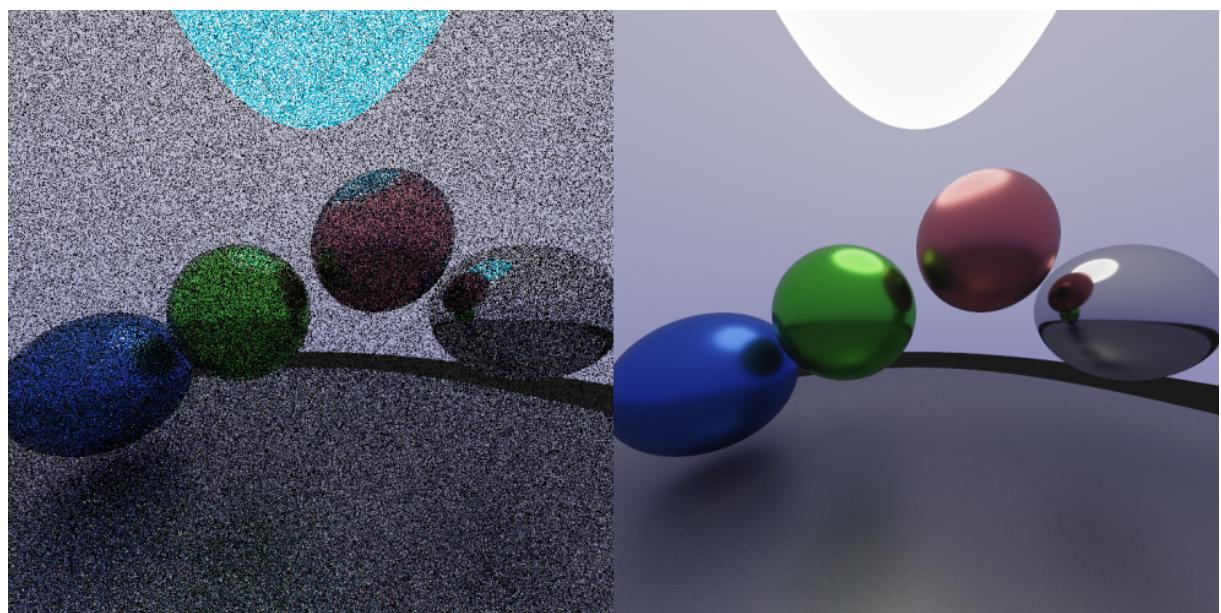
Gebruikt 4.5 om de richting en intensiteit van de volgende straal te berekenen, en geeft deze.

### 5.2.6 `vec3 trace(ray, balls, faces)`

Voert herhaaldelijk de `getHit()`-code uit en gebruikt de verkregen informatie en de functie `BRDF()` om een nieuwe richting te vinden voor de reflectielichtstraal. Past het filtersysteem toe van 4.4 en geeft de eindthroughput als eindwaarde.

### 5.2.7 `void main()`

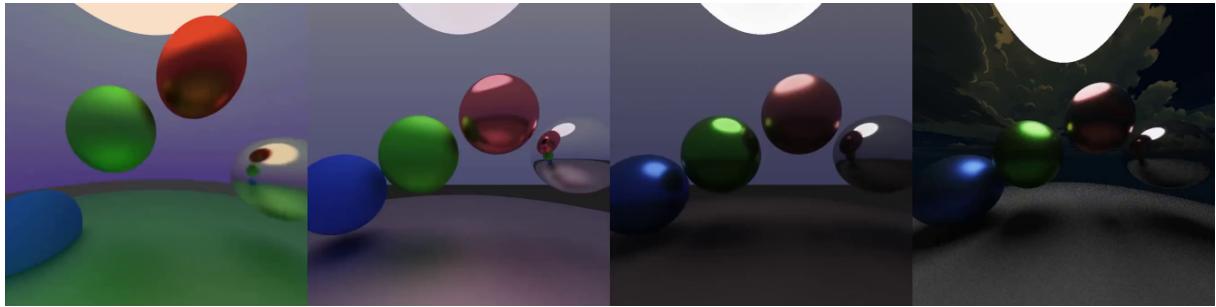
Voert de kernloop uit; wordt aangeroepen aan het begin van het programma. Berekent de startrichtingsvector met de pixelcoördinaten en `camdir()` en start `trace()`. Herhaalt dit proces aan de hand van de ingestelde ray-per-pixel; het kan efficiënter zijn om dit deels binnen de gpu te houden, dus het is het beste om zowel op de cpu als de gpu herhalingen uit te voeren.



Figuur 5.2: Een vergelijking van rays per pixel. Links een rpp van 10, en rechts 10.000.

# Hoofdstuk 6

## Proces



Figuur 6.1: De ontwikkeling van het rendersysteem.

### 6.1 AI

In dit profielwerkstuk is gebruikgemaakt van AI, maar enkel in een context waarin dit in mijn ogen mijn fundamentele werk niet vervangt. Dit is niet erg lastig te vervullen, gezien complete ray-tracingcode gewoon te complex is voor taalmodellen om te begrijpen. Ik heb meerdere keren codestukken gestuurd met de opdracht om te laten zien wat er fout is, als eerste filter om slordigheden en domme fouten eruit te halen. Alle genoemde algoritmen zijn zelf herleid of via genoemde bronnen gevonden. Ook specifieke dingen over de syntax van de gebruikte programmeertaal zijn aan AI gevraagd. Alle tekst is door mij geschreven, en een zeer klein deel (korte hulpfuncties) van de broncode is door AI geschreven. Als dit zo is, staat het boven de functie benoemd in de broncode.

### 6.2 Tijdsontwikkeling

Ik heb niet erg veel tijd besteed aan vooronderzoek, want in het uitzoeken of dit onderwerp geschikt zou zijn als profielwerkstuk had ik al erg veel basiskennis opgebouwd. Eerst was ik begonnen met WebGL, een webversie van OpenGL, en heb ik hierin een simpele bewegende bol laten werken. Ik liep echter vrij snel tegen het probleem aan dat alleen GLSL versie 1 hier goed wordt ondersteund. Ik ben om deze reden overgestapt naar pymodernGL, waar dit geen probleem was.

#### 6.2.1 De ontwikkeling van de BRDF

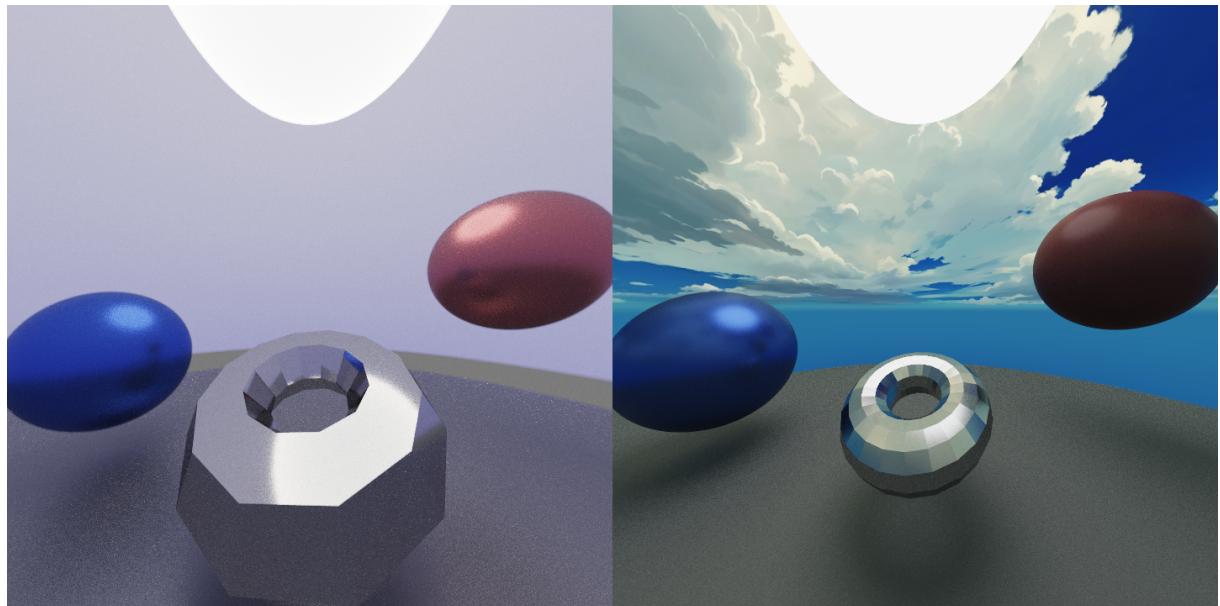
Het lichtsysteem is een onderwerp dat erg vaak is geïtereerd in de loop van het project. Eerst begon ze als een simpel binair systeem: als het object geraakt wordt, toon dan een kleur. Hierna had ik cosinusweging toegevoegd, wat de bol een schaduw gaf. Daarna kwam pas het tracingsysteem: de kleur van de bol werd afhankelijk van de kleur van een reflectie in willekeurige richting. Dit staat bekend als de Lambertian Diffuse. Even later ontwikkelde dit zich tot het Phong-model: er werd geinterpoleerd tussen de lambertian diffuse en een bekende reflectieverctor om de grofheid van een materiaal na te bootsen. Hier bleef het een tijdje op, totdat ik inzag dat de tijd ervoor was om de grootste verbetering te doen: Een werkelijk realistisch model: de Cook-Torrance GGX BRDF. Dit heeft zeer realistische beelden kunnen genereren.

### 6.2.2 De ontwikkeling van objectondersteuning

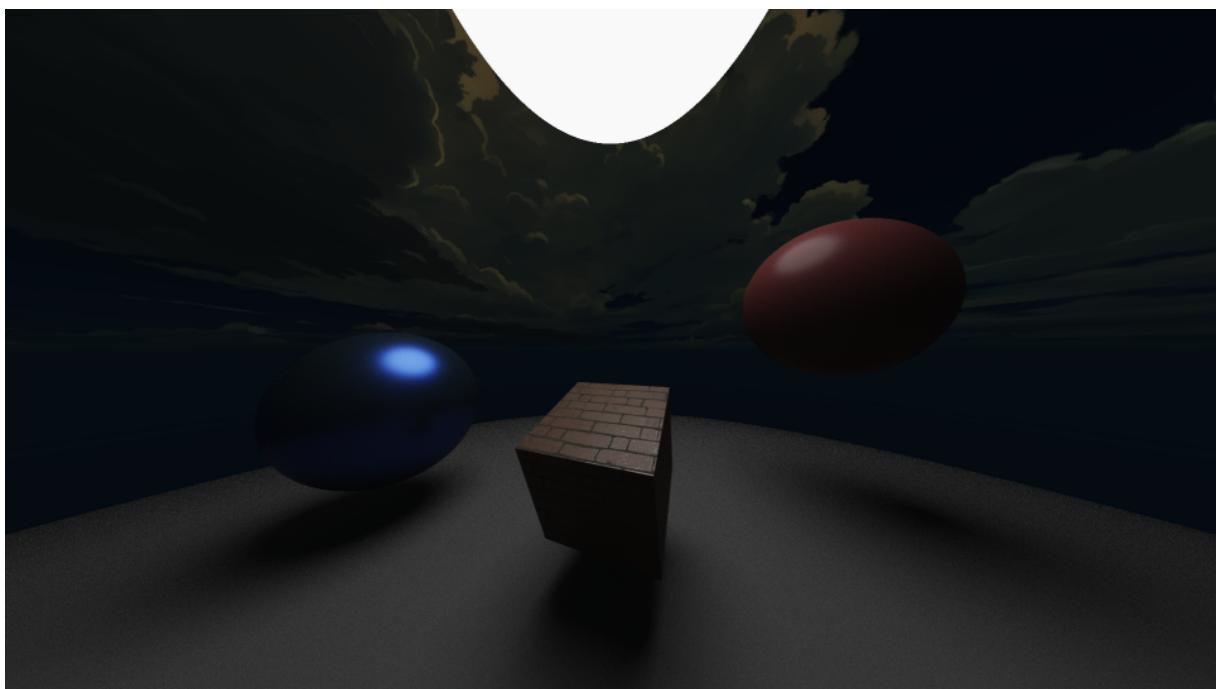
Aan het begin deed ik al mijn tests met bollen: deze zijn wiskundig simpel en een zeer grote groene bol kan goed dienen als ondergrond voor een eenvoudige scène. De eerste grote verandering was om deze bollen naar een buffer te verplaatsen, zodat het systeem binnen de CPU beter aangestuurd kon worden. Hierna kwam de grootste stap: Naast een bolbuffer had ik een vertexbuffer toegevoegd. Deze vertices konden na veel debuggen ook op het scherm worden getoond, al is dit wel tegen hoge snelheidskosten. Toen had ik textures toegevoegd: Mijn snijdingstest voor de faces gaf als bijproduct ook een relatieve locatie op de driehoek, wat perfect ruimte bood voor het toevoegen van texture-data, zodat objecten uit meerdere materialen per face kunnen bestaan. Als laatste hebben bounding volumes deze vertices erg versneld en dus bijgedragen aan een hoger limiet voor het aantal vertices.

### 6.2.3 Problemen

Ik ben tegen zeer veel problemen aangekomen tijdens het schrijven van code en heb een groot deel van mijn uren besteed in een staat van verwarring over deze codefouten. Zo heb ik een keer, bij het toevoegen van vertices en het testen van reflectie, lang nagedacht over de mogelijke reden waarom de kubus doorzichtig bleek te zijn. Uiteindelijk kwam ik erachter dat mijn kubus niet doorzichtig was, maar dat de perfecte reflectie van de kubus, in een testscène die perfect draaisymmetrisch is, exact hetzelfde eruitzag als transparantie.



Figuur 6.2: Een vergelijking van een hoeveelheid faces. Dankzij bounding box-optimisatie is de rendertijd van de tweede foto korter dan die van de eerste, sinds deze dit niet gebruikte.



Figuur 6.3: Een baksteentexture op een kubus.

# Hoofdstuk 7

## Conclusie

In dit profielwerkstuk is het schrijven van een ray-tracer, gebaseerd op natuurkundige principes, beschreven. Dit is gedaan op basis van de wiskunde van het theoretisch kader, met GLSL gpu-code en pymodernGL als brug tussen Python en OpenGL. Door gebruik te maken van de genoemde algoritmen kan er een beeld worden gevormd dat toeneemt in precisie met de toename in rendertijd.

De gestelde eisen zijn vervuld, en een groot deel van de extra doelen is ook vervuld. Ray-tracing is een erg breed onderwerp en ik heb op zeer directe wijze geleerd over onderwerpen van theoretische aard, zoals lineaire algebra en statistiek, tot meer toegepast onderzoek, zoals texture-atlassen, de opbouw van 3d-modellen en uv-mapping.

Enkel zijn een paar extra onderdelen, geschreven voor het geval van een extreem tijdoverschot, zoals glasrefractie, BVH's en ray differential LOD's, niet toegevoegd. Andere extra velden, zoals het ondersteunen van textures, zijn nog wel binnen de verkregen tijd vervuld.

# Hoofdstuk 8

## Discussie

Dit profielwerkstuk is grotendeels goed verlopen, met een paar lichte problemen onderweg. Zo is er gewerkt aan BVH's, maar dit is niet afgemaakt. Dit is te wijten aan mijn interesse in het algehele onderwerp, wat zeer gunstig is geweest voor het investeren van tijd in het werkstuk; vaak voelde het schrijven van code voor mij niet als een schoolopdracht en diende het zien van een extra niveau van realisme in mijn eindresultaat als motivatie om er meer aan te werken. Dit heeft er echter wel voor gezorgd dat er minder tijd overbleef voor het schrijven van het werkstuk zelf, en met daarbovenop een licht schaatsongeluk dat mijn hand drie dagen lang de vaardigheid ontnam om te typen, is er tegen het einde van de deadline een zekere tijdsdruk ontstaan.

Ook had er verbetering kunnen zitten in mijn onderzoeks methode. Ik maak erg veel gebruik van wetenschappelijk onderzoek in mijn profielwerkstuk, maar de wijze waarop ik deze vond, was niet vaak door middel van een wetenschappelijk artikel, maar via eigen herleiding, voorkennis of een verbetering in mijn eigen herleiding door ChatGPT. Bij een tweede onderzoek zou ik bij het implementeren van een nieuw codestuk ook eerst dit algoritme in Google Scholar of iets vergelijkbaars invoeren om direct een bron van kwaliteit te vinden, in plaats van gebruik te maken van dit onderzoek door een derde partij.

Weer als consequentie van mijn intrinsieke interesse in dit onderwerp zou ik wellicht wat betreft tijd op een meer verspreide wijze werken. Op sommige vakantiedagen heb ik mijn hele dag besteed aan programmeren, en hoewel ik dat enigszins leuk vind, kan dat mentaal erg vermoeiend zijn en voelen als een dag die nooit heeft plaatsgevonden. Op andere momenten heb ik wel eens wekenlang niets gewerkt aan mijn profielwerkstuk, en het zou wellicht beter zijn om in deze tijd ook wat gewerkt te hebben.

# Hoofdstuk 9

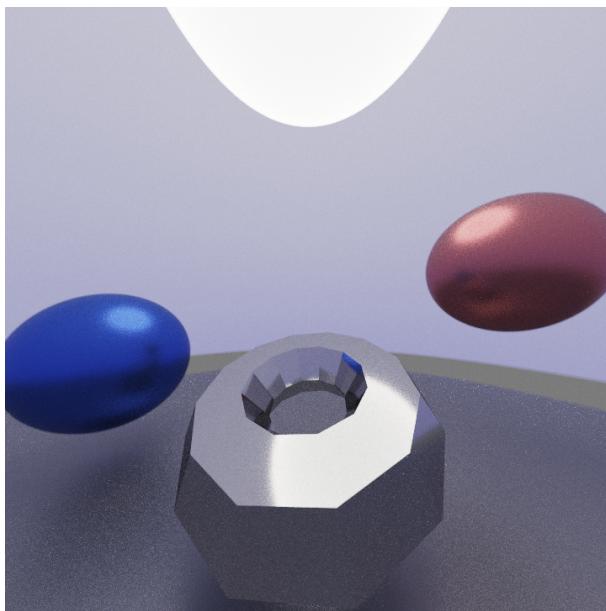
## Bijlage

### 9.1 Logboek

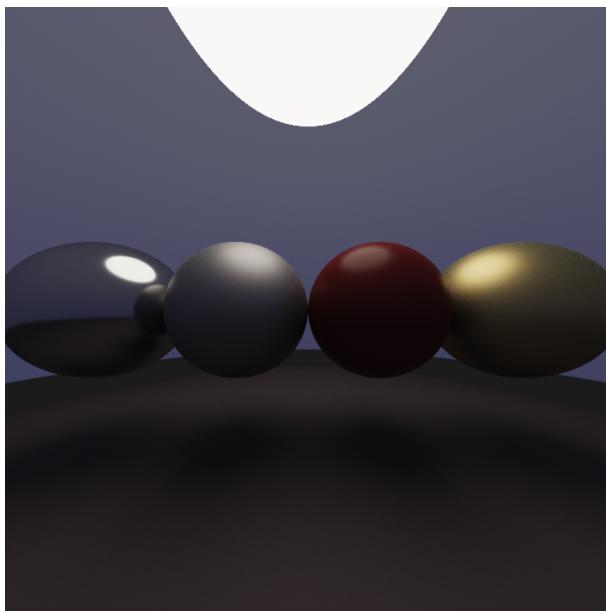
Datum	Taak	Tijd
20 April 2025	eerste concept hoofd- en deelvragen	10 minuten
21 April 2025	Github-repo aangemaakt, boilerplate code gekopieerd van testproject en een basis LaTeX bestand gemaakt	31 minuten
21 April 2025	schrijven simpel plan	42 minuten
21 April 2025	leren LaTeX	13 minuten
30 Mei 2025	deadline 2 schrijven doelstelling etc	120 minuten
28 Juni 2025	lezen introductie pbrt	19 minuten
28 Juni 2025	onderzoeken welk systeem het beste is voor mijn doel	21 minuten
30 Juni 2025	schrijven begin PWS in overleaf LaTeX	56 minuten
3 Juli 2025	PWS ochtend en thuis werken aan Theor. kader	236 minuten
3 Juli 2025	afmaken theoretisch kader	100 minuten
28 Juli 2025	lezen pbrt monte carlo integratie	52 minuten
28 Juli 2025	lezen radiometrie	64 minuten
31 Juli 2025	begin schrijven rust	33 minuten
7 Augustus 2025	oefenen/proberen p5js	66 minuten
7 Augustus 2025	schrijven p5js basis raytracer: een bewegende eenkleurige cirkel zonder belichting voor een camera	83 minuten
8 Augustus 2025	bewegende bol met 1 lichtpunt en lamberian diffuse	80 minuten
10 Augustus 2025	toevoegen path tracing	481 minuten
10 Augustus 2025	denoising more	44 minuten
10 Augustus 2025	fix noise	61 minuten
11 Augustus 2025	proberen DFS, opsplitsen hitInfo en trace functie	85 minuten
11 Augustus 2025	blinn-phong speculaire reflecties	78 minuten
11 Augustus 2025	accumulator toegevoegd en support voor supersampling met vastgezette tijd per frame	276 minuten
11 Augustus 2025	objecten en objetlocatie omzetten naar ssbo: config file gemaakt	70 minuten
12 Augustus 2025	progress vertices	294 minuten
12 Augustus 2025	vertices bijna af	202 minuten
13 Augustus 2025	vertices werken eindelijk	152 minuten
30 Augustus 2025	camera quaternion deel 1	145 minuten
1 Oktober 2025	documentatie	121 minuten
2 Oktober 2025	documentatie	105 minuten

14 Oktober 2025	verbeteren lichtsysteem	47 minuten
14 Oktober 2025	verbeterd lichtsysteem, post-processing support	158 minuten
15 Oktober 2025	werk aan betere brdf	25 minuten
15 Oktober 2025	werk brdf	7 minuten
15 Oktober 2025	uitzoeken brdf importance sampling	85 minuten
15 Oktober 2025	werk brdf	59 minuten
16 Oktober 2025	debuggen brdf	303 minuten
18 Oktober 2025	weeerk brdf	72 minuten
20 Oktober 2025	eindelijk probleem gevonden, gebruik nu ggx brdf	37 minuten
20 Oktober 2025	vernieuwde camera logica en werk live bewegingssysteem	277 minuten
21 Oktober 2025	afmaken movement	74 minuten
26 Oktober 2025	schrijven brdf docu	45 minuten
26 Oktober 2025	documenteren en manim	245 minuten
26 Oktober 2025	schrijven methode	186 minuten
1 November 2025	rework materiaal systeem, werken textures	141 minuten
1 November 2025	meer werk textures	250 minuten
2 November 2025	textures	206 minuten
20 November 2025	schrijven inleiding en schrijven andere dingen en bespreking	390 minuten
18 December 2025	fixen texturess	183 minuten
19 December 2025	bugfixes	191 minuten
20 December 2025	code herschreven om bug weg te halen	85 minuten
20 December 2025	texture channel fix werk	25 minuten
21 December 2025	fixen texture sampling bug	95 minuten
21 December 2025	bounding box werk	87 minuten
21 December 2025	toegevoegd bounding box	168 minuten
22 December 2025	aanpassing texture systeem en inlezen texture mipmap LOD op basis van differentiaal werk textures	39 minuten
22 December 2025	werk textures	107 minuten
23 December 2025	skybox	209 minuten
23 December 2025	skybox, torus render tests	60 minuten
26 December 2025	backend bvh	240 minuten
26 December 2025	werk bvh	55 minuten
30 December 2025	schrijven	81 minuten
31 December 2025	schrijven	77 minuten
1 Januari 2026	schrijven	97 minuten
3 Januari 2026	schrijven	621 minuten
4 Januari 2026	schrijven	541 minuten
Totaal	-	145 uur

## 9.2 Renders



Figuur 9.1: Een 1000-rpp 164-face render.



Figuur 9.2: Render van een bol van spiegel, metaal, plastic en goud.

# Bibliografie

- [1] B. T. Phong, “Illumination for Computer Generated Pictures,” *Communications of the ACM*, jrg. 18, nr. 6, p. 311–317, 1975. DOI: [10.1145/360825.360839](https://doi.org/10.1145/360825.360839).
- [2] B. Walter, S. R. Marschner, H. Li en K. E. Torrance, “Microfacet Models for Refraction through Rough Surfaces.” *Rendering techniques*, jrg. 2007, 18th, 2007. DOI: [10.5555/2383847.2383874](https://doi.org/10.5555/2383847.2383874).
- [3] A. Nguyen, “Implicit bounding volumes and bounding volume hierarchies,” *Doctor of Philosophy, Stanford University*, 2006.
- [4] H. Igehy, “Tracing Ray Differentials,” in *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques*, reeks SIGGRAPH ’99, New York, NY, USA: ACM, 1999, p. 179–186. DOI: [10.1145/311535.311555](https://doi.org/10.1145/311535.311555).
- [5] J. T. Kajiya, “The rendering equation,” in *Proceedings of the 13th annual conference on Computer graphics and interactive techniques*, 1986, p. 143–150. DOI: [10.1145/15922.15902](https://doi.org/10.1145/15922.15902).
- [6] M. Pharr, W. Jakob en G. Humphreys, *Physically Based Rendering: From Theory to Implementation*, 4th. The MIT Press, 2023, ISBN: 9780262048026.