KU LEUVEN

FACULTEIT INGENIEURSWETENSCHAPPEN

realtime renderen van vele lichtbronnen

Martinus Wilhelmus Tegelaers

Thesis voorgedragen tot het behalen van de graad van Master of Science in de ingenieurswetenschappen: computerwetenschappen, hoofdspecialisatie Mens-machine communicatie

Promotor:
Prof. dr˙ ir. P. Dutre

Assessor:
T. Do

Begeleider:
T.O. Do

Academiejaar 2016 – 2017

Voorwoord

Dit is mijn dankwoord om iedereen te danken die mij bezig gehouden heeft. Hierbij dank ik mijn promotor, mijn begeleider en de voltallige jury. Ook mijn familie heeft mij erg gesteund natuurlijk.

Martinus Wilhelmus Tegelaers

Inhoudsopgave

Samenvatting

In dit environment wordt een al dan niet uitgebreide samenvatting van het werk gegeven. De bedoeling is wel dat dit tot 1-bladzijde beperkt blijft.

Lijst van figuren en tabellen

Lijst van figuren

Lijst van tabellen

Hoofdstuk 1

Inleiding

Dit is de algemene inleiding placeholder

1.1 Probleemstelling

probleem stelling placeholder

1.2 Overzicht Thesis

Wat volgt is een kort overzicht van de hoofdstukken binnen deze thesis:

Chapter 2: Theorie Het theorie hoofdstuk zal een korte inleiding geven tot het renderen van 3d computer graphics en de basis technieken gebruikt binnen deze thesis.

Chapter 3: Literatuur Studie De literatuur studie bouwt verder op de brede inleiding van chapter 2, en behandelt de specifieke papers waarop deze thesis verder bouwt. Hierbij worden zowel de rendering technieken besproken die in de thesis geimplementeerd zijn, als de datastructuren gebruikt in het voorgestelde algoritme.

Chapter 4: Methode: Eigen Algoritme Hier wordt het eigen algoritme besproken

Chapter 5: Methode: Implementatie Hier wordt de implementatie besproken

Chapter 6: Resultaten Hier worden de resultaten besproken

Chapter 7: Discussie en conclusie Hier worden de resultaten bediscussieerd

Hoofdstuk 2

Theorie

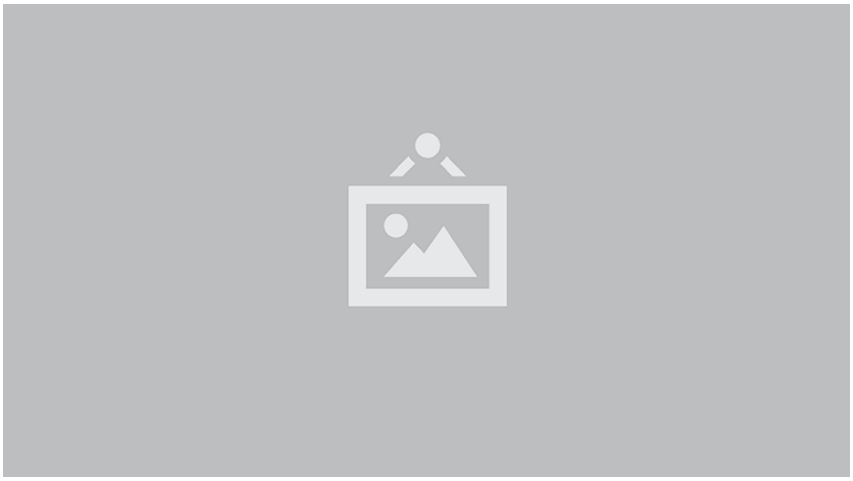
Zoals besproken in de inleiding draait de thesis om het renderen van drie dimensionale scenes in real-time. Het doel is hierbij veelal om geloofwaardige afbeeldingen te creeeren uit een bepaalde drie dimensionale scene beschrijving. In veel gevallen betekent dit dat de scene fotorealistisch dient afgebeeld te worden, echter andere stylistische keuzes zijn tevens gebruikelijk. In alle gevallen echter is het concept van geloofwaardigheid in grote mate afhankelijk van de manier hoe mensen de wereld om zich heen waarnemen. Voordat dan ook verder ingegaan wordt op de algoritmes om dergelijke afbeelding te produceren, zal eerst ingegaan worden op deze perceptie. Nadat vastgesteld is wat bereikt dient te worden met renderen, zal ingegaan worden op hoe dit mathematisch voor te stellen, en welke algoritmes gebruikt worden om deze problemen op te lossen. Als laatste zal besproken worden hoe dit binnen huidige generatie videokaarten op hardware niveau geimplementeerd is.

2.1 Fysische werkelijkheid

De fysische wereld waarin de mens zich bevindt wordt gedicteerd door alle fysische wetten. De mens neemt deze wereld door middel van zintuigen. Voor computer graphics, waarneming is het belangrijkste zintuig. Door middel van waarneming wordt de drie dimensionale wereld om de mens heen geinterpeteerd. Deze interpretatie zal de fysische werkelijkheid genoemd worden binnen deze thesis. Zowel de fysische wereld als de manier waarop deze waargenomen wordt bepaald dus de fysische werkelijkheid.

2.1.1 Waarneming

De mens neemt de wereld waar door het oog. Het menselijk oog interpreteert de drie dimensionale wereld door stralen van licht te focussen op een enkel punt, doormiddel van een lens. Het enkele punt dat licht omzet naar neurosignalen wordt de retina genoemd. Een camera bootst het oog na, en projecteert licht op een electronische photosensor, die het signaal op zet naar een digitaal signaal. Dit is weergegeven in figuur 2.1.



Figuur 2.1: Waarneming doormiddel van het oog en camera.

Deze manier van projectie heeft twee belangrijke gevolgen:

- Objecten worden als kleiner waargenomen naarmate ze verder van de waarnemer af staan.

- Objecten worden waargenomen met Foreshortening, i.e. de dimensies van een object parallel aan het gezichtsveld, worden als kleiner waargenomen dan dimensies van hetzelfde object loodrecht aan het gezichtsveld.

De mens verwacht dat deze eigenschappen aanwezig zijn, om beelden te interpeteren.

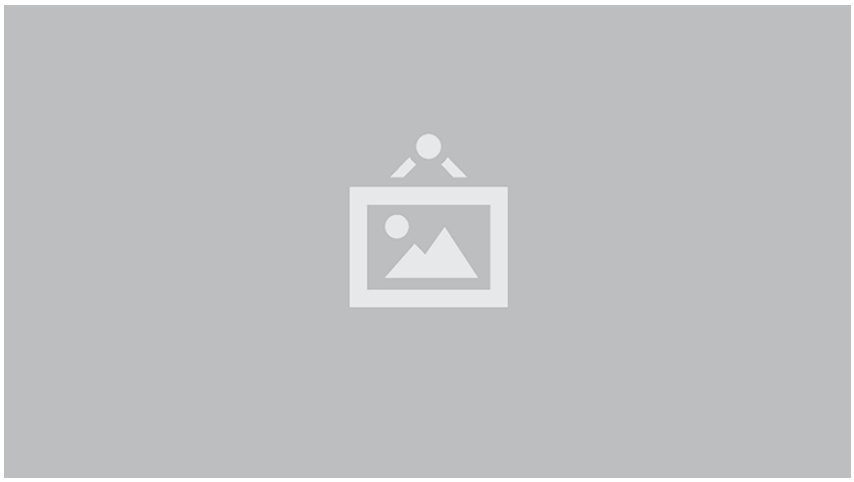
2.1.2 Licht

Het tweede belangrijke inzicht bij waarneming is dat de wereld wordt waargenomen door middel van licht. Dit betekent dat bij afwezigheid van licht, het niet mogelijk is om iets waar te nemen. Verder betekent dat ook dat het gedrag van licht een grote invloed heeft op de manier hoe de wereld wordt waargenomen.

Licht is electromagnetische straling. Voor Computer Graphics is met name de optica van belang. Hierin wordt licht, en de interactie tussen licht en materie bestudeerd. Deze wetten vormen veelal de basis om licht te simuleren. Licht zal onder normale omstandingheden, zich altijd in een rechte lijn zolang het binnen hetzelfde medium blijft. Indien het licht in contact komt met een nieuw medium zijn er verschillende fenomen die kunnen gebeuren:

Absorptie Het licht wordt geabsorbeerd door de atomen van het nieuwe medium, en uitgestoten als warmte. Hierbij gaat het licht verloren.

Reflectie Het licht wordt gereflecteerd op het oppervlakte van het nieuwe medium. Hierbij wordt het licht terug de scene ingestuurd. De hoek van reflectie hangt



Figuur 2.2: Absorptie, reflectie en transmissie van licht.

af van het type medium. Indien het materiaal zich gedraagd als een spiegel, en zal het licht teruggekaatst worden met dezelfde hoek gespiegeld om de normaal. Indien het materiaal licht diffuus weerspiegelt betekent dat de hoek van inval niet uitmaakt voor de reflectie en deze min of meer willekeurig is.

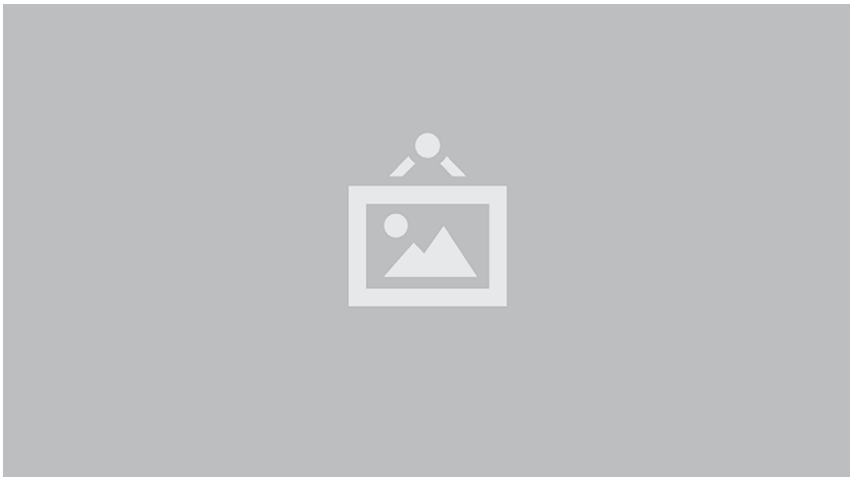
Transmissie Het licht plant zich verder voort door het nieuwe medium, opnieuw in een rechte lijn, met mogelijk een iets andere richting op basis van de brekingsindex van het nieuwe en het oude medium.

Deze fenomenen zijn verder geillustreerd in figuur 2.2. Ze zijn niet exclusief aan elkaar. Een medium kan dus bijvoorbeeld een gedeelte van het licht absorberen en een ander gedeelte reflecteren.

Zoals eerder vermeld, neemt het oog de wereld waar door licht op te vangen. Het merendeel van het licht dat opgevangen wordt is gereflecteerd via een of meerdere oppervlaktes. Een belangrijke constatering is dat objecten slechts zichtbaar zijn als er binnen het medium geen (onderzichtige) andere media liggen tussen het object en de lens. Dit is een triviale constatering in de fysische werkelijkheid echter dit zal niet triviale consequenties hebben binnen de computer graphics zoals later zal worden beschreven.

Om de interactie van licht te simuleren is het van belang dat licht meetbaar is. Er zijn hiervoor twee sets van eenheden, radiometrie en fotometrie. Binnen radiometrie wordt slechts de lichtkracht over alle golflengte gemeten. Bij fotometrie wordt deze kracht gewogen, aan de hand van het gestandardiseerde model voor de perceptie van helderheid. Fotometrie is van belang voor computer graphics, omdat het inzicht geeft in de perceptie van de mens. Echter binnen deze thesis zal slechts kort ingegaan worden op radiometrie.

De belangrijkste termen van Radiometrie zijn opgesteld in tabel **??**. Binnen computer graphics is de belangrijkste eenheid radiantie. Dit is de flux per eenheid geprojecteerde opperlakte per eenheid ruimtehoek. Radiantie meet de flux op een



Figuur 2.3: Het mengen van kleuren volgens een additief model.

willekeurig punt in de ruimte, komende van een specifieke hoek en gemeten over een oppervlakte eenheid op een denkbeeldige oppervlakte loodrecht op de hoek. Radiantie heeft de volgende eigenschappen die van belang zullen zijn indien deze berekend dient te worden gedurende de simulatie van licht:

- Radiantie is constant binnen een straal die zich voortplant door vacuum. Tevens is het gelijk in beide richtingen die een straal zich voort kan planten
- Indien het punt van meting op een oppervlakte wordt genomen, maakt het niet uit of de flux binnenkomt, of uitgaand is. Het maakt zelfs niet uit of de flux geabsorbeerd, gereflecteerd, of doorgelaten wordt door het materiaal.

2.1.3 Kleur

Een tweede belangrijk aspect van licht voor computer graphics is het concept kleur. Het concept kleur is niet een fysisch verschijnsel, maar een consequentie van hoe ogen licht interpeteren. De mens neemt slechts een gedeelte van al het licht waar. Dit wordt het zichtbare licht genoemd. Het menselijk oog interpeteert het licht Door het zowel een intensiteit als een kleur toe te kennen. De kleur die waargenomen wordt van een lichtstraal is afhankelijk van het licht. Een gemiddeld persoon is in staat om 3 verschillende primaire kleuren waar te nemen, rood, blauw en groen. Elke zichtbare kleur kan voorgesteld worden als een mix van deze primaire kleuren. De manier om deze kleuren te mengen is afgebeeld in figuur 2.3 Belangrijk om hierbij te vermelden is dat licht zich gedraagd als additieve kleurmenging. Dit houdt in dat indien verschillende kleuren licht op het zelfde punt worden afgebeeld, dit punt zal worden waargenomen als de kleur gelijk aan de optelling van deze lichten.

Objecten kunnen tevens een kleur hebben. Reeds is besproken dat objecten worden waargenomen door de reflectie van hun licht. De kleur van een object is dan ook het gevolg van de gedeeltelijk absorptie van licht. In het geval dat een gekleurd object wordt verlicht met puur wit licht, zal slechts het licht dat overeenkomt

in frequentie met de kleur van het object weerspiegelt worden. De frequenties tegenovergesteld aan de kleur van het object, zullen worden geabsorbeerd door het object.

2.1.4 Simulatie

Computer graphics heeft als doel om de fysische werkelijkheid te benaderen. Echter hiervoor is het niet nodig om de volledige fysische werkelijkheid te benaderen. Het simuleren van de fysische werkelijkheid om een afbeelding te verkrijgen, het renderen, kan dus in grofweg in twee problemen ingedeeld worden:

- Wat is zichtbaar binnen een scene vanuit het huidige gezichtspunt.
- Hoe ziet datgene wat zichtbaar is er uit binnen onze afbeelding.

Wat afgebeeld wordt op een afbeelding, hangt af van twee aspecten, hoe wordt de 3d scene op het 2d beeld geprojecteerd. En wat van elk object dat geprojecteerd kan worden is daadwerkelijk zichtbaar op de uiteindelijk afbeelding. Het eerste wordt perspectief projectie genoemd. Het tweede probleem wordt het visibiliteitsprobleem probleem genoemd.

Hoe hetgene wat afgebeeld wordt, er uiteindelijk uit ziet binnen onze afbeelding, wordt in de tweede stap bepaald. Deze stap wordt shading genoemd, en alle berekeningen gerelateerd aan kleur, absorptie, weerspiegeling etc, vallen hier onder.

2.2 Notaties en definities

Voordat de achterliggende theorie verder behandelt wordt, zal eerst kort ingegaan worden op de verschillende notaties en definities binnen deze thesis.

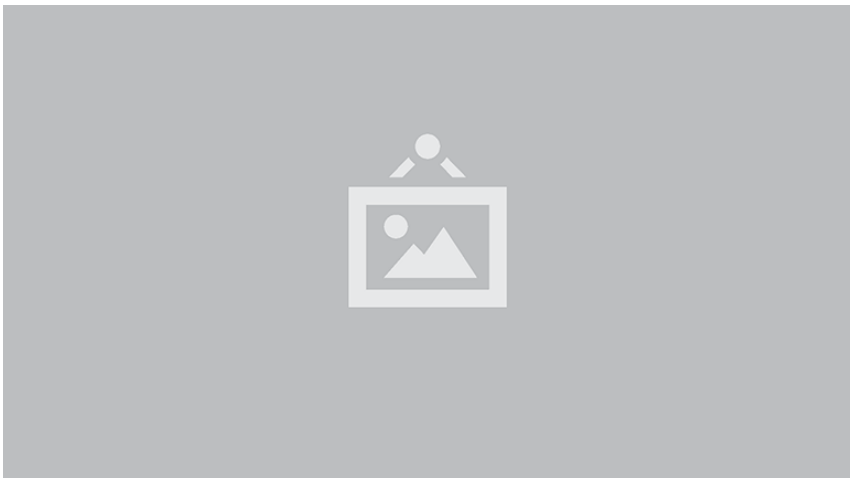
2.2.1 Geometrische definities

Voordat het mogelijk is om drie dimensionale omgevingen om te zetten naar afbeeldingen is het nodig om een beschrijving van deze scenes te hebben. De basis render primitieven die gebruikt worden door de meeste grafische hardware zijn punten, lijnen en driehoeken. Een punt wordt beschreven met homogene coordinaten:

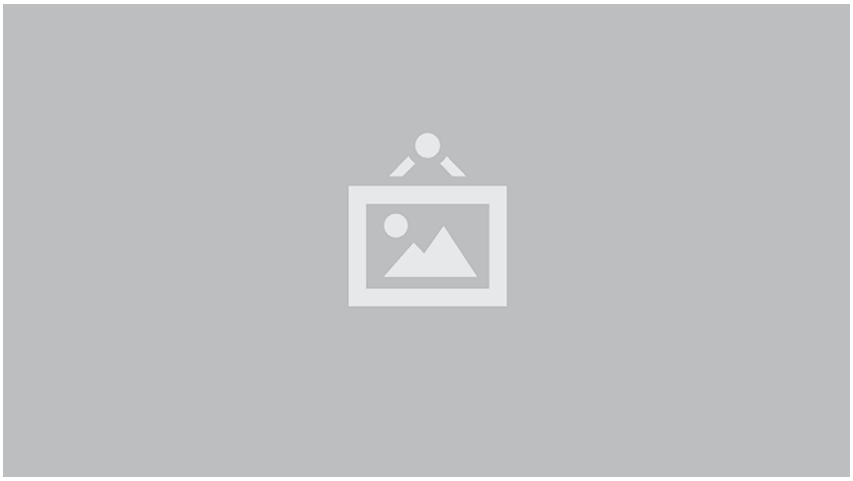
$$\mathbf{v} = \begin{pmatrix} v_x \\ v_y \\ v_z \\ v_w \end{pmatrix}$$

Indien gesproken wordt van primitieven zal, als niet anders is aangegeven, gedoeld worden op driehoeken. De notatie voor een driehoek is:

$$\triangle \mathbf{v_1 v_2 v_3}$$



Figuur 2.4: Voorstelling van objecten doormiddel van driehoeken.



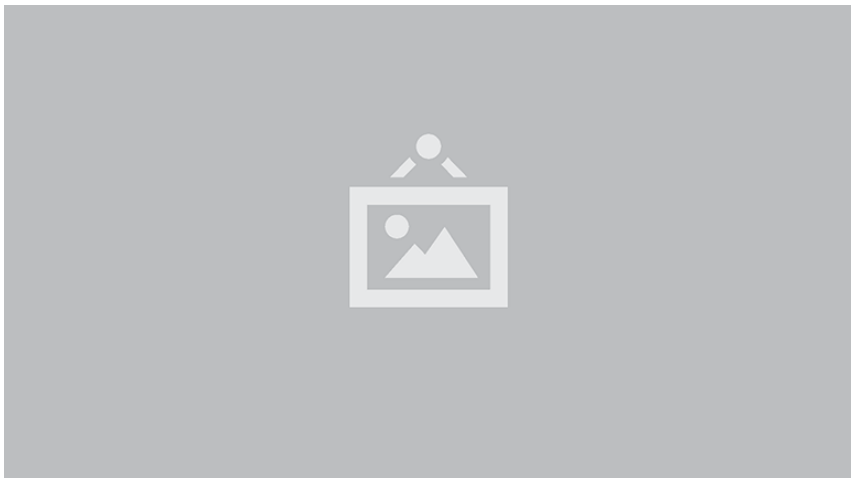
Figuur 2.5: Het standaard camera model.

Een *mesh* is een collectie van driehoeken die samen de vorm van een object vormen. Een *object* bevat zowel een *mesh* als een transformatie matrix die die de locatie, schaal, en rotatie van het *object* vastlegt. De verschillende geometrische definities zijn geillustreerd in figuur 2.4. De volledige beschrijving van een virtuele omgeving zal een *scene* genoemd worden. Deze bevat een set van objecten, de lichten, en eventuele definities van gezichtspunten.

2.2.2 Camera model

Het gezichtspunt, of *camera* binnen deze thesis is het standaard camera model binnen computer graphics. Zoals geillustreerd in figuur
2.5.

De volgende vectoren en punten zijn hiervoor gedefinieerd



Figuur 2.6: Perspectief projectie.

Up De locale y-as van de camera.
Eye (x, y, z) positie van de camera in wereld coordinaten.
Centre (x, y, z) positie waarnaar de camera kijkt
Z-near de near z plane waarop het beeld wordt geprojecteerd.
Z-far Het vlak waarachter fragmenten niet meer worden weergegeven.

De *Z-near* en *Z-far* in combinatie met *Eye* creeert de view frustrum. Slechts Primitieven binnen dit view frustum zullen worden gerenderd.

2.3 Perspectief projectie en het visibileitsprobleem

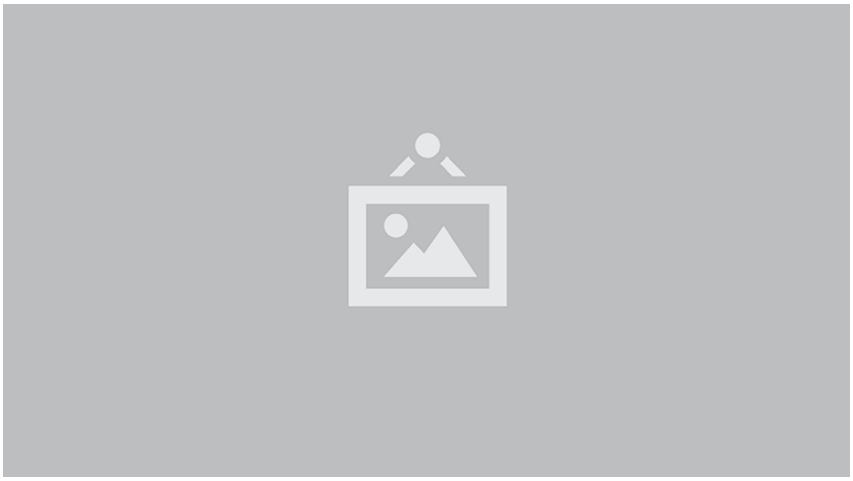
In de sectie ... zijn de twee problemen vastgesteld die opgelost dienen te worden om geloofwaardige afbeeldingen te generen. In deze sectie zal de eerste geadresseerd worden, wat is zichtbaar binnen een scene vanuit het huidige gezichtspunt. Om te bepalen wat zichtbaar is dient zowel het perspectief gesimuleerd te worden, als bepaald te worden welk van de objecten in perspectief als eerste zichtbaar is.

2.3.1 Perspectief projectie

Zoals eerder besproken zijn de twee eigenschappen van perspectief:

- Objecten worden als kleiner waargenomen naarmate ze verder van de waarnemer af staan.

- Objecten worden waargenomen met Foreshortening, i.e. de dimensies van een object parallel aan het gezichtsveld, worden als kleiner waargenomen dan dimensies van hetzelfde object loodrecht aan het gezichtsveld.



Figuur 2.7: Projectie van een enkel punt.

Deze effecten kunnen gesimuleerd worden door de 3d scene te projecteren naar het oogpunt en af te beelden op het canvas. Zoals weergegeven in figuur 2.6.

Zoals eerder beschreven bestaan de objecten binnen de scenes uit meshes van primitieven. Omdat elk van deze driehoeken gedefinieeerd kan worden door zijn drie vertices, is het niet nodig om elke mogelijk punt binnen de driehoek af te beelden op de canvas, maar is het genoeg om slechts deze drie vertices te projecteren.

In figuur 2.7 is de projectie $\mathbf{p}'$ van een enkel punt $\mathbf{p}$ op een enkele dimensie van de canvas weergegeven. De hoek $\angle\mathbf{abc}$ Hier is te zien dat de hoek tussen C en AB'C' gelijk is. Dit betekent dat we het punt C' kunnen bereken doordat de verhouding geldt

$$\frac{BC}{AB} = \frac{B'C'}{AB'}$$

Verder is de afstand van het oogpunt tot de canvas, AB', bekend. Wat ertoe leidt dat we het geprojecteerde punt gemakkelijk kunnen berekenen.

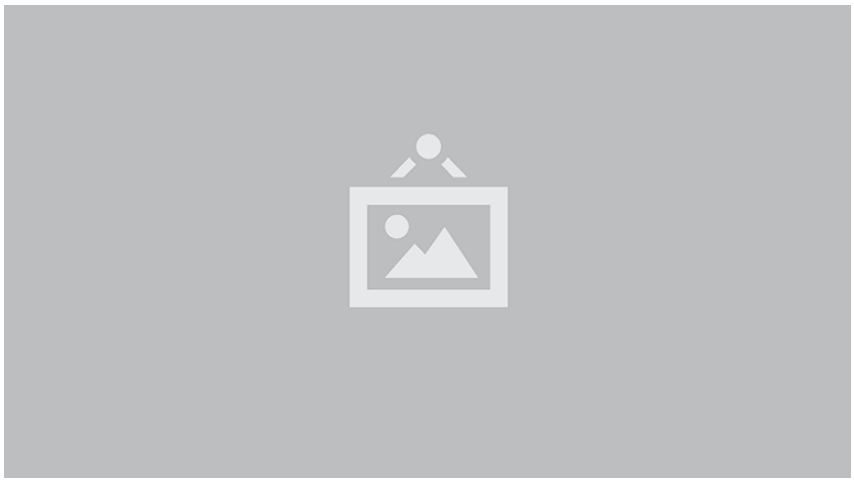
$$p'_x = d\frac{p_x}{p_z}$$

$$p'_y = d\frac{p_y}{p_z}$$

$$p'_z = d$$

$$p'_w = 1$$

waar d de afstand van het oogpunt tot de canvas is. Binnen computer graphics wordt deze stap de perspectief deling genoemd. We kunnen deze berekeningen samenvoegen tot een enkele matrix $\mathbf{P}$ die een punt in een specifiek coordinaten stelsel omzet naar een punt geprojecteerd op de canvas. Wat leidt tot de perspectief projectie van punten.



Figuur 2.8: Visibiliteitsprobleem in een scene met meerdere primitieven.

$$\mathbf{Pp} = \mathbf{p}'$$

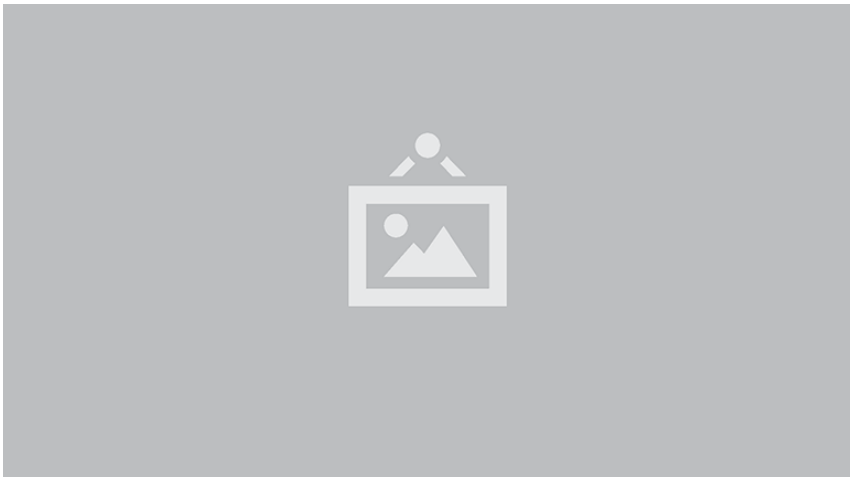
Of deze projectie daadwerkelijk nodig is, is afhankelijk van de gekozen rendering techniek. Binnen rasterisatie is het nodig om deze stap expliciet uit te voeren. Raytracing neemt de perspectief projectie impliciet mee.

2.3.2 Visibiliteitsprobleem

Er is nu vastgesteld hoe objecten in perspectief afgebeeld op de canvas kunnen worden. Echter, hiermee is nog niet volledig vastgesteld wat daadwerkelijk zichtbaar gaat zijn op het canvas, zoals weergegeven in figuur 2.8. Hiervoor is het tevens nodig om te bepalen welke delen van objecten zichtbaar zijn, en welke verborgen zijn achter andere objecten. Dit probleem wordt onder andere het visibiliteitsprobleem genoemd, en was een van de eerstegrote problemen binnen computer graphics.

De oplossing voor dit probleem is de realisatie dat het visibiliteitsprobleem intrinsiek een sorteerprobleem is. Stel er bestaat een minimale oppervlakte O op het canvas, waarvoor gekeken wordt welk deel zichtbaar is. Door middel van perspectief projectie is het mogelijk om te bepalen welke objecten op O worden afgebeeld. Er van uitgaande dat er een object A bestaat die afgebeeld wordt op O. Dan is dit object daadwerkelijk zichtbaar in O als er geen andere objecten op O worden geprojecteerd die dichterbij het oogpunt liggen dan object A. Wanneer alle objecten gesorteerd zijn is het per punt of het canvas mogelijk om het dichtstbijzijnde object te selecteren, en deze weer te geven op het scherm.

De algoritmes om dit efficient te doen worden verborgen oppervlakte bepalings (hidden surface determination) algoritmes genoemd. Deze kunnen grofweg ingedeeld worden in twee categorien, raytracing en rasterisatie. Hierbij zou, in theorie, geen verschil in resultaat hoeven zijn. Beide klasses van algoritmes hetzelfde doel hebben, het produceren van realistische beelden op basis van een 3d scene.



Figuur 2.9: Forwaards raytracen.

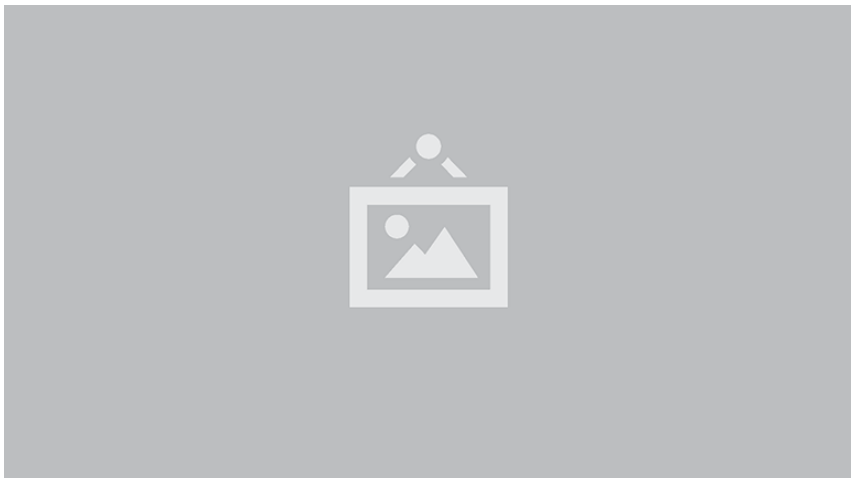
Raytracing werkt op basis van het trekken van zogenoemde stralen door $\mathbf{p}'$ waarbij de eerste $\mathbf{p}$ wordt bepaald. Rasterisation daarentegen, bepaald alle mogelijke $\mathbf{p}'$ op basis van alle mogelijke $\mathbf{p}$ die geprojecteerd worden op $\mathbf{p}$. Vervolgens wordt bepaald welke $\mathbf{p}'$ daadwerkelijk zichtbaar is.

In de volgende secties zal er kort ingegaan worden op beide technieken.

2.3.3 Raytracing

Raytracing simuleert de werking van licht en het menselijk ook, en lost hiermee zowel het visibiliteitsprobleem als het perspectief op. Er is reeds vastgesteld dat menselijke waarneming berust op het waarnemen van licht dat valt op de lens en geprojecteerd wordt op de retina, de lichtsensor. In theorie is het mogelijk om beelden op een zelfde manier op te bouwen, zoals dit gebeurd in het oog. In dit geval zouden vanuit lichten willekeurige stralen geschoten kunnen worden. Hierbij is een straal gedefinieerd als zijnde een vector met een beginpunt en een richting. Wanneer een dergelijke straal door de canvas op het oogpunt valt, wordt deze meegeteld. Dit is geillustreerd in figuur 2.9. Deze techniek wordt forwaards tracen genoemedt. Echter vaak is het canvas van een camera velen malen kleiner dan de scene in kwestie. Dit zorgt ervoor dat de kans dat de canvas geraakt wordt, uitermate klein is. Hierdoor is een groot aantal lichtstralen nodig, voordat een geloofwaardige afbeelding wordt verkregen.

Met de realisatie dat we uiteindelijk slechts de stralen nodig hebben, die door het canvas op het oogpunt vallen kunnen we de techniek omdraaien. In plaats van willekeurige stralen te schieten vanuit lichten, schieten we, per punt dat we willen weten op de canvas, een straal. Deze straal zal dus altijd het oogpunt raken, en door punt $\mathbf{p}'$ gaan. Vervolgens dient gekeken te worden welk object, als er een bestaat, deze straal raakt. Hiermee wordt punt $\mathbf{p}$ gevonden. Vanaf $\mathbf{p}$ kunnen we bepalen welke lichten dit punt raken, en dus hoe het punt $\mathbf{p}'$ gekleurd dient te worden. Dit zal verder besproken worden in de sectie over shading.



Figuur 2.10: Raytrace algoritme.

Deze techniek, waarbij gestart wordt vanuit de camera wordt, achterwaardse tracing genoemd. Belangrijk om hierbij op te merken is dat ray tracing, dus voor elk punt $\mathbf{p}'$ een punt $\mathbf{p}$ vindt. Wanneer gesteld wordt dat punt $\mathbf{p}'$ een (sub)pixel is, zou een algoritme dus bestaan uit twee loops. In de eerste plaats wordt per pixel een straal gegenereerd. Vervolgens wordt per straal gekeken over alle objecten welke object zowel door de straal geraakt wordt en het dichtstbij ligt. Doordat de buitenste loop over de pixels loopt, worden raytracing algoritmes dan ook wel beeldcentrische algoritmes genoemd. De pseudo code zal er als volgt uitzien:

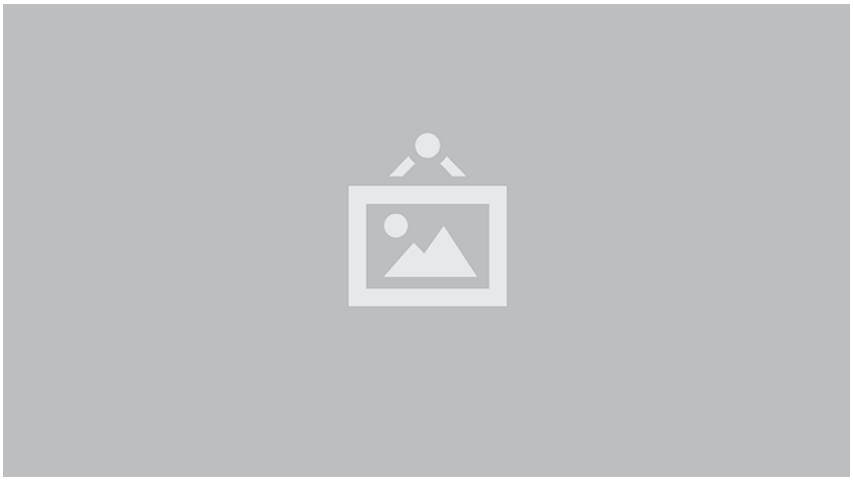
```
for pixel in canvas:
    ray = construct_ray(eye, pixel)

    for object in scene:
        closest = None
        if (ray.hits(object) and
           (closest == None or distance(object, eye) < distance(closest, eye))):
            closest = object

    do_shading(closest, ray)
```

Waarbij `do_shading` gebruikt wordt om de kleur te bepalen van de specifieke pixel. Dit is verder geillustreerd in figuur 2.10

Dit concept is de basis voor alle raytracing algoritmes. Merk hierbij verder op, dat er geen expliciete perspectief projectie plaats vindt. Doordat stralen opgebouwd zijn beginnend in het oogpunt door punt $\mathbf{p}'$, wordt het perspectief impliciet gedefineerd.



Figuur 2.11: Het rasterisatie algoritme.

2.3.4 Rasterisatie

Rasterisatie algoritmes lossen perspectief projectie en het visibiliteitsprobleem op een verschillende volgorde op dan hoe het binnen raytracing algoritmes wordt opgelost. Waar raytracing uitgaat van het punt $\mathbf{p}'$ en kijkt welk object hier op valt, begint een rasterisatie algoritme met het afbeelden van alle objecten op de canvas, om vervolgens te bepalen op welke pixels deze objecten invloed hebben. In dit geval wordt uitgegaan van de punten $\mathbf{p}$ en worden de punten $\mathbf{p}'$ gevonden. Waar raytracing algoritmes dus beeld centrisch zijn, zijn rasterisatie algoritmes object centrisch. Hierbij wordt in de buitenste loop over alle objecten gelopen. En daarna per object gekeken welke pixels door dit object worden beinvloedt. Dit leidt tot de volgende pseudo code:

```
for object in canvas:
    projection = project(object, eye)

    for pixel in projection:
        do_shading(pixel, object)
```

Dit is tevens afgebeeld in figuur 2.11.

Om een enkele primitief dus af te beelden op het canvas dient eerst, voor elke hoek van dit primitief de perspectief deling uitgevoerd te worden. Hierna dient het resultaat omgezet te worden naar raster-ruimte, zodat de punten binnen pixels vallen. Vervolgens dienen de pixels overlopen te worden, om na te gaan of deze binnen of buiten het object valt of niet. Dit leidt uiteindelijk tot een set van $\mathbf{p}'$, i.e. een set van pixels, die behoren tot het object. Om deze pixels efficient te overlopen, wordt meestal een bounding box in raster-ruimte gecreeerd. Slechts voor de pixels binnen deze bounding box wordt nagegaan of het object behoort tot hen of niet. Dit is weer gegeven in figuur ...

Hiermee is vastgesteld dat de oplossing voor de perspectief projectie bestaat uit twee simpele stappen, die goedkoop uit te rekenen zijn. Echter, dit lost nog niet het visibiliteitsprobleem op, doordat het mogelijk is dat verschillende objecten op het punt $\mathbf{p}'$ worden afgebeeld. Om het visibiliteitsprobleem op te lossen zijn verschillende algoritmes voorgesteld. In het volgende stuk zal het z-buffer algoritme besproken worden. Dit is het algoritme waar grafische kaarten gebruik van maken, en is daarom van belang.

Zoals opgemerkt bij de bespreking van het visibiliteitsprobleem, is dit intrinsiek een sorteer probleem, waarbij objecten geordend dienen te worden ten opzichte van de kijkas, de **z**-as. Om het zichtbare object binnen een punt $\mathbf{p}'$ te bepalen, dient dus bepaald te worden welk object de kleinste **z**-as waarde heeft ten opzichte van het oogpunt. De oplossing voor dit probleem is dan ook simpel. Voor elke pixel wordt de kleinste gevonden **z**-as waarde bijgehouden in een corresponderende twee dimensionale array. Deze array wordt een z-buffer, of een diepte-buffer (depth-buffer) genoemd. Wanneer een pixel gevonden wordt met een kleinere **z**-waarde, wordt zowel het object in punt $\mathbf{p}'$ als de nieuwe diepte bijgewerkt. Wanneer alle objecten overlopen zijn zal er dus per pixel bekend zijn welke objecten gebruikt dienen te worden om de shading berekening uit te voeren.

2.4 Shading

In de voorgaande secties is besproken hoe visibiliteit opgelost kan worden. Echter dit is slechts de eerste stap in het genereren van beelden. Nu vastgesteld is welke vorm objecten in de scene hebben, en welke delen van objecten daadwerkelijk zichtbaar zijn, is het tevens nodig om te bepalen hoe deze objecten er uit zien. Shading is het proces waarbij vergelijkingen worden gebruikt om te bepalen welke kleur punten dienen te hebben. Hierbij wordt verder gebouwd op de kennis van sectie .... In deze sectie zal een mathematische beschrijving worden gegeven van shading.

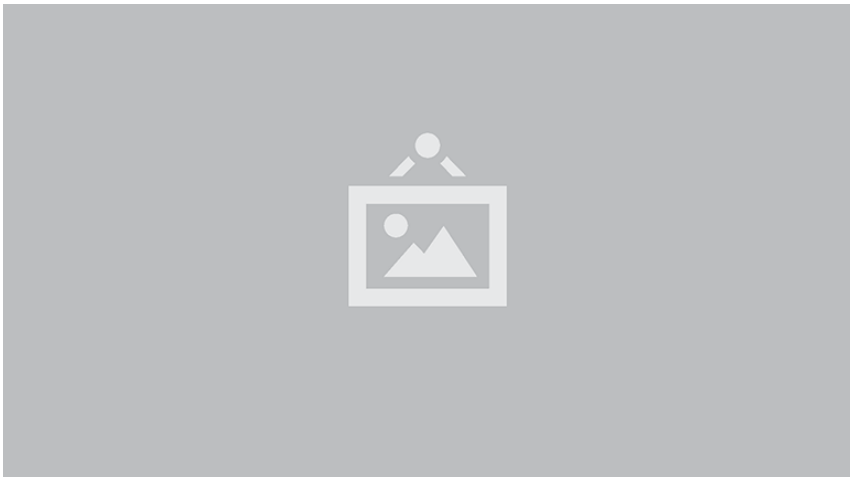
2.4.1 Mathematische modelering

De belangrijkste vergelijking binnen computer graphics is de render vergelijking (rendering equation):

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

Hier weergegeven in hemisfeer vorm. Deze vergelijking toont stabiele toestand van de stralingsenergie balans binnen een scene. Hierbij is $L_o((p), \omega_o$ de radiantie uitgezonden vanuit punt $\mathbf{p}$ over ω_o. Deze radiantie kan gedefinieerd worden aan de hand van de som van gereflecteerde radiantie, en de radiantie die door het punt $\mathbf{p}$ zelf wordt uitgestraald. De uitgestraalde radiantie is $L_e(\mathbf{p}, \omega_o)$. De reflectie van radiantie wordt beschreven door het tweede deel van de render vergelijking:

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



Figuur 2.12: Uitstraling van radiantie over ω_o vanuit $\mathbf{p}$

Dit wordt de reflectie vergelijking genoemd. Hierbij wordt geintegreerd over de gehele hemisfeer om de volledige binnenkomende radiantie te berekenen. Vervolgens specificeert een zogenoemde bidirectionele reflectie distributie functie (bidirectional reflectance distribution function BRDF), hoe de radiantie over een bepaalde ruimtehoek ω_i bijdraagt aan de uitgaande radiantie in punt $\mathbf{p}$ over ruimtehoek ω_o. Hiermee kan precies worden vastgelegd wat de uitgaande radiantie is in punt $\mathbf{p}$ over ruimtehoek ωo. Dit is verder geillustreerd in fig. **??**.

De BRDF in essentie is de wiskundige functie die beschrijft hoe een materiaal zich gedraagd ten opzichte van licht. Deze functies hebben een aantal eigenschappen die gebruikt kunnen worden bij de berekening van de kleur van een punt.

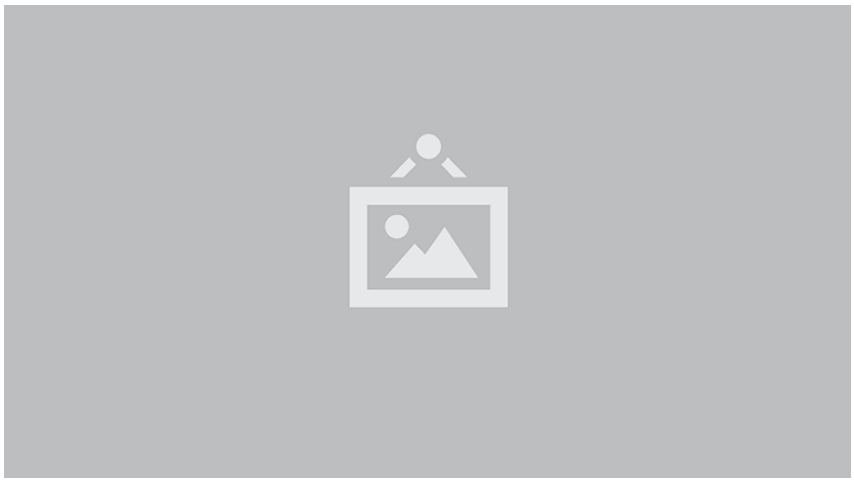
Helmholtz reciprociteit De waarde van een BRDF blijft gelijk indien ω_i en ω_o worden omgedraaid.

$$f_r(\mathbf{p}, \omega_i, \omega_o) = f_r(\mathbf{p}, \omega_o, \omega_i)$$

Lineariteit De totale gereflecteerde radiantie is gelijk aan de som van alle BRDFs op dit specifieke punt. Hierdoor wordt het mogelijk om een materiaal voor te stellen met meerdere BRDFs in hetzelfde punt.

Conservatie van energie De totaal ingevallen radiantie is gelijk aan de som van het uitgezonden licht, en het geabsorbeerde licht. Dit houdt in dat L_o niet groter dan 1 kan zijn over alle ω_o.

Het is nu mogelijk om een beschrijving van de kleur in elk punt te geven aan de hand van de radiantie die berekend kan worden met de rendering vergelijking. Hierbij zullen de materialen van objecten beschreven zijn met BRDFs. Echter er is hier nog wel een groot probleem. De radiantie die uitgezonden wordt vanuit $\mathbf{p}$ over ω_o is afhankelijk van alle radiantie binnenkomend over de gehele hemisfeer in



Figuur 2.13: Lambertiaanse BRDF.

punt **p**. De binnenkomende radiantie is gelijk aan de radiantie uitgezonden vanuit alle punten op de hemisfeer, volgens de Helmholtz reciprociteit. Als gevolg heeft dit dat om de radiantie te berekenen, het nodig is om alle radiantie in de scene al van te voren te weten. Dit is niet mogelijk, en dus zullen alle shading algoritmes pogen een benadering te geven van de daadwerkelijke oplossing van de rendering vergelijking. De kwaliteit van de benadering hangt af van meerdere aspecten, een grote beperkende factor binnen real-time graphics is de beschikbare rekentijd.

2.4.2 Lambertiaanse Bidirectionele Reflectie Distributie Functie

Materialen kunnen gedefinieerd worden als set van BRDFs, die het gedrag van het licht beschrijven indien het in contact komt met een object. De simpelste BRDF is de lambertiaanse BRDF. Deze BRDF beschrijft een puur diffuus oppervlakte, wat inhoudt dat de richting waarin een binnenkomende straal licht wordt gereflecteerd puur willekeurig is. Dit is weergegeven in fig. **??**. Deze BRDF heeft als uitkomst een constante waarde. Deze constante waarde wordt veelal gedefinieerd als de de *diffuse kleur* $c_{\texttt{dif}}$ van dit object. Dit leidt tot de volgende functie:

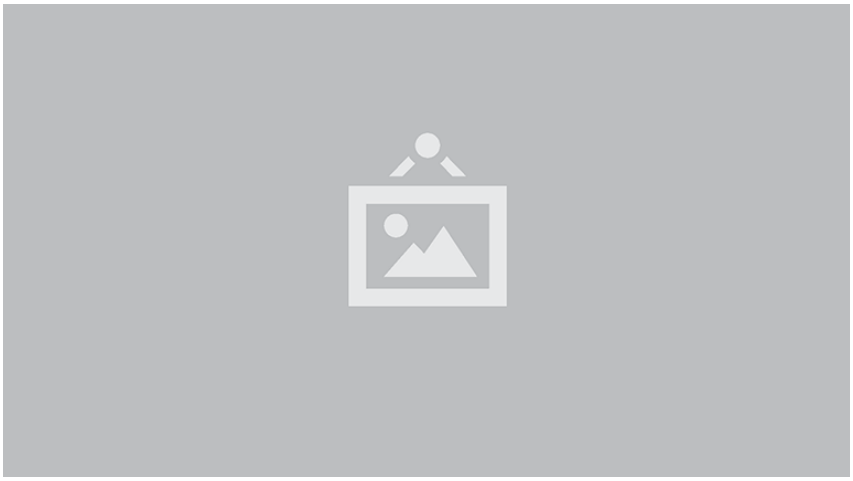
$$f(\omega_i, \omega_o) = \frac{c_{\texttt{dif}}}{\pi}$$

Hierbij is de deling door π een gevolg van de integratie van de cosinus factor over de hemisfeer.

De lambertiaanse BRDF is als standaard materiaal gebruikt binnen deze thesis. Indien niet anders vermeld zullen afbeeldingen en testen gegeneerd zijn met deze functie.

2.4.3 Definitie van licht

Zoals eerder benoemd, draait de kern van deze thesis om het optimaliseren van het aantal lichtberekeningen in real-time toepassingen. Om deze reden is het belangrijk



Figuur 2.14: Afstandsdempings curves.

om het concept licht zoals gebruikt in deze thesis te definieren. Wanneer er gesproken wordt van een licht, of een lichtbron zal altijd gedoeld worden op een eindige puntlichtbron die zich bevindt op punt **p** binnen de scene.

In de fysische wereld zijn lichten nooit eindig, echter de invloed die ze hebben op de omringende wereld zal bij grotere afstand 0 benaderen. Binnen de fysische wereld is dit een gevolg van absorptie door het medium waardoor het licht zich beweegt. Dit proces wordt afstands demping genoemd. Binnen de physica wordt deze relatie vastgelegd met de wet van Lambert-Beer gedefinieerd voor uniforme demping als.

$$T = e^{-\mu l}$$

waar l de padlengte van de straal licht door het medium is en μ de dempingscoefficient is.

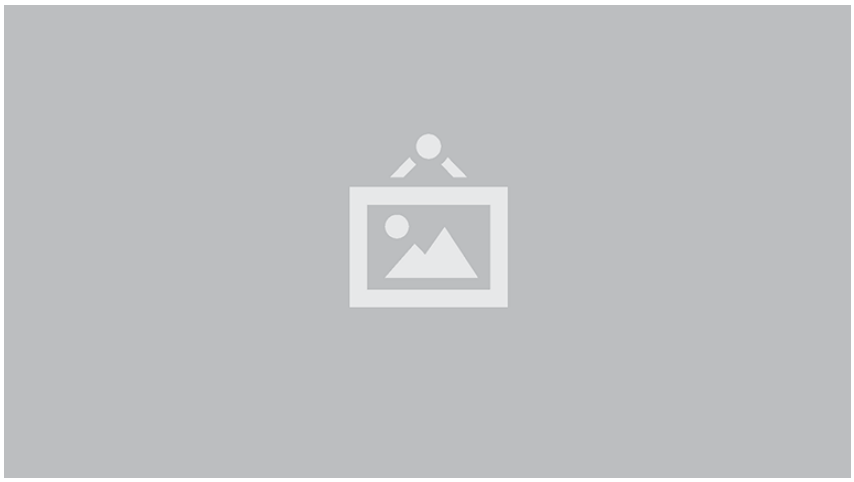
Hierbij wordt de demping van het licht gerelateerd aan het medium waardor het zich beweegt. Dit leidt tot afstandsdempings (distance attenuation) curves zoals weergegeven in figuur 2.14.

Binnen veel real-time rendering toepassingen wordt afgestapt van dit fysische model. Er wordt gebruikt gemaakt van een eindige benaderingen van deze lichtbronnen. Waarbij wordt gesteld dat het licht geen invloed meer heeft na afstand r.

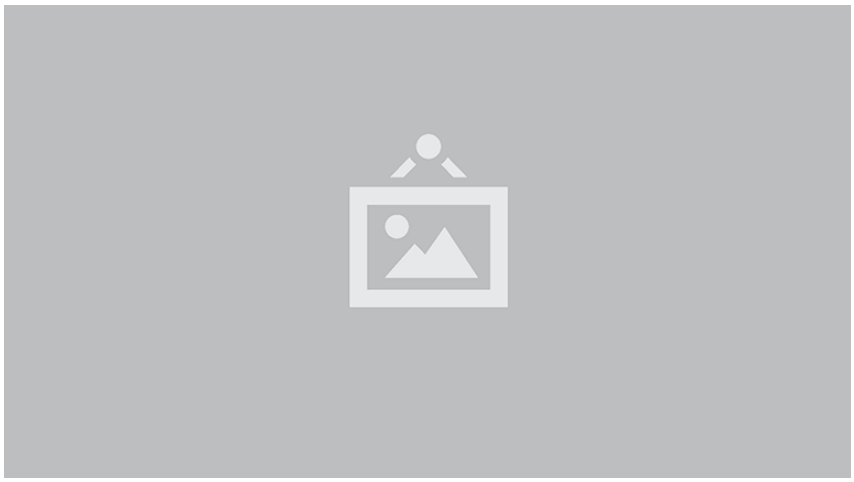
Dit leidt tot een voorstelling als weergegeven in figuur 2.15.
Hierbij is de invloed in de oorsprong gelijk aan 1, en op afstand r en groter 0.

Echter om de illusie te wekken dat de lichten fysiek accuraat zijn, dient tevens een benadering gemaakt te worden van de afstandsdempings functies. Deze functies dienen te voldoen aan de eerder gestelde voorwaarde, waarbij de invloed een is in de oorsprong, en nul op afstand r. Enkele veel gebruikte benaderingen als wel de daadwerkelijke afstandsdemping zijn gegeven in figuur 2.16

Binnen de thesis zelf is gekozen voor de benadering:



Figuur 2.15: Voorstelling van licht.



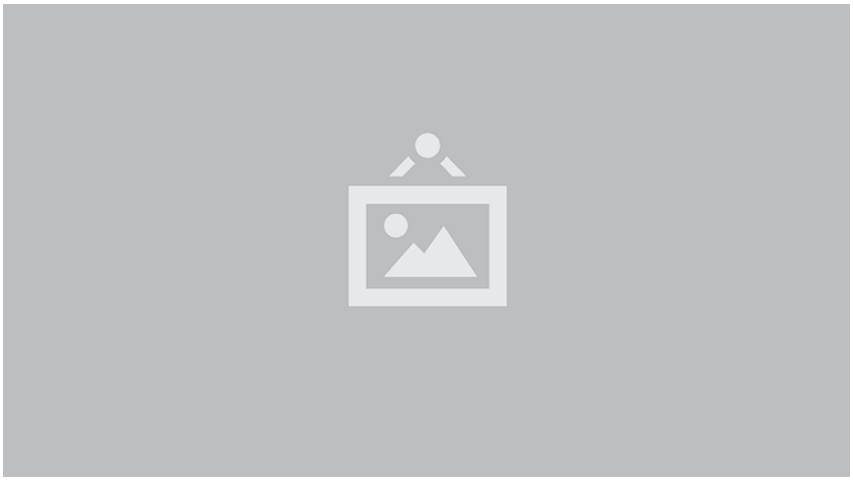
Figuur 2.16: Afstandsdempings curves voor eindige lichtbronnen.

$$(\frac{l}{r}\Big|_{[0,1]})^2$$

Als laatste dienen we intensiteit van de lichtbron vast te leggen. Zoals gebruikelijk binnen computer graphics, is hier gekozen voor een rgb voorstelling. Waarbij de waardes zich bevinden in het bereik van 0 tot en met 1.

Dit alles leidt ertoe dat we een lichtbron kunnen definieren als de set van de volgende eigenschappen:

- De positie $\mathbf{p}$ van het licht ten opzichte van een coordinatenstelsel met oorsprong O
- Een afstand r die de invloed van de lichtbron bepaald
- Een afstandsdempingsfunctie f die het verval van invloed moduleert



Figuur 2.17: De logische onderverdeling van de Moderne Grafische Pijplijn.

- Een intensiteit **i** die de kleur en kracht van de lichtbron bepaald

2.5 Moderne Grafische Pijplijn

De voorgaande secties hebben een grof overzicht gegeven van zowel de problemen als oplossingen binnen het renderen van afbeeldingen. Het onderzoek binnen deze thesis richt zich op het real-time renderen. Het onderliggende gereedschap, verantwoordelijk voor het renderen van de afbeeldingen is de real-time grafische pijplijn. In deze sectie zal eerst conceptueel de opbouw van de grafische pijplijn beschreven worden, waarna in meer detail de `openGL` implementatie besproken wordt, waarmee het onderzoek binnen deze thesis is gebouwd.

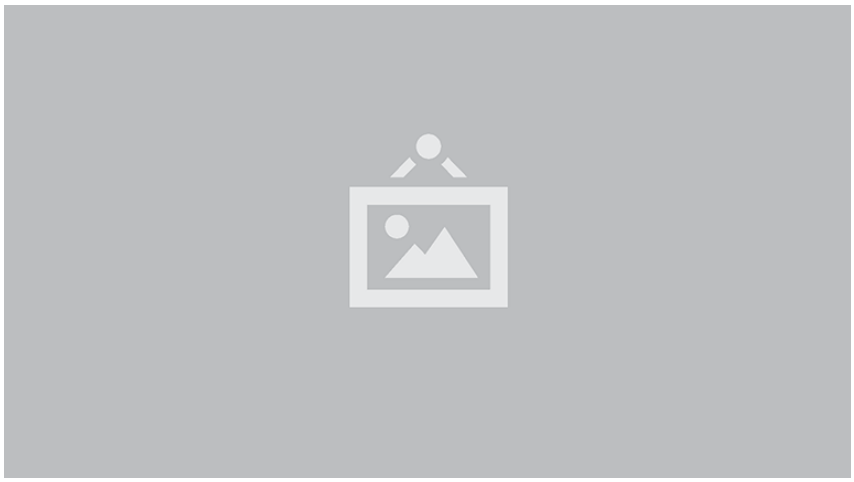
2.5.1 Conceptuele architectuur

In de fysieke wereld is een pijplijn een verzameling van stappen. Elke stap transformeert de uitkomst van de vorige stap in een volgende uitkomst. Elke stap bouwt dus verder op de voorgaande stap, echter elk van de stappen kunnen wel in parallel uitgevoerd worden. Dit betekent dat de snelheid van de pijplijn bepaald wordt door de stap met de langste bewerking.

De grafische pijplijn is op een vergelijkbare manier opgebouwd en kan grofweg onderverdeeld worden in 3 conceptuele stappen.

- Applicatie stap
- Geometrie stap
- Rasteriser stap

Deze zijn tevens weergegeven in figuur **??**.



Figuur 2.18: De logische onderverdeling van de Moderne Grafische Pijplijn.

Applicatie stap

De applicatie stap beschrijft alle berekeningen die plaatsvinden binnen de applicatie van de ontwikkelaar. Belangrijke aspecten hier zijn onder de verwerking van invoer van gebruikers, het opzetten van datastructuren, collision detection, etc. Aan het einde van de applicatie stap dient een verzameling primitieven gestuurd te worden naar de geometrie stap.

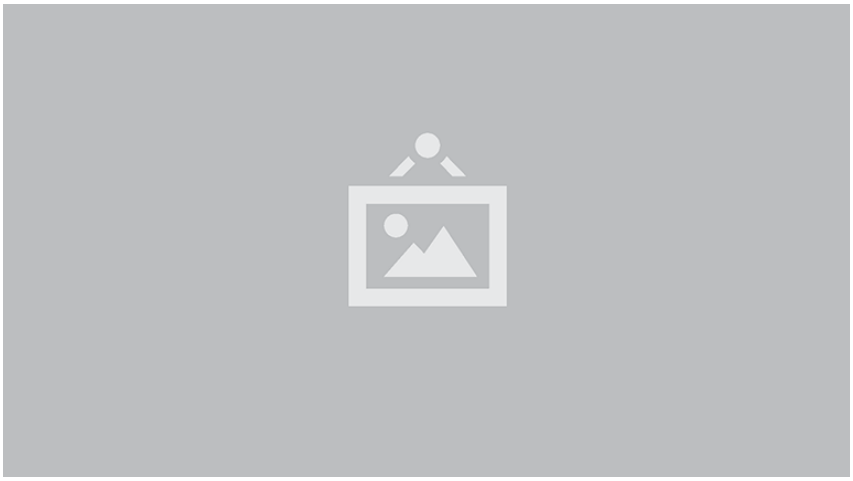
Geometrie stap

De geometrie stap is verantwoordelijk voor het merendeel van per-polygon operaties. Deze stap kan verder onderverdeelt worden zoals weergegeven in figuur **??**. De volgende sub-stappen kunnen onderscheden worden:

- Model- en zichtstransformaties
- Vertex shading
- Projectie
- Clipping
- Canvas afbeelding

Belangrijk hierbij is dat de conceptuele beschrijving in sommige opzichten kan verschillen van daadwerkelijke implementaties.

De geometrie stap zorgt dat objecten geproduceerd door de applicatie stap, omgezet worden naar een set van data die in de rasterisatie stap omgezet kan worden in een daadwerkelijke afbeelding. Eerst worden objecten getransformeerd zodanig dat alle primitieven zich in hetzelfde coordinaten systeem bevinden. Hierna vindt een eerste shading stap plaats die wordt uitgevoerd voor alle vertices van alle primitieven. Dit is een eerste stap in het oplossen van het shading probleem. Nadat de shading berekend is per vertex, wordt de perspectief projectie uitgevoerd en worden niet zichtbare objecten weggesneden uit het resultaat. Als laatste worden de primitieven



Figuur 2.19: De logische onderverdeling van rasterisatie stap.

die over zijn omgezet naar canvas coordinaten. De set van primitieven in canvas coordinaten, en corresponderende shading data wordt vervolgens doorgestuurd naar de laatste stap.

Rasterisatie stap

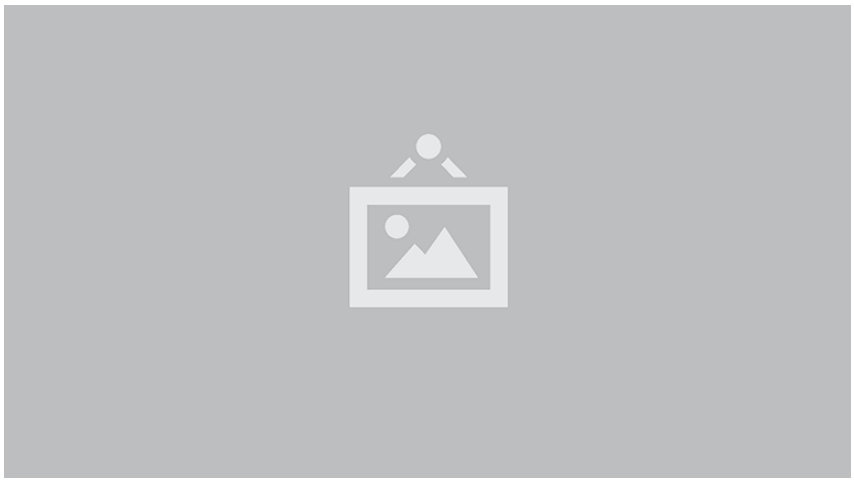
In de rasterisatie stap wordt de daadwerkelijke kleuren van de afbeelding berekend. Hiervoor wordt een rasterisatie algoritme uitgevoerd als beschreven in . Dit leidt tot de onderverdeling zoals weergegeven in figuur **??**. De volgende sub-stappen kunnen onderscheden worden:

- Driehoek opzet
- Driehoek doorkruizing
- Pixel shading
- Samenvoeging

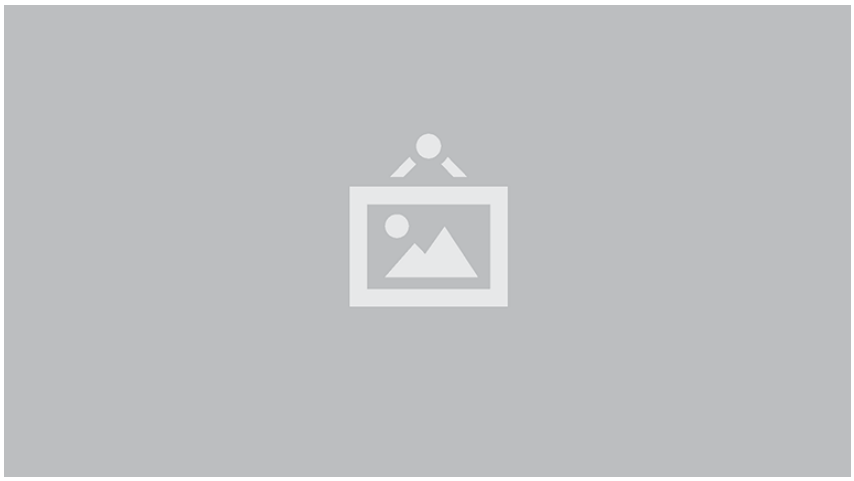
De eerste twee stappen komen overeen met het rasterisatie algoritme. Hierbij wordt shading data uit de geometrie stap geinterpoleerd. Dit leidt tot een set van fragmenten met geassocieerde geinterpoleerde shading data. Deze worden met behulp van pixel shading verwerkt tot een specifieke kleur voor een specifieke pixel. Als laatste wordt doormiddel van het z-buffer algoritme en specificaties van de ontwikkelaar, elk van deze potentiele pixel waardes samengevoegd tot een specifieke kleur waarde die weergegeven kan worden binnen de afbeelding.

2.5.2 Moderne Grafische Pijplijn implementatie

De moderne grafische pijplijn wordt gedefinieerd als een programmeerbare pijplijn. Dit houdt in dat de ontwikkelaar in staat is om zelf algoritmes te implementeren. Er zijn verschillende `APIs` beschikbaar waarmee een ontwikkelaar de grafische pijplijn



(a) openGL



(b) Direct3D

Figuur 2.20: De stappen van zowel de openGL als Direct3D implementaties.

kan gebruiken. De twee meest gebruikte industrie standaarden zijn `openGL` en `Direct3D`. `openGL` is een niet platform specifieke specificatie. Om deze reden is gekozen voor het gebruik van `openGL` binnen deze thesis. `openGL` en `Direct3D` zijn in grote mate vergelijkbaar, echter verschillen in nomenclatuur. In deze uitleg zal gebruik gemaakt worden van de naamgeving zoals geintroduceerd in `openGL`. Indien een nieuwe term geintroduceerd wordt zal waar nodig ook de `Direct3D` equivalent worden gespecificeerd.

De moderne grafische pijplijn is een grote mate programmeerbaar, echter in is wegens efficientie redenen slechts configureerbaar. Een overzicht van de verschillende stappen van de pijplijn voor respectievelijk `openGL` en `Direct3D` zijn gegeven in figuur **??** en **??**. Door middel van kleuren is de mate van programmeerbaarheid, aangegeven. Tevens zijn optionele stappen aangegeven door een stippellijn. In beide

APIs zijn 9 stappen terug te vinden.

- Vertex Specificatie
- Vertex Shader
- Tesselatie
- Geometrie shader
- Vertex Post-Processing
- Primitieven assemblage
- Rasterisatie
- Fragment Shader
- Per-Sample operaties

De programmeerbaarheid van de pijplijn volgt uit de programmeerbare shaders, de *vertex shader*, *geometrie shader*, en *fragment shader*. Deze hebben respectievelijk invloed op vertices, primitieven en fragmenten. Fragmenten zijn de punten die worden teruggegeven nadat de primitieven zijn verwerkt door het rasterisatie algoritme, veelal komen deze overeen met pixels of subpixels.

Terugkijkend op de conceptuele beschrijving van de grafische pijplijn, komen de stappen als volgt overeen met de conceptuele stages. De applicatie stage is niet gedefinieerd binnen de openGL pijplijn, deze bevindt zich voordat de openGL pijplijn wordt aan gesproken. De applicatie stap eindigt met de vertex specificatie. De applicatie specificeert hierbij een set van primitieven. Vervolgens wordt deze verzameling van vertices verwerkt door een of meerdere vertex shaders. De ontwikkelaar heeft hier volledige controle over, echter veelal vinden hier de model en gezichts transformaties plaats. Ook is het mogelijk dat hier een eerste stap in shading plaatsvindt, wat vervolgens geinterpoleerd zal worden in volgende stappen. De tesselatie en geometrie kunnen gebruikt worden om primitieven aan te passen, of zelfs volledige nieuwe geometrie te produceren of juist weg te filteren. Deze stappen zijn optioneel. De vertex post-processing, assemblage en rasterisatie zijn allemaal fixed function operaties. Deze komen overeen met de stappen, clipping, canvas afbeelding, driehoek opzet en driehoek doorkruizing. Hierin vindt op hardware niveau de uitvoering van het rasterisatie algoritme plaats, en wordt een set van fragmenten geproduceerd. Binnen de fragmentshader worden deze fragmenten gebruikt om per-pixel shading uit te voeren, waarmee een specifieke kleur wordt gegenereerd voor elk fragment. Als laatste vind dan de samenvoeging van fragmenten plaats in de per-sample operaties. Belangrijk hierbij is dat hier tevens het z-buffer algoritme wordt uitgevoerd. Dit betekent dat per-pixel shading wordt uitgevoerd voor alle pixels, en dus tevens voor pixels die uiteindelijk helemaal geen invloed hebben op de uiteindelijke afbeelding.

Als laatste zal kort ingegaan worden op de vertex en fragment shaders, en de per-sample operaties.

Vertex Shader

De vertex shader behandelt exclusief de punten, vertices, die gespecifieerd worden door de applicatie. De shaders zelf heeft geen kennis hoe elk van de vertices zich

verhoudt tot primitieven. Veelal is de vertex shader verantwoordelijk voor het omzetten van de coordinaten van model naar camera of wereld ruimte, afhankelijk van de specificatie van de fragment shader. Tevens dient hier de locatie gezet te worden van de specifieke geprojecteerde locatie.

Fragment Shader

Nadat de primitieven omgezet zijn naar een set van fragmenten, wordt op elk van de fragmenten de gespecificeerde fragment shader uitgevoerd. De rasterisatie stap produceert een set van data, waaronder specifieke locatie van fragmenten, en interpolatie van berekende waardes binnen de vertex shader. Deze kunnen vervolgens gebruikt worden door de fragment shader, om de shading van dat fragment te berekenen.

De fragmentshader berekent de kleur voor elk fragment, voordat deze wordt samengevoegd in de per-sample operaties stap. Dit is veelal de stap binnen de grafische pijplijn die de meeste berekeningsmiddelen vereist. In moderne applicaties wordt hier veelal de benadering van de renderingsvergelijking berekend. Tevens is het mogelijk voor de fragmentshader, om resultaten naar meerdere verschillende renderdoelen (*multiple render targets*) weg te schrijven.

Per-Sample Operaties

De laatste stap van een enkele uitvoering van de renderpijplijn bestaat uit de per-sample operaties. Hierin worden de verschillende fragmenten samengevoegd en weggeschreven naar de framebuffer, door middel van het z-buffer algoritme. Verder kunnen hier stappen plaatsvinden zoals compositie en het mixen van kleuren, wat belangrijk is voor de ondersteuning van transparantie.

Zoals eerder vermeld is een belangrijke observatie dat de fragmentshader wordt uitgevoerd voor elk fragment, ongeacht of deze daadwerkelijk zichtbaar is. Dit kan leiden tot een grote mate van onnodige berekeningen, indien de scene bestaat uit veel primitieven. Oplossingen hiervoor zullen verder besproken worden in volgende hoofdstukken.

2.6 Conclusie en verder informatie

In dit hoofdstuk is een introductie gegeven tot real-time computer graphics. De perceptie en fysische eigenschappen van licht vormen de onderliggende basis voor computergraphics. Hierbij dienen de eigenschappen van licht en perceptie gesimuleerd te worden om geloofwaardige afbeeldingen te creeeren. Deze simulatie kan grofweg in twee problemen worden opgedeeld.

- Wat is zichtbaar
- Hoe wordt het zichtbare waargenomen.

Het eerste probleem leidt tot perspectief, en het visibiliteit probleem. Het tweede probleem wordt opgelost doormiddel van het benaderen van de renderingvergelijking.

Binnen realtime graphics wordt dit alles gerealiseerd door middel van de moderne grafische pijplijn. Om deze pijplijn te gebruiken wordt binnen deze thesis gebruik gemaakt `openGL`.

Voor een uitgebreidere beschrijving van de behandelde problemen, is het mogelijk om te referen naar de volgende boeken. Voor de psychologische grondslag van kleur en perceptie, kan gerefereerd worden naar Wolfe's Sensation and Perception ([9]). De fysica die ten grondslag ligt aan computer graphics kan gevonden worden in Optics door Eugene Hecht ([3]). Meer toegepast op computer graphics, en tevens voor de behandeling van shading en raytracing, zijn de boeken Raytracing from the ground up ([6]), en Physical Based Rendering ([5]), een goede basis. Voor de basis van real-time rendering en de moderne grafische pijplijn vormt Real-Time Rendering ([1])een goede basis. De basis voor `Direct3D` en `openGL` kan het beste begonnen worden bij hun respectievelijke documentatie websites.

Hoofdstuk 3

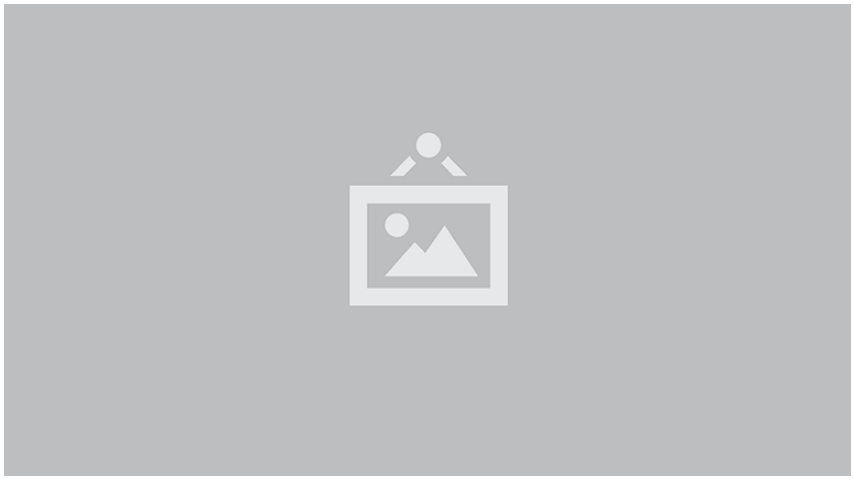
Literatuur-studie

3.1 Forward en deferred shading

Binnen sectie **??** is vastgesteld dat bij standaard uitvoering, pas na uitvoering van de fragment shader wordt bepaald welke fragmenten daadwerkelijk zichtbaar zijn. Dit betekent dat ook voor fragmenten die geen enkele invloed zullen hebben op de scene, de shading berekening worden uitgevoerd. De shading complexiteit is dus direct gekoppeld aan de scene complexiteit. Voor simpele scenes is dit geen probleem, echter wanneer scenes complexer worden kan dit leiden tot een grote mate van verspilde rekenkracht. Een simpel voorbeeld van een dergelijke scene, waar belichtingsberekening onnodig worden uitgevoerd is weergegeven in figuur 3.1.

Een logische stap om dit probleem op te lossen is het ontkoppelen van visibiliteit en shading. Dit leidt tot twee discrete stappen, een visibiliteitsstap waar rasterisatie plaats vindt en de informatie van de zichtbare fragmenten als uitvoer wordt gegeven.

Deze oplossing is niet nieuw binnen computer graphics, zo werd in de scan-line polygon renderer van Watkins ([8]) in de jaren 70 al bepaald welke oppervlakte het



Figuur 3.1: Een scene met een grote hoeveelheid van verborgen geometrie.

dichtst bij de camera lag en slechts hiervoor de lichtberekeningen uitgevoerd. Andere hardware en software implementaties zijn tevens in de jaren 80 geintroduceerd []. Een specifieke beschrijving van deferred shading is gegeven in []. Moderne deferred shading algoritmes maken veelal gebruik van het concept GBuffers. GBuffers waren oorspronkelijk geintroduceerd in de context van het non-photorealistsch renderen om visuele begrijpbaarheid te verbeteren.[] Echter het concept van GBuffers leent zich om de data tussen de visibiliteitsstap en de shading stap op te slaan.

In deze sectie zal ingegaan worden op het algoritme van deferred shading om de koppeling tussen geometrie complexiteit en shading op te lossen. Vervolgens zal gekeken worden naar nieuwe algoritmes die verder bouwen op deferred shading.

3.1.1 Definities

Binnen deze thesis zal de volgende terminologie gebruikt worden. *Forward shading* beschrijft de standaard uitvoering van de render pijplijn. Hier wordt niet expliciet de fragmenten op diepte gefilterd, en dus zal voor elk fragment de belichtingsberekeningen uitgevoerd worden.

Deferred shading beschrijft het algoritme waarbij expliciet de render pijplijn wordt onderverdeeld in twee discrete stappen, een geometriestap en een belichtingsstap. De visibiliteitsstap maakt hierbij gebruik van een *GBuffer* om de geometrie data op te slaan.

Light-pre pass beschrijft het algoritme waarbij eerst een z-pass wordt uitgevoerd voordat de shading stap wordt uitgevoerd.

3.1.2 Deferred shading

Zoals besproken in sectie 3.3.1, is de rendering vergelijking gegeven door:

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

Uitgaande dat een punt geen licht uitstraalt, dan zou de rendering vergelijking benaderd kunnen worden door middel van een beschrijving van directe belichting:

$$L_o(\mathbf{p}, \omega_o) = \sum_{k=1}^{n} f_r(\mathbf{p}, l_k, \omega_o) L_i(\mathbf{p}, l_k) \cos\theta_i$$

Hierbij is licht k gedefinieerd als l_k. Een licht bevat alle relevante informatie, positie, intensiteit, etc. De inkomende radiantie van een enkel licht l_k kan gedefinieerd worden als:

$$L_i(\mathbf{p}, l_k) = f_{\mathtt{att}}(d) L_k$$

waar d gedefinieerd als de afstand tussen punt $\mathbf{p}$ en licht l_k. Zoals beschreven in de definitie van licht.

Binnen forward shading wordt deze radiantie berekening uitgevoerd in fragment shader. Dit betekent zowel dat voor elk fragment deze berekening dient te worden

uitgevoerd, en dat alle informatie per fragment beschikbaar is. Pas nadat de radiantie berekening is uitgevoerd, wordt bekeken of een fragment daadwerkelijk wordt opgeslagen of niet. Om deze berekening te ontkoppelen, moet de lichtstap na de zichtbare fragment bepaling worden uitgevoerd. Dit betekent dat de verwerking van fragmenten al is voltooid, voordat de radiantie wordt berekend. Hierdoor zijn de gegevens over elk fragment niet meer impliciet beschikbaar, en zullen deze expliciet moeten worden opgeslagen ten tijde van de visibiliteit bepaling.

Bij inspectie van de functie om L_o te berekenen, kunnen de volgende attributen geidentificeerd worden:

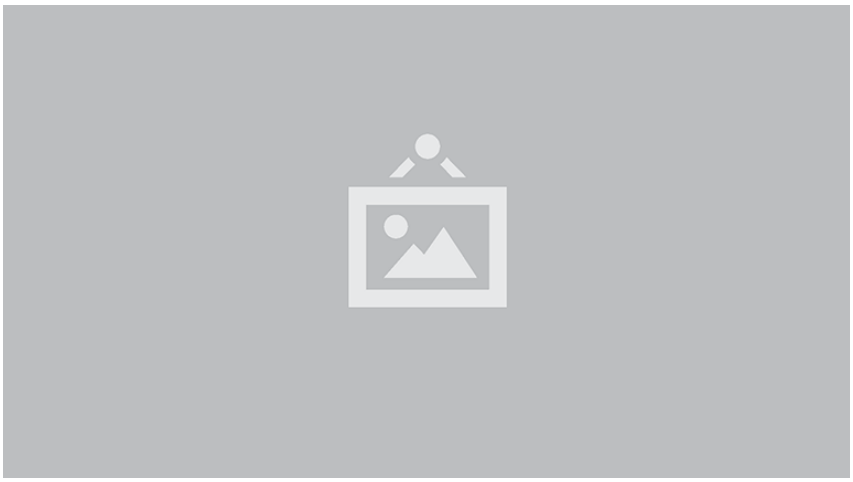
- De geometrie informatie van punt **p**.
  - De positie van punt **p**
  - De normaal in punt **p**
- Informatie met betrekking tot de oppervlakte in punt **p**
  - De kleur van het oppervlakte
  - Eventuele extra attributen zoals reflectie coefficient, ruwheid etc.
- Informatie van de lichten
  - Positie
  - Intensiteit
  - Afstandsdempingfunctie

Binnen de geometrie stap is het nodig om de benodigde informatie van de geometrie en oppervlakte expliciet op te slaan. De informatie van de lichten is niet afhankelijk van de fragmenten en kan op een zelfde manier voor de shading stap beschikbaar gemaakt worden als gedaan wordt in forward shading. Indien deze eigenschappen worden bepaald voor elk van de fragmenten, zullen de per pixel operaties ervoor zorgen dat slechts voor de zichtbare fragmenten deze attributen zijn opgeslagen. De shading stap zal vervolgens deze waardes uit het geheugen uitlezen en gebruiken om de radiantie te berekenen.

Gbuffer

Om het opslaan van deze attributen te faciliteren wordt een techniek gebruik die GBuffers genoemd wordt. Dit is een object bestaande uit meerdere textures, elk verantwoordelijk voor een attribuut dat opgeslagen dient te worden tussen geometry en shading pass. Een voorbeeld van deze textures voor een enkel frame is gegeven in figuur 3.2. Hierin zijn de positie, normaal, diffuse kleur en textuur coordinaten weergegeven.

Moderne grafische kaarten hebben de mogelijkheid om te renderen naar meerdere texturen in een enkele uitvoering van de pijplijn. Deze mogelijkheid wordt meerdere render doelen (Multiple Render Targets (MRT)) genoemd. Hiervan wordt gebruik gemaakt om de GBuffer te vullen met informatie in de geometrie pass. Deze textures zullen in het geheugen van de grafische kaart beschikbaar zijn, en opgevraagd kunnen worden gedurende de shading stap.



Figuur 3.2: De texturen in een GBuffer.

Algoritme

Als een laatste optimalisatie kan opgemerkt worden dat veelal lichten slechts, een beperkte invloed hebben op de scene. Slechts pixels binnen het lichtvolume kunnen worden gekleurd door een licht. Het is mogelijk om de lichtvolumes te rasteriseren, en slechts de licht benadering uit te voeren voor deze fragmenten. Dit leidt tot het volgende algoritme.

```
# Geometry pass
# ------------------------------------------------
for obj in scene_objects:
    fragments = rasterise(obj)

    for frag in fragments:
        write_to(gbuffer, frag.attributes)

per_pixel_operations()

# Shading pass
# ------------------------------------------------
for light in scene_lights:
    fragments = rasterise(light.volume)

    for frag in fragments:
        attributes = look_up(gbuffer, frag.pos)
        canvas[frag.pos] += do_shade(frag, attributes, light)

per_pixel_operations()
```

Nadelen

Het grootste probleem met het gepresenteerde algoritme is de hoge bandbreedte vereiste. Bij een toenemend aantal lichten zal voor elk fragment de attributen meerdere malen moeten worden opgevraagd. Doordat de geheugenbandbreedte binnen een grafische kaart beperkt is kan dit een knelpunt worden. Tevens zal de GBuffer bij toenemende complexiteit van de shader, meer attributen moeten opslaan, wat leidt tot een groter geheugen verbruik.

Een tweede belangrijk minpunt is dat dit algoritme slechts werkt voor ondoorzichtige objecten. De GBuffers zijn niet in staat om meer dan een laag geometrie op te slaan, in het geval dat er transparantie nodig is binnen een is het niet mogelijk om de attributen hiervan op te slaan. Om transparantie te ondersteunen binnen deferred shading zullen of wel meerdere gbuffers nodig zijn, of wel een aparte rendering pass voor alle transparante objecten. Het bijhouden van meerdere GBuffers leidt tot groot geheugenverbruik, en is daarom veelal onbruikbaar.

Een derde probleem is dat anti-aliasing niet meer triviaal ondersteun wordt. Binnen de shading pass is het niet mogelijk om sub-pixels te bemonsteren, gezien ook deze data expliciet opgeslagen moet worden. Hiervoor bestaan wel alternatieve anti-aliasing technieken die werken binnen een deferred shading context, voorbeelden hiervan zijn ....

3.1.3 Forward+ shading

Forward+ shading, lighting pre-pass, deferred lighting, zijn een collectie van vergelijkbare algoritmes die verder bouwen op deferred shading, met als belangrijkste doel het geheugen verbruik en bandbreedte te verminderen.

Het gaat uit van het concept dat de set van BRDFs die de kleur van een oppervlakte bepaald veelal bestaat uit individuele lichttermen, difuus en speculair. Indien de benadering van de rendering vergelijking wordt herschreven in een vorm van een diffuus en een speculair component, dan wordt de functie:

$$L_o(\mathbf{p}) = \sum_{k=1}^{n} \left(\mathbf{c}_{\texttt{diff}} * f_{\texttt{diff}}(L_{i_k}, \mathbf{l}_k, \mathbf{n}) + \mathbf{c}_{\texttt{spec}} * f_{\texttt{spec}}(L_{i_k}, \mathbf{l}_k, \mathbf{n}, \mathbf{p}, m) \right)$$

waarbij $f_{\texttt{spec}}$ en $f_{\texttt{diff}}$ de shading functies zijn, onafhankelijk van de mogelijke kleur van een punt. Deze worden bepaald door de intensiteit van het licht L_{i_k}, de lichtinvals hoe op het punt $\mathbf{p}$ en de normaal $\mathbf{n}$ in punt $\mathbf{p}$. Doordat de speculaire en diffuse term onafhankelijk zijn is het mogelijk om de benadering van de rendering vergelijk te herschrijven tot:

$$L_o(\mathbf{p}) = \mathbf{c}_{\texttt{diff}} * \sum_{k=1}^{n} \left(f_{\texttt{diff}}(L_{i_k}, \mathbf{l}_k, \mathbf{n}) \right) + \mathbf{c}_{\texttt{spec}} * \sum_{k=1}^{n} \left(f_{\texttt{spec}}(L_{i_k}, \mathbf{l}_k, \mathbf{n}, \mathbf{p}, m) \right)$$

Algoritme

Deze opdeling leidt tot het volgende algoritme

1. Rasteriseer de ondoorzichte geometrie, waarbij de normaal vector $\mathbf{n}$ en schrijf per fragment de speculaire spreiding m weg naar een texture. Deze $\mathbf{n}m$ buffer is vergelijkbaar met een GBuffer maar bevat signifacant minder geheugen.
2. Rasteriseer de licht volumes en bereken per fragment de shading functies f. In het geval dat $f_{\texttt{spec}}$ en $f_{\texttt{diff}}$ berekend dienen te worden, zullen hier twee textures voor nodig zijn, waarin de waardes geaccummuleerd worden.

$$G_{\texttt{diff}} = \sum_{k=1}^{n} \left(f_{\texttt{diff}}(L_{i_k}, \mathbf{l}_k, \mathbf{n}) \right)$$
$$G_{\texttt{spec}} = \sum_{k=1}^{n} \left(f_{\texttt{spec}}(L_{i_k}, \mathbf{l}_k, \mathbf{n}, \mathbf{p}, m) \right)$$

3. Rasteriseer opnieuw de ondoorzichte geometrie, ditmaal wordt de data uit de accumulatie fase uitgelezen en vermenigvuldigd met de kleuren van het punt $\mathbf{p}$, tevens worden in deze stap andere licht berekening gedaan die niet uitgevoerd zijn in de accumulatie fase. Dit leidt tot:

$$L_o(\mathbf{p}) = \mathbf{c}_{\texttt{diff}} * G_{\texttt{diff}} + \mathbf{c}_{\texttt{spec}} * G_{\texttt{spec}}$$

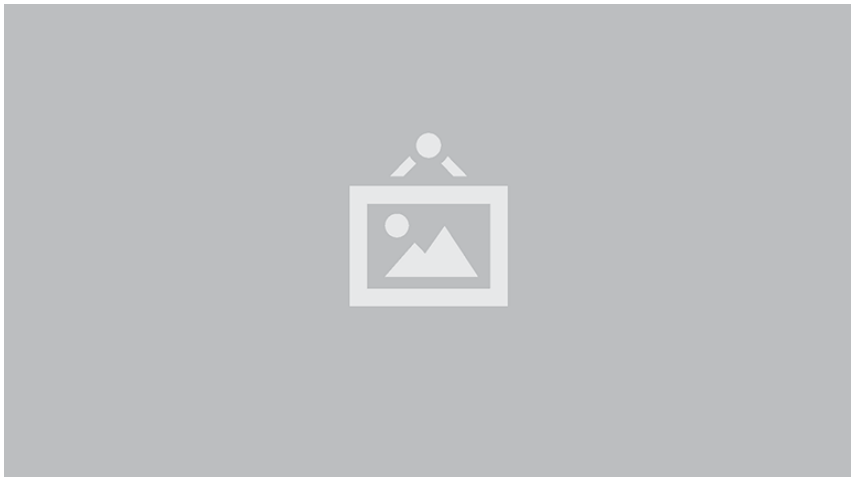
3.1.4 Conclusie

In deze sectie zijn twee algoritmes geintroduceerd als oplossing voor de koppeling tussen geometrie complexiteit en shading complexiteit. Beide delen het renderprocess op in meerdere stappen, zodat slechts de shading berekeningen uitgevoerd hoeven te worden voor de zichtbare elementen.

3.2 Tiled Shading

Een eerste stap in de optimalisatie van de lichttoekenning binnen beide render pijplijnen is Tiled Shading. Tiled shading beperkt het aantal lichtberekeningen per fragment door het canvas in te delen in een rooster. Voor elk vlak wordt berekend welke lichten daadwerkelijk invloed hebben binnen dit vlak. Vervolgens worden niet meer alle lichten binnen de scene overlopen, maar wordt slechts gekeken naar de lichten binnen een vlak waarin een fragment valt. Dit leidt tot een onderverdeling van het gezichtsfrustum, zoals weergegeven in figuur 3.3.

Binnen deferred tiled shading lost deze aanpak tevens het bandbreedte probleem op zoals deze geintroduceerd is binnen de sectie 3.1.2. Binnen het oorspronkelijke algoritme werden lichten een voor een behandeld, door het licht volume van elk licht te rasteriseren, en vervolgens data op te halen uit de G-buffer. Binnen tiled deferred shading loopt de binneste loop over de pixels, en kan per fragment direct opgehaald worden welke lichten invloed hebben op het fragment in kwestie.



Figuur 3.3: De onderverdeling van het gezichtsveld dat plaatsvindt in Tiled Shading.

De techniek zelf is al langere tijd in gebruik binnen de game industrie ([2], [7]). Een formelere beschrijving en evaluatie van de resultaten kan gevonden worden in [4].

De techniek heeft verder als voordelen voor beide renderpijplijnen dat:

- Gemeenschappelijk termen in de rendering vergelijking kunnen buiten de vergelijking gehaald worden.
- Licht berekening vindt plaats op register niveau.
- Dezelfe shading functies kunnen gebruikt worden.

Verder zijn er nog enkele specifieke voorden voor zowel forwaards als deferred. In het geval van deferred rendering:

- G-Buffers worden slechts een enkele keer per fragment gelezen
- De framebuffer wordt slechts een enkele keer naar weggeschreven.
- Fragmenten binnen dezelfde tile ondergaan dezelfde berekening.

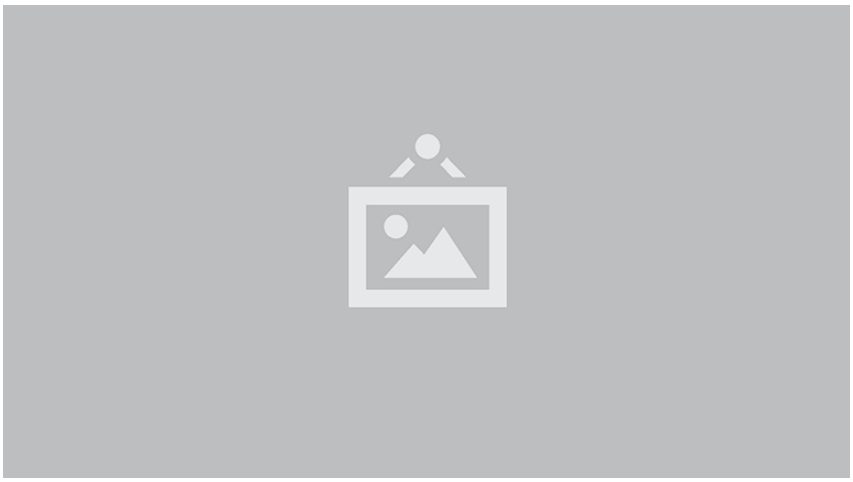
Voor forwaards renderen:

- Lichtmanagement is losgekoppeld van de geometrie.
- Licht data hoeft slechts een enkele keer naar de GPU worden geladen per scene.
- FSAA werkt zoals verwacht.

3.2.1 Algoritme beschrijving

Het algoritme bestaat uit de volgende stappen:

1. Bepaal het canvas rooster dat over de gehele frame buffer valt op basis van een vaste vlak grootte $t = (x, y)$ bijvoorbeeld, 32×32



Figuur 3.4: De datstructuur van Tiled Shading.

2. Voor elk licht, bepaal de vlakken waar het licht invloed heeft, en voeg aan deze vlakken de identifier van het licht toe.
3. Voer de standaard render stap uit, tot aan de fragment shader.
4. Voor elk fragment, bepaal het vlak waartoe deze behoort. Op basis van dit vlak bepaal de lichten die invloed hebben op dit fragment, en verzamel de licht contributie op basis van deze lichten.

Opbouwen van het rooster

De vlakgrootte $\mathbf{s}$ is een variable waar de ontwikkelaar verantwoordelijk voor is. De keuze is een afweging tussen geheugen en rekentijd. Een kleiner waarde s heeft als gevolg dat er meer geheugen nodig is voor het opslaan van het rooster, gezien bij een vergelijkbare grootte van canvas er meer vlakken in het rooster bijgehouden dienen te worden. Bij het opstellen van het rooster zal tevens een iets grotere berekeningstijd nodig zijn, doordat elk licht aan meer roosters toegevoegd dient te worden. Echter gezien dit een triviale stap is, zal de invloed op de berekeningstijd van de applicatie minimaal zijn. Wanneer de fragment shader bereikt is zal minder berekeningstijd nodig zijn, doordat er minder lichten aan elk vlak toegekend zijn, kleinere vlakken zullen een betere benadering van lichten geven. Binnen deze thesis zal, als niet anders vermeld is, een vlakgrootte $\mathbf{s}$ van 32×32 gebruikt zijn.

Licht projectie

Met de grootte van vlakken vastgesteld is het nodig om te bepalen welke lichten invloed hebben op welke vlakken. Dit kan gedaan worden door de lichtbollen te projecteren op de canvas, en te bepalen welke vlakken overlappen met de projectie.

Datastructuren

De shaders moeten instaat zijn om op te zoeken welke lichten invloed hebben op specifieke fragmenten. Om dit instaat te stellen, wordt gebruik gemaakt van drie lijsten:

Globale Lichtlijst De globale lichtlijst is een simpele lijst die elk van de lichten bevat, met de relevante data om een licht berekening uit te voeren.

Vlak-licht-index-lijst De vlak-licht-index lijst is een opeenvolging van indices die referen naar lichten binnen de globale lichtlijst, om zo het gebruikte geheugen te beperken. Deze is zodanig opgesteld dat elk vlak in het rooster refereert naar een subset van deze index-lijst doormiddel van een bepaalde afstand van het begin en een lengte van lichten die behoort tot dit vlak.

Lichtrooster Het lichtrooster definieert de afstand en lengte van elk vlak binnen de vlak-licht-index-lijst. Elk vlak heeft een unieke positie binnen het lichtrooster, dat gebruik kan worden om de licht indices in de index-lijst op te zoeken.

Deze datastructuren en hun onderlinge afhankelijkheid is geillustreerd in figuur 3.4.

3.2.2 Besluit

Binnen deze sectie is kort de werking van Tiled Shading beschreven. De implementatie kan gevonden worden in sectie ... De techniek bouwt op de observatie dat niet alle lichten effect zullen hebben op alle pixels. Door middel van een rooster van vlakken wordt bijgehouden welke lichten invloed hebben op welke set van pixels. Hierdoor is het mogelijk om de hoeveelheid lichten die berekend dient te worden terug te brengen van alle lichten die in de scene vallen tot slechts die worden afgebeeld op elk vlak.

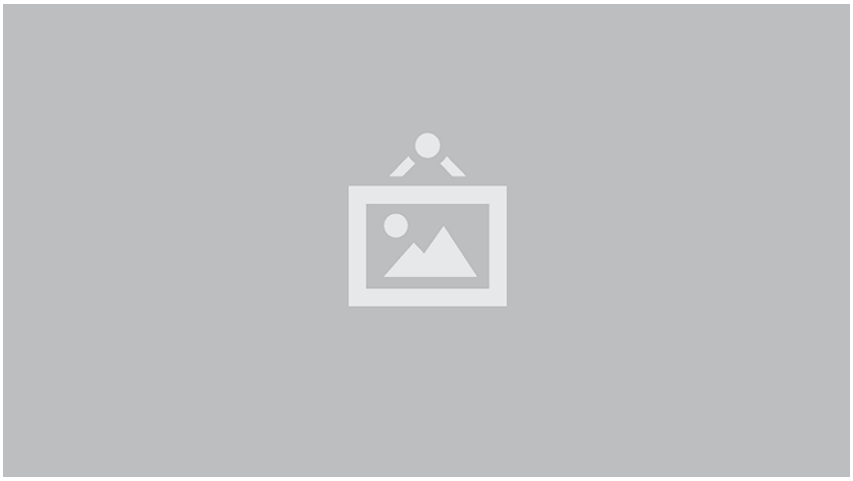
3.3 Clustered Shading

Clustered Shading bouwt verder op de light toekenning geintroduceerd in tiled shading. Tiled shading presteert slechter wanneer een hoop lichten achter elkaar geplaatst zijn, die fragmenten in de diepte verlichten, zoals de straat scene in figuur 3.5. Zoals te zien is, zijn er vlakken die alle lichten bevatten, echter fragmenten binnen deze vlakken zullen niet verlicht worden door elk van deze lichten. Clustered shading lost dit op door van vlakken naar hogere dimensies clusters te gaan. In de eerste plek wordt de diepte meegenomen voor de bepaling van clusters, verder kunnen andere attributen zoals bijvoorbeeld normalen gebruik worden om verder onderscheid tussen fragmenten te maken.

3.3.1 Algoritme

Het algoritme voor Clustered Shading bestaat uit de volgende stappen:

1. Render de scene naar de GBuffers.



Figuur 3.5: Een straat scene waar tiled shading slecht op prestreert.

2. Bereken de clusters.
3. Bepaal de unieke clusters.
4. Ken de lichten toe aan de clusters.
5. Voer de licht berekeningen uit voor shading.

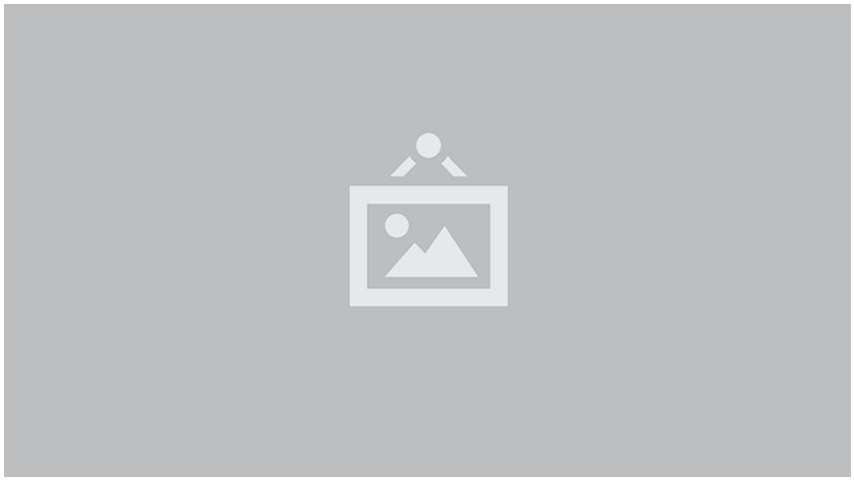
In het geval van forward clustered shading wordt in de eerste stap de z-buffer gevuld, en eventuele alternatatieve buffers die gebruikt worden binnen de clusters. Deze zullen dan vervolgens worden gebruikt voor de bepaling van de unieke clusters.

Cluster bepaling

Het doel van cluster bepaling is om een functie op te stellen die een fragment afbeeldt op een integer, zodanig dat alle vergelijkbare fragmenten dezelfde waarde toegekend krijgen. Binnen clustered shading wordt hiervoor de locatie van een fragment gebruikt, en eventueel de normaal, echter andere attributen zijn ook mogelijk.

Ideaal gezien bevat een cluster waartoe een fragment behoort slechts de lichten die invloed hebben op dat fragment. Hiervoor is het nodig dat de clusters klein zijn en bestaan uit vergelijkbare fragmenten, zodanig dat er weinig lichten toegekend hoeven te worden aan een cluster die geen invloed hebben op een deel van de fragmenten. Hier staat tegenover dat het de efficientie van de cluster berekening, en de lichttoekenning, en het geheugengebruik ten goede komt als er zoveel mogelijk fragmenten binnen een zelfde cluster zitten.

Gezien vergelijkbare fragmenten veelal ruimtelijk dicht bij elkaar liggen, is het logisch om een ruimtelijk opdeling te maken. Binnen clustered shading is gekozen voor een onderverdeling van de het zichtsfrustrum in kleinere sub frustrums. De uitgangsbasis hiervoor is het rooster opgesteld in tiled shading. Elk van de vlakken vormt een ruimte strekkend van de $z_{\texttt{near}}$ tot de $z_{mathttfar}$. Deze worden opgedeeld om zo subfrustrums binnen het zichtsfrustrum te krijgen. Om er voor te zorgen dat



Figuur 3.6: De exponentiele opdeling van het zichtsfrustrum in de z-as.

de dimensies van de subfrustrums in elke richting vergelijkbaar zijn, is er gekozen voor een exponentiele opdeling, zoals weergegevin in figuur 3.6.

De functie waarmee fragmenten afgebeeld worden op de sleutel van clusters kan nu als volgt gegeven worden. We stellen dat de minimale sleutel van een cluster gedefinieerd wordt door de tuple van (i, j, k), waarmee de gediscretiseerde locatie in het zichtsfrustrum wordt vastgelegd. De x en y waardes komen respectievelijk overeen met i en j. Deze kunnen berekend worden vergelijkbaar hoe deze berekend worden in tiled shading. Uitgaande dat een fragment zich bevindt op de positie p_x en p_y in pixel coordinaten en de grote van een rooster is gedefinieerd als $\mathbf{s}$, dan geldt:

$$i = \lfloor p_x / s_x \rfloor$$
$$j = \lfloor p_y / s_y \rfloor$$

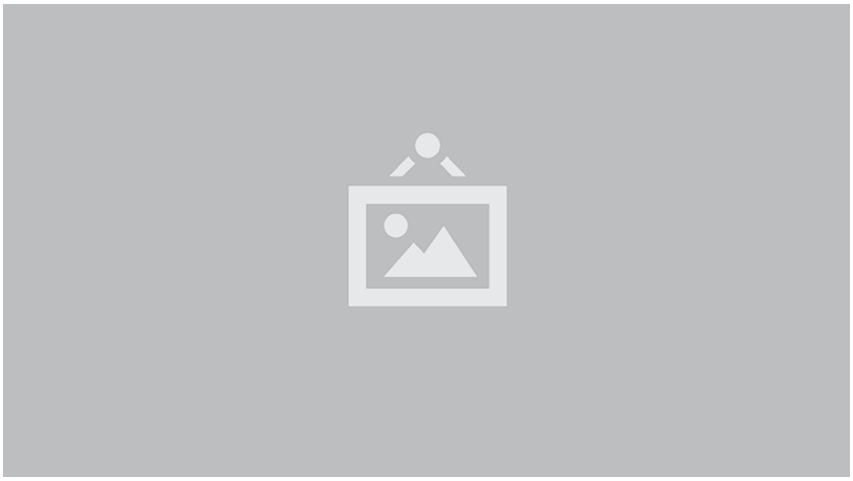
Uitgaande dat de waarde f_z in camera coordinaten is, en de afstand van het frustrum gegeven wordt door $(z_{\texttt{near}}, z_{\texttt{far}})$ dan kan de waarde voor k kan als volgt berekend worden:

$$k = \lfloor \frac{\log(\frac{f_z}{z_{\texttt{far}}})}{\log(1 + \frac{2 \tan \theta}{*} h)} \rfloor$$

waarbij h het percentage van de hoogte van een enkel vlak is tenopzichte van de gehele hoogte. Deze tuple kan vervolgens uitgebreid worden met andere attributen zoals de normaal. Elk cluster aanwezig in een frame is uniek gedefineerd door een dergelijke tuple.

Unieke cluster bepaling

Nadat voor elk fragment bepaald is tot welke cluster deze behoort, dienen alle unieke clusters bepaald te worden. Gezien slechts clusters belicht dienen te worden, is het



Figuur 3.7: De sorteer en comprimeer stap uitgevoerd over een enkel vlak.

niet nodig om voor elk cluster de invloeden van lichten te berekenen, en deze bij te houden in het geheugen.

Het vinden van unieke clusters kan gedaan worden door een sorteer en comprimeer stap uit te voeren. Deze technieken zijn veelal beschikbaar in bibliotheken om uitgevoerd te worden op grafische kaarten, en zijn tevens makkelijk te implementeren. Deze stappen zijn geillustreerd in figuur 3.7. Sorteren blijft echter een dure operatie. Een optimilasie ten opzichte van het globaal sorteren van clusters, kan geimplementeerd worden door slechts lokaal te sorteren per vlak. Sorteren over verschillende vlakken is onnodig, gezien deze per definitie van de tuples verschillend zullen zijn.

Light toekenning

In de meeste tiled shading algoritmes wordt licht toegekend aan vlakken door intersecties te vinden tussen vlakken en het lichtvolume. Dit is ook mogelijk voor clustered shading indien kleine hoeveelheden licht en clusters gebruikt worden. Echter indien de hoeveelheid lichten en clusters toeneemt, leidt dit tot performantie problemen.

Om een betere performantie te verkrijgen, worden lichten in de vorm van een begrensings volume hierarchie (bounding volume hierachy) opgeslagen. Voor elk cluster wordt deze BVH gebruikt om efficient te bepalen welke lichten invloed hebben op het cluster.

Shading

De shading functie gebruikt in de fragment shader komt in grote mate overeen met tiled shading. Door middel van de clusters is het opnieuw mogelijk om een fragment aan een bepaalde afstand en lengte binnen de licht index lijst te verkrijgen. Echter, niet alle clusters zullen expliciet bijgehouden worden, dit zou leiden tot onnodig geheugen verbruik. In dit geval bestaat er geen directe koppeling tussen de cluster

sleutel van een fragment, en de positie binnen een lijst van clusters. Om te bepalen tot welk cluster een fragment behoort zal niet tijdens de shading stap opnieuw de sleutel berekend worden, maar zal een textuur opgesteld worden tijdens de sorteer en comprimeer stap, bestaande uit de index van elk uniek cluster. Tijdens de shading stap zal deze worden uitgelezen en gebruikt worden om de relevante lichten te bepalen. Dit is weergegeven in figuur **??**.

3.3.2 Besluit

Clustered shading bouwt verder op tiled shading door de licht toekenning binnen het zichtsfrustrum toe te passen. Hiervoor worden lichten toegekend aan clusters die gedefinieerd worden door de posite van fragmenten, en eventuele andere attributen. Per frame wordt een clustering opgesteld die gebruikt wordt om shading functies uit te voeren.

3.4 Octree Datastructuren

3.4.1 Octree datastructuur

3.4.2 GPU structuren

Hoofdstuk 4

Implementatie

Hoofdstuk 5

Methode

Hoofdstuk 6

Resultaten

Hoofdstuk 7

Discussie

Hoofdstuk 8

Conclusie

Bibliografie

[1] Akenine-Möller, T., Haines, E., and Hoffman, N. *Real-Time Rendering, Third Edition*. CRC Press, 2016.

[2] Balestra, C., and Engstad, P.-K. The technology of uncharted: Drakes fortune. In *Game Developer Conference* (2008).

[3] Hecht, E. *Optics, Global Edition*. Always learning. Pearson Education, Limited, 2016.

[4] Olsson, O., and Assarsson, U. Tiled shading. *Journal of Graphics, GPU, and Game Tools 15*, 4 (2011), 235–251.

[5] Pharr, M., Jakob, W., and Humphreys, G. *Physically Based Rendering: From Theory to Implementation*. Elsevier Science, 2016.

[6] Suffern, K. *Ray Tracing from the Ground Up*. Taylor & Francis, 2007.

[7] Swoboda, M. Deferred lighting and post processing on playstation 3. In *Game Developer Conference* (2009).

[8] Watkins, G. S. A real time visible surface algorithm. Tech. rep., DTIC Document, 1970.

[9] Wolfe, J. M., Kluender, K. R., Levi, D. M., Bartoshuk, L. M., Herz, R. S., Klatzky, R. L., Lederman, S. J., and Merfeld, D. M. *Sensation & perception*, 4 ed. Sinauer Sunderland, MA, 2015.

Fiche masterproef

Student: Martinus Wilhelmus Tegelaers

Titel: realtime renderen van vele lichtbronnen

Engelse titel: Realtime rendering of many light sources

UDC: 621.3

Korte inhoud:

Thesis voorgedragen tot het behalen van de graad van Master of Science in de ingenieurswetenschappen: computerwetenschappen, hoofdspecialisatie Mens-machine communicatie

Promotor: Prof. dr˙ ir. P. Dutre

Assessor: T. Do

Begeleider: T.O. Do