

Forwards en deferred hashed shading

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

© Copyright KU Leuven

Zonder voorafgaande schriftelijke toestemming van zowel de promotor als de auteur is overnemen, kopiëren, gebruiken of realiseren van deze uitgave of gedeelten ervan verboden. Voor aanvragen tot of informatie i.v.m. het overnemen en/of gebruik en/of realisatie van gedeelten uit deze publicatie, wend u tot het Departement Computerwetenschappen, Celestijnenlaan 200A bus 2402, B-3001 Heverlee, +32-16-327700 of via e-mail info@cs.kuleuven.be.

Voorafgaande schriftelijke toestemming van de promotor is eveneens vereist voor het aanwenden van de in deze masterproef beschreven (originele) methoden, producten, schakelingen en programma's voor industrieel of commercieel nut en voor de inzending van deze publicatie ter deelname aan wetenschappelijke prijzen of wedstrijden.

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

Voorwoord	i
Samenvatting	iv
Lijst van figuren en tabellen	v
1 Inleiding	1
1.1 Probleemstelling	1
1.2 Motivatie	1
1.3 Overzicht Thesis	1
2 Theorie	3
2.1 Fysische werkelijkheid	3
2.2 Notaties en definities	7
2.3 Perspectief projectie en het visibiliteitsprobleem	14
2.4 Shading	22
2.5 Moderne Grafische Pijplijn	26
2.6 Probleemstelling	32
2.7 Conclusie en verder informatie	32
3 Implementatie overzicht	33
3.1 Overzicht	33
3.2 Shading: implementatie	34
3.3 Meet methodes	35
3.4 Test suite	36
4 Deferred Shading	43
4.1 Theorie	44
4.2 Algoritme	46
4.3 Resultaten	47
4.4 Conclusie	51
4.5 Discussie	54
5 Tiled Shading	57
5.1 Theorie	57
5.2 Algoritme	60
5.3 Testen en resultaten	60
5.4 Discussie	60
6 Hashed Shading	61

INHOUDSOPGAVE

6.1	Theorie	61
6.2	Algoritme	73
6.3	Implementatie in nTiled	84
6.4	Testen en resultaten	84
6.5	Discussie	84
7	Besluit	85
	Bibliografie	87

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

2.1	Waarneming doormiddel van het oog en camera.	4
2.2	Absorptie, reflectie en transmissie van licht.	5
2.3	Het mengen van kleuren volgens een additief model.	6
2.4	Voorstelling van objecten doormiddel van driehoeken.	11
2.5	Het standaard camera model.	13
2.6	Rechtshanding Carthesisch coördinatenstelsel.	14
2.7	Verschillende Carthesische coördinatenstelsels en bijbehorende transformaties.	15
2.8	Perspectief projectie.	16
2.9	Projectie van een enkel punt.	17
2.10	Visibiliteitsprobleem in een scene met meerdere primitieven.	18
2.11	Forwaards raytracen.	19
2.12	Raytrace algoritme.	20
2.13	Het rasterisatie algoritme.	21
2.14	Uitstralung van radiantie over ω_o vanuit \mathbf{p}	22
2.15	Lambertiaanse BRDF.	24
2.16	Afstandsdempings curves.	25
2.17	Voorstelling van licht.	25
2.18	Afstandsdempings curves voor eindige lichtbronnen.	26
2.19	De logische onderverdeling van de Moderne Grafische Pijplijn.	27
2.20	De logische onderverdeling van geometrie stap.	28
2.21	De logische onderverdeling van rasterisatie stap.	29
2.22	De stappen van zowel de OpenGL als Direct3D implementaties.	30
3.1	Een oppervlakte gerenderd met de standaard lambert shader binnen nTiled.	36
3.2	Grafisch overzicht van de indoor spaceship scene.	38
3.3	De verschillende aspecten van de piper's alley scene.	39
3.4	De verschillende afbeeldingen van de ziggurat stadsscene.	40
4.1	Een scene met een grote hoeveelheid van verborgen geometrie.	43

LIJST VAN FIGUREN EN TABELLEN

4.2	De texturen in de GBuffer gebruikt in killzone 2, geproduceerd door Guerrilla Games[30].	46
4.3	De texturen in de GBuffer gebruikt in nTiled.	47
4.4	Overzicht van de executie tijd per frame voor de drie test scenes bij verschillende resolutie en groottes van aantal lichten.	49
4.5	Indoor spaceship frames waarbij het aantal fragmenten één benaderd. .	50
4.6	Ziggurat stad frames waarbij de camera van positie verspringt.	50
4.7	Overzicht van de executie tijd als functie van het aantal lichten, per test scene bij een resolutie van 2560.	52
4.8	Overzicht van de executie tijd als functie van de resolutie, per test scene.	53
5.1	De onderverdeling van het gezichtsveld dat plaatsvindt in Tiled Shading.	57
5.2	De datstructuur van Tiled Shading.	59
6.1	De onderverdelingen binnen Tiled en Clustered Shading	62
6.2	Een binaire ruimte partitie.	63
6.3	Een roosterdatastructuur.	63
6.4	De roosterdatastructuur voorgesteld als BRP.	64
6.5	Weergave van een octree bestaande uit drie lagen, links is de 3d representatie weergegeven, rechts de pointers	64
6.6	De octree datastructuur als BRP.	65
6.7	De kd-boom datastructuur.	65
6.8	De r-boom datastructuur.	66
6.9	Een voorbeeld octree.	70
6.10	Een voorbeeld hoe een series van lagen van een octree gecodeerd kan worden met behulp van een ruimtelijke hash functies.	71
6.11	Een voorbeeld van de representatie van een octree met behulp van een verbindingloze octree.	72
6.12	Een visuele weergave van het algoritme om data uit de verbindingloze octree te halen.	72
6.13	Een overzicht van Hashed shading.	73
6.14	Voorstelling van de enkele licht boom.	74
6.15	Opbouw van het initiele rooster van de enkele lichtboom.	77
6.16	De lichtoctree.	78
6.17	Visuele weergave van de datastructuren binnen de constructie van de verbindingloze octree.	81
6.18	Visuele representatie van het hashed shading algoritme.	84

Lijst van tabellen

2.1	Wiskundige notaties.	7
2.2	Wiskundige operatoren.	8

6.1 De mogelijke situaties wanneer een enkele lichtboomknoop wordt toegevoegd aan een octreeeknoop.	80
---	----

Hoofdstuk 1

Inleiding

Dit is de algemene inleiding placeholder

1.1 Probleemstelling

probleem stelling placeholder

1.2 Motivatie

1.3 Overzicht Thesis

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

Hoofdstuk 2: Theorie Binnen het theorie hoofdstuk wordt een inleiding tot 3D computer grafieken gegeven. Hierbij worden visibiliteit, shading en real-time graphics besproken. Tevens wordt de benodigde wiskunde behandeld, en wordt de terminologie die gebruikt wordt binnen deze thesis uitgelegd.

Hoofdstuk 3: Implementatie overzicht In het implementatie overzicht wordt het programma waarin de verschillende algoritmes van deze thesis zijn geïmplementeerd besproken. Tevens wordt een overzicht gegeven van de gebruikte test-hardware en de gebruikte test-scenes.

Hoofdstuk 4: Forward en Deferred shading Binnen Forward en Deferred shading worden de algoritmes geïntroduceerd waarmee geometrische complexiteit ontkoppeld kan worden van shading complexiteit.

Hoofdstuk 5: Tiled en clustered shading Binnen het hoofdstuk Tiled en clustered shading worden het tiled lichttoekenningsalgoritme en het clustered lichttoekenningsalgoritme geïntroduceerd. Deze twee algoritmes vormen de basis voor Hashed shading, het nieuwe licht toekenningsalgoritme dat geïntroduceerd wordt binnen deze thesis. Voordat ingegaan wordt op Hashed shading zal de theorie en effectiviteit van deze twee algoritmes besproken worden.

Hoofdstuk 6: Hashed shading Het Hashed shading hoofdstuk introduceert een nieuw algoritme voor lichttoekenning. Dit algoritme ontkoppeld de lichttoe-

1. INLEIDING

kenning van het zichtsveld zonder een significant grotere aanspraak op het grafisch geheugen te maken. Dit wordt bereikt door gebruik te maken van een octree datastructuur op basis van hash functies, zodat deze compact en efficient gebruikt kan worden op de grafische kaart.

Hoofdstuk 7: Besluit In het besluit worden alle individuele algoritmes vergeleken, en wordt ingegaan op hun relatieve efficiency en geheugengebruik.

Hoofdstuk 2

Theorie

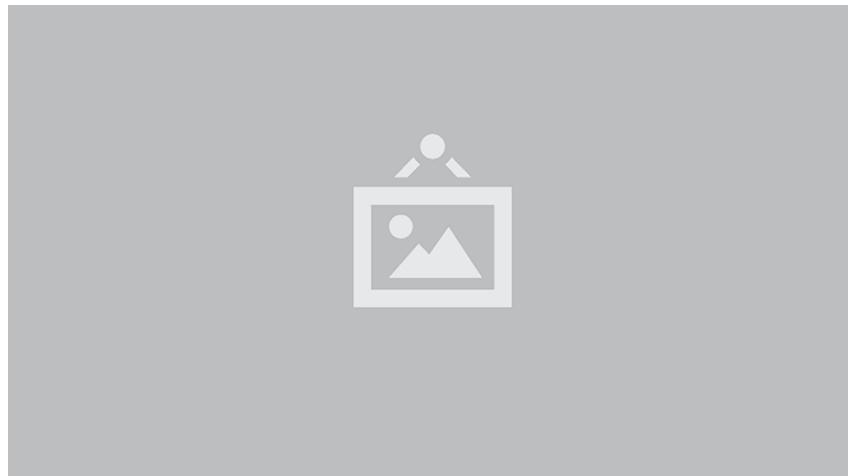
Zoals besproken in de inleiding draait de thesis om het renderen van drie dimensionale scenes in real-time. Het doel is hierbij veelal om geloofwaardige afbeeldingen te creeëren uit een bepaalde drie dimensionale scene beschrijving. In veel gevallen betekent dit dat de scene fotorealistisch afgebeeld dient te worden, echter andere stylistische keuzes zijn tevens mogelijk. Echter in alle gevallen 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 geïmplementeerd is.

2.1 Fysische werkelijkheid

De fysische wereld waarin de mens zich bevindt wordt gedierteerd 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 geïnterpreteerd. 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 interpreteren.

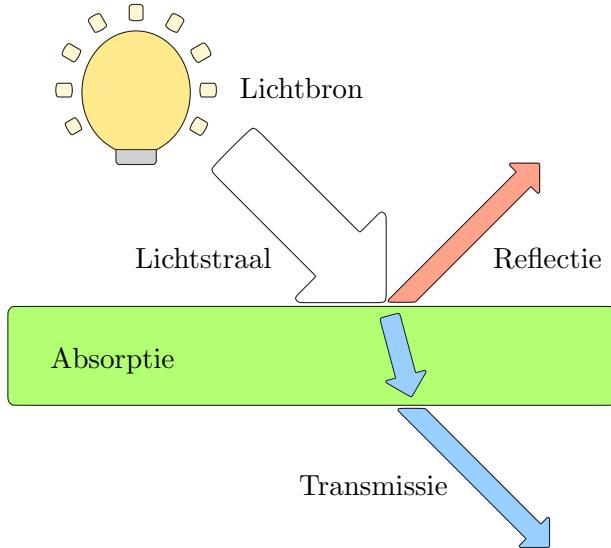
2.1.2 Licht

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 omstandigheden, 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 fenomenen 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 scène ingestuurd. De hoek van reflectie hangt



Figuur 2.2: Absorptie, reflectie en transmissie van licht.

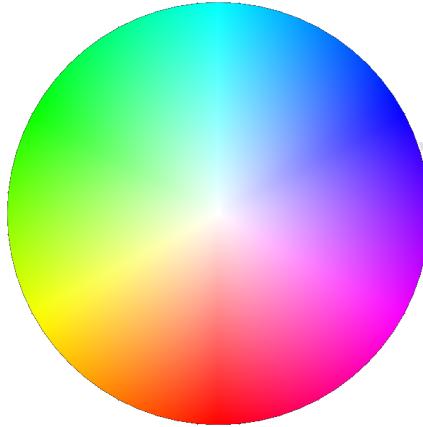
af van het type medium. Indien het materiaal zich gedraagt als een spiegel, en zal het licht teruggekaatst worden met dezelfde hoek gespiegeld om de normaal. Indien het materiaal licht diffuus weerspiegelt betekent dat de hoek van inval niet uitmaakt voor de reflectie en deze min of meer willekeurig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



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

De belangrijkste termen van Radiometrie zijn opgesteld in tabel ???. Binnen computer graphics is de belangrijkste eenheid radiantie. Dit is de flux per eenheid geprojecteerde oppervlakte per eenheid ruimtehoek. Radiantie meet de flux op een 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 binnentkomt, 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 interpreteren. De mens neemt slechts een gedeelte van al het licht waar. Dit wordt het zichtbare licht genoemd. Het menselijk oog interpreteert 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 gedraagt als additieve kleurmenging. Dit houdt in dat indien verschillende kleuren licht op hetzelfde 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

Type	Notatie	Voorbeelden
Hoek	Griekse kleine letters	θ
Scalar	Cursieve kleine letters	c
Vector of punt	Dikgedrukte kleine letters	\mathbf{p}
Matrix	Dikgedrukte Hoofdletters	\mathbf{M}
Functie	Cursieve (kleine) letters	f

Tabel 2.1: Wiskundige notaties.

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 scène 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 scène 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.

Operator	Notatie	Voorbeelden
klem	$\dots _{[a,b]}$	$x _{[0,1]}$

Tabel 2.2: Wiskundige operatoren.

2.2.1 Mathematische notaties

Binnen deze thesis zal veelvuldig gebruik gemaakt worden van verschillende mathematische notaties. Tabel 2.1 beschrijft de wiskundige notatie van de verschillende symbolen. Tevens zijn de meest gebruikte operatoren gegeven in tabel 2.2. In de volgende subsections zullen de belangrijkste mathematische concepten verder worden uitgewerkt.

Euclidische ruimtes

De standaard ruimte waarin binnen Computer Graphics gewerkt wordt is de Euclidische ruimte, genoteerd als \mathbb{R}^n . Een vector binnen deze ruimte is gedefinieerd als een geordende lijst bestaande uit n reële getallen:

$$\mathbf{v} \in \mathbb{R}^n \Leftrightarrow \mathbf{v} = \begin{pmatrix} v_0 \\ v_1 \\ \vdots \\ v_{n-1} \end{pmatrix} \text{ waar } v_i \in \mathbb{R}$$

Een vector beschrijft een richting en grootte relatief aan de oorsprong van het coördinatenstelsel. Een punt wordt tevens voorgesteld als een geoordende lijst bestaande uit n reële getallen:

$$\mathbf{p} \in \mathbb{R}^n \Leftrightarrow \mathbf{p} = \begin{pmatrix} p_0 \\ p_1 \\ \vdots \\ p_{n-1} \end{pmatrix} \text{ waar } p_i \in \mathbb{R}$$

Een punt beschrijft een positie in de ruimte \mathbb{R}^n waartoe deze behoort.

Zowel punten als vectoren kunnen worden gemanipuleerd met matrices. Een matrix is gedefinieerd als een blok van $p \times q$ reële getallen:

$$\mathbf{M} = \begin{bmatrix} m_{00} & m_{01} & \dots & m_{0,q-1} \\ m_{10} & m_{11} & \dots & m_{1,q-1} \\ \vdots & \vdots & \ddots & \vdots \\ m_{p-1,0} & m_{p-1,1} & \dots & m_{p-1,q-1} \end{bmatrix} = [m_{ij}]$$

Enkele veel voorkomende transformaties die voorgesteld kunnen worden als matrix zijn rotatie, schaling en scheer:

$$\begin{aligned} \text{Rotatie over hoek } \theta \quad & \mathbf{R}_x = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos \theta & -\sin \theta \\ 0 & \sin \theta & \cos \theta \end{bmatrix} \\ & \mathbf{R}_y = \begin{bmatrix} \cos \theta & 0 & -\sin \theta \\ 0 & 1 & 0 \\ \sin \theta & 0 & \cos \theta \end{bmatrix} \\ & \mathbf{R}_z = \begin{bmatrix} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \\ \text{Schaling met } s \quad & \mathbf{S} = \begin{bmatrix} s & 0 & 0 \\ 0 & s & 0 \\ 0 & 0 & s \end{bmatrix} \\ \text{Scheer met } \lambda \quad & \mathbf{H}_{0,2} = \begin{bmatrix} 1 & 0 & \lambda \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \end{aligned}$$

Matrices kunnen samengesteld worden met behulp van matrix vermenigvuldiging. Dit leidt er bijvoorbeeld toe dat de transformatie waarbij een vector \mathbf{v} eerst wordt geroteerd over de x -as:

$$\mathbf{v}' = \mathbf{R}_x \mathbf{v}$$

en vervolgens geschaald met een waarde s :

$$\mathbf{v}'' = \mathbf{S} \mathbf{v}'$$

gelijk is aan de samenstelling van de transformaties R_x en S .

$$\mathbf{v}'' = \mathbf{S}(\mathbf{R}_x \mathbf{v}) = (\mathbf{S}\mathbf{R}_x)\mathbf{v}$$

Homogene coordinaten

In theorie is het mogelijk om zowel punten en vectoren in de drie dimensionale Euclidische ruimte, \mathbb{R}^3 , voor te stellen als een 3-tupel van reële getallen. Echter binnen deze voorstelling is het niet mogelijk om de translatie transformatie als matrix voor te stellen. Dit heeft geen invloed op vectoren, gezien translatie niet gedefineerd is voor vectoren. Echter translaties hebben wel betekenis voor punten.

2. THEORIE

Om dit probleem op te lossen worden punten en vectoren in de drie dimensionale Euclidische ruimte \mathbb{R}^3 niet met drie maar met vier reële getallen voorgesteld. Deze voorstelling van coördinaten wordt het homogene coördinatenstelsel genoemd.

Een vector in homogene coördinaten wordt dan voorgesteld als:

$$\mathbf{v} \in \mathbb{R}^4 \Leftrightarrow \mathbf{v} = \begin{pmatrix} v_x \\ v_y \\ v_z \\ 0 \end{pmatrix} \text{ waar } v_i \in \mathbb{R}$$

Een punt in homogene coördinaten wordt voorgesteld als:

$$\mathbf{p} \in \mathbb{R}^4 \Leftrightarrow \mathbf{p} = \begin{pmatrix} p_x \\ p_y \\ p_z \\ 1 \end{pmatrix} \text{ waar } p_i \in \mathbb{R}$$

Verder worden de transformatie matrices uitgebreid van 3×3 elementen naar 4×4 elementen:

$$\mathbf{M}_{4 \times 4} = \begin{bmatrix} m_{00} & m_{01} & m_{02} & 0 \\ m_{10} & m_{11} & m_{12} & 0 \\ m_{20} & m_{21} & m_{22} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

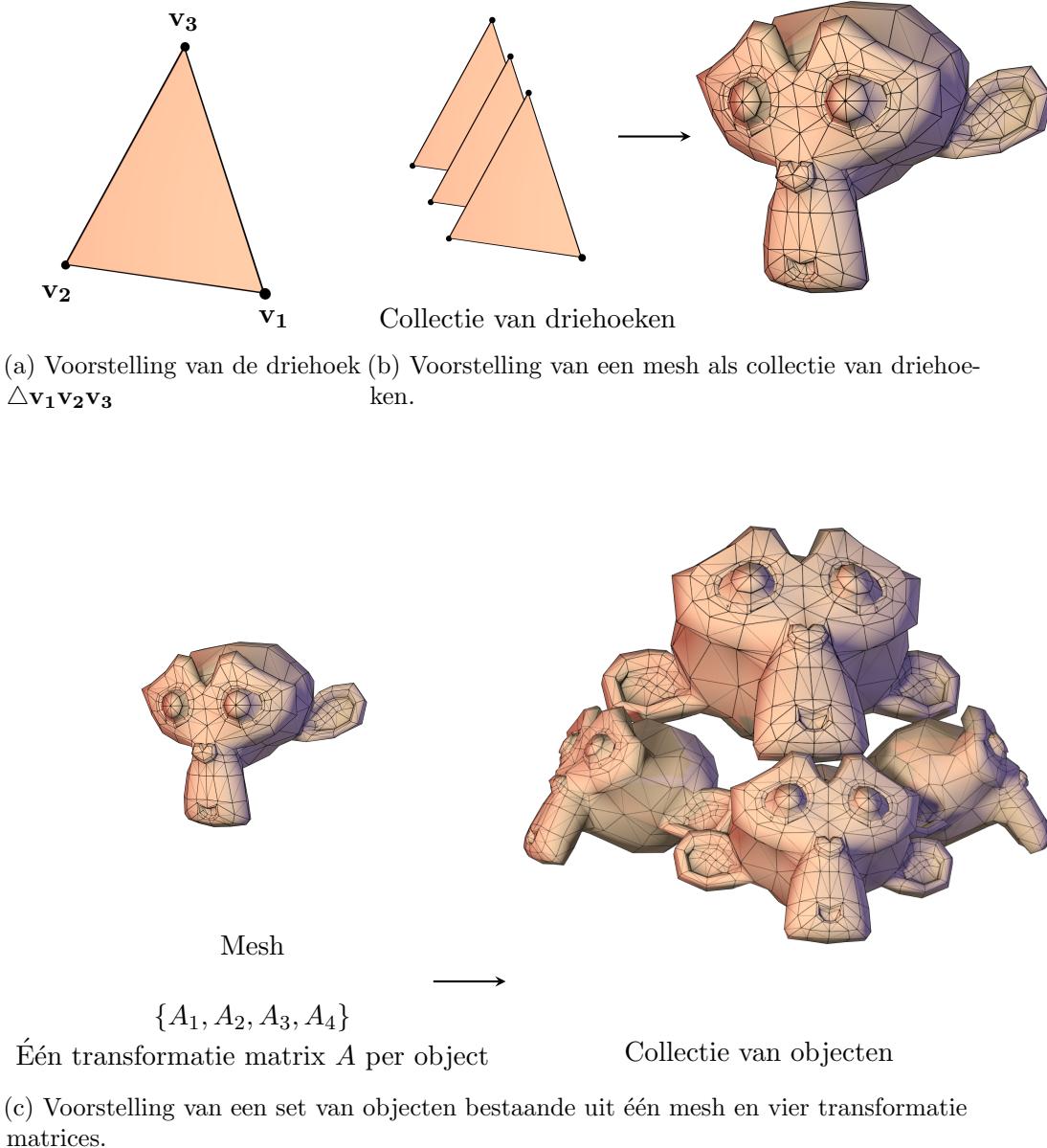
waarbij \mathbf{M} vervangen kan worden door de rotatie, schalings en scheer matrices om de equivalenten transformatie matrices in homogene coördinaten te verkrijgen.

Nu is het mogelijk om de translatie matrix te definiëren als:

$$\mathbf{T} = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Deze transformatie zal geen invloed hebben op vectoren, doordat $p_w = 0$, maar zal wel punten met \mathbf{t} translaten doordat $p_w = 1$.

Het is mogelijk dat voor p_w een waarde anders dan 0 of 1 gevonden wordt. In dat geval dient het punt gedeeld te worden door p_w om de correcte Euclidische coördinaten p_x , p_y en p_z te verkrijgen. Dit wordt homogenisatie genoemd.



Figuur 2.4: Voorstelling van objecten doormiddel van driehoeken.

2.2.2 Geometrische definities

Om een afbeelding te creeëren van een virtuele drie dimensionale omgeving, is het in de eerste plaats nodig om een beschrijving te hebben van deze omgeving. Om een omgeving voor te stellen wordt een set van primitieven gedefinieerd. Deze primitieven maken het mogelijk om een beschrijving te geven van elk object binnen de scene. De basis render-primitieven die gebruikt worden in grafische render-hardware zijn punten, lijnen en driehoeken. Een enkel punt kan beschreven worden in homogene coördinaten als:

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

Doormiddel van twee en drie punten kunnen respectievelijk lijnen en driehoeken voorgesteld worden. Een driehoek wordt genoteerd als:

$$\triangle \mathbf{v}_1 \mathbf{v}_2 \mathbf{v}_3$$

Indien gesproken wordt binnen deze thesis over primitieven, zal, indien niet anders aangegeven, gedoeld worden op driehoeken. Met behulp van driehoeken is het mogelijk om de geometrie van alle objecten in een scene te beschrijven. Dit proces is weergegeven in figuur 2.4. Door een collectie van verschillende driehoeken te nemen, wordt een *mesh* gevormd. Een *mesh* beschrijft het oppervlakte van een object, zoals weergegeven in fig. 2.4b. Vervolgens wordt een *object* gedefinieerd doormiddel van een referentie naar een *mesh* en een transformatiematrix *A*. Deze transformatiematrix beschrijft de locatie, schaal en rotatie van de *mesh* van het *object* binnen de drie dimensionale omgeving. Hierdoor is het mogelijk om meerdere objecten op verschillende plekken in de wereld voor te stellen met dezelfde *mesh* en een verschillende matrix. Een voorbeeld van een set van vier objecten met dezelfde *mesh* maar verschillende transformatiematrices is weergegeven in figuur 2.4c.

De volledige beschrijving van een omgeving zal een *scene* genoemd worden. Deze bevat de definities van de verschillende objecten binnen de wereld, als ook de lichten en eventuele cameras en gezichtspunten.

2.2.3 Camera model

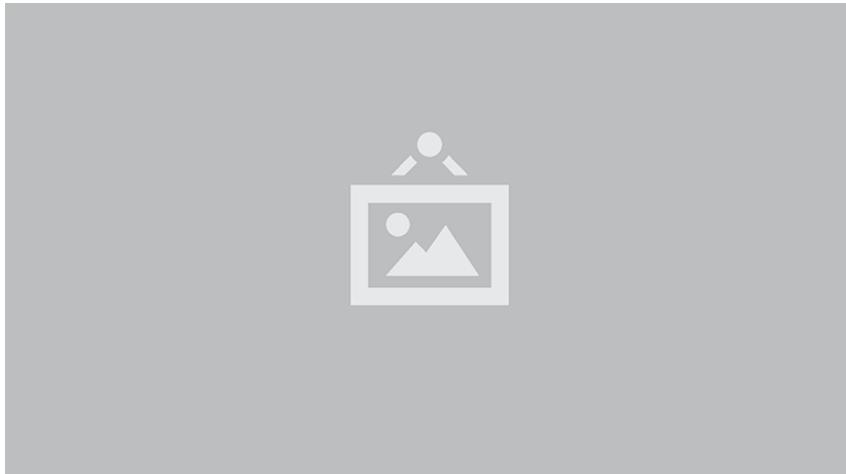
Het gezichtspunt, of *camera* binnen deze thesis is het standaard camera model binnen computer graphics. Zoals geïllustreerd in figuur 2.5.

De volgende vectoren en punten zijn hiervoor gedefinieerd

Up De locale y-as van de camera.

Eye (x, y, z) positie van de camera in wereld coördinaten.

Centre (x, y, z) positie waarnaar de camera kijkt



Figuur 2.5: Het standaard camera model.

Z-near de near z plane waarop het beeld wordt geprojecteerd.

Z-far Het vlak waarachter fragmenten niet meer worden weergegeven.

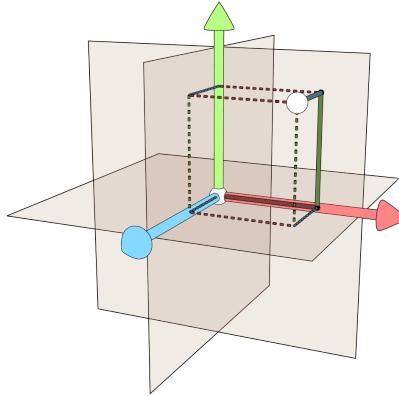
De *Z-near* en *Z-far* in combinatie met *Eye* creeert de view frustum. Slechts Primitieven binnen dit view frustum zullen worden gerenderd.

2.2.4 Coördinaten Stelsels

Zoals reeds vermeld in sectie 2.2.1 wordt een drie dimensionale ruimte \mathbb{R}^3 gebruikt om de scenes voor te stellen. Waarbij punten zich bevinden op positie \mathbf{p} . Voordat een dergelijk punt echter een betekenis heeft, dient een oorsprong en een set van drie onafhankelijke vectoren te worden gedefinieerd. Deze combinatie van oorsprong en basis wordt een coördinatenstelsel genoemd. Hiermee is het mogelijk om punt \mathbf{p} eenduidig vast te leggen in de Euclidische ruimte. De oorsprong beschrijft het nulpunt, waaraan alle punten relatief zijn. De drie onafhankelijke vectoren, of assen, definiëren vervolgens hoe de positie van punt \mathbf{p} relatief aan de oorsprong verkregen kan worden.

Binnen de meeste 3D pakketten wordt een Carthesisch rechtshandig coördinatenstelsel gebruikt. Een Carthesisch coördinatenstelsel is gedefinieerd aan de hand van drie loodrechte vlakken, waarbij een punt de positie op elke as beschrijft. Dit is weergegeven in figuur 2.6 De orientatie van een coördinatenstelsel is afhankelijk van de richting en volgorde van de assen. Binnen deze thesis wordt altijd gebruik gemaakt van rechtshandige coördinatenstelsels, zodanig dat de y -as omhoog wijst, en de x -as naar rechts. De negatieve z -as wijst vervolgens voorwaarts. Positieve rotaties zijn in dit geval tegen-de-klok-in.

Een belangrijke observatie is dat het mogelijk is om van coördinatenstelsel te veranderen door transformatie en projectie matrices toe te passen op de basis en oorsprong, en dus op alle objecten binnen een bepaalde ruimte. Hierdoor is het



Figuur 2.6: Rechtshanding Carthesisch coördinatenstelsel.

mogelijk om verschillende coördinatenstelsels te definiëren. Om een scene op te stellen worden de volgende coördinatenstelsels gebruikt.

Per mesh wordt gebruik gemaakt van een *modelruimte* om de posities van de vertices en daarmee dus de triangles te definiëren. Dit referentiekader is onafhankelijk van andere meshes.

Zoals eerder vermeld in sectie 2.2.2 is voor elk object een aparte transformatiematrix opgesteld. Hiermee wordt de geassocieerde mesh omgezet van *modelcoördinaten* naar *wereldcoördinaten*. Met 'e'en scene is 'e'en *wereldruimte* geassocieerd. Dit coördinatenstelsel beschrijft de onderlinge relatie tussen objecten, lichten, en cameras die zich bevinden in de scene.

Om de wiskunde van het renderen te vergemakkelijken wordt tevens nog een cameraruimte opgesteld, waarbij de scene zodanig wordt getransformeerd dat de oorsprong overeenkomt met de positie van de camera. De negatieve z -as komt overeen met de kijkrichting, en de x - en y -assen komen overeen met de viewport van de camera. Deze transformatie wordt bereikt met behulp van de LookAt matrix, die verder zal worden toegelicht in de volgende secties.

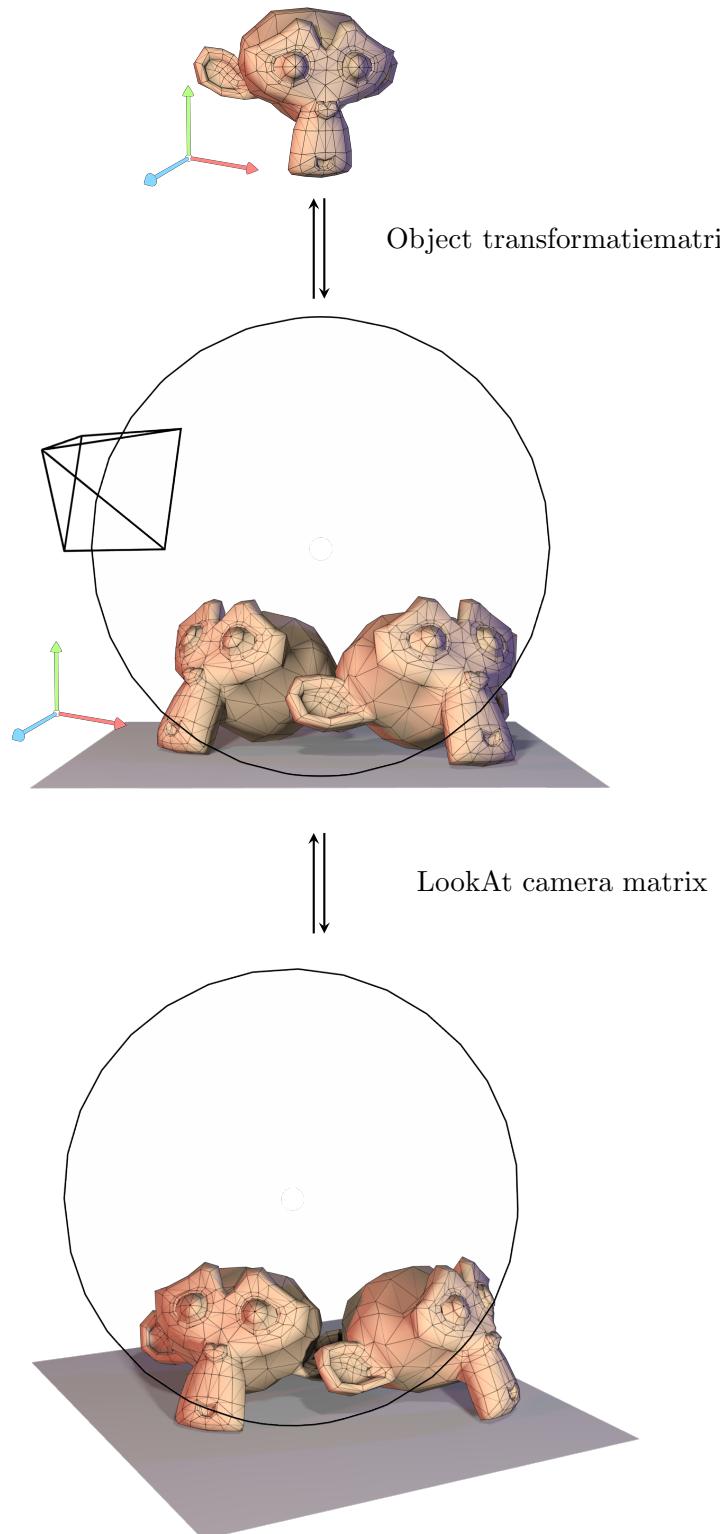
In de laatste stap voordat het renderalgoritme plaatsvindt, dient de cameraruimte nog omgezet te worden naar Normalised Device Coordinates (NDC). Dit is een tweedimensionaal coördinatenstelsel die de afbeelding van de scene op de viewport beschrijft in coördinaten van -1 tot 1 . Ook dit zal verder toegelicht worden in de volgende secties.

De onderlinge relatie van de verschillende ruimtes, en hoe deze in elkaar om te zetten zijn, is weergegeven in figuur 2.7.

2.3 Perspectief projectie en het visibleitsprobleem

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

2.3. Perspectief projectie en het visibiliteitsprobleem



Figuur 2.7: Verschillende Carthesische coördinatenstelsels en bijbehorende transformaties.



Figuur 2.8: Perspectief projectie.

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.

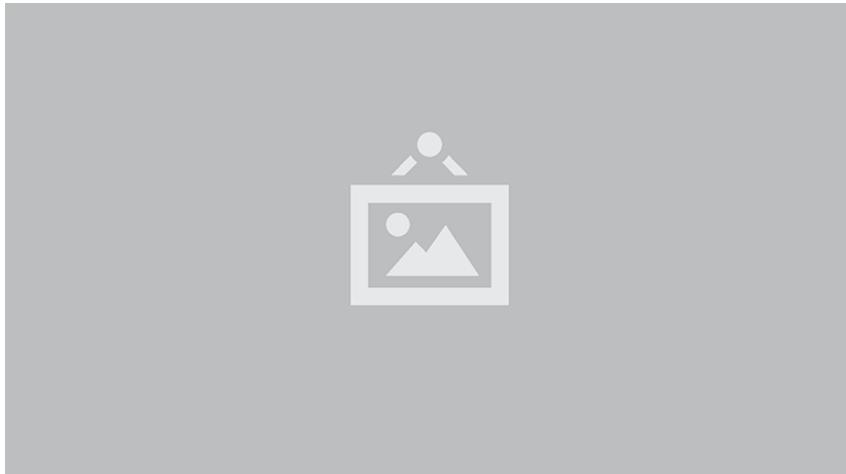
Deze effecten kunnen gesimuleerd worden door de 3d scene te projecteren naar het oogpunt en af te beelden op het canvas. Zoals weergegeven in figuur 2.8.

Zoals eerder beschreven bestaan de objecten binnen de scenes uit meshes van primitieven. Omdat elk van deze driehoeken gedefinieerd 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 2.9 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 \mathbf{C} en $\mathbf{AB'C'}$ gelijk is. Dit betekent dat we het punt \mathbf{C}' kunnen berekenen 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.



Figuur 2.9: Projectie van een enkel punt.

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

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

$$p'_z = d$$

$$p'_w = 1$$

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 coördinaten stelsel omzet naar een punt geprojecteerd op de canvas. Wat leidt tot de perspectief projectie van punten.

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

Of deze projectie daadwerkelijk nodig is, is afhankelijk van de gekozen rendering techniek. Binnen rasterisatie is het nodig om deze stap explicet 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 gaan zijn op het canvas, zoals weergegeven in figuur 2.10. 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

2. THEORIE



Figuur 2.10: Visibiliteitsprobleem in een scène met meerdere primitieven.

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.

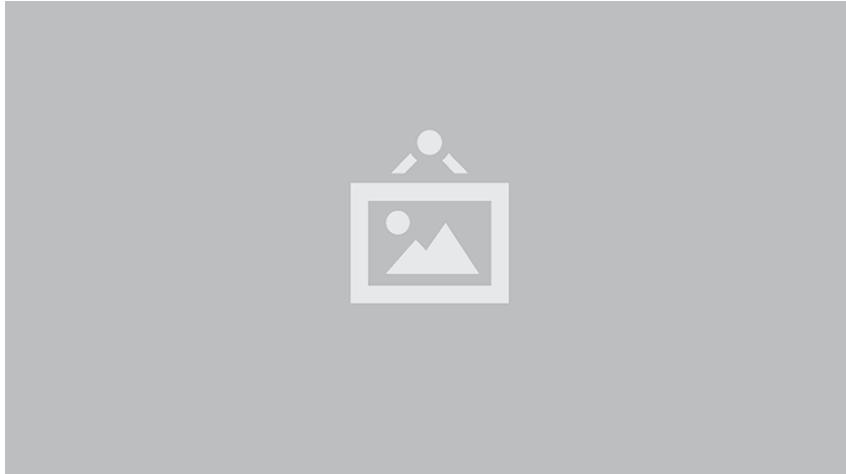
De algoritmes om dit efficient te doen worden verborgen oppervlakte bepalings (hidden surface determination) algoritmes genoemd. Deze kunnen grofweg ingedeeld worden in twee categorien, raytracing en rasterisatie. Hierbij zou, in theorie, geen verschil in resultaat hoeven zijn. Beide klassen van algoritmes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

Raytracing werkt op basis van het trekken van zogenoemde stralen door \mathbf{p}' waarbij de eerste \mathbf{p} wordt bepaald. Rasterisation daarentegen, bepaalt 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 oog, 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 eenzelfde 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 geïllustreerd in figuur 2.11. Deze techniek wordt forwaards tracen genoemd. Echter vaak is het



Figuur 2.11: Forwaards raytracen.

canvas van een camera velen malen kleiner dan de scène 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.

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

```



Figuur 2.12: Raytrace algoritme.

```
do_shading(closest, ray)
```

Waarbij `do_shading` gebruikt wordt om de kleur te bepalen van de specifieke pixel. Dit is verder geïllustreerd in figuur 2.12

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.

2.3.4 Rasterisatie

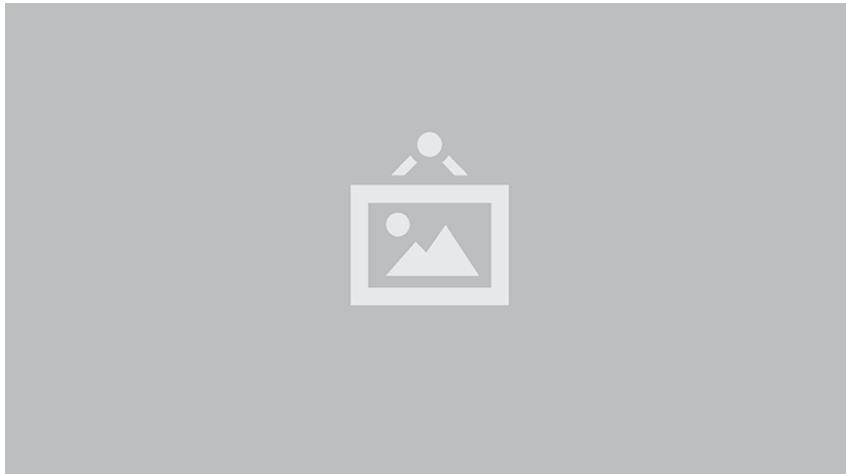
Rasterisatie algoritmes lossen perspectief projectie en het visibiliteitsprobleem op een verschillende volgorde op dan hoe het binnen raytracing algoritmes wordt opgelost. Waar raytracing uitgaat van het punt \mathbf{p}' en kijkt welk object hier op valt, begint een rasterisatie algoritme met het afbeelden 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 beïnvloedt. 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 2.13.

2.3. Perspectief projectie en het visibiliteitsprobleem



Figuur 2.13: Het rasterisatie algoritme.

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 sorteert probleem, waarbij objecten geordend dienen te worden ten opzichte van de kijkas, de \mathbf{z} -as. Om het zichtbare object binnen een punt \mathbf{p}' te bepalen, dient dus bepaald te worden welk object de kleinste \mathbf{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 \mathbf{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 \mathbf{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.



Figuur 2.14: Uitstraling van radiantie over ω_o vanuit \mathbf{p}

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 scène 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 2.1. 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 de stabiele toestand van de stralingsenergie balans binnen een scène. 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:

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

Dit wordt de reflectie vergelijking genoemd. Hierbij wordt geïntegreerd over de gehele hemisfeer om de volledige binnengesloten 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 ω_o . Dit is verder geïllustreerd in fig. 2.14.

De BRDF in essentie is de wiskundige functie die beschrijft hoe een materiaal zich gedraagt 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 binnengkomend over de gehele hemisfeer in punt \mathbf{p} . De binnengkomende 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 scène al van te voren te weten. Dit is niet mogelijk, en dus zullen alle shading algoritmes pogingen 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 binnengkomende straal licht wordt gereflecteerd puur willekeurig is. Dit is weergegeven in fig. 2.15. Deze BRDF heeft als uitkomst een constante waarde. Deze constante waarde wordt veelal gedefinieerd als de de *diffuse kleur* c_{dif} van dit object. Dit leidt tot de volgende functie:

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



Figuur 2.15: Lambertiaanse BRDF.

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 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 \mathbf{p} binnen de scène.

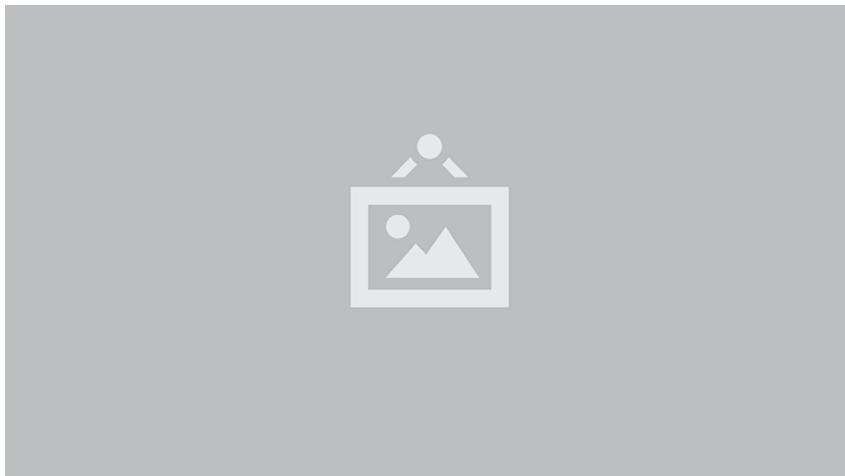
In de fysische wereld zijn lichten nooit eindig, echter de invloed die ze hebben op de omringende wereld zal bij grotere afstand 0 benaderen. Binnen de fysische wereld is dit een gevolg van absorptie door het medium waardoor het licht zich beweegt. Dit proces wordt afstands demping genoemd. Binnen de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

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

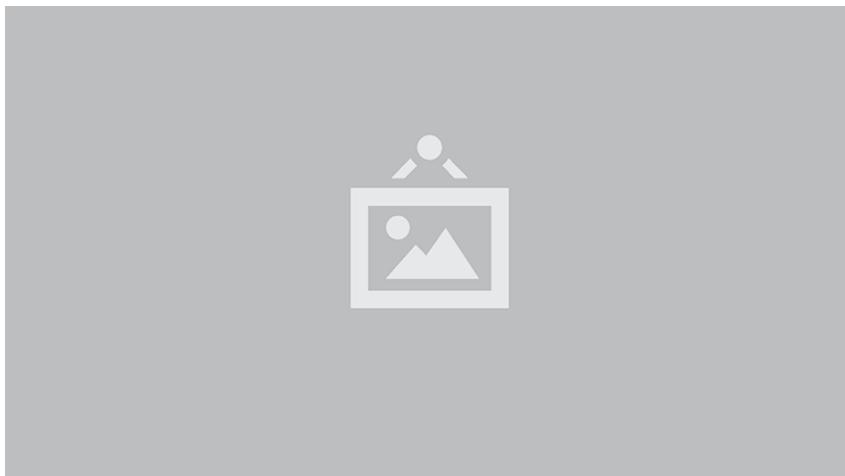
waar l de padlengte van de straal licht door het medium is en μ de dempingscoefficient is.

Hierbij wordt de demping van het licht gerelateerd aan het medium waardoor het zich beweegt. Dit leidt tot afstandsdempings (distance attenuation) curves zoals weergegeven in figuur 2.16.

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



Figuur 2.16: Afstandsdempings curves.



Figuur 2.17: Voorstelling van licht.

Dit leidt tot een voorstelling als weergegeven in figuur 2.17.

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 2.18

Binnen de thesis zelf is gekozen voor de benadering:

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

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



Figuur 2.18: Afstandsdempings curves voor eindige lichtbronnen.

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 coördinatenstelsel met oorsprong O
- Een afstand r die de invloed van de lichtbron bepaalt
- Een afstandsdempingsfunctie f die het verval van invloed moduleert
- Een intensiteit \mathbf{i} die de kleur en kracht van de lichtbron bepaald

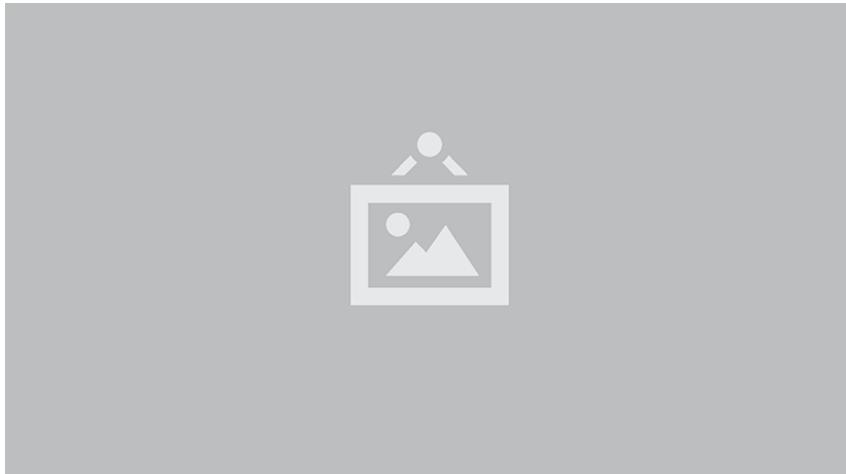
2.5 Moderne Grafische Pijplijn

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.



Figuur 2.19: De logische onderverdeling van de Moderne Grafische Pijplijn.

- Applicatie stap
- Geometrie stap
- Rasteriser stap

Deze zijn tevens weergegeven in figuur 2.19.

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 onderverdeeld worden zoals weergegeven in figuur 2.20. De volgende sub-stappen kunnen onderscheiden 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.

2. THEORIE



Figuur 2.20: De logische onderverdeling van geometrie stap.

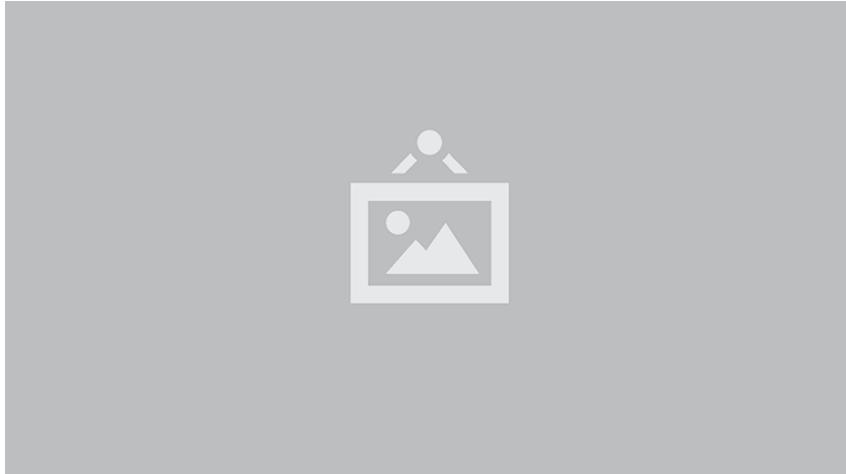
De geometrie stap zorgt dat objecten geproduceerd door de applicatie stap, omgezet worden naar een set van data die in de rasterisatie stap omgezet kan worden in een daadwerkelijke afbeelding. Eerst worden objecten getransformeerd zodanig dat alle primitieven zich in hetzelfde coördinaten systeem bevinden. 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 die over zijn omgezet naar canvas coördinaten. De set van primitieven in canvas coördinaten, 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 2.21. De volgende sub-stappen kunnen onderscheiden 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



Figuur 2.21: De logische onderverdeling van rasterisatie stap.

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 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 geïntroduceerd in `openGL`. Indien een nieuwe term geïntroduceerd 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 configurerbaar. Een overzicht van de verschillende stappen van de pijplijn voor respectievelijk `openGL` en `Direct3D` zijn gegeven in figuur 2.22a en 2.22b. Door middel van kleuren is de mate van programmeerbaarheid, aangegeven. Tevens zijn optionele stappen aangegeven door een stippe lijn. In beide APIs zijn 9 stappen terug te vinden.

- Vertex Specificatie
- Vertex Shader
- Tesselatie
- Geometrie shader
- Vertex Post-Processing
- Primitieven assemblage
- Rasterisatie

2. THEORIE



(a) openGL



(b) Direct3D

Figuur 2.22: De stappen van zowel de openGL als Direct3D implementaties.

- 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 specificert 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 geinterpolateerd 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 gespecificeerd 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 coördinaten 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 waarden 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.

2. THEORIE

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 Probleemstelling

2.7 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 creëren. 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 refereren naar de volgende boeken. Voor de psychologische grondslag van kleur en perceptie, kan gerefereerd worden naar Wolfe's *Sensation and Perception* ([32]). De fysica die ten grondslag ligt aan computer graphics kan gevonden worden in *Optics* door Eugene Hecht ([13]). Meer toegepast op computer graphics, en tevens voor de behandeling van shading en raytracing, zijn de boeken *Raytracing from the ground up* ([27]), en *Physical Based Rendering* ([24]), 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

Implementatie overzicht

Voordat ingegaan wordt op de specifieke technieken die geïmplementeerd zijn, zal eerst een overzicht gegeven worden van de geïmplementeerde functionaliteit en de gebruikte test scenes. Deze sectie beschrijft de opzet van het programma dat geïmplementeerd is om deze testen uit te voeren. Eerst zal een overzicht worden gegeven van de opbouw van het programma en de geïmplementeerde functionaliteit. Vervolgens zal gekeken worden naar de meet methodes en zullen de test scenes behandeld worden.

3.1 Overzicht

`nTiled` is de software geschreven om de verschillende lichtmethoden te testen. Het is geschreven in C++ en gebouwd op de volgende libraries.

`openGL 4.4` `openGL` met behulp van `GLAD`¹ zorgt voor de rendering pijplijn.

`glfw` `glfw` is de window manager, verantwoordelijk voor het aan maken van de applicatie in het besturingssysteem.²

`assimp` `assimp` is gebruikt om de geometrie objecten in `nTiled` te laden.³

`glm` `glm` is een mathematische library gebruikt voor de vector en matrix berekeningen aan de C++ kant.⁴

`rapidjson` `rapidjson` wordt gebruikt om de `json` configuratie bestanden in te lezen.
⁵

`dear, imgui dear, imgui` is de GUI libary.⁶

Het programma zelf kan grofweg worden ingedeeld in de render functionaliteit die getest wordt binnen deze thesis, en zeven ondersteunende modules:

¹glad project pagina: github.com/Dav1dde/glad

²glfw website: www.glfw.org

³assimp project pagina: github.com/assimp/assimp

⁴glm website: glm.g-truc.net/0.9.8/index.html

⁵rapidjson project pagina: github.com/miloyip/rapidjson

⁶dear, imgui project pagina: github.com/ocornut/imgui

3. IMPLEMENTATIE OVERZICHT

Camera De camera implementeert het camera model zoals beschreven in het theorie hoofdstuk. Hierbij is het mogelijk om data opgeslagen in `json` bestanden in te lezen.

GUI De GUI implementeert de gebruikers interface met behulp van `dear`, `imgui`
log De log module implementeert de logging capiciteiten.

main Implementeert de functionaliteit gerelateerd aan het opstarten en sluiten van het programma.

math In math zijn de extra wiskundige functies geimplementeerd die gebruikt worden binnen `nTiled`.

state State bevat de staat nodig voor een enkele uitvoering van `nTiled`. Tevens bevat het de functionaliteit om alle configuratie bestanden in te lezen.

world Implementeert de functionaliteit om de geometrie en lichten in te lezen, en te beheren binnen een uitvoering van `nTiled`

De pipeline module implementeert de rendering pipeline. Hierin wordt zowel de voorwaartse als de deferred pijplijn geimplementeerd, in respectief de `forward` en `deferred` module. Tevens bevindt zich hierin de implementatie van de drie belichtings methodes in de module `light-management`.

Voor beide types pijplijnen zijn de volgende shaders geimplementeerd:

Geattenueerd De shaders zonder versnellingsstructuren.

Tiled De shaders die de tiled versnellingsstructuur bevatten.

Clustered De shaders die de clustered versnellingsstructuur bevatten.

Hashed De shaders die de hashed versnellingsstructuur bevatten zoals voorgesteld in deze thesis.

Elke shader bestaat uit een C++ implementatie en corresponderende `glsl` bestanden. Voor een compleet overzicht van de implementatie zie de documentatie ⁷ en repository van `nTiled`.⁸

3.2 Shading: implementatie

Alle shading binnen deze thesis is uitgevoerd met dezelfde rendering functie. Deze functie wordt berekend voor alle relevante lichten binnen de fragment shader. De gebruikte rendering functie is een simpele direchte licht benadering van een Lamber-tiaans wit oppervlakte, zoals beschreven in het theorie hoofdstuk. Deze is als volgt gedefinieerd:

```
struct Light {  
    vec4 position;  
    vec3 intensity;  
    float radius;  
};
```

⁷de `nTiled` documentatie kan gevonden worden op ntiled.readthedocs.io/en/latest/index.html

⁸de `nTiled` repository kan gevonden worden op github.com/BeardedPlatypus/nTiled

```

struct GeometryParam {
    vec4 position;
    vec3 normal;
    vec3 colour;
};

vec3 computeLight(Light light,
                  GeometryParam param) {
    vec3 light_direction =
        vec3(light.position - param.position);
    float d = length(L);
    light_direction /= d;

    float attenuation =
        clamp(1.0 - (d / light.radius), 0.0f, 1.0f);
    attenuation *= attenuation;

    float cos_angular_incidence =
        clamp(dot(param.normal, light_direction), 0.0f, 1.0f);
    return (param.colour * light.intensity *
            cos_angular_incidence * attenuation);
}

```

Deze functie komt overeen met de volgende formule

$$L(l_i, \mathbf{p}) = c_{\mathbf{p}} * I_i * \cos \theta * f_{\text{att}}$$

waar

$$f_{\text{att}} = \left(1 - \frac{d}{r_i}\right)_{0,1}$$

Om de volledige kleur van een fragment te berekenen dient deze functie uitgevoerd te worden voor elk licht dat invloed heeft op dit fragment. Hiervoor is het nodig om de positie en de normaal van het punt te hebben. Deze worden doorgegeven vanuit de vertex shader.

Dit alles gecombineerd leidt tot oppervlaktes zoals weergegeven in figuur 3.1.

3.3 Meet methodes

Alle testen zijn uitgevoerd op dezelfde hardware onder zelfde omstandigheden. Voor alle testen zijn de executie tijden van individuele functies binnen elke lichttoekennings methode, gemeten. Hiervoor is gebruik gemaakt van de `QueryPerformanceCounter`,

3. IMPLEMENTATIE OVERZICHT



Figuur 3.1: Een oppervlakte gerenderd met de standaard lambert shader binnen nTiled.

functionaliteit aangeboden binnen `Windows.h`. Hierdoor is het mogelijk om waardes te rapporteren in de orde van grootte van $\mu.s$.

Elke test is drie maal uitgevoerd, indien niet anders vermeld. De gerapporteerde waardes zijn gemiddelden.⁹.

3.3.1 Hardware

Alle testen zijn uitgevoerd op de hardware in de volgende tabel:

Besturingssysteem	Windows 10 64-bit
Processor	Inter Core i7 6700 HQ @ 2.60 Ghz
Geheugen	16 GB
Grafische kaart	NVIDIA GeForce GTX 960M
Grafische kaart drivers	372.70

3.4 Test suite

De test suite bestaat uit 3 verschillende scenes waarvoor, voor elke scene, geometrie, een camera pad, en verschillende lichtconfiguraties zijn gedefinieerd. Elk van de testen die volgt zijn uitgevoerd met deze scenes.

De gedefinieerde scenes zijn:

- Ruimteschip: indoor scene

⁹De rauwe data kan gevonden worden in de repository: github.com/BeardedPlatypus/thesis-data-suite.

- Piper's Alley: straatzicht
- Ziggurat stad: stads scene

Deze zijn geselecteerd zodanig dat deze scenes representatief zijn voor de verschillende schalen die voor kunnen komen binnen games. Een afgesloten indoor ruimte , een open lucht straat met details in de diepte, en een grote open lucht scene. Door het gebruik van deze scenes is het mogelijk om de performantie van de verschillende rendertechnieken tevens te vergelijken op het niveau van verschillende scenes.

Voor elk van de scenes zijn verschillende licht configuraties gedefinieerd. Elk van de lichten zijn statisch, met eenzelfde radius, indien niet anders aangegeven. De lichten zelf zijn uniform over de scenes verdeeld aan de hand van licht volumes waarin een gespecificeerd aantal lichten worden geplaatst.

3.4.1 Indoor: Spaceship

Het indoor spaceship scene is gebaseerd op de CG lighting challenge #18, gemodelleerd door Juan Carlos Silva.¹⁰ Deze scene staat model voor indoor game scenes, waarbij visibiliteit beperkt is. De scene bestaat uit ... tussen stukken verbonden met hoekstukken en 1 centraal midden. De grootste bron van details komt uit de panelen aan weerszijden van elk tussenstuk. De scene bestaat uit ... polygonen, de verdeling van deze polygonen is gegeven in figuur 3.2c. Het camera pad zoals weergegeven in figuur 3.2b bestaat uit ... frames. Voor elk component in de scene is een lichtvolume gedefinieerd waarin de lichten zijn gegenereerd. Dit is weergegeven in figuur 3.2d.

3.4.2 Straatzicht: Piper's Alley

De piper's alley scene is gebaseerd op de CG lighting challenge #42, gemodelleerd door Clint Rodrigues.¹¹ Deze scene beschrijft een enkele straat, met grote diepte, waar aan weerszijden huizen zijn geplaatst. Er zijn totaal ... polygonen binnend de scène, de verdeling hier van is te zien in figuur 3.3c. De camera beweegt over deze straat gedurende ... frames. De lichtvolumes zijn geplaatst over de straat zoals weergegeven in figuur 3.3d.

3.4.3 Stadsscene: Ziggoerat

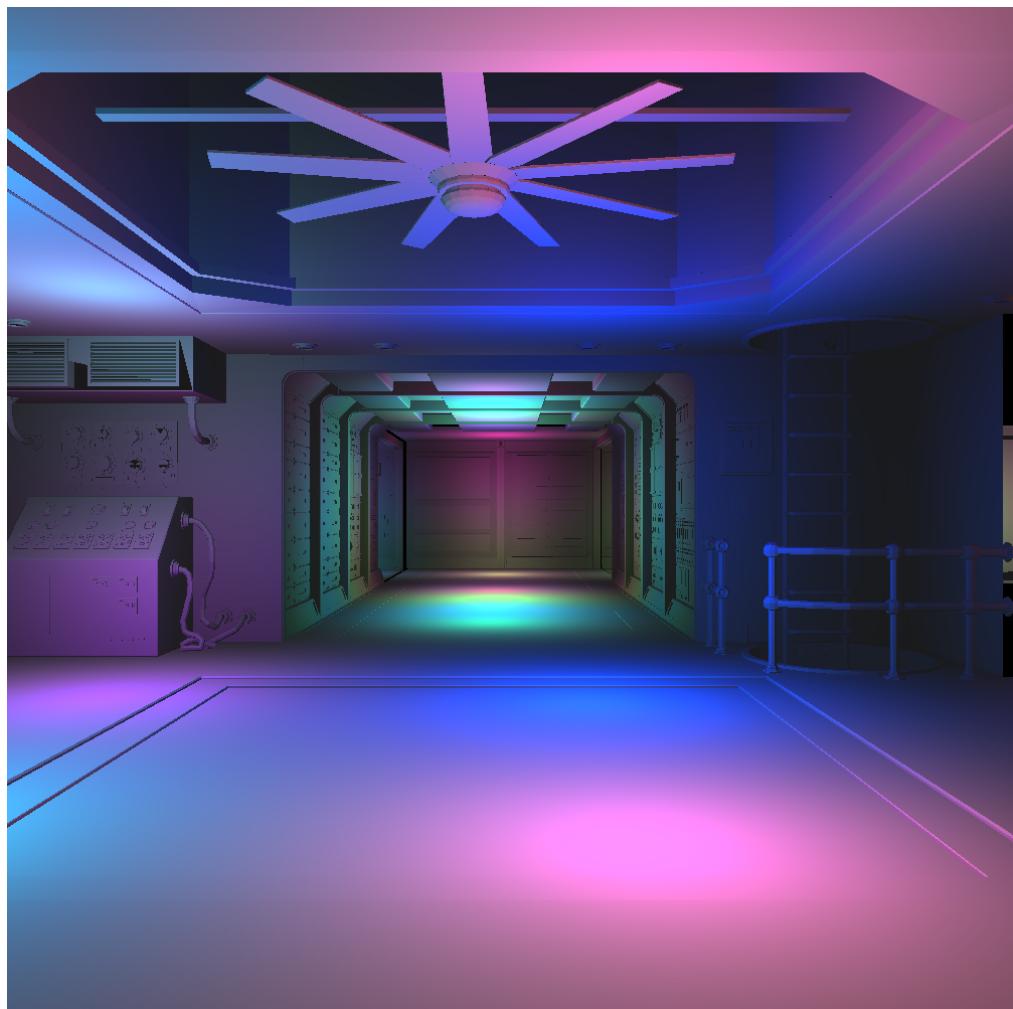
De ziggoerat stadsscene is gemodelleerd als onderdeel van de open film sintel.¹² De scene bestaat uit een gedetailleerde ziggoerat, waaromheen in lagere resolutie huizen en een stadsmuur liggen. De scene bestaat uit totaal 1,6 miljoen polygonen, de distributie is te zien in figuur 3.4c. De camera beweegt eerst omhoog de ziggoerat, om vervolgens over de stad heen te pannen. Dit gebeurt over 460 frames. Het pad is te zien in figuur 3.4b. De lichten zijn onderverdeeld in twee sets, de lichten op de ziggoerat zijn kleiner en met een hogere dichtheid geplaatst, ten opzichte van de

¹⁰Scene url: www.3drender.com/challenges/

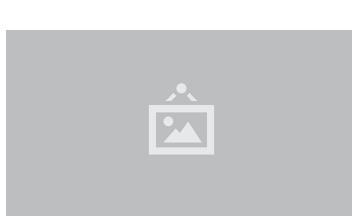
¹¹Lighting challenge url: forums.cgsociety.org/archive/index.php?t-1309021.html

¹²Open film url: durian.blender.org

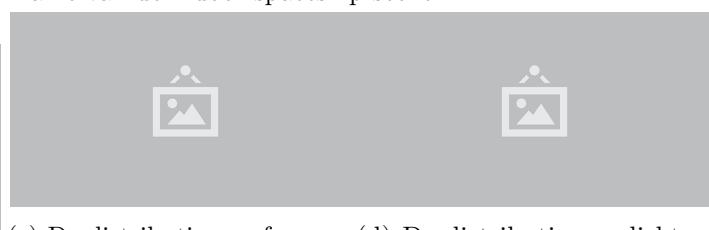
3. IMPLEMENTATIE OVERZICHT



(a) Een frame van de indoor spaceship scene.

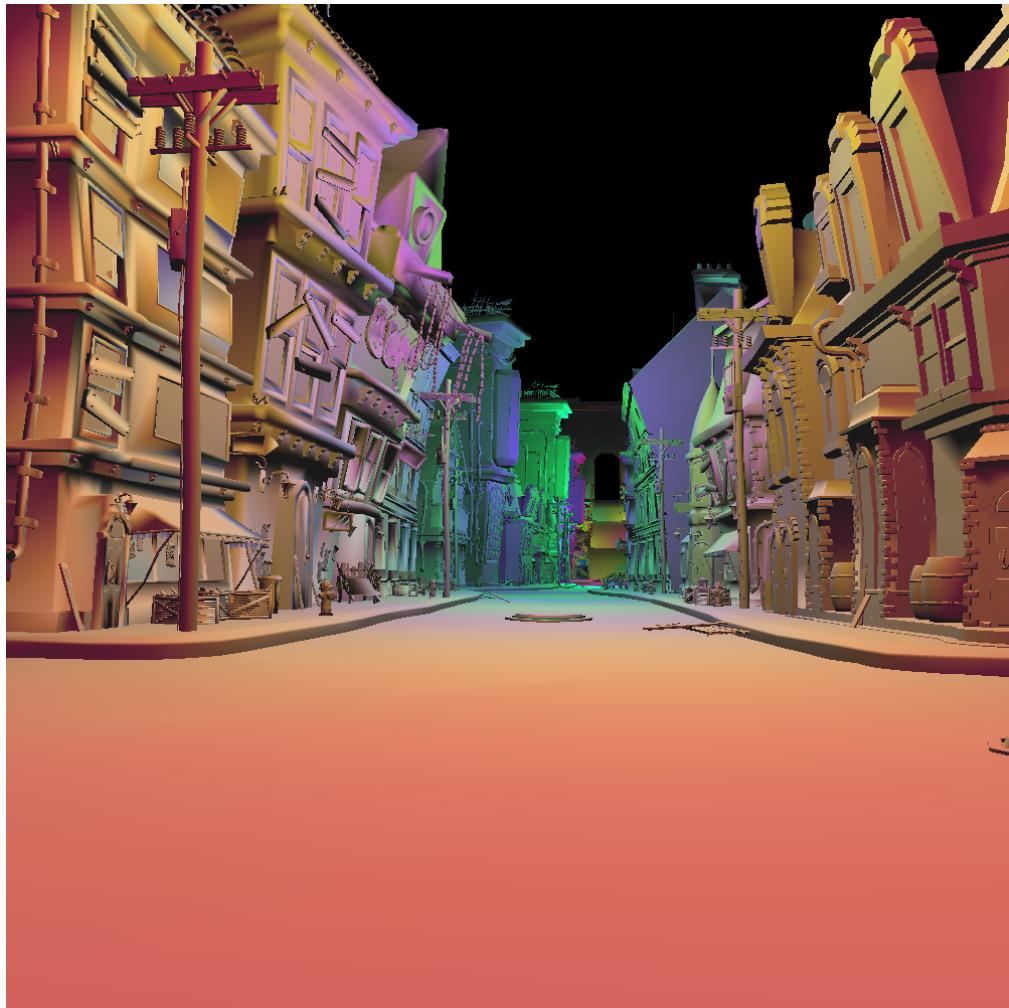


(b) Het camera pad in de indoor spaceship scene.

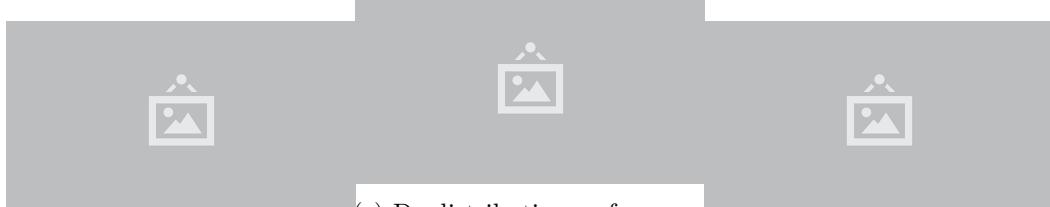


(c) De distributie van fragmen(d) De distributie van lichten
ten binnenden de indoor spaceship scene.
binnen de indoor spaceship scene.

Figuur 3.2: Grafisch overzicht van de indoor spaceship scene.



(a) Een frame van de piper's alley scene.



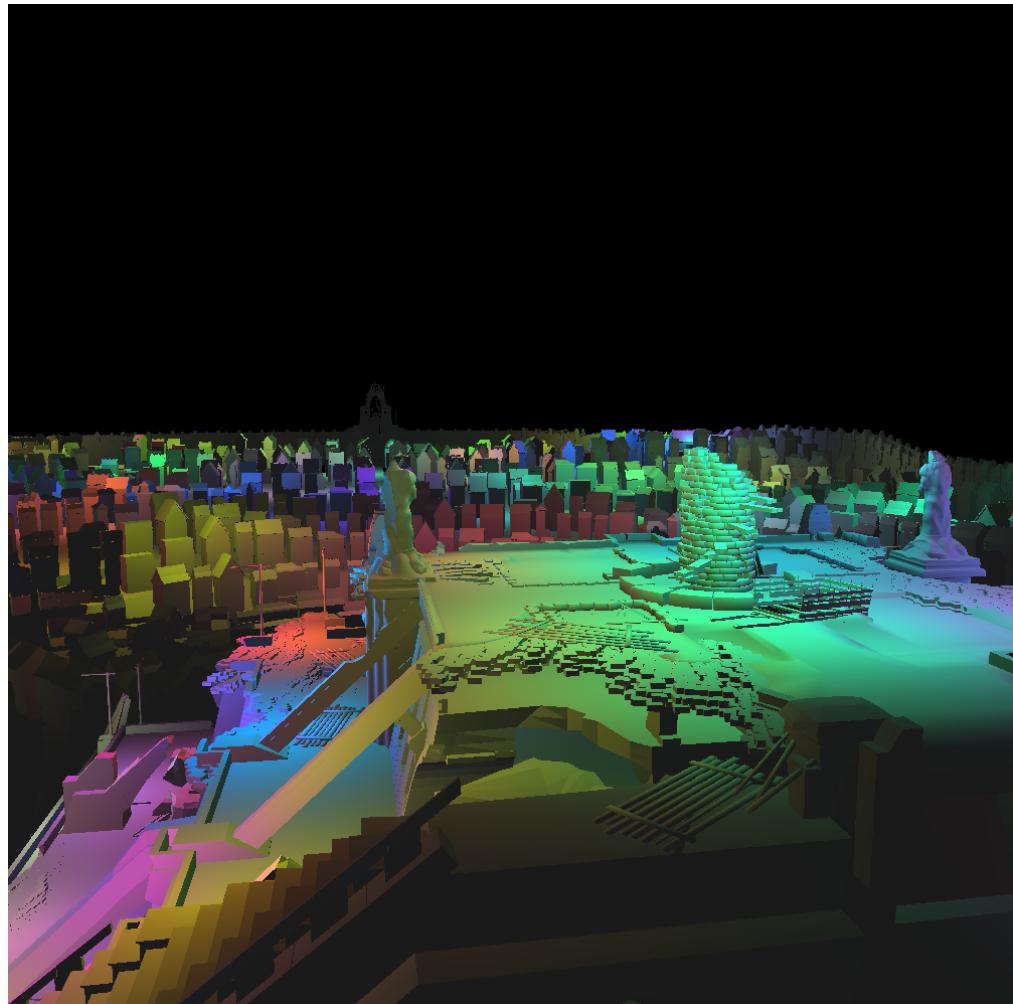
(c) De distributie van fragmen-

(b) Het camera pad in de pi-ten binnen de piper's alley scene.

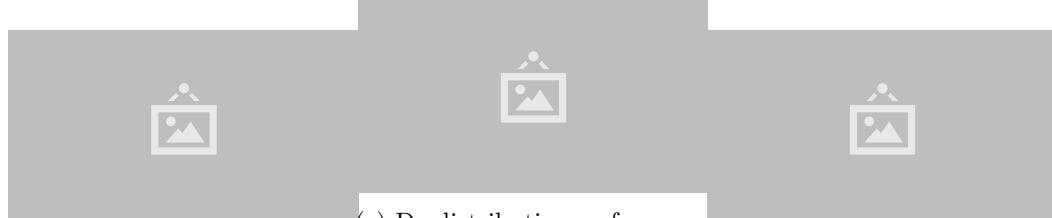
(d) De distributie van lichten binnnen de piper's alley scene.

Figuur 3.3: De verschillende aspecten van de piper's alley scene.

3. IMPLEMENTATIE OVERZICHT



(a) Een frame van de ziggurat stadsscene.



(c) De distributie van fragmen-

(b) Het camera pad in de zig-ten binnen de ziggurat stads-

(d) De distributie van lichten
scene.

binnen de ziggurat stadsscene.

Figuur 3.4: De verschillende afbeeldingen van de ziggurat stadsscene.

3.4. Test suite

lichten die geplaatst zijn over de stad. De lichtvolumes zijn weergegeven in figuur 3.4d.

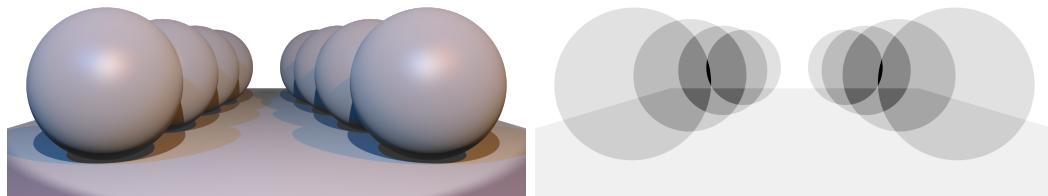
Hoofdstuk 4

Deferred Shading

Binnen sectie 2.5 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 niet zichtbaar zijn in de gerenderde frame de belichtingsberekening wordt uitgevoerd. De shading complexiteit is dus direct gekoppeld aan de geometrische complexiteit van de scène. Voor simple scenes is dit geen probleem. Echter wanneer scenes complexer worden, leidt dit tot verspilde rekenkracht. Een simpel voorbeeld van een dergelijke scène, waar de belichtingsberekening onnodig wordt uitgevoerd is weergegeven in figuur 4.1. Elk van de bollen creeert fragmenten die niet zichtbaar zijn. Het aantal fragmenten dat per pixel gecreeëerd wordt, is visueel weergegeven in de warmtekaart, fig. 4.1b.

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 en een renderstap, waar de informatie van de zichtbare fragmenten opnieuw wordt ingelezen, en de belichtingsberekening wordt uitgevoerd.

In de volgende secties zal eerst de theorie toegelicht worden, vervolgens zal ingegaan worden op het algoritme en de implementatie binnen `nTiled`. Als laatste zal de effectiviteit behandeld worden aan de hand van uitgevoerde testen.



(a) Rendering van de scène.

(b) Warmtekaart van de fragmenten.

Figuur 4.1: Een scène met een grote hoeveelheid van verborgen geometrie.

4.1 Theorie

De afhankelijkheid tussen geometrische complexiteit en de complexiteit van de belichtingsberekening werd al in de begin jaren van computer graphics herkend als probleem. De opsplitsing van visibiliteit en belichtingsberekeningen werd toen al voorgesteld. Zo werd in de scan-line polygon renderer van Watkins ([31]) in de jaren 70 al bepaald welke oppervlakte het dichtst bij de camera lag en slechts hiervoor de lichtberekening uitgevoerd. Andere hardware en software-implementaties zijn tevens in de jaren 80 geïntroduceerd.[7, 10, 23, 11] Tebbs, Neumann, Eyles, Turk and Ellsworth ([29]) geven een gedetailleerd overzicht van deferred shading in 1990.

Moderne deferred shading algoritmes maken veelal gebruik van GBuffers. GBuffers waren oorspronkelijk geïntroduceerd in de context van het non-fotorealistisch renderen om visuele begrijpbaarheid te verbeteren. [26] Echter GBuffers lenen zich tevens goed om de data tussen de visibiliteitsstap en de lichtberekeningsstap op te slaan.

Deferred shading is een veel gebruikt algoritme in moderne game engines. Voorbeelden van engines die gebruik maken van deferred shading zijn: Unreal 4[14], Frostbyte 2[18], Unity[25] en CryEngine 3[20].

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.

4.1.1 Definities

Binnen deze thesis zal de volgende terminologie gebruikt worden.

Forward shading beschrijft de standaard uitvoering van de renderpijplijn. 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.

4.1.2 Deferred shading

Zoals besproken in sectie 2.4, 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 de directe belichting[1]:

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

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:

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

waar d de afstand tussen punt \mathbf{p} en licht l_k is, zoals beschreven in de definitie van licht.

Binnen forward shading wordt deze radiantieberekening uitgevoerd in fragment shader. Dit betekent dat voor elk geproduceerd fragment deze berekening dient te worden uitgevoerd, en dat alle informatie per fragment beschikbaar is. Pas nadat de radiantie berekening is uitgevoerd, wordt bekijken of een fragment daadwerkelijk wordt opgeslagen of niet. Om deze berekening te ontkoppelen, moet de lichtberekeningsstap na de bepaling van zichtbare fragmenten worden uitgevoerd. Dit betekent dat de verwerking van fragmenten al is voltooid, voordat de radiantie wordt berekend. Dit heeft als bijkomend effect dat de gegevens over elk fragment niet meer impliciet beschikbaar zijn. Het is dus nodig om deze expliciet op te slaan ten tijde van de visibiliteitbepaling.

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

- De geometrie informatie van punt \mathbf{p} .
 - De positie van punt \mathbf{p}
 - De normaal in punt \mathbf{p}
- Informatie met betrekking tot de oppervlakte in punt \mathbf{p}
 - De kleur van het oppervlakte
 - Eventuele extra attributen zoals reflectie coëfficient, ruwheid etc.
- Informatie van de lichten
 - Positie
 - Intensiteit
 - Afstandsdiminuutfunctie

Binnen de geometriestap is het nodig om de benodigde informatie van de geometrie en de 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 met behulp van de grafische pipeline, zullen de per-pixel-operaties ervoor zorgen dat slechts voor de zichtbare fragmenten deze attributen zijn opgeslagen. De belichtingsstap zal vervolgens deze waardes uit het geheugen lezen en gebruiken om de radiantie te berekenen.

4. DEFERRED SHADING



Figuur 4.2: De texturen in de GBuffer gebruikt in killzone 2, geproduceerd door Guerrilla Games[30].

4.1.3 Gbuffer

Om het opslaan van deze attributen te faciliteren wordt een datastructuur gebruik die de *GBuffer* genoemd wordt. Dit is een object bestaande uit meerdere texturen, elk verantwoordelijk voor een attribuut dat opgeslagen dient te worden in de geometriestap zodat deze beschikbaar is in de belichtingsstap. Een voorbeeld van deze texturen zoals deze gebruikt worden in het computerspel Killzone 2 van Guerrilla Games is gegeven in figuur 4.2. Hierin zijn de diepte, normaal, diffuse kleur en shinyness weergegeven.

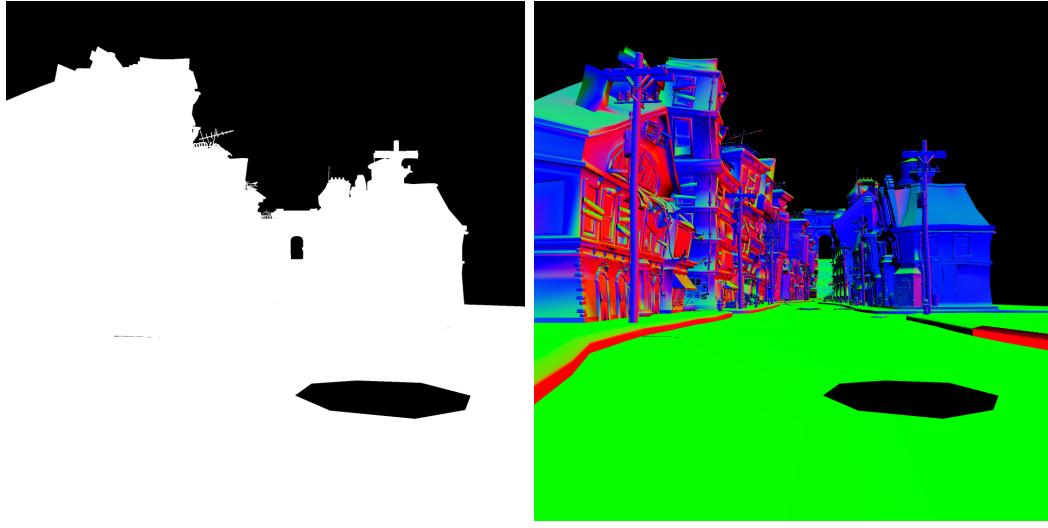
Moderne grafische kaarten hebben de mogelijkheid om te renderen naar meerdere texturen in een enkele uitvoering van de pijplijn. Deze mogelijkheid wordt meerdere render doelen (Multiple Render Targets (MRT))¹ genoemd. Hiervan wordt gebruik gemaakt om de GBuffer te vullen met informatie in de geometrie pass. Deze texturen zullen in het geheugen van de grafische kaart beschikbaar zijn, en opgevraagd kunnen worden gedurende de belichtingsstap.

4.2 Algoritme

De voorgestelde opsplitsing van forward shading, weergegeven in listing 1, leidt tot het volgende algoritme.

- Rasteriseer de geometrie en schrijf de relevante attributen weg naar de GBuffer
- Rasteriseer een vierkant vlak overeenkomend met het gezichtsveld

¹[https://msdn.microsoft.com/en-us/library/windows/desktop/bb147221\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/bb147221(v=vs.85).aspx)



(a) Diffuus.

(b) Normaal.

Figuur 4.3: De texturen in de GBuffer gebruikt in **nTiled**.

- Per fragment van dit vlak, bereken de lichtbijdrage van elke licht met dit fragment.

Hierbij wordt de renderingspijplijn dus twee maal doorlopen. Eerst om de visibiliteit te bepalen. Vervolgens om de lichtberekening uit te voeren.

Dit proces kan nog verder geoptimaliseerd worden door in te zien dat de lichtvolumes tevens gerasteriseerd kunnen worden om zo de fragmenten te verkrijgen waarop zij mogelijk invloed hebben. Voor de fragmenten die geproduceerd worden met een lichtvolume wordt de belichtingsberekening uitgevoerd met het corresponderende licht en de geometrie attributen bepaald in de geometrie stap. Deze bijdrages worden vervolgens gesommeerd in het geheugen van de grafische kaart. Dit leidt tot het volledige deferred algoritme waarvan de pseudocode is weergegeven in listing 2.

Binnen **nTiled** is deze laatste optimalisatie niet geïmplementeerd, en is gekozen om gebruik te maken van viewport-overdekkend vlak. De forward en deferred algoritmes zonder extra versnellingsstructuren worden binnen **nTiled** respectievelijk **Forward Attenuated** en **Deferred Attenuated** genoemd. Voor de GBuffer binnen **nTiled** is gekozen voor drie texturen, positie, diffuse kleur en de normalen. Een voorbeeld van de diffuus en normaal texturen van de GBuffer is weergegeven in figuur 4.3.

4.3 Resultaten

De performantie van het deferred algoritme dat geïntroduceerd werd in de vorige sectie, is geëvalueerd aan de hand van de drie test scenes. Eerst zal het gedrag per frame gedurende een enkele uitvoering besproken worden. Vervolgens wordt gekeken

4. DEFERRED SHADING

naar de gemiddelde executietijd per frame als functie van het aantal lichten, en de resolutie.

Binnen de testen refereert *forward* naar de `forward_attenuated` shader en *deferred* naar de `deferred_attenuated` shader van nTiled.

4.3.1 frames

In figuur 4.4 zijn de gemiddelde executietijden per frame gedurende een complete uitvoering weergegeven. Links zijn de waardes weergegeven van de uitvoeringen met het laagste aantal lichten per scene bij een resolutie van 160×160 . Rechts zijn de waardes weergegeven van het hoogste aantal lichten per scene bij een resolutie van 2560×2560 .

Gelijk valt hierbij op dat bij een lage resolutie en een klein aantal lichten forward en deferred shading elkaar nauwelijks ontlopen. Het verschil in executietijd is echter significant bij een hoge resolutie en een groot aantal lichten. Niet alleen is de executietijd kleiner bij deferred shading, het is tevens consistenter. Dit is belangrijk voor computerspellen en andere real-time toepassingen waar een consistente framerate nodig is voor een overtuigende virtuele ervaring.

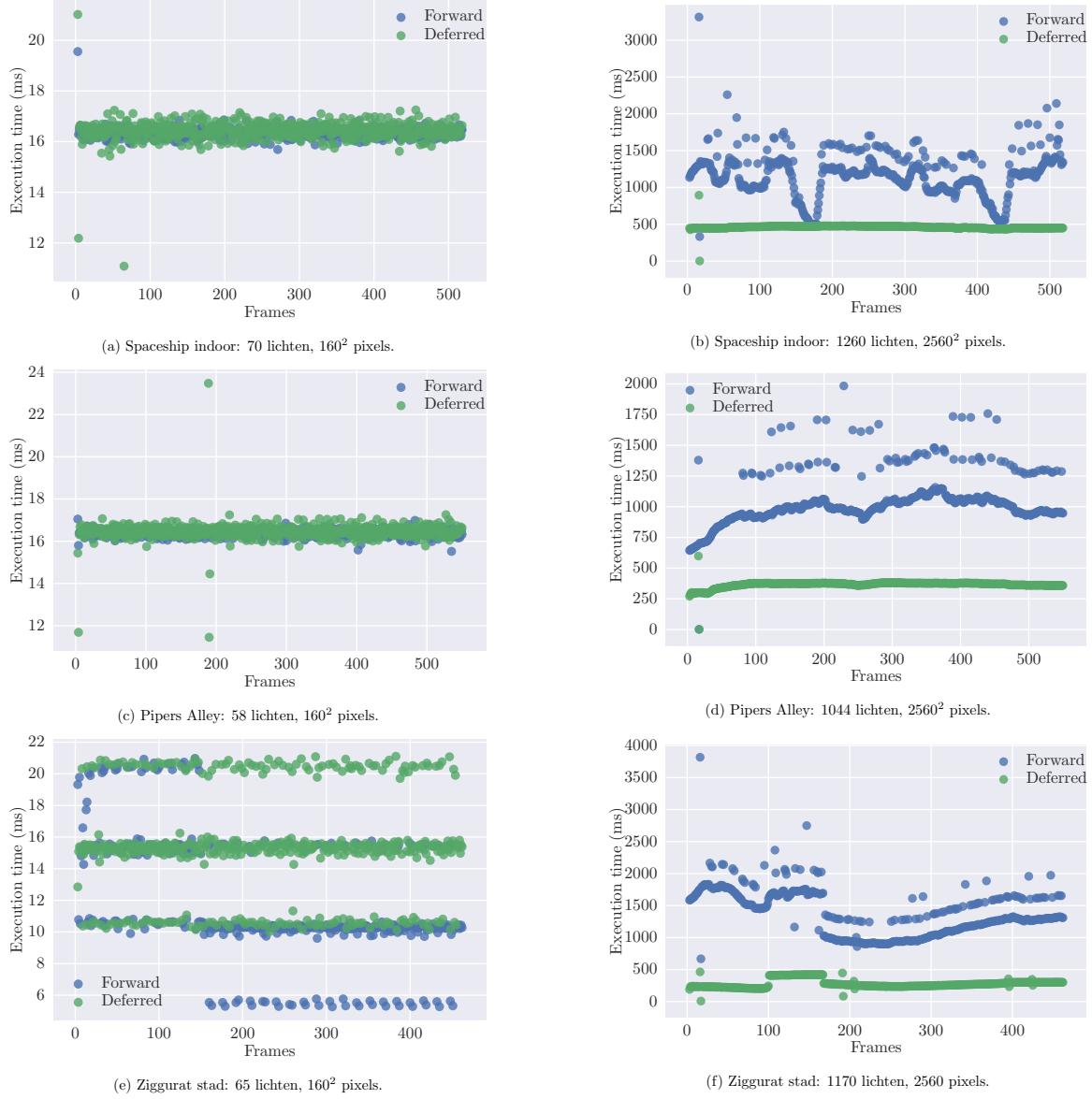
De consistentie in executietijd kan verklaard worden doordat de tijdsbeperkende stappen niet afhankelijk zijn van het aantal fragmenten die zichtbaar zijn. Het wegschrijven van fragmenten naar de GBuffer is een simpele operatie die geen berekeningen vereist. Een verschil in fragmenten zal hierdoor geen significant verschil in executietijd veroorzaken. Tevens zal de belichtingsberekening slechts uitgevoerd worden voor een enkel fragment, waardoor het aantal fragmenten ook geen invloed op de belichtingsstap heeft. Bij forward shading is de lichtberekening tijdsbepalend en deze is direct gekoppeld aan het aantal fragmenten dat gecreeëerd wordt.

Deze koppeling tussen visibiliteit en shadingcomplexiteit is dan ook direct terug te zien in het tijdsgedrag van forward shading. Zo is in de indoor spaceship scene te zien dat de executietijd van forwardshading de executietijd van deferred shading benaderd rond frame 170 en frame 430. Deze frames komen overeen met een zicht op

```
for obj in scene_objects:  
    fragments = rasterise(obj)  
  
    for frag in fragments:  
        for l in lights:  
            canvas[frag.pos] +=  
                do_shade(frag,  
                         frag.attributes,  
                         light)  
  
per_pixel_operations()
```

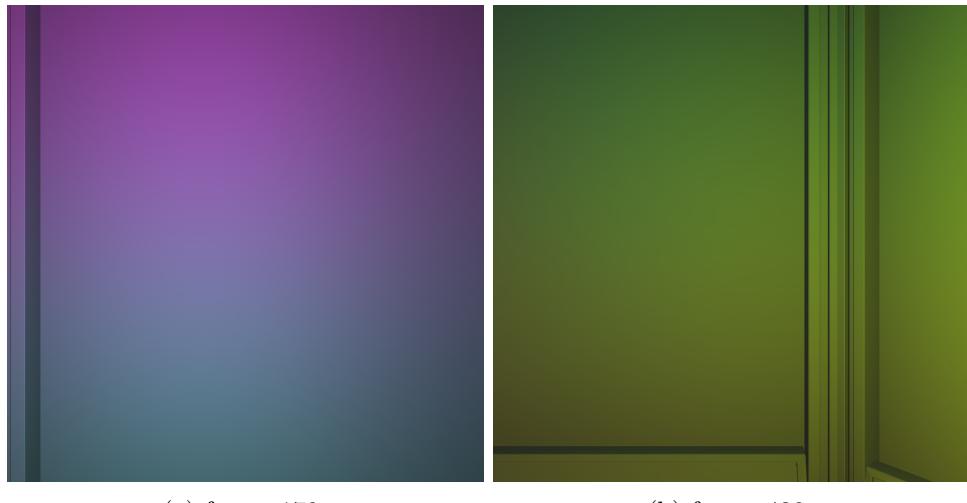
Listing 1: Forward shading.

4.3. Resultaten



Figuur 4.4: Overzicht van de executie tijd per frame voor de drie test scenes bij verschillende resolutie en groottes van aantal lichten.

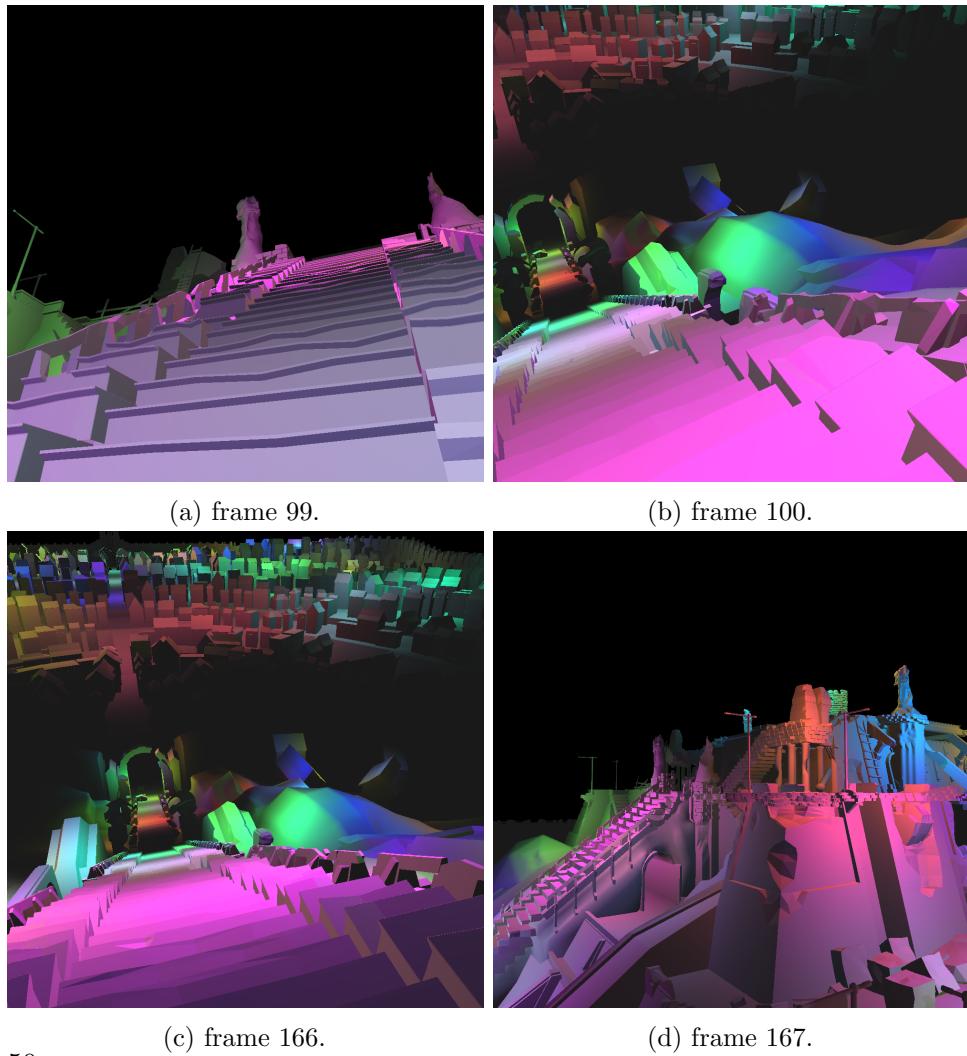
4. DEFERRED SHADING



(a) frame 170.

(b) frame 430.

Figuur 4.5: Indoor spaceship frames waarbij het aantal fragmenten één benaderd.



(a) frame 99.

(b) frame 100.

(c) frame 166.

(d) frame 167.

een enkele muur, zoals weergegeven in figuur 4.5a en 4.5b. Hier benaderd het aantal fragmenten per pixel 'e'en.

Verder is er binnen de ziggurat stadsscene, fig. 4.4f nog een sprong in executietijd te zien bij frame 100 en frame 167. Deze sprongen komen overeen met de verandering van camera, zoals te zien is in figuur 4.6. Het verschil in executietijd is een gevolg van het percentage lege ruimte in de drie camera standpunten. Waar het zicht tussen frame 100 en frame 167 voornamelijk bestaat uit geometrie, en in de frames voor 100 en na 167 ook een lege lucht waar te nemen is.

4.3.2 Lichten

In figuur 4.7 is de gemiddelde executietijd per frame weergegeven als functie van het aantal lichten in de scène bij een resolutie van 2560×2560 . Hierbij zijn alle executietijden per frame gemiddeld over alle frames in een uitvoering.

Er is een lineair verband waar te nemen bij zowel forward als deferred shading. Dit komt overeen met de verwachting dat het aantal lichtberekeningen per fragment lineair schaalt met het aantal lichten. Deferred shading schaalt met een kleinere factor dan forward shading, doordat er slechts voor een enkel fragment de extra lichten geëvalueerd dienen te worden.

4.3.3 Resolutie

In figuur 4.8 is de gemiddelde executietijd per frame weergegeven als functie van de resolutie weergegeven bij het grootste aantal lichten per scène. Hierbij komt een resolutie van n overeen met een gebruikte resolutie van $n \times n$. De executietijden per frame zijn op eenzelfde manier gemiddeld als de executietijd per aantal lichten.

In elk van de grafieken is een kwadratisch verband waarneembaar. Dit komt overeen met de kwadratische toename van pixels, en dus fragmenten.

4.4 Conclusie

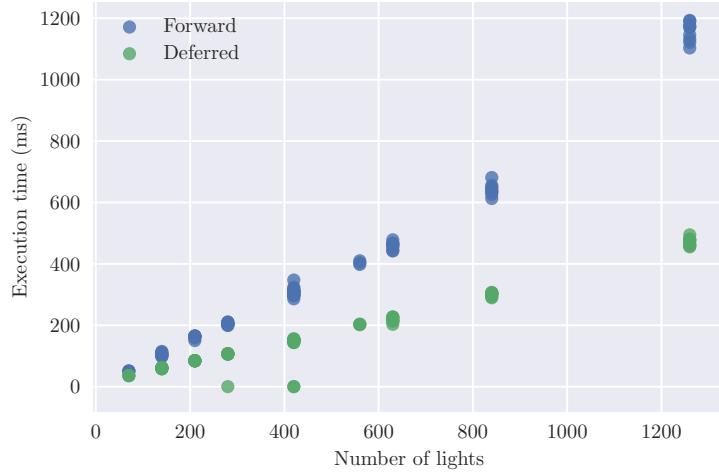
Uit alle grafieken blijkt dat de gemiddelde executietijd kleiner is voor deferred shading dan voor forward shading. Echter beide technieken hebben eenzelfde tijdscomplexiteit ten opzichte van het aantal lichten.

Deferred shading heeft als bijkomend voordeel dat de executietijd consistent is. Dit is een belangrijke eigenschap voor game-engines, waar een constante framerate belangrijk is voor de menselijke interpretatie, als ook voor het verdelen van de berekeningstijd over de verschillende subsystemen.

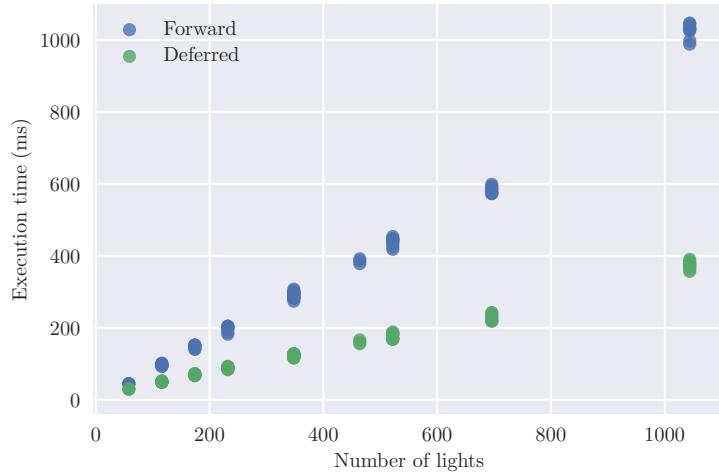
Tegenover de verbetering in de executietijd staat wel dat een set van texturen met een grootte gelijk aan de viewport bijgehouden dient te worden. Dit betekent dat deferred shading een grotere geheugenvoetafdruk heeft. Het aantal texturen dat bijgehouden dient te worden is direct afhankelijk van de complexiteit van de shader.

Tevens maakt het gebruik van een GBuffer bij deferred shading transparantie onmogelijk, doordat het niet mogelijk is om meerdere lagen geometrie op te slaan. Deferred shading is dus beperkt tot slechts ondoorzichtige geometrie.

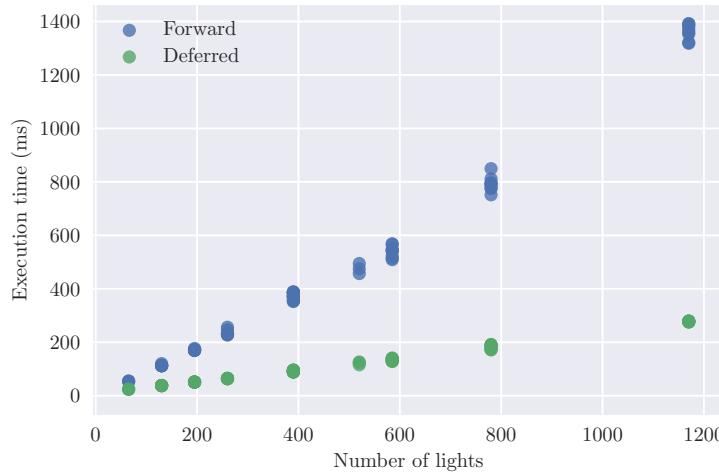
4. DEFERRED SHADING



(a) Spaceship indoor

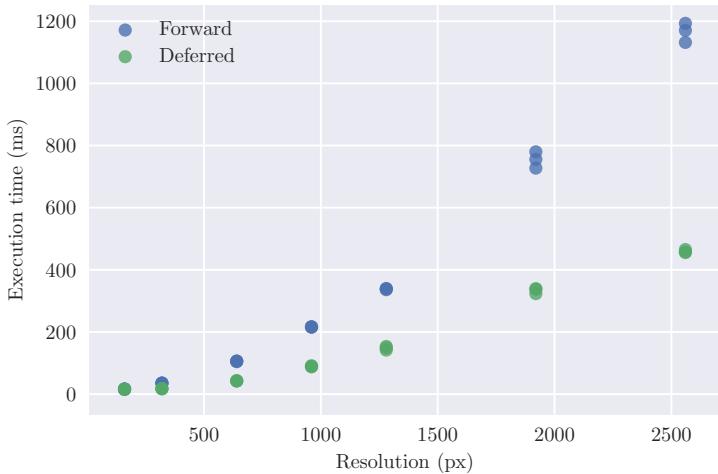


(b) Pipers Alley

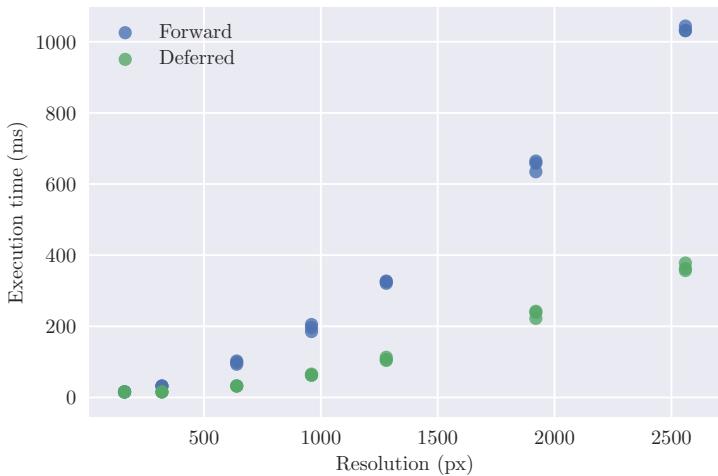


(c) Ziggurat stad

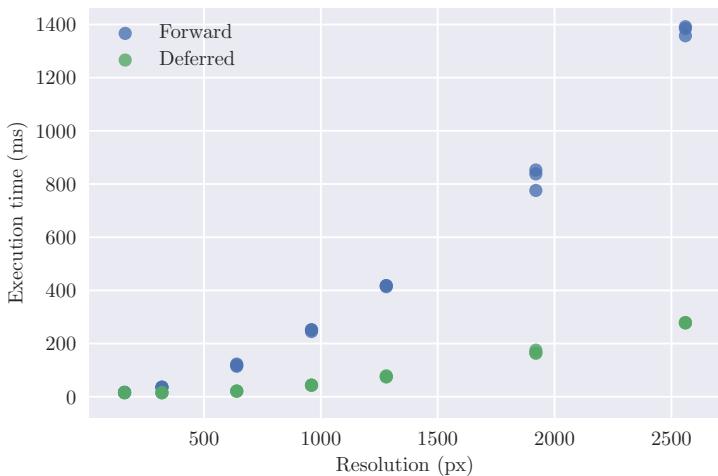
Figuur 4.7: Overzicht van de executie tijd als functie van het aantal lichten, per test scene bij een resolutie van 2560.



(a) Spaceship indoor, 1260 lichten



(b) Pipers Alley, 1044 lichten



(c) Ziggurat stad, 1170 lichten

Figuur 4.8: Overzicht van de executie tijd als functie van de resolutie, per test scene.

4. DEFERRED SHADING

Het laatste minpunt 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.

4.5 Discussie

4.5.1 Rasterisatie van lichtvolumes

Zoals vermeld in het sectie 4.2, is het mogelijk om de lichtvolumes te rasteriseren, en vervolgens per licht slechts voor deze fragmenten de lichtberekening uit te voeren. Dit verlaagt het aantal lichtberekeningen ten opzichte van de implementatie in `nTiled`. Het heeft echter als nadeel dat dit de geheugenbandbreedte significant vergroot. In plaats dat de texturen eenmalig per fragment bemonsterd worden, worden deze dan in het slechtse geval even vaak bemonsterd als er lichten in de scene zijn.

Doordat de geheugenbandbreedte binnen een grafische kaart beperkt is kan dit een knelpunt worden. Helemaal indien de shader complex is, en de GBuffer veel attributen dient op te slaan.

De effecten hiervan zijn niet geëvalueerd in deze thesis, er kan echter aangenomen worden dat dit de executietijd verlaagd zou hebben.

4.5.2 Ondersteunen van transparantie

Transparantie kan niet worden ondersteund met een standaard deferred shading implementatie. Het is onrealistisch om per transparantie laag, een extra GBuffer bij te houden, gezien elke GBuffer een vergelijkbare geheugenvoetafdruk bezit. Binnen veel moderne engines wordt dit ondersteund door een aparte voorwaardse renderingstap uit te voeren voor de transparante objecten, nadat de ondoorzichtige objecten gerenderd zijn.

4.5.3 Ondersteunen anti-aliasing

Het bemonsteren van subpixels binnen forward shading is triviaal. Binnen deferred shading is het echter niet meer mogelijk om subpixels te bemonsteren zonder dat de data hiervan opgeslagen wordt in de GBuffer. Er zijn verschillende aanpakken voorgesteld die anti-aliasing mogelijk maken binnen een deferred pijplijn.[16]

4.5.4 Alternatieven voor deferred shading

Er zijn verschillende alternatieven voorgesteld die elk op verschillende manieren de nadelen besproken in de conclusie proberen te verhelpen. Lighting pre-pass[8], en vergelijkbare algoritmes bouwen verder op deferred shading, met als belangrijkste doel het geheugenverbruik en de geheugenbandbreedte te verminderen.

In het volgende hoofdstuk zal ingegaan worden op Tiled shading, met als doel om de geheugenbandbreedte te verlagen in vergelijking tot deferred shading waarbij de lichtvolumes gerasteriseerd worden.

```
# 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()
```

Listing 2: Deferred shading.

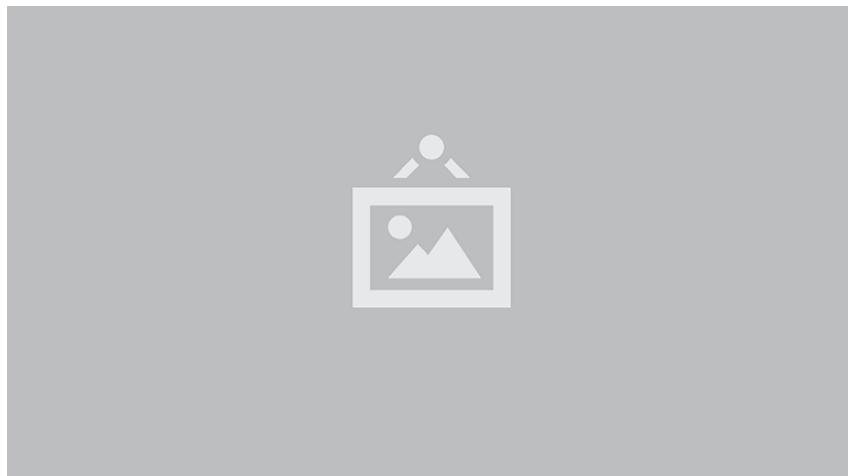
Hoofdstuk 5

Tiled Shading

5.1 Theorie

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 5.1.

Binnen deferred tiled shading lost deze aanpak tevens het bandbrede probleem op zoals deze geïntroduceerd is binnen de sectie 4.1.2. 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 5.1: De onderverdeling van het gezichtsveld dat plaatsvindt in Tiled Shading.

5. TILED SHADING

De techniek zelf is al langere tijd in gebruik binnen de game industrie ([2], [28]). Een formelere beschrijving en evaluatie van de resultaten kan gevonden worden in [22].

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.
- Dezelfde shading functies kunnen gebruikt worden.

Verder zijn er nog enkele specifieke voordelen 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.

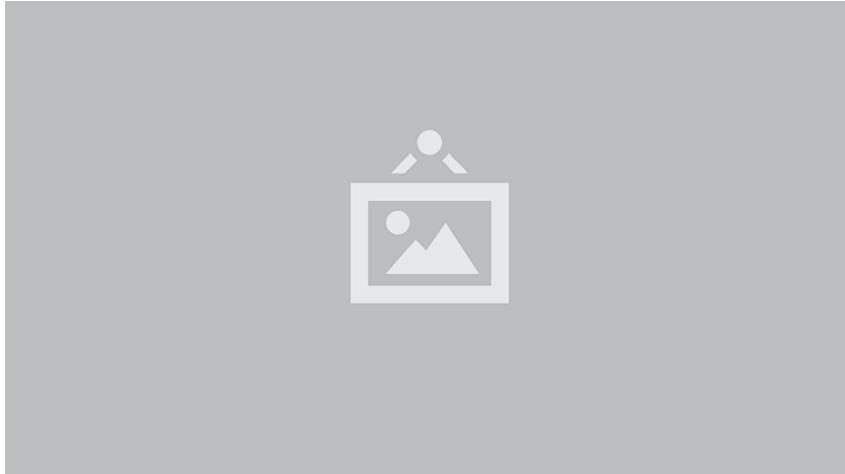
5.1.1 Algoritme beschrijving

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
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 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



Figuur 5.2: De datstructuur van Tiled Shading.

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 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 referent 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 geïllustreerd in figuur 5.2.

5.2 Algoritme

5.3 Testen en resultaten

5.4 Discussie

Hoofdstuk 6

Hashed Shading

In de voorgaande hoofdstukken zijn Tiled en Clustered shading geïntroduceerd. Het doel van deze lichttoekenningsalgoritmes is om voor elk fragment een set van relevante lichten te bepalen, om zodanig het aantal lichtberekeningen terug te brengen. In dit hoofdstuk wordt Hashed shading als alternatief lichttoekenningsalgoritme geïntroduceerd.

Zowel Tiled als Clustered shading zijn camera-afhankelijk en de geassocieerde datastructuren dienen om deze reden per frame opnieuw opgebouwd te worden. Hashed shading daarentegen gebruikt camera-onafhankelijke datastructuren, waardoor deze hergebruikt kan worden tussen frames. Tegelijkertijd behaald het nog steeds een vergelijkbare versnelling en wordt niet significant meer geheugen gebruikt. Dit wordt bereikt door de lichten binnen de scène te representeren als een octree.

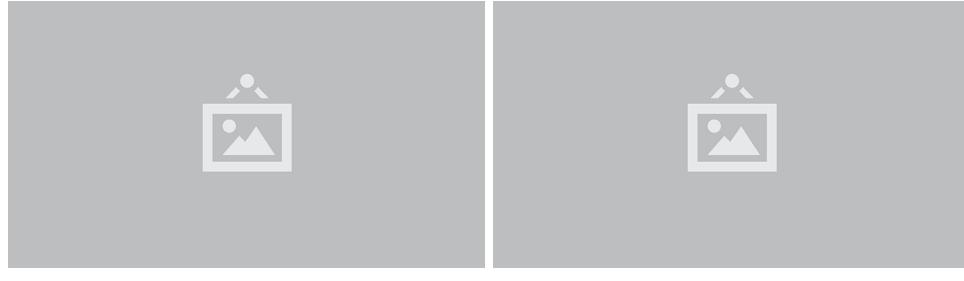
In de volgende secties zal eerst de achterliggende theorie besproken worden, waarbij ingegaan wordt op de keuze voor de octree als spatiale datastructuur, en hoe deze voorgesteld kan worden op de GPU. Vervolgens zal het algoritme behandeld worden. Hierna zal de efficiëncy en het geheugengebruik aan de hand van de testscenes geëvalueerd worden. Als laatste zullen deze resultaten vergeleken worden met Tiled en Clustered Shading.

6.1 Theorie

In het algemeen kan gesteld worden dat lichttoekenning er om draait om de ruimte zo efficiënt mogelijk op te delen, zodat voor elk fragment slechts de lichten die daadwerkelijk invloed hebben, snel opgehaald kunnen worden. De datastructuur die gebruikt wordt om de ruimte onder te verdelen dient dus de volgende attributen te bezitten

- Voor elk punt in de ruimte dient de datastructuur efficiënt de relevante lichten met zo'n groot mogelijke precisie terug te geven
- De datastructuur dient compact van aard te zijn.
- De datastructuur moet dynamisch aan te passen zijn, indien lichten van positie of grootte veranderen.

6. HASHED SHADING



(a) Tiled Shading

(b) Clustered Shading

Figuur 6.1: De onderverdelingen binnen Tiled en Clustered Shading

- De datastructuur dient efficiënt te construeren zijn.

Bij het renderen zal bij elke frame voor elk fragment de set van relevante lichten opgehaald dienen te worden. Dit is dus de operatie die het meest uitgevoerd zal worden. Tevens is het belangrijk dat de datastructuur compact is. Het beschikbare geheugen op de grafische kaart is beperkt, en een compacte voorstelling verkleint de gebruikte geheugenbandbreedte.

In veel moderne toepassingen zijn lichten dynamisch van aard. Denk hierbij aan lichten die geassocieerd zijn met objecten binnen de scenes, zoals koplampen van bewegende autos, als ook lokale lichten die veranderen in felheid door veranderingen in de scene, zoals uitdovende vuren, of explosies. Een datastructuur dient instaat te zijn om dergelijke effecten te modeleren, zonder dat de renderingstijd negatief beïnvloed wordt. Het is mogelijk om de datastructuur volledig opnieuw op te bouwen per frame, echter in veel gevallen zijn deze veranderingen tussen frames klein en lokaal van aard, waardoor het veelal efficiënter is om de al opgestelde datastructuur (gedeeltelijk) her te gebruiken.

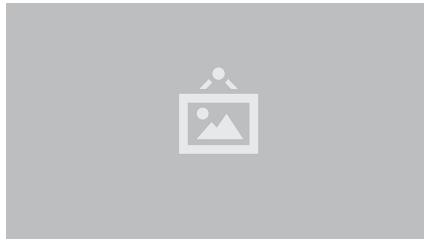
Indien een datastructuur efficiënt kan worden bijgewerkt gedurende de uitvoering, zal de efficiëncie het initieel opstellen van de datastructuur van minder groot belang zijn, gezien deze stap als een pre-process stap uitgevoerd kan worden.

Binnen tiled en clustered shading wordt de viewport ruimte onderverdeeld zoals weergegeven in figuur 6.1. Hierin wordt de compactheid bereikt door slechts een klein deel van de ruimte te behandelen. Het dynamisch karakter wordt in beide technieken behaald door per frame de complete datastructuur opnieuw op te bouwen.

In de volgende secties zullen eerst enkele veel voorkomende spatiale datastructuren behandeld worden. Hierna zal toegelicht worden aan de hand van de bovengestelde eisen, waarom gekozen is voor de octree datastructuur. Vervolgens zal de achterliggende theorie van de octree datastructuur behandeld worden en hoe deze voorgesteld kan worden op de GPU.



Figuur 6.2: Een binaire ruimte partitie.



Figuur 6.3: Een roosterdatastructuur.

6.1.1 Overzicht van Spatiale Datastructuren

Binaire ruimte partitionering

Binaire ruimte partitionering is een methode om een d dimensionale ruimte onder te verdelen. Hiervoor wordt aan de hand van hypervlakken de ruimte recursief opgedeeld.[21] Waarbij een hypervlak een d dimensionaal vlak is. Deze recursieve opdeling wordt opgeslagen in een binaire-ruimte-partitioneringsboom. In het geval van een 1-dimensionale ruimte, zoals een lijst, komt dit overeen met een binaire zoekboom, waarbij de hypervlakken een punt binnen de lijst is. In het geval van een 3-dimensionale ruimte zullen de hypervlakken standaard vlakken zijn, een illustratie voor een drie dimensionale ruimte is weergegeven in figuur 6.2. De volgende datastructuren kunnen gedefinieerd worden als specifieke implementaties van binaire ruimte partitionering.

Rooster

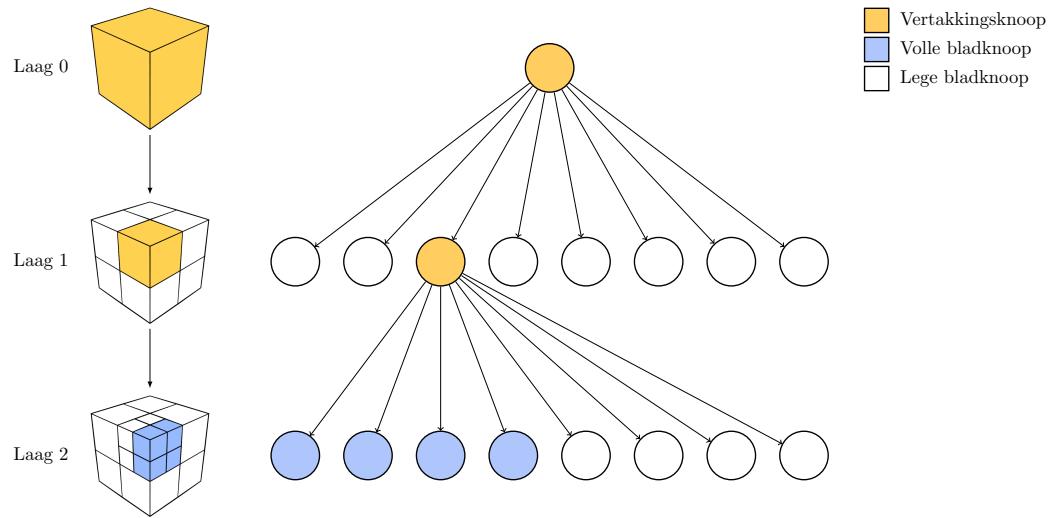
Het rooster is de meest simpele spatiale datastructuur. De relevante ruimte wordt grofweg in cellen van een specifieke grootte onderverdeeld, zoals weergegeven in fig. 6.3. Binnen elke cel wordt vervolgens een verwijzing naar de relevante data bijgehouden.

Een dergelijke datastructuur kan, indien de grootte van de cellen klein wordt genomen, een hoge nauwkeurigheid van relevante lichten opleveren. Tevens zullen cellen snel en eenvoudig toegankelijk zijn. Echter hier staat tegenover dat bij een naïve implementatie het geheugengebruik zeer snel toe neemt, gezien voor elke cel binnen het rooster deze verwijzing dient te worden opgeslagen.

6. HASHED SHADING



Figuur 6.4: De roosterdatastructuur voorgesteld als BRP.



Figuur 6.5: Weergave van een octree bestaande uit drie lagen, links is de 3d representatie weergegeven, rechts de pointers .

Clustered Shading maakt gebruik van een vorm van een rooster over de viewport. Hierbij wordt de data compacter voorgesteld door slechts cellen die zowel licht als geometrie bevatten, op te slaan.

Een rooster kan simpelweg gedefinieerd worden als een drie dimensionale lijst. Indien dit als binaire ruimte partitionering wordt weergegeven, wordt recursief in elke dimensie steeds 'e'en cel gedefinieerd, zoals weergegeven in figuur 6.4.

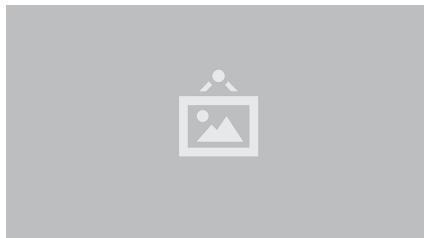
Octrees

De octree is een boomdatastructuur waarbij elke takknoop precies acht kinderen bezit. Elke knoop kan voorgesteld worden als een kubus. De kinderen van een tak knoop verdelen de ruimte geassocieerd met de takknoop in acht equivalenten kubussen.[19] Dit is weergegeven in fig 6.5.

Data kan ofwel in elke knoop opgeslagen worden, ofwel slechts in de bladknopen. Indien een knoop een ruimte volledig beschrijft is het niet nodig om deze verder onder te verdelen. Deze hierarchische structuur zorgt ervoor dat het mogelijk is om de ruimte in hoge resolutie te beschrijven, en tevens niet tegen de geheugenproblemen



Figuur 6.6: De octree datastructuur als BRP.



Figuur 6.7: De kd-boom datastructuur.

van het rooster aan te lopen, doordat grote homogene ruimtes door slechts een enkele knoop kunnen worden voorgesteld.

De octree definieert een ruimtepartitionering door per takknoop drie hypervlakken op te stellen. Deze zullen altijd het middelpunt van de octreeknoop snijden, zoals weergegeven in figuur 6.6.

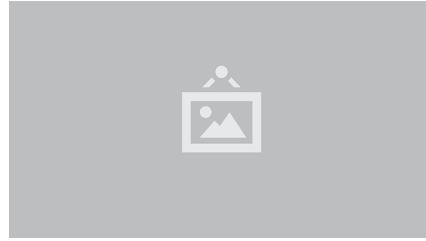
kd-boom

Een kd-boom is een boomstructuur in d dimensies, waarbij k discriminatoren zijn gedefinieerd.^[3] De discriminator associeert een specifieke orientatie van een hypervlak met een punt. Alle knopen op eenzelfde niveau delen dezelfde discriminator. Een takknoop definieert vervolgens de positie van het hypervlak met orientatie k_i . Hierbij wordt de ruimte geassocieerd met de knoop opgedeeld in twee delen. Dit is weergegeven in figuur 6.7.

Dit is een specifiek geval van binaire ruimte partitionering, op elk niveau wordt de ruimte in twee helften ingedeeld. Het verschil met generieke binaire ruimte partitionering is dat de set van hypervlakken slechts een subset is van alle mogelijke hypervlakken.

R boom

De r-boom is een n -dimensionale zelf-balancerende boom. Elke knoop bevat een n -dimensionaal vierkant. In de bladknopen bevindt zich binnen dit vierkant de data geassocieerd met deze knoop. In het geval van de takknoop bevinden alle kinderen zich binnen het vierkant geassocieerd met de knoop. [12] Hierbij hebben de vierkanten twee belangrijke eigenschappen:



Figuur 6.8: De r-boom datastructuur.

- De vierkanten overlappen niet
- De vierkanten bevatten zo min mogelijk lege ruimte

In combinatie met het feit dat de boom gebalanceerd is, zorgt dit voor de mogelijkheid om spatiale data efficiënt op te halen.

6.1.2 Keuze voor de octree-datastructuur

Het doel van lichttoekenning is om de ruimte zodanig onder te verdelen dat de relevante lichten voor een punt in de wereld efficiënt opgehaald kunnen worden. De spatiale datastructuur die gebruikt wordt voor het ophalen van de lichten dient te voldoen aan de eerder genoemde eisen.

Een belangrijke observatie is dat dit geen dichtste-buur-probleem is. Onder perfecte omstandigheden zou de datastructuur alle lichten waarvan het punt in het lichtvolume valt, teruggeven, en geen enkel ander licht. De punten zijn hierbij zijn de fragmenten die geprojecteerd zijn op de viewport.

Zowel de kd-boom als de R-boom zijn ontwikkeld met het dichtste-buur-probleem in gedachte, waarbij de dataset bestaat uit een set van punten. Deze datastructuren zijn daarom minder geschikt voor lichttoekenning, waarbij de data geen punten maar volumes zijn.

Zowel het rooster als de octree zijn geschikter voor lichttoekenning, omdat de manier van opdeling bepaald wordt door het volume, en niet door de data. In het geval van het rooster levert dit uniforme kubussen op waarvoor voor elk volume bepaald kan worden met welke lichtvolumen het overlapt. Dit betekent dat ook voor grote uniforme ruimtes een groot aantal kubussen zijn gedefinieerd. Hierdoor schaalt het rooster slecht met de grootte van de ruimte. De octree is in staat om dergelijke uniforme ruimtes efficiënt voor te stellen. Om deze redenen is gekozen voor de octree-datastructuur als basis voor het camera-onafhankelijke lichttoekenningsalgoritme.

6.1.3 Octrees datastructuur

6.1.4 Hash functies

Stel $U = \{0, 1, 2, \dots, u - 1\}$ is een ruimte voor een positieve integer u . Neem S een set van n unieke elementen (sleutelwaardes) binnen U . Een hash functie is gedefinieerd

als een functie $h : U \rightarrow M$ die de sleutelwaarden in S afbeeldt op een gegeven interval $[0, m - 1]$.

Het is mogelijk om de ruimte M te gebruiken als adres ruimte geassocieerd met een opslagvolume. De adressen in deze ruimte M kunnen gebruikt worden om elementen geassocieerd met de sleutelwaarden op te slaan in het opslagvolume. Het opslag volume wordt in dit geval een hash table genoemd. Het gebruik van hash functies in combinatie met hash tables maakt het mogelijk om data verspreid over een grote ruimte compact op te slaan en op deze manier geheugen gebruik terug te brengen.[5, 15]

Indien twee sleutels op een zelfde adres worden afgebeeld, wordt gesproken van een botsing. De sleutels corresponderende met dit adres worden synoniemen van elkaar genoemd. Botsingen leiden er toe dat waardes geassocieerd met synoniemen niet meer in een enkele stap uit de hash table kunnen worden opgehaald. Er zijn verschillende manieren om om te gaan met dergelijke botsingen.

Indien elke sleutel op een uniek adres wordt afgebeeld, wordt gesproken van een perfecte hash functie, of een 1-probe hash functie. Dergelijke functies zijn dus gedefinieerd als:

$$h : U \rightarrow [0, m - 1] \forall x, y \in S : x \neq y \rightarrow h(x) \neq h(y)$$

Hieruit volgt dat de adres ruimte M geassocieerd met de set van sleutelwaardes S , een grootte gelijk of groter dan de set van sleutel waardes dient te hebben, $m \geq n$.

Wanneer geldt dat $m = n$ wordt de gesproken van een perfecte minimale hash functie. Een dergelijke minimale hash functie bestaat voor elke set van sleutel waardes, doordat geldt dat voor twee eindige sets X en Y van gelijke grootte die lineair geordend zijn er altijd een injectie functie bestaat zodanig dat X op Y wordt afgebeeld[6]:

$$h : X \rightarrow Y$$

hiervoor geldt dat

$$\forall x_1, x_2 \in X : x_1 \leq x_2 \Leftrightarrow h(x_1) \leq^* h(x_2)$$

waar \leq en \leq^* de respectievelijke lineaire orde functies zijn.

Het blijkt echter niet triviaal om dergelijke functies te vinden. Dit wordt verder gecompliceerd doordat het geheugen gebruik en rekentijd die dergelijke functies gebruiken ook van belang zijn. In de praktijk is het veelal niet doenlijk om minimale perfecte hash functies op te stellen.

Er zijn verschillende algoritmes om perfecte hash functies op te stellen die slechts zorgen voor een klein extra geheugen gebruik ten opzicht van de minimale perfecte hash functie.

6.1.5 Ruimtelijke hash functies

In Levebvre en Hoppe[17] wordt een algoritme besproken dat verder bouwt op perfecte hash functies gedefinieerd met behulp van hulp tabellen. Het gepresenteerde

6. HASHED SHADING

algoritme spitst zich toe op grafische applicaties. Het heeft als doel om verspreide data binnen d -dimensionale ruimtes compact op te slaan, en efficiënt op te halen uit geheugen. Er wordt gebruik gemaakt van een perfecte hash functie, zodat de grafische kaart efficiënt gebruik kan maken van de datastructuur, doordat er geen conditietakken ontstaan.

Terminologie

Het ruimte U is gedefinieerd als een drie dimensionaal rooster met grootte $u = \dot{u}^3$, waarbij de posities zijn gedefinieerd als

$$\mathbf{p} \in \mathbb{Z}_{\dot{u}}^3 = [0, (\dot{u} - 1)]^3$$

Binnen dit rooster bevindt zich een set $S \subset U$ met n elementen, waarbij voor elke positie $\mathbf{p} \in S$ een data element is geassocieerd zodanig dat $D(\mathbf{p}) = d$.

Het doel van de ruimtelijke hash functie is om de data in D op te slaan in een compacte hash tabel H , zodanig dat

$$D(\mathbf{p}) = H[h(\mathbf{p})]$$

Hash tabel H heeft een grootte van $m = \dot{m}^3 \leq n$ en bevat de data elementen uit D . De hash functie $h(\mathbf{p})$ is gedefinieerd als

$$h = h_0(\mathbf{p}) + \Phi[h_1(\mathbf{p})] \mod \dot{m}$$

Zowel h_0 als h_1 zijn imperfecte hash functies. De offset die volgt uit de offset tabel Φ zorgt er voor dat botsingen in h_0 worden voorkomen. Hierdoor gedraagt $h(\mathbf{p})$ zich als een perfecte hash functie. De offset tabel Φ is gedefinieerd als een hash tabel bevattende offset coördinaten in de vorm van drie dimensionale vectoren. De grootte van offset tabel Φ wordt gesteld op $r = \dot{r}^3 \leq \sigma n$. De hash functies in $h(\mathbf{p})$ zijn gedefinieerd als

$$\begin{aligned} h_0 : \mathbf{p} &\rightarrow \mathbf{p} \mod \dot{m} \\ h_1 : \mathbf{p} &\rightarrow \mathbf{p} \mod \dot{r} \end{aligned}$$

De eisen aan deze functies en hash tabellen zijn:

1. $h(\mathbf{p})$ is een perfecte hash functie.
2. Φ en H zijn zo compact mogelijk.
3. De toegang tot de hash tabellen, $\Phi[h_1(\mathbf{p})]$ en $H[h(\mathbf{p})]$ moet goede ruimtelijke coherentie tonen rekeninghoudend met \mathbf{p} .

Constructie

De eerste stap in de constructie van een ruimtelijke hash functie is het vast stellen van de parameters van de hash tabellen. \dot{m} wordt gesteld op de kleinste waarde

zodat voldaan wordt aan $m = \dot{m}^3 \leq n$. De offset waardes worden voorgesteld met behulp van drie 8 bit integers. In het geval dat $\dot{m} > 255$ dan wordt m gesteld op $m = \dot{m}^3 \leq n * 1.01$, om een perfecte hash functie mogelijk te maken.

De grootte van \dot{r} kan op twee manieren worden geconstrueerd, verschillend in berekeningstijd en compactheid van de offset tabel.

- Voor snelle constructie wordt \dot{r} initieel gesteld op $r = \dot{r}^3 \leq \sigma n$ waarbij de factor σ gesteld wordt op $\frac{1}{6}$. Omdat elke offset uit 8 bits per dimensie bestaat, leidt dit tot vier extra bits per data element. Indien geen perfecte hash functie gevonden kan worden, wordt \dot{r} geometrisch vergroot.
- Voor een optimale constructie met betrekking tot geheugenverbruik wordt gebruik gemaakt van een binaire zoek functie over \dot{r} . Er wordt gebruik gemaakt van een gretig probabilistisch algoritme om \dot{r} zo klein mogelijk te definieren, zodanig dat de gevonden hash functie de perfecte minmale hashfunctie benadert.

Nadat de groottes van de data tabel en offset tabel gedefinieerd zijn, dienen de expliciete wardes te worden toegekend aan de hash tabel elementen. Zoals eerder vermeld dienen de offset waardes in de offset tabel Φ botsingen in $h(\mathbf{p})$ te voorkomen. Elke waarde in de offset tabel dient zodanig gekozen te worden, dat de synoniemen voor een locatie in Φ geen botsingen veroorzaken in $h(\mathbf{p})$. De ruimte O definieert het mogelijke domein van Φ .

$$O : \mathbb{Z}_{\dot{r}}^3 = [0, (\dot{r} - 1)]^3$$

Vervolgens kan deze eis gedefinieerd worden als

$$\forall \mathbf{o} \in O : \forall \mathbf{p} \in \{\mathbf{e} | \mathbf{e} \in S \wedge h_1(\mathbf{e}) = o\} : \nexists \mathbf{p}' h(\mathbf{p}') = ((h_0(\mathbf{p}) + \Phi(o)) \mod \dot{m})$$

Wanneer de set $\{\mathbf{e} \in S \wedge h_1(\mathbf{e}) = o\}$ van synoniemen van een waarde o groot is, zal het vinden van een offset die botsingen in $h(\mathbf{p})$ voorkomt moeilijker worden.[9] Om een correcte toekenning van offset waardes te vergemakkelijken, zullen de elementen $o \in O$ geordend worden op de grootte van hun respectievelijke synoniemen set. De waardes o met de grootste synoniemen sets zullen als eerste toegekend worden.

De toegekende offset waardes dienen verder zodanig gekozen te worden dat de ruimtelijke coherentie, en daarmee de textuur raadpleging in H , coherent blijven. Doordat $h_1(\mathbf{p})$ slechts een modulus operatie is, is deze per definitie coherent.

Coherentie tussen twee elementen kan worden gedefinieerd als

$$N_S(\mathbf{p}_0, \mathbf{p}_1) := \begin{cases} 1 & : \|\mathbf{p}_0, \mathbf{p}_1\| = 1 \\ 0 & : \text{otherwise} \end{cases}$$

Het doel is om de coherentie van data queries in hash tabel H te maximaliseren. De coherentie in hash tabel H kan gedefinieerd worden als



Figuur 6.9: Een voorbeeld octree.

$$\begin{aligned}\mathcal{N}_H &= \sum_{\mathbf{p}_0, \mathbf{p}_1 | N_S(\mathbf{p}_0, \mathbf{p}_1) = 1} N_H(h\mathbf{p}_0), h(\mathbf{p}_1) \\ &= \sum_{\mathbf{p}_0, \mathbf{p}_1 | N_H(h\mathbf{p}_0), h(\mathbf{p}_1) = 1} N_S(\mathbf{p}_0, \mathbf{p}_1)\end{aligned}$$

De laatste uitdrukking kan gemeten worden tijdens constructie. Wanneer deze gemaximaliseerd leidt dit tot de uitdrukking

$$\max_{\Phi[\mathbf{o}]} (\mathcal{C}(\Phi[\mathbf{o}])), \mathcal{C}(\Phi[\mathbf{o}] = \sum_{\mathbf{p} \in h_1^{-1}(\mathbf{o}), |Vert\Delta| = 1} N_S(h^{-1}(h_0(\mathbf{p}) + \Phi[\mathbf{o}] + \Delta), \mathbf{p}).$$

Doordat het testen van alle mogelijke waardes computationeel te veel eisend zou zijn, wordt in plaats hiervan een heuristiek gebruikt. Indien de offset tabel lokaal constant is, dan is de afhankelijke hash functie coherent. Dus wordt bij het toewijzen van een offset bij \mathbf{o} eerst gekeken naar de naburige cellen van \mathbf{o} , wanneer deze een offset bevatten die tevens correct is voor \mathbf{o} , wordt deze ook aan \mathbf{o} toegewezen.

$$\Phi[\mathbf{o}] \in \{\Phi[\mathbf{o}'] | \| \mathbf{o} - \mathbf{o}' \| < 2\}$$

Indien geen van de omliggende offset waardes correct is, wordt willekeurig een correcte kandidaat gevonden.

6.1.6 Verbindingloze octree

Met behulp van de ruimtelijke hash functie is het mogelijk om een efficiënte octree implementatie voor GPUs op te stellen.[4] Zoals eerder vermeld werkt de standaard CPU implementatie met pointers, waarbij recursief in de boom wordt afgedaald.



Figuur 6.10: Een voorbeeld hoe een series van lagen van een octree gecodeerd kan worden met behulp van een ruimtelijke hash functies.

Dergelijke datastructuren die sterk leunen op controle structuren zijn niet erg efficiënt wanneer deze gebruikt worden op de GPU.

De data die opgeslagen wordt in een octree is in zeer grote mate verspreid over de drie dimensionale ruimte. Indien gekeken wordt naar figuur 6.9, is te zien dat een groot deel van de cellen in deze octree leeg zijn, hiervoor dient dus geen data opgeslagen te worden. Het is mogelijk om de octree voor te stellen als een stapel van lagen, met elk een fijnere granulariteit. Elk van deze lagen kan gecodeerd worden als een ruimtelijke hash functie. Dit is weergegeven in figuur 6.10

Elke cel die zich bevindt in een octree laag dient te worden voorgesteld. Een cel kan ofwel een blad, of wel een tak knoop zijn binnen de octree. Verder kunnen blad knopen of wel leeg zijn, of wel data bevatten. Een cel kan dus worden voorgesteld met twee bits, de eerste bit die aangeeft of een cel een blad knoop is, en de tweede bit die aangeeft of een cel data bevat of niet. Om efficiency redenen worden hierbij acht nodes samengenomen, zodat deze voorgesteld kunnen worden met behulp van twee integers van elk acht bits. Omdat elke tak knoop acht kinderen bevat, geeft deze representatie dus een tak knoop weer, waarin elk van de kinderen is geencodeerd.

De data geassocieerd met elke knoop in de octree kan worden opgeslagen binnen het data element in de hash tabel van deze knoop, hiervoor wordt dan boven op de twee 8 bits integers een set van 8 data elementen toegevoegd. Echter in het geval dat dergelijke data elementen groot zijn, of indien een groot aantal knopen geen data bevat, is het efficiënter om niet per cel data ruimte te reserveren, maar deze apart op te slaan. Hiervoor kan per laag, een tweede ruimtelijke hash functie worden bijgehouden, die alle relevante data elementen compact op slaat. Dit leidt tot een voorstelling zoals weergegeven in figuur 6.11. De hash tabel die de octree encodeert zal de octree tabel genoemd worden, en de hash tabel die de data encodeert zal de data tabel genoemd worden.

Indien data dient te worden opgehaald uit een verbindingloze octree, wordt per

6. HASHED SHADING



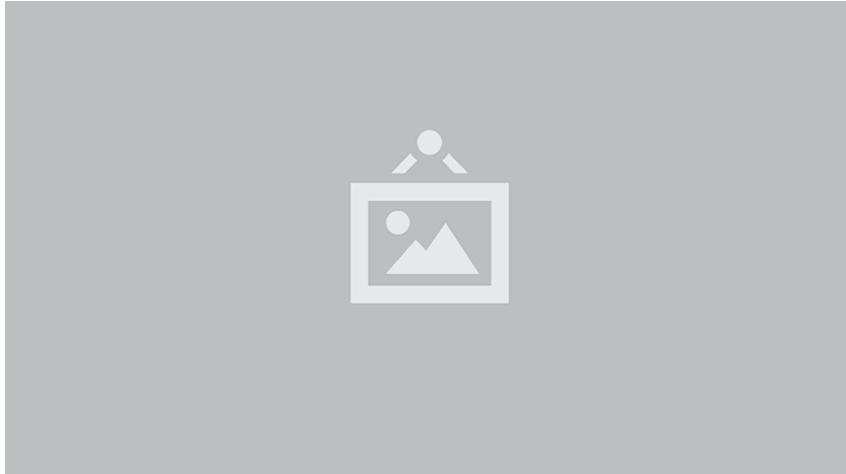
Figuur 6.11: Een voorbeeld van de representatie van een octree met behulp van een verbindingloze octree.



Figuur 6.12: Een visuele weergave van het algoritme om data uit de verbindingloze octree te halen.

laag de knoop uit de octreetabel berekend, en bepaald of de knoop in deze laag een blad knoop is. Is dit niet het geval dan wordt de knoop in de octreetabel van de volgende laag berekend. Is dit wel het geval, dan wordt gekeken of deze blad knoop ook data bevat. Indien de knoop niet leeg is wordt ofwel de data geassocieerd met de gevonden knoop in de octreetabel teruggegeven, ofwel wordt de knoop in de datatabel van deze laag berekend en teruggegeven. Dit algoritme is visueel weergegeven in figuur 6.12.

Doordat de basis van de ruimtelijke hash functie een combinatie van simpele berekeningen en texturen is in plaats van het recursief doorlopen van een boom structuur, kan deze efficient worden geïmplementeerd op de grafische kaart. Dit maakt het mogelijk om deze datastructuur als basis te gebruiken voor het hashed



Figuur 6.13: Een overzicht van Hashed shading.

shading algoritme. Deze is beschreven in de volgende sectie.

6.2 Algoritme

In deze sectie wordt het algoritme beschreven dat de basis vormt voor de implementatie binnen `nTiled`. De volledige set van datastructuren en algoritmes zal Hashed shading genoemd worden, naar de ruimtelijke hash-functies die de basis vormen van de octree implementatie, zoals eerder beschreven in de theorie.

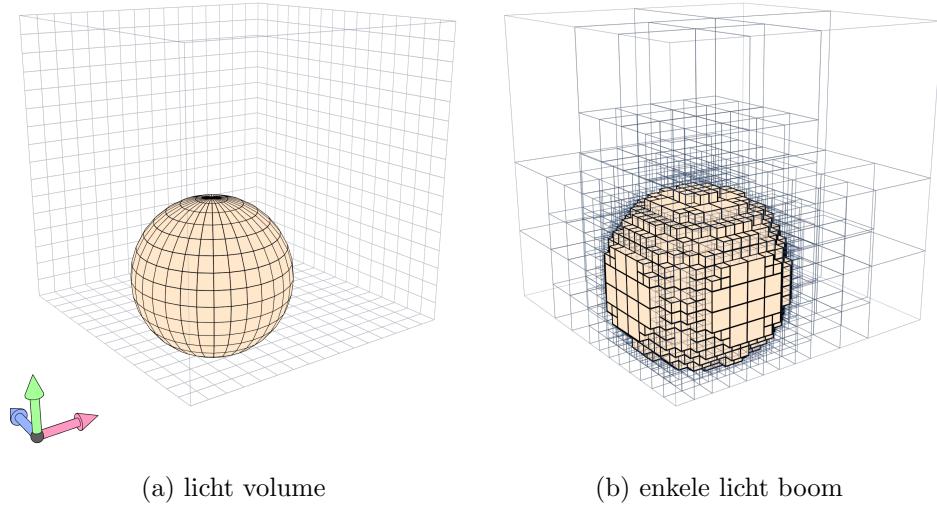
6.2.1 Overzicht

Het hashed shading algoritme vervangt de toekenning van clusters per pixel, met een verbindingsloze octree. De octree geeft een gevoxeliseerde beschrijving van de lichten binnen een scene. Elke blad knoop binnen bevat de lichten waarmee de knoop overlapt. Deze lichten worden beschreven met een afstand en lengte binnen de licht index lijst, zoals deze gespecificeerd is in tiled en clustered shading. De grootte van de voxels kunnen door de gebruiker gespecificeerd worden. Als de afstand en lengte binnen de licht indices lijst vast gesteld is, kan de licht berekening plaatsvinden zoals in tiled en clustered shading.

Bovenop de linkloze octree, en de lijsten van lichten en licht indices, wordt er aan de CPU kant een traditionele octree bijgehouden die alle lichten bevat, en per licht een octree die het licht binnen de traditionele octree beschrijft. Deze set van octrees wordt gebruikt om de verbindingsloze octree op te stellen en aanpassingen aan de staat van de lichten efficiënt up te daten.

Het hashed shading algoritme kan dus in drie componenten worden onderverdeeld;

- Opzoeken van licht informatie
- Constructie van de verschillende datastructuren
- Updaten van data binnen de lichtstructuur



Figuur 6.14: Voorstelling van de enkele licht boom.

Deze datastructuren en hoe deze samenhangen met deze componenten zijn geïllustreerd in figuur 6.13.

Binnen `nTiled` zijn de eerste twee componenten gerealiseerd. Een strategie voor het updaten van de datastructuur zal besproken worden in de discussie.

De constructie stap binnen hashed shading is een pre-process stap en kan worden uitgevoerd voordat de daadwerkelijke rendering plaatsvindt. Deze constructie valt uit een in drie stappen. Eerst zal voor elk licht een enkele licht boom berekend worden. Uit deze enkele lichtbomen wordt vervolgens de overkoepelende lichtoctree gebouwd worden. Nadat de lichtoctree opgesteld is, is het mogelijk om een verbindingloze octree op te stellen. De verbindingloze octree zal vervolgens gebruikt worden om de scenes mee te shaden.

In deze sectie zal de constructie van elk van de datastructuren worden toegelicht. Tevens zal het algoritme voor het opzoeken van de licht informatie worden toegelicht.

6.2.2 Enkele Licht Boom

De enkele lichtboom (Single Light Tree) beschrijft de octreerepresentatie van een enkel licht binnen de gehele lichtoctree. Een voorbeeld hiervan is gegeven in figuur 6.14. Elke bladknoop beschrijft of het volume geassocieerd met de bladknoop een gedeelte van het lichtvolume van de lichtbron bevat. De verzameling van alle enkele lichtbomen wordt samengevoegd om de volledige licht octree te verkrijgen.

Om een enkele lichtboom op te stellen is de volgende informatie nodig:

- De lichtbron en zijn lichtvolume.
- Een grootte voor de ribben van de kleinste mogelijke bladknoop
- De oorsprong van de lichtoctree

Voor de lichtoctree oorsprong moet gelden dat deze kleiner is in elke dimensie dan de oorsprong van het licht minus de radius:

$$\forall d \in \{x, y, z\} : p_d - r > o_d$$

Waar p_d de oorsprong van de lichtbron is in dimensie d , r de radius van de lichtbron en o_d de oorsprong van de licht octree in dimensie d .

De enkele lichtboom wordt opgebouwd in twee stappen. Eerst wordt een rooster opgesteld bestaande uit knopen van minimale lengte. Voor deze knopen wordt vastgesteld of zij overlappen met het lichtvolume van de lichtbron. Vervolgens wordt dit rooster gebruikt om van bovenaf de enkele lichtboom op te stellen. Beginnend vanaf de wortelknoop wordt bepaald of een knoop een tak- of een bladknoop is.

Het rooster in de eerste stap wordt zodanig gekozen dat het bestaat uit het minimale aantal knopen dat het volledige lichtvolume omvat. Vervolgens dient per knoop bepaald te worden of deze overlapt met het lichtvolume. Hiervoor wordt gekeken of het punt \mathbf{v} binnen de knoop dat het dichtst bij de oorsprong van de lichtbron ligt binnen de radius van het licht valt. Dit punt kan gevonden worden door een klemoperatie toe te passen op de oorsprong van de lichtbron. Voor elke dimensie wordt de positie van de lichtbron geklemd tussen de uiterste waarden van de knoop:

$$\mathbf{v} = \begin{pmatrix} p_x|_{[k_x, k_x+l]} \\ p_y|_{[k_y, k_y+l]} \\ p_z|_{[k_z, k_z+l]} \end{pmatrix}$$

Waarbij p_d de oorsprong van de lichtbron in dimensie d is, k_d oorsprong van de knoop in dimensie d is, en l de lengte van de ribben van de knoop.

Vervolgens wordt de afstand van dit punt tot de lichtbron vergeleken met de radius van de lichtbron. Dit algoritme is weergegeven in de volgende pseudocode

```
def node_in_light(node, light) -> bool:
    closest_point =
        vec3( clamp(node.x, light.orig.x, node.x + node.size)
              , clamp(node.y, light.orig.y, node.y + node.size)
              , clamp(node.z, light.orig.z, node.z + node.size)
        )
    p = closest_point - light.orig
    return ((p.x * p.x + p.y * p.y + p.z * p.z) >
            light.radius * light.radius )
```

Het is niet nodig om deze berekening uit te voeren voor elke knoop in het rooster. In plaats hiervan kan een breedte-eerst flood-fill algoritme gebruikt worden. Er wordt in dit geval ofwel vanuit gegaan dat elke knoop in geen licht bevat waarna de nodes gevuld worden die wel overlappen. Of er wordt aangenomen dat alle lichten overlappen, waarna alle knopen die niet overlappen leeg worden gemaakt.

6. HASHED SHADING

In het geval van een puntlicht kan het volume worden gedefinieerd als dat van een bol:

$$V = \frac{4}{3}\pi r^3$$

gezien het volume van bol iets meer dan de helft van de omsluitende kubus bevat, is voor het puntlicht gekozen om ervan uit te gaan dat alle knopen overlappen met het licht en vervolgens de niet overlappende knopen te markeren. Als beginpunten worden de hoeken van de kubus gebruikt. Dit gehele proces is weergegeven in figuur 6.15.

In de tweede stap wordt van bovenaf de enkele lichtboom opgebouwd. Voor elke knoop wordt bepaald of dit een tak- of een bladknoop is. In het geval van een takknoop, wordt tevens voor de kinderen van deze knoop bepaald, welke type knoop zij zijn. Het type knoop wordt bepaald aan de hand van het rooster. Er zijn initieel drie mogelijke situaties:

- De knoop overlapt niet met het rooster
- De knoop overlapt gedeeltelijk met het rooster
- De knoop valt binnen het rooster

In het eerste geval is de knoop altijd een lege bladknoop, gezien het onmogelijk is dat er volle knopen van minimale lengte buiten het rooster liggen.

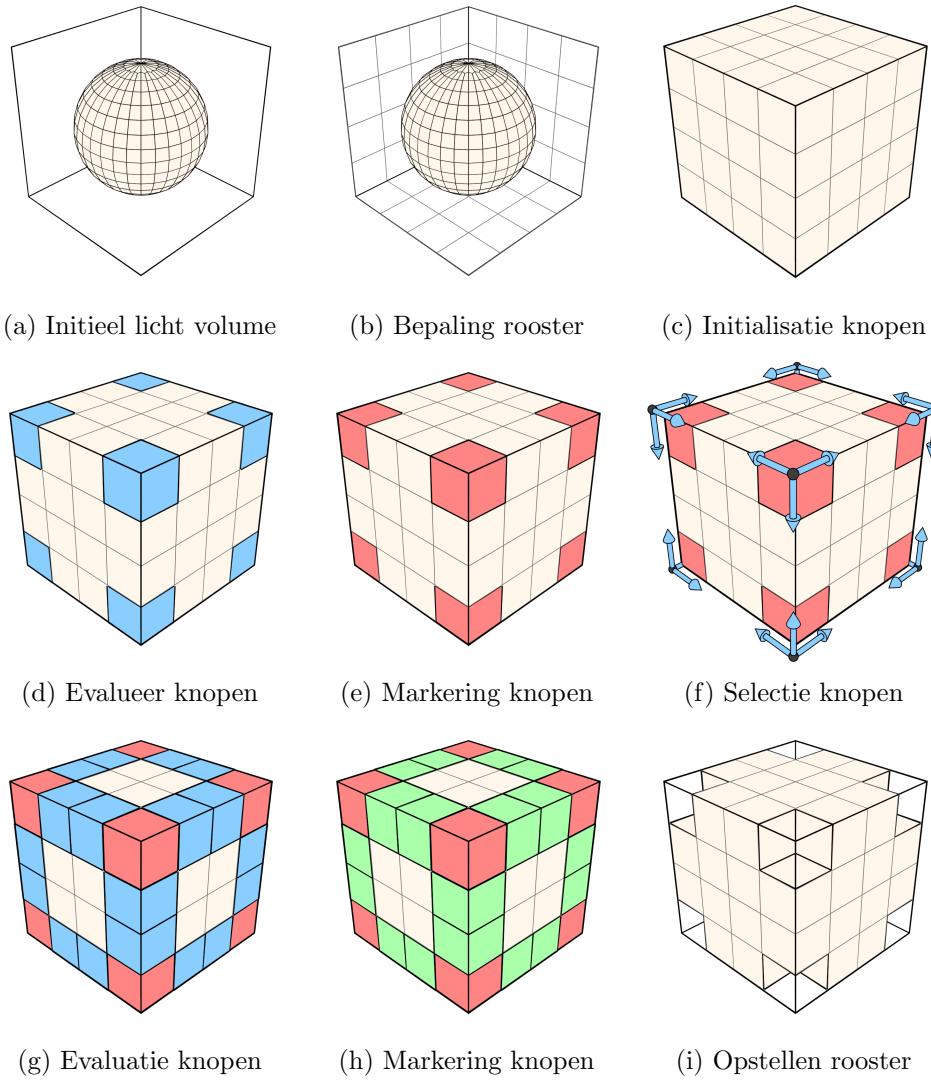
In het tweede geval zijn er twee mogelijkheden ofwel er liggen volle roosterknopen binnen de enkele lichtboomknoop, ofwel er liggen slechts lege roosterknopen in de enkele lichtboomknoop. Wanneer er slechts lege knopen binnen de lichtboomknoop liggen, is deze knoop zelf ook leeg, anders is het een takknoop.

Het punt binnen de lichtboomknoop dat het dichtstbij de lichtbronnoorsprong ligt, bevindt zich in de roosterknoop die overlapt met de lichtboomknoop en het dichtst bij de oorsprong van de lichtbron ligt. Indien deze roosterknoop leeg is, zijn alle andere roosterknopen tevens leeg. Dus om te bepalen of in het tweede geval een enkele lichtboomknoop een takknoop of een lege bladknoop is, dient slechtst gekeken te worden naar {'}e}{'}e}n specifieke roosterknoop.

In het laatste geval zijn er drie mogelijkheden

- De enkele lichtboomknoop omvat alleen lege roosterknopen.
- De enkele lichtboomknoop omvat alleen volle roosterknopen.
- De enkele lichtboomknoop omvat zowel lege als volle roosterknopen.

De enkele lichtboomknoop is dan respectievelijk een volle bladknoop, lege bladknoop of takknoop. Indien de roosterknoop dichtstbij de lichtbronnoorsprong binnen de lichtboomknoop leeg is, dan is de enkele lichtboom knoop tevens leeg. Dit volgt dezelfde redenering als in het geval van gedeeltelijke overlapping met het rooster. Indien deze roosterknoop vol is, dan is het mogelijk dat er ofwel slechts volle roosterknopen binnen de enkele lichtboomknoop vallen, ofwel dat er zowel lege als volle knopen zijn. Hiervoor wordt gekeken naar de roosterknoop binnen de lichtboomknoop die het verste weg van de lichtbron ligt. Indien deze ook vol is, dan is de enkele lichtboomknoop



Figuur 6.15: Opbouw van het initiele rooster van de enkele lichtboom.



Figuur 6.16: De lichtoctree.

tevens vol. Anders bevat de enkele lichtboomknoop zowel lege als volle knopen, en wordt deze gesteld op een takknoop.

Op basis van dit algoritme is het mogelijk om een enkele lichtboom op te stellen. Deze zal gebruikt worden om de volledige lichtoctree te construeren in de volgende sectie.

6.2.3 Licht octree

De lichtoctree beschrijft de octree-representatie van de set van alle lichten binnen de scène. Een illustratie van een lichtoctree is te vinden in figuur 6.16. Elke bladknoop bevat een lijst van lichten waarvan het lichtvolume (gedeeltelijk) overlapt met de bladknoop. Hierdoor is het mogelijk om de set van relevante lichten voor een punt \mathbf{p} te beperken van de set van alle lichten binnen de scène, tot de set van lichten die opgeslagen is in de bladknoop waartoe punt \mathbf{p} behoort.

De lichtoctree volgt een standaard pointer implementatie. Voordat deze bruikbaar is binnen de grafische kaart, dient de lichtoctree eerst omgezet te worden naar een verbindingoctree voorstelling.

Om de lichtoctree voor een scène te construeren is de volgende informatie nodig:

- De set van lichten die zich bevindt in de scène
- Een grootte voor de ribben van de kleinste mogelijke bladknoop

De oorsprong van de octree is gedefinieerd als het punt met de grootste mogelijke dimensies waarvoor geldt dat er geen lichtvolumes kleiner zijn dan dit punt.

$$\begin{aligned}
\mathbf{p}_o := \mathbf{p} - b : & (\forall l \in Sp. x < l.p.x - l.r \wedge \\
& p.y < l.p.y - l.r \wedge \\
& p.z < l.p.z - l.r) \wedge \\
& (\exists \mathbf{p}' (p'.x > p.x \vee \\
& p'.y > p.y \vee \\
& p'.z > p.z) \wedge \\
& (\forall l' \in Sp'. x < l.p.x - l.r \wedge \\
& p'.y < l.p.y - l.r \wedge \\
& p'.z < l.p.z - l.r))
\end{aligned}$$

Waarbij \mathbf{p}_o de oorsprong van de lichtoctree is b een offset-waarde is om afrondingsfouten te voorkomen.

Nadat de oorsprong is vastgesteld kan voor elk licht de corresponderende enkele lichtboom worden opgesteld. Deze lichtbomen zullen vervolgens iteratief worden toegevoegd aan aan een lichtoctree geïnitialiseerd met een lege bladknoop.

Het toevoegen van een enkele lichtboom vindt plaats in twee stappen. Eerst wordt afgedaald in de lichtoctree totdat de octreeknoop overeenkomend met de wortel van de enkele lichtboom is bereikt. Indien een bladknoop wordt tegengekomen wordt deze vervangen met een takknoop waarvan de kinderen dezelfde lichten bevatten als de bladknoop die vervangen wordt. Vervolgens wordt de wortel van de enkele lichtboom toegevoegd aan de bereikte octree knoop. Elke keer dat een lichtboomknoop wordt toegevoegd aan een octreeknoop zijn er vijf mogelijke situaties. Deze zijn weergegeven in tabel 6.1.

6.2.4 Verbindingloze octree

Nu de lichtoctree opgesteld is, is het mogelijk om deze voor te stellen als een verbindingloze octree. De structuur van de lichtoctree kan voorgesteld worden zoals beschreven in sectie 6.1.6.

Naast de structuur dient tevens, per gevulde bladknoop, bijgehouden te worden welke lichten van invloed zijn op de corresponderende ruimte. Omdat de gevulde bladknopen slechts een kleine subset van alle knopen zijn is er gekozen om deze data op te slaan in aparte hashtabellen. Hierbij is gekozen voor een vergelijkbare aanpak als in Tiled en Clustered shading. Er zijn drie verschillende datastructuren gespecificeerd:

- Een lijst van alle lichten in de scene.
- Een lijst van lichtindices
- De data-hashtabellen

De lijst van lichtindices bestaan uit integers die wijzen naar specifieke lichten in de lichtlijst. De data-hashtabellen bevat vervolgen per volle bladknoop in een

6. HASHED SHADING

Lichtoctreeknoop o		Enkele Lichtboomknoop l	
		Bladknoop	Takknoop
Leeg	Vol		
Bladknoop	Stop	De index van lichtboomknoop l wordt toegevoegd aan de lijst van indices van octreeknoop o	De octreeknoop o wordt vervangen door een takknoop o' waarvan de kinderen dezelfde indices bevatten als de octreeknoop o . De lichtboomknoop l wordt toegevoegd aan de nieuwe octreeknoop o' .
Takknoop	Stop	Lichtboomknoop l wordt toegevoegd aan elk van de kinderen van octreeknoop o .	Elk van de kinderen van lichtboomknoop l wordt toegevoegd aan het overeenkomstige kind van octreeknoop o .

Tabel 6.1: De mogelijke situaties wanneer een enkele lichtboomknoop wordt toegevoegd aan een octreeknoop.

laag een vector van twee integers. De eerste integer specificeert een beginpunt in de lichtindexlijst. De tweede integer specificeert het aantal lichten behorende bij de gevulde bladknoop. Op deze manier wordt een subset van de lichtindexlijst gespecificeerd. Deze subset komt overeen met alle indices relevant voor de gevulde bladknoop. Deze datastructuren zijn geïllustreerd in figuur ??

Wanneer een laag bestaat uit slechts een klein aantal knopen, zal de voorstelling als spatiale hashfunctie relatief veel geheugen gebruiken, door de overhead van de texturen. Om deze reden worden veelal de eerst i lagen niet explicet voorgesteld maar samengenomen in de eerste voorgestelde laag.

Het algoritme om een lichtoctree om te zetten naar een verbindingloze octree kan dan als volgt worden voorgesteld.

- Haal alle knopen uit de lichtoctree op behorende tot laag j , waarbij bladknopen in laag $j' < j$ worden opgesplits in kleinere bladknopen.
- Voor elke knoop in laag j , bereken de octreetabelwaarde, gespecificeerd als:



Figuur 6.17: Visuele weergave van de datastructuren binnen de constructie van de verbindingloze octree.

$$\text{octree_node}_i := \left(\sum_{k=0}^7 \text{is_leaf}(\text{sub_node}_k) \times 2^k, \sum_{k=0}^7 \text{is_empty}(\text{sub_node}_k) \times 2^k \right)$$

$$\text{is_leaf}(\text{node}) = \begin{cases} 1 & : \text{node is een blad knoop.} \\ 0 & : \text{anders} \end{cases}$$

$$\text{is_empty}(\text{node}) = \begin{cases} 1 & : \text{node is geen blad knoop, of bevat minimaal 1 relevant licht.} \\ 0 & : \text{anders} \end{cases}$$

- Indien een kind van een knoop in laag j
 - een takknoop is: Voeg deze toe aan de lijst van takknopen van laag $j + 1$.
 - een gevulde bladknoop is: Voeg deze toe aan de lijst van bladknopen van laag j .
- Bouw de octreetabel voor alle octree knopen
- Voor alle bladknopen bepaal de datatabelwaarde gespecificeerd als:

```
data_node = (len(light_indices), len(node.indices))
```

vervolgens worden de knoopindices toegevoegd aan de lijst van lichtindices.

- Bouw de datatabel voor alle data knopen
- Herhaal totdat de volgende laag geen takknopen meer bevat.

Na uitvoering zijn n octreetabellen, $m \leq n$ datatabellen en een lichtindexlijst opgesteld. Deze kunnen op de GPU geladen worden en gebruikt worden in de shaders.

6.2.5 Licht toekenning

Nu de verbindingloze octree is opgesteld kan deze gebruikt worden voor de lichttoekenning. Het berekenen van de relevante lichten voor een fragment \mathbf{p} komt neer op een afdaling in de verbindingloze octree. Voor elke stap in de afdaling dient eerste de octree beschrijving opgehaald te worden uit de verbindingloze octree. Deze is gedefinieerd als

$$\begin{aligned} (\text{is_leaf}_i, \text{is_empty}_i) &= H_i [\mathbf{k}_i + \Phi_i [\mathbf{k}_i] \mod m_i] \\ \mathbf{k}_i &= \lfloor * \mathbf{p}' / s_i \rfloor \end{aligned}$$

Hierbij is \mathbf{k}_i de positie in laag i , \mathbf{p}' de positie van fragment \mathbf{p} ten opzichte van de octree oorsprong $\mathbf{O}_{\text{octree}}$ en s_i de grootte van een knoop in laag i . De waarden `is_leaf` en `is_empty` beschrijven de structuur van de hashcluster behorende tot fragment \mathbf{p} en de hashclusters die behoren tot dezelfde ouder. Om de relevante waarde voor fragment \mathbf{p} te vinden dient eerst vastgesteld te worden tot welk kind fragment \mathbf{p} behoort. De positie binnens de ouder kan berekend worden als:

$$\mathbf{k}_{\text{local}} = \mathbf{k}_{i+1} - 2 * \mathbf{k}_i$$

waarbij de bit locatie van fragment \mathbf{p} gesteld kan worden als

$$\sum_{j=0}^2 \mathbf{k}_{\text{local},j} * 2^j$$

Als laatste kan de relevante bit met behulp van bitverschuivingen verkregen worden:

```
# bit masking of k-th bit in n
int mask = 1 << k
int masked_n = n & mask
int bit = masked_n >> k
```

Wanneer de waarden `is_leaf` en `is_empty` berekend zijn, zijn er 3 mogelijkheden:

- De knoop in laag i is een tak knoop, en de hashcluster wordt berekend voor laag $i + 1$.
- De knoop is een niet lege bladknoop. De hashcluster geassocieerd met fragment \mathbf{p} wordt opgehaald uit de datahashtabel geassocieerd met laag i .
- De knoop is een lege bladknoop en het hashcluster $(0, 0)$ wordt teruggegeven.

Wanneer het hash cluster gevonden is, kan de shading plaats vinden. Dit is exact hetzelfde als bij Tiled en Clustered shading. De pseudo code voor dit algoritme is weergegeven in lst 3. Verder is een illustratie van de bepaling van het relevante hash cluster voor punt \mathbf{p} weergegeven in figuur 6.18.

In deze sectie is een compleet overzicht gegeven van het hashed shading algoritme op basis van de verbindingloze octree. In de volgende sectie zal de efficiëntie worden bepaald aan de hand van de test scenes.

```

for (uint layer_i = 0;
    layer_i < OCTREE_DEPTH;
    layer_i++) {
    octree_coord_cur = octree_coord_next;

    next_node_size_den *= 2;
    octree_coord_next =
        ivec3(floor(fragment_octree_position *
            next_node_size_den));
    index_dif = octree_coord_next -
        octree_coord_cur * 2;
    index_int = index_dif.x +
        index_dif.y * 2 +
        index_dif.z * 4;

    octree_data =
        obtainNodeFromSpatialHashFunction(
            octree_coord_cur,
            octree_offset_tables[layer_i],
            octree_data_tables[layer_i]);

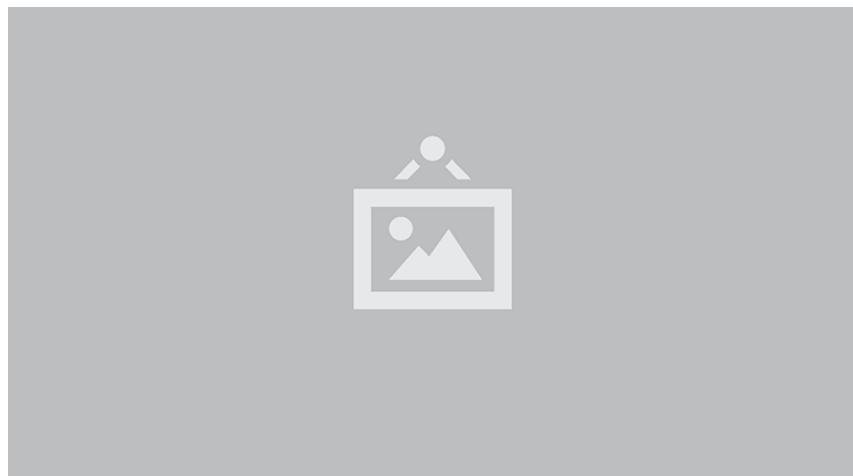
    if(extractBit(octree_data.x, index_int)) {
        if(extractBit(octree_data.y, index_int)) {
            light_data =
                obtainNodeFromSpatialHashFunction(
                    octree_coord_next,
                    light_offset_tables[layer_i],
                    light_data_tables[layer_i]);
        }
        break;
    }
}

uint offset = light_data.x;
uint n_lights = light_data.y;

for (uint i = offset; i < offset + n_lights; i++) {
    light_acc += computeLight(lights[light_indices[i]], param);
}

```

Listing 3: Lichttoekenning in de GLSL shader.



Figuur 6.18: Visuele representatie van het hashed shading algoritme.

6.3 Implementatie in nTiled

6.4 Testen en resultaten

6.5 Discussie

Hoofdstuk 7

Besluit

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] Bentley, J. L. Multidimensional binary search trees used for associative searching. *Communications of the ACM* 18, 9 (1975), 509–517.
- [4] Choi, M. G., Ju, E., Chang, J.-W., Lee, J., and Kim, Y. J. Linkless octree using multi-level perfect hashing. In *Computer Graphics Forum* (2009), vol. 28, Wiley Online Library, pp. 1773–1780.
- [5] Cormen, T. H. *Introduction to algorithms*. MIT press, 2009.
- [6] Czech, Z. J., Havas, G., and Majewski, B. S. Perfect hashing. *Theoretical Computer Science* 182, 1-2 (1997), 1–143.
- [7] Deering, M., Winner, S., Schediwy, B., Duffy, C., and Hunt, N. The triangle processor and normal vector shader: a vlsi system for high performance graphics. In *ACM SIGGRAPH Computer Graphics* (1988), vol. 22, ACM, pp. 21–30.
- [8] Engel, W. The light pre-pass renderer: Renderer design for efficient support of multiple lights. *SIGGRAPH Course: Advances in realtime rendering in 3D graphics and games* (2009).
- [9] Fox, E. A., Heath, L. S., Chen, Q. F., and Daoud, A. M. Practical minimal perfect hash functions for large databases. *Communications of the ACM* 35, 1 (1992), 105–121.
- [10] Fuchs, H., Poulton, J., Eyles, J., Greer, T., Goldfeather, J., Ellsworth, D., Molnar, S., Turk, G., Tebbs, B., and Israel, L. Pixel-planes 5: a heterogeneous multiprocessor graphics system using processor-enhanced memories. In *ACM Siggraph Computer Graphics* (1989), vol. 23, ACM, pp. 79–88.
- [11] Glassner, A. S. *Algorithms for efficient image synthesis*. PhD thesis, University of North Carolina at Chapel Hill, 1988.

- [12] Guttman, A. *R-trees: a dynamic index structure for spatial searching*, vol. 14. ACM, 1984.
- [13] Hecht, E. *Optics, Global Edition*. Always learning. Pearson Education, Limited, 2016.
- [14] Karis, B. Real shading in unreal engine 4. *Proc. ACM SIGGRAPH Courses* (2013), 22.
- [15] Kleinberg, J., and Tardos, E. *Algorithm design*. Pearson Education India, 2006.
- [16] Lauritzen, A. Deferred rendering for current and future rendering pipelines. *SIGGRAPH Course: Beyond Programmable Shading* (2010), 1–34.
- [17] Lefebvre, S., and Hoppe, H. Perfect spatial hashing. In *ACM Transactions on Graphics (TOG)* (2006), vol. 25, ACM, pp. 579–588.
- [18] Magnusson, K. Lighting you up in battlefield 3. In *Proc. 25th Annual Game Developers Conference* (2011).
- [19] Meagher, D. Geometric modeling using octree encoding. *Computer graphics and image processing* 19, 2 (1982), 129–147.
- [20] Mittring, M. A bit more deferred–cryengine 3. In *Triangle Game Conference* (2009), vol. 4.
- [21] Naylor, B. F. A tutorial on binary space partitioning trees. In *Computer Games Developer Conference Proceedings* (1998), pp. 433–457.
- [22] Olsson, O., and Assarsson, U. Tiled shading. *Journal of Graphics, GPU, and Game Tools* 15, 4 (2011), 235–251.
- [23] Perlin, K. An image synthesizer. *ACM Siggraph Computer Graphics* 19, 3 (1985), 287–296.
- [24] Pharr, M., Jakob, W., and Humphreys, G. *Physically Based Rendering: From Theory to Implementation*. Elsevier Science, 2016.
- [25] Pranckevičius, A., and Dude, R. Physically based shading in unity. In *Game Developer’s Conference* (2014).
- [26] Saito, T., and Takahashi, T. Comprehensible rendering of 3-d shapes. In *ACM SIGGRAPH Computer Graphics* (1990), vol. 24, ACM, pp. 197–206.
- [27] Suffern, K. *Ray Tracing from the Ground Up*. Taylor & Francis, 2007.
- [28] Swoboda, M. Deferred lighting and post processing on playstation 3. In *Game Developer Conference* (2009).

- [29] Tebbs, B., Neumann, U., Eyles, J., Turk, G., and Ellsworth, D. Parallel architectures and algorithms for real-time synthesis of high quality images using deferred shading. Tech. rep., DTIC Document, 1989.
- [30] Valient, M. The rendering technology of killzone 2. In *Game Developers Conference* (2009).
- [31] Watkins, G. S. A real time visible surface algorithm. Tech. rep., DTIC Document, 1970.
- [32] 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: Forwaards en deferred hashed shading

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