

Forward en Deferred Hashed Shading voor het Realtime Renderen van Grote Aantallen Lichtbronnen

Martinus Wilhelmus Tegelaers

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

Promotor:

Prof. dr. ir. Philip Dutré

Assessor:

Ir. Tuur Stuyck
Dr. Dominique Devriese

Begeleider:

Ir. Matthias Moulin
Ir. Jeroen Baert

© 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

Als eerste wil ik mijn promotor Philip Dutré bedanken, niet alleen voor zijn rol binnen mijn thesis, maar ook voor het aanwakkeren van mijn passie voor computergrafieken. Door zijn colleges ambieer ik een carrière binnen dit vakgebied.

Veel dank gaat ook uit naar mijn begeleiders, eerst Jeroen Baert en daarna Matthias Moulin. Ik was niet altijd de makkelijkste student en zonder hun geduld en ondersteuning zou ik hier niet gestaan hebben. Verder ook dank aan de voltallige jury.

Als laatst wil ik mijn familie en vrienden bedanken. Mijn ouders en zus hebben mij gedurende mijn hele studie ondersteund en zonder hen was ik waarschijnlijk het spoor bijster geraakt. Daarnaast heeft mijn vriendin mij door de jaren heen gesteund. Als laatste wil ik mijn vrienden in zowel België als Nederland bedanken. Jullie maakten de slechte momenten draagbaar, en de goede momenten des te beter.

Martinus Wilhelmus Tegelaers

Inhoudsopgave

Voorwoord	i
Samenvatting	iv
Lijst van figuren en tabellen	v
Lijst van afkortingen en symbolen	ix
1 Inleiding	1
1.1 Situering	1
1.2 Doelstelling	2
1.3 Methodologie	2
1.4 Contributie	3
1.5 Overzicht	3
2 Theorie	5
2.1 Fysische werkelijkheid	5
2.2 Virtuele voorstelling	10
2.3 Perspectiefprojectie en het visibiliteitsprobleem	12
2.4 Shading	17
2.5 Moderne Grafische Pijplijn	22
2.6 Besluit	28
3 Methode-overzicht	29
3.1 Software	29
3.2 Hardware	32
3.3 Testsuite	32
3.4 Data-analyse	36
4 Forward en Deferred Shading	37
4.1 Theorie	38
4.2 Algoritme	40
4.3 Resultaten	42
4.4 Conclusie	45
4.5 Discussie	47
5 Tiled Shading	49
5.1 Theorie	49
5.2 Algoritme	51
5.3 Resultaten	54

5.4 Conclusie	63
5.5 Discussie	64
6 Clustered Shading	67
6.1 Theorie	68
6.2 Algoritme	72
6.3 Resultaten	78
6.4 Conclusie	82
6.5 Discussie	84
7 Hashed Shading	85
7.1 Theorie	85
7.2 Algoritme	98
7.3 Resultaten	111
7.4 Conclusie	131
7.5 Discussie	131
8 Besluit	133
8.1 Vergelijking van de resultaten	133
8.2 Verder onderzoek	135
A Poster	149
B Paper	153
Bibliografie	169

Samenvatting

In deze thesis wordt een nieuw lichttoekenningsalgoritme geïntroduceerd in de context van realtime rendering op basis van de verbindingloze octree. Het doel hiervan is om een betere performantie te verkrijgen dan de huidige lichttoekenningsalgoritmes Tiled en Clustered Shading, door de opgestelde datastructuren tussen frames te hergebruiken. Hiervoor wordt de ruimte opgedeeld aan de hand van een een camera-onafhankelijke octree. Veranderingen in de scène zijn relatief klein tussen frames, waardoor de kost van het aanpassen van de datastructuur ook klein is.

Om de performantie van dit nieuwe algoritme te evalueren is gekeken naar de uitvoeringsstijd en het aantal lichtberekeningen per frame bij verschillende scènes, resoluties en aantallen lichtbronnen. Ter referentie zijn ook de lichttoekenningsalgoritmes Tiled en Clustered Shading geïmplementeerd binnen hetzelfde programma. De resultaten van deze implementaties zijn vergeleken.

De Hashed Shading implementatie vereist de helft van het aantal lichtberekeningen per frame ten opzichte van Tiled Shading. Hierdoor is de Forward Hashed Shading implementatie een factor twee sneller dan Forward Tiled Shading. De beperking in het aantal lichtberekeningen met Hashed Shading met de kleinst geëvalueerde knopen komt overeen met de reductie van het aantal lichtberekeningen behaald binnen Clustered Shading. Daarnaast schaalt Hashed Shading in kleine mate beter met de resoluties, doordat opdeling hier niet direct afhankelijk van is.

Daarnaast vereist Hashed Shading geen complete herberekening van de datastructuren bij een verandering van het camerastandpunt, waardoor de kost ten opzichte van Tiled en Clustered Shading afneemt. Op basis hiervan kan gesteld worden dat het doel van deze thesis bereikt is. Er zijn echter nog wel enkele beperkingen in de huidige implementatie van Hashed Shading. Op dit moment is slechts gekeken naar statische lichten. Het ondersteunen van dynamische lichten vereist wel dat de datastructuren van Hashed Shading bijgewerkt worden. Dit zal een extra kost met zich mee brengen. Echter doordat de verschillen tussen twee opeenvolgende frames klein zullen zijn, zal deze kost beperkt blijven. Daarnaast schaalt het geheugengebruik slecht met de knoopgrootte, waardoor veel geheugen gebruikt wordt bij kleine knoopgroottes. Voor beide beperkingen worden oplossingen voorgesteld.

Lijst van figuren en tabellen

Lijst van figuren

1.1	Progressie van grafische kwaliteit in de Halo series[Bun01]	2
2.1	Waarneming door middel van het oog en camera.	6
2.2	Absorptie, reflectie en transmissie van licht.	7
2.3	Het mengen van kleuren volgens een additief model.	8
2.4	Voorstelling van objecten door middel van driehoeken.	11
2.5	Het Cameramodel.	12
2.6	Perspectiefprojectie.	13
2.7	Visibiliteitsprobleem in een scène met meerdere primitieven.	13
2.8	Mogelijkheden raytrace-algoritmes	14
2.9	Raytrace-algoritme.	16
2.10	Het rasterisatie-algoritme.	16
2.11	Lambertiaanse BRDF.	19
2.12	Afstandsdempingscurves voor eindige omnipuntlichtbronnen.	20
2.13	Voorstelling van een puntlicht.	22
2.14	De stappen van zowel de OpenGL als Direct3D implementaties.	25
3.1	Een oppervlakte gerenderd met de standaard lambertshader binnen nTiled.	32
3.2	De Spaceship Indoor scène.	33
3.3	De Piper's Alley scène.	34
3.4	De Ziggoerat stadsscene.	35
3.5	Het Cubehelix kleurenpalet lopende van 0 tot 1.	36
4.1	Een scène met een grote hoeveelheid verborgen geometrie.	37
4.2	De texturen in de GBuffer gebruikt in Killzone 2, geproduceerd door Guerrilla Games[Val09].	40
4.3	De texturen in de GBuffer gebruikt in nTiled.	41
4.4	Overzicht van de uitvoeringstijd per frame voor de drie testscènes bij verschillende resolutie en aantal lichten.	43
4.5	Spaceship Indoor frames waarbij het aantal fragmenten één benadert.	44
4.6	Ziggurat City frames waarbij de camera van positie verspringt.	44
4.7	Aantal lichten.	46

LIJST VAN FIGUREN EN TABELLEN

4.8	Resolutie.	46
5.1	Opdeling van het zichtveld.	50
5.2	Het Tiled Shading algoritme voor de Forward en Deferred pijplijn.	51
5.3	De datstructuren van Tiled Shading.	52
5.4	Projectie van lichtbol op de a -as.	53
5.5	Overzicht van de uitvoeringstijd voor Forward Shading per frame voor de drie testscènes bij verschillende resoluties en aantallen lichten.	55
5.6	Overzicht van de uitvoeringstijd voor Deferred Shading per frame voor de drie testscènes bij verschillende resoluties en aantallen lichten.	56
5.7	Overzicht van het aantal lichtberekeningen per frame voor Deferred Shading voor de drie testscènes bij verschillende resoluties en aantallen lichten.	57
5.8	Renders en hittekaarten van de verschillende scènes voor Tiled Shading.	58
5.9	Overzicht van de uitvoeringstijd per aantal lichten bij een resolutie van 2560^2 pixels voor de drie testscènes.	59
5.10	Lichtberekeningen.	60
5.11	Lichtroosteropbouw.	60
5.12	Overzicht van de uitvoeringstijd per resolutie voor de drie testscènes.	61
5.13	Lichtberekeningen.	62
5.14	Lichtroosteropbouw.	62
6.1	Clustering op basis van diepte, en diepte en normaal[OBA12].	68
6.2	Opdeling van het zichtfrustum	69
6.3	Normaalkegel op basis van een eenheidskubus.	70
6.4	Lichtruiming op basis van de normaalkegel.	71
6.5	Het Clustered Shading algoritme voor de Forward en Deferred pijplijn. .	72
6.6	De sorteer- en comprimeerstap uitgevoerd over een enkel vlak.	74
6.7	De datastructuren gebruikt binnen Clustered Shading.	76
6.8	Overzicht van de uitvoeringstijd voor Deferred Clustered Shading per frame voor de drie testscènes bij verschillende resoluties en aantallen lichten.	79
6.9	Overzicht van het aantal lichtberekeningen per frame voor Deferred Shading voor de drie testscènes bij verschillende resoluties en aantallen lichten.	80
6.10	Renders en hittekaarten van de verschillende scènes voor Clustered Shading.	81
6.11	Lichtberekeningen.	83
6.12	Resolutie.	83
7.1	De onderverdelingen binnen Tiled en Clustered Shading	86
7.2	Weergave van een octree bestaande uit drie lagen, links is de 3d representatie weergegeven, rechts de pointerrepresentatie	88
7.3	Afbeelding van U op M	90
7.4	De perfecte spatiale hashfunctie $h(\mathbf{p})$ in twee dimensies.	92

7.5	De toekenning van een enkele offset-waarde om de perfecte spatiale hashfunctie $h(\mathbf{p})$ op te bouwen.	94
7.6	De verschillende knopen in een octree per laag.	97
7.7	Een voorbeeld van de representatie van een octree met behulp van een verbindingloze octree.	98
7.8	Overzicht van de datastructuren gebruikt in Hashed Shading	99
7.9	Een voorbeeld van de opdeling van de scène met behulp van een octree.	100
7.10	Bepaling of een knoop overlapt met het lichtvolume.	102
7.11	Opbouw van het minimale rooster voor een enkel licht.	104
7.12	Voorstelling van een enkele licht als octree.	105
7.13	De beslissingsboom om het type van een octreeknoop te bepalen.	106
7.14	Een overzicht van het algoritme om de datastructuren van Hashed Shading op te bouwen.	108
7.15	Overzicht van de constructietijd van de verbindingloze octree bij verschillende seed-waardes.	111
7.16	Overzicht van het geheugengebruik van de verbindingloze octree bij verschillende seed-waardes.	112
7.17	Diepte van de octree als functie van de knoopgrootte.	114
7.18	Constructietijd van de Hashed Shading datastructuren.	114
7.19	Constructietijd per stap als functie van de knoopgrootte.	115
7.20	Aantal pixels.	116
7.21	Aantal lichtindices.	116
7.22	Geheugen gebruik per laag van de Hashed Shading datastructuren als functie van de knoopgrootte.	117
7.23	Uitvoeringstijd: 320^2 pixels.	119
7.24	Lichtberekeningen: 320^2 pixels.	119
7.25	Uitvoeringstijd: 2560^2 pixels.	120
7.26	Lichtberekeningen: 2560^2 pixels.	120
7.27	Aantal pixels.	121
7.28	Aantal lichtindices.	121
7.29	Visualisatie van het aantal lichtberekeningen bij verschillende knoopgroottes relatief aan het naïeve Deferred Shading algoritme.	122
7.30	Constructietijd als functie van de begindiepte.	123
7.31	Aantal pixels als functie van de begindiepte.	123
7.32	Aantal pixels als functie van de begindiepte.	124
7.33	Aantal lichtindices als functie van de begindiepte.	124
7.34	De uitvoeringstijd per frame voor Forward Shading.	126
7.35	De uitvoeringstijd per frame voor Deferred Shading.	126
7.36	Het aantal lichtberekeningen voor Deferred Shading.	127
7.37	De uitvoeringstijd per frame als functie van het aantal lichten voor Forward Shading.	128
7.38	De uitvoeringstijd per frame als functie van het aantal lichten voor Deferred Shading.	128
7.39	De uitvoeringstijd per frame als functie van de resolutie voor Forward Shading.	129
7.40	De uitvoeringstijd per frame als functie van de resolutie voor Deferred Shading.	129

LIJST VAN FIGUREN EN TABELLEN

7.41 Het aantal lichtberekeningen als functie van het aantal lichten.	130
7.42 Het aantal lichtberekeningen als functie van de resolutie.	130
8.1 Reductie van het aantal lichtknopen met behulp van de geometrie.	136
8.2 Opdeling van de scèneruimte.	137
8.3 Opdelingstrategie van diepere lagen van de verbindingloze octree.	138
8.4 2D weergave van de mogelijke transformaties van puntlichten en hun invloed op de octree structuur.	143

Lijst van tabellen

2.1 Radiometrische grootheden en eenheden.	8
7.1 De mogelijke situaties wanneer een lichtoctreeknoop wordt toegevoegd aan een scène-octreeknoop.	107
8.1 De mogelijke situaties wanneer een lichtindex wordt toegevoegd aan een octreeknoop.	141
8.2 De mogelijke situaties wanneer een lichtindex wordt verwijderd uit een octreeknoop.	141

Lijst van afkortingen en symbolen

Lijst van symbolen

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}
Ruimte	Hoofdletters	U
Verzameling	Hoofdletters	S
Functie	Cursieve (kleine) letters	f

Operator	Notatie	Voorbeelden
Floor	$\lfloor \dots \rfloor$	$\lfloor x \rfloor$
Ceil	$\lceil \dots \rceil$	$\lceil x \rceil$
Klem	$\dots _{[a,b]}$	$x _{[0,1]}$

Hoofdstuk 1

Inleiding

1.1 Situering

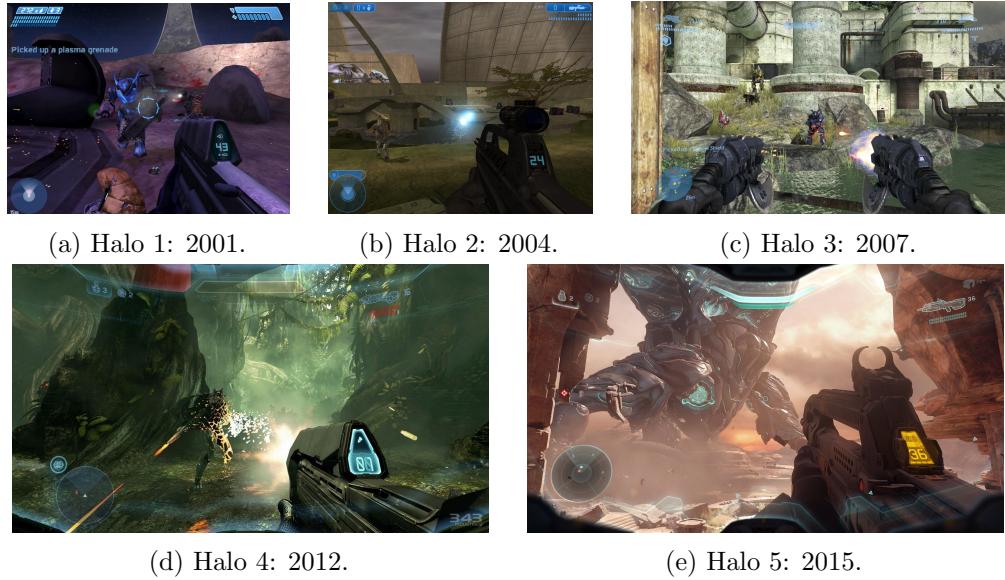
Realtime 3D grafische applicaties zijn niet meer weg te denken uit de moderne maatschappij. De bekendste categorie van realtime 3D grafische applicaties zijn games, maar ook buiten de entertainment industrie wordt er veelvuldig gebruik gemaakt van dergelijke applicaties. Denk hierbij aan interactieve productvisualisaties, interactieve simulaties, en opkomende technologien zoals Augmented en Virtual Reality.

Onder realtime 3D grafische applicaties worden alle applicaties verstaan die in realtime 3D data omzetten naar beelden. Om de illusie van bewegende beelden te creëren, worden deze beelden veelal met een framerate van 24 tot 60 frames per seconde gegenereerd. Deze beelden worden niet van te voren berekend, waardoor de applicatie in staat moet zijn om een beeld binnen 42 tot 17 milliseconden op te stellen. Dit maakt het mogelijk om interactief om te gaan met de 3D wereld die afgebeeld wordt.

In onder andere games is het doel veelal om de 3D wereld met een vorm van realisme weer te geven. Om de immersie te vergroten is er een constante vraag om de wereld met een grotere grafische getrouwheid weer te geven: de wereld dient in een hogere resolutie weergegeven te worden, complexere grafische fenomenen moeten gesimuleerd worden, en Scènes dienen verlicht te worden met meer lichtbronnen. Dit alles leidt tot steeds complexere berekeningen, die meer middelen vereisen om in realtime uitgevoerd te worden. Deze trend in grafische vooruitgang is duidelijk terug te zien wanneer gekeken wordt naar games die de afgelopen jaren ontwikkeld zijn. In Figuur 1.1 zijn screenshots van de games Halo 1 tot en met Halo 5 en hun jaar van publicatie weergegeven. Zowel de toename in complexiteit van de wereld als de belichting is duidelijk zichtbaar. Het berekenen van dergelijke complexe scènes in realtime wordt enerzijds mogelijk gemaakt door de toename in beschikbare rekenkracht en geheugen. Anderzijds worden algoritmes vereist die de hoeveelheid werk die verzet moet worden minimaliseren.

Om een afbeelding te genereren van een 3D wereld dient per pixel de kleur berekend te worden. Dit kan gedaan worden door per lichtbron het effect op een pixel na te gaan. Wanneer deze effecten gesommeerd worden, wordt de kleur van een pixel verkregen.

1. INLEIDING



Figuur 1.1: Progressie van grafische kwaliteit in de Halo series[Bun01].

Om de hoeveelheid werk te beperken kan gekeken worden welke lichtbronnen een invloed hebben op een pixel. Lichtbronnen die geen invloed hebben hoeven vervolgens niet geëvalueerd te worden voor deze pixel. De klasse van algoritmes die nagaan welke lichten relevant zijn voor pixels worden lichttoekenningsalgoritmes genoemd. Twee veelgebruikte lichttoekenningsalgoritmes zijn Tiled Shading[OA11] en Clustered Shading[OBA12]. Deze algoritmes bepalen per frame de relevante lichtbronnen per pixel.

1.2 Doelstelling

Het doel van deze thesis is om te evalueren of het hergebruik van lichttoekenningsdatastructuren tussen frames uitgebuit kan worden om een performantieverbetering te verkrijgen ten opzichte van Tiled en Clustered Shading. Hiervoor dient een nieuw lichttoekenningsalgoritme ontwikkeld te worden, waarin de datastructuren hergebruikt worden. Dit algoritme dient minimaal een vergelijkbare versnelling en reductie in aantal lichtberekeningen te realiseren als Tiled en Clustered Shading.

1.3 Methodologie

Om de performantie van het nieuwe lichttoekenningsalgoritme te evalueren zijn dit algoritme, en de Tiled en Clustered Shading algoritmes geïmplementeerd in één programma. Zowel de uitvoeringstijd en het aantal lichtberekeningen van Hashed Shading zijn vergeleken met die van Tiled en Clustered Shading, zodat een accuraat inzicht in de performantie wordt verkregen.

1.4 Contributie

Binnen deze thesis wordt een nieuw lichttoekenningsalgoritme voorgesteld op basis van de octree datastructuur. Doordat deze octree onafhankelijk van het zichtfrustum is, is het mogelijk om de datastructuren te hergebruiken tussen frames, ongeacht of het camerastandpunt verandert. Verder worden uitbreidingen voorgesteld om het geheugengebruik van dit nieuwe algoritme terug te dringen.

1.5 Overzicht

Hoofdstuk 2: Theorie Binnen het theorie hoofdstuk wordt een algemene inleiding gegeven tot 3D computergrafieken. Hierbij wordt ingegaan op de visibiliteit, shading, en realtime computergrafieken. Tevens wordt de gebruikte terminologie en wiskunde behandeld. Als laatste zal aan de hand van de geïntroduceerde concepten de probleemstelling en doel van de thesis in meer detail worden gedefinieerd.

Hoofdstuk 3: Methode-overzicht Binnen het methode-overzicht wordt ingegaan op de gebruikte software en testscènes. De implementatie van de ontwikkelde software wordt toegelicht. Als laatste zal ingegaan worden op de data-analyse.

Hoofdstuk 4: Forward en Deferred Shading Binnen Forward en Deferred Shading wordt een veelgebruikte methode geïntroduceerd om de geometrische complexiteit te ontkoppelen van de shading berekening.

Hoofdstuk 5: Tiled Shading Binnen Tiled Shading wordt het eerste veelgebruikte lichttoekenningsalgoritme geïntroduceerd. De resultaten van dit lichttoekenningsalgoritme worden vergeleken met de resultaten van een implementatie zonder versnellingsstructuren.

Hoofdstuk 6: Clustered Shading Binnen Clustered Shading wordt het tweede veelgebruikte lichttoekenningsalgoritme geïntroduceerd. De resultaten worden vergeleken met die van Tiled Shading en de naïeve implementatie.

Hoofdstuk 7: Hashed Shading Binnen het Hashed Shading hoofdstuk wordt het lichttoekenningsalgoritme geïntroduceerd dat binnen deze thesis is ontwikkeld. Hiervoor wordt eerst ingegaan op de achterliggende literatuur. Vervolgens worden het algoritme en de implementatie behandeld. Daarna worden de resultaten van Hashed Shading vergeleken met de resultaten van zowel Tiled en Clustered Shading als de naïeve implementatie.

Hoofdstuk 8: Besluit In het besluit wordt ingegaan op de verschillende conclusies getrokken in de voorgaande hoofdstukken. Daarna wordt mogelijk verder onderzoek toegelicht.

Hoofdstuk 2

Theorie

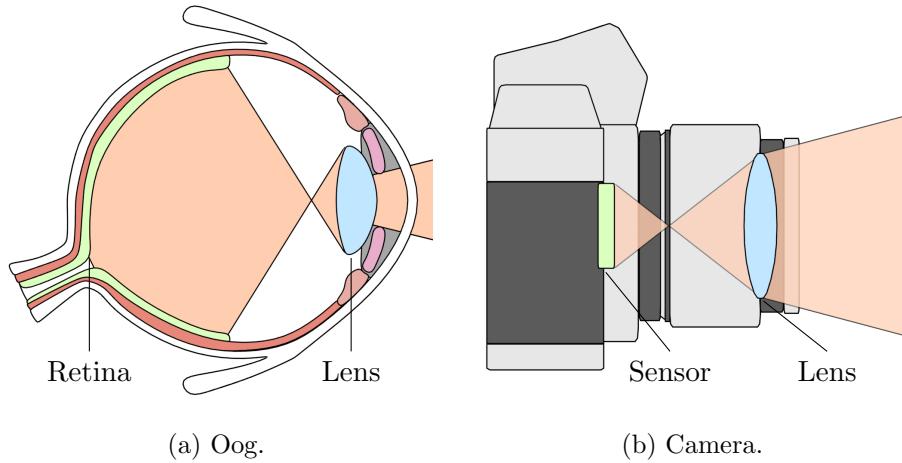
Zoals besproken in de inleiding draait de thesis om het renderen van driedimensionale scènes in realtime. Het doel is hierbij veelal om geloofwaardige afbeeldingen te creeren uit een bepaalde driedimensionale scène beschrijving. In veel gevallen betekent dit dat de scène fotorealistisch afgebeeld dient te worden, echter andere stijlistische keuzes zijn tevens mogelijk. 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 afbeeldingen 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 hardwareniveau geïmplementeerd is.

2.1 Fysische werkelijkheid

De fysische wereld waarin de mens zich bevindt, wordt gedicteerd door alle fysische wetten. De mens neemt deze wereld waar door middel van zintuigen. Voor de computergrafieken is waarneming het belangrijkste zintuig. Door middel van waarneming wordt de driedimensionale 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, bepaalt dus de fysische werkelijkheid.

2.1.1 Waarneming

De mens neemt de wereld waar door de ogen. Het menselijk oog interpreert de driedimensionale wereld door stralen van licht te focussen op een enkel punt, met behulp van een lens. Het enkele punt dat licht omzet naar neurosignalen wordt de retina genoemd[WKL⁺15]. Een camera bootst het oog na, en projecteert licht op een elektronische fotosensor, die het licht omzet naar een digitaal signaal. Dit is weergegeven in figuur 2.1.



Figuur 2.1: Waarneming door middel 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.

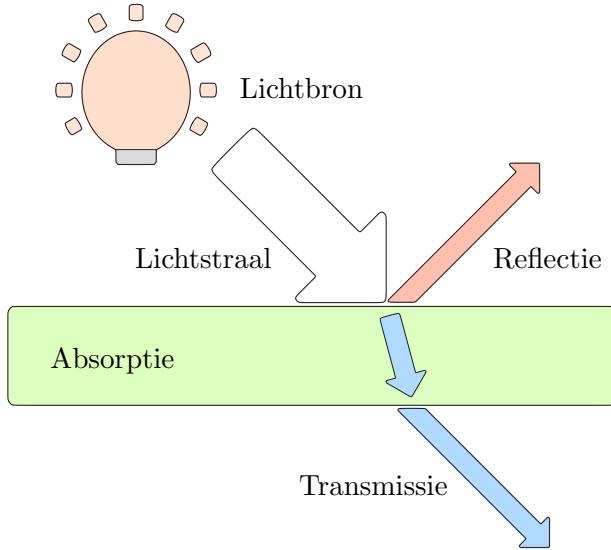
De mens verwacht dat deze eigenschappen aanwezig zijn, om beelden te interpreteren.

2.1.2 Licht

Het tweede belangrijke inzicht met betrekking tot 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 een vorm van elektromagnetische straling. Dit betekent dat de fysische wetten met betrekking tot elektromagnetische straling ook van invloed zijn op licht. Voor de computergrafieken 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 zich onder normale omstandigheden altijd in een rechte lijn voortbewegen, zolang het binnen hetzelfde medium blijft. Wanneer het licht in contact komt met een nieuw medium zijn er verschillende fenomenen die kunnen plaatsvinden:

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



Figuur 2.2: Absorptie, reflectie en transmissie van licht.

Reflectie Het licht wordt gereflecteerd op het oppervlak van het nieuwe medium. Hierbij wordt het licht terug de scène ingestuurd. De hoek van reflectie hangt af van het type medium. Indien het materiaal zich gedraagt als een ideale spiegel zal het licht teruggekaatst worden met dezelfde hoek gespiegeld aan de oppervlakte 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 aangepaste richting op basis van de brekingsindex van het nieuwe en het oude medium.

Deze fenomenen zijn geïllustreerd 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.

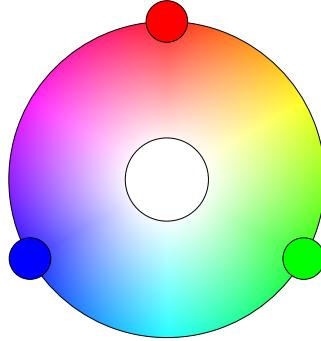
Zoals eerder vermeld is, neemt het oog de wereld waar door licht op te vangen. Het merendeel van het licht dat opgevangen wordt is gereflecteerd via een of meerdere oppervlaktes. Een belangrijke constatering is dat een object slechts zichtbaar is als er geen occlusie van het object is. Dit is een triviale constatering in de fysische werkelijkheid. Echter dit zal niet-triviale consequenties hebben binnen de computergrafieken zoals later zal worden beschreven.

Om de interactie van licht te simuleren is het van belang dat licht meetbaar is. Er zijn hiervoor twee verzamelingen van eenheden, radiometrie en fotometrie[Suf07]. Binnen radiometrie wordt de lichtkracht over alle golflengte gemeten. Bij fotometrie wordt deze kracht gewogen, aan de hand van het gestandaardiseerde model voor de perceptie van helderheid. Fotometrie is van belang binnen de computergrafieken

2. THEORIE

Grootheid	Eenheid
Stralingsenergie (Q)	Joule (J)
Stralingsflux (Φ)	Watt (W)
Irradiantie (E)	Watt per vierkante meter ($\text{W}\cdot\text{m}^{-2}$)
Radiantie (L)	Watt per vierkante meter per ruimtehoek ($\text{W}\cdot\text{m}^{-2}\cdot\text{sr}^{-1}$)

Tabel 2.1: Radiometrische grootheden en eenheden.



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

omdat het inzicht geeft in de perceptie van de mens. Binnen deze thesis zal slechts kort ingegaan worden op radiometrie.

De belangrijkste termen van Radiometrie zijn opgesteld in tabel 2.1. De stralingsenergie Q is de basiseenheid van elektromagnetische straling. Deze energie bevindt zich in de fotonen. Flux Φ beschrijft de stralingsenergie die zich door een ruimte per seconde heen verplaatst. Binnen de computergrafieken is de belangrijkste eenheid radiantie L . Dit is de stralingsflux van fotonen die zich beweegt door een kegel. Wanneer de grootte van de basis van de kegel nul benaderd, bewegen de fotonen zich in een rechte lijn voort. Radiantie heeft de volgende eigenschappen die van belang zijn bij de simulatie van licht:

- Radiantie is constant langs een straal die zich voortplant door vacuüm. Tevens is de binnenkomende straal gelijk aan de uitgaande straal.
- Indien het meetpunt op een oppervlak wordt genomen, maakt het niet uit of de flux ingaand, of uitgaand is. Het maakt zelfs niet uit of de flux geabsorbeerd, gereflecteerd, of doorgelaten wordt door het materiaal.

De radiantie kan dus gebruikt worden om de kleur van een oppervlakte te bepalen.

2.1.3 Kleur

Een tweede belangrijk aspect van licht voor computergrafieken is het concept kleur. Kleur is niet een fysisch verschijnsel, maar een gevolg van hoe ogen licht interpreteren. De mens neemt slechts een gedeelte van al het EM-spectrum 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. Indien verschillende lichten op hetzelfde punt worden afgebeeld, zal dit worden waargenomen als een licht gelijk aan de optelling van de individuele lichten.

Objecten kunnen tevens een kleur hebben. Reeds is besproken dat objecten worden waargenomen door de reflectie van licht op het oppervlak van het object. De kleur van een object is het gevolg van de gedeeltelijk absorptie van het licht dat op het oppervlak valt. 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 weerspiegeld worden. De frequenties tegenovergesteld aan de kleur van het object, zullen worden geabsorbeerd door het object.

2.1.4 Animatie

Beweging kan gesimuleerd worden door statische beelden snel genoeg na elkaar af te beelden. Deze illusie wordt *schijnbare beweging* genoemd[WKL⁺15]. In de computergrafieken wordt hier dankbaar gebruik van gemaakt om bewegende beelden te creëren.

Om de illusie van beweging te creëren, dienen de beelden met een hoge framerate weergegeven te worden. Veelal worden hier framerates tussen de 20 en 60 frames per seconde voor gebruikt. Een hogere framerate leidt tot een vloeindere beweging.

2.1.5 Simulatie

Binnen de computergrafieken is het doel veelal om de waarneming van 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 worden opgedeeld:

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

Het eerste probleem kan worden opgelost met behulp van perspectief projectie. Het tweede probleem wordt het visibiliteitsprobleem genoemd.

2. THEORIE

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.

Voor een uitgebreidere beschrijving van de behandelde onderwerpen kan gerefereerd worden naar de volgende boeken: Voor de psychologische grondslag van kleur en perceptie, kan gerefereerd worden naar Wolfe's *Sensation and Perception*[WKL⁺15]. De fysica die ten grondslag ligt aan computergrafieken kan gevonden worden in *Optics* door Eugene Hecht [Hec16].

2.2 Virtuele voorstelling

Om een afbeelding te creëren van een virtuele driedimensionale omgeving die de fysische werkelijkheid benadert, is het in de eerste plaats nodig om een beschrijving te hebben van deze omgeving. Daarnaast is het nodig om een virtuele camera te definiëren waarmee deze virtuele wereld wordt waargenomen.

2.2.1 Voorstelling van de geometrie

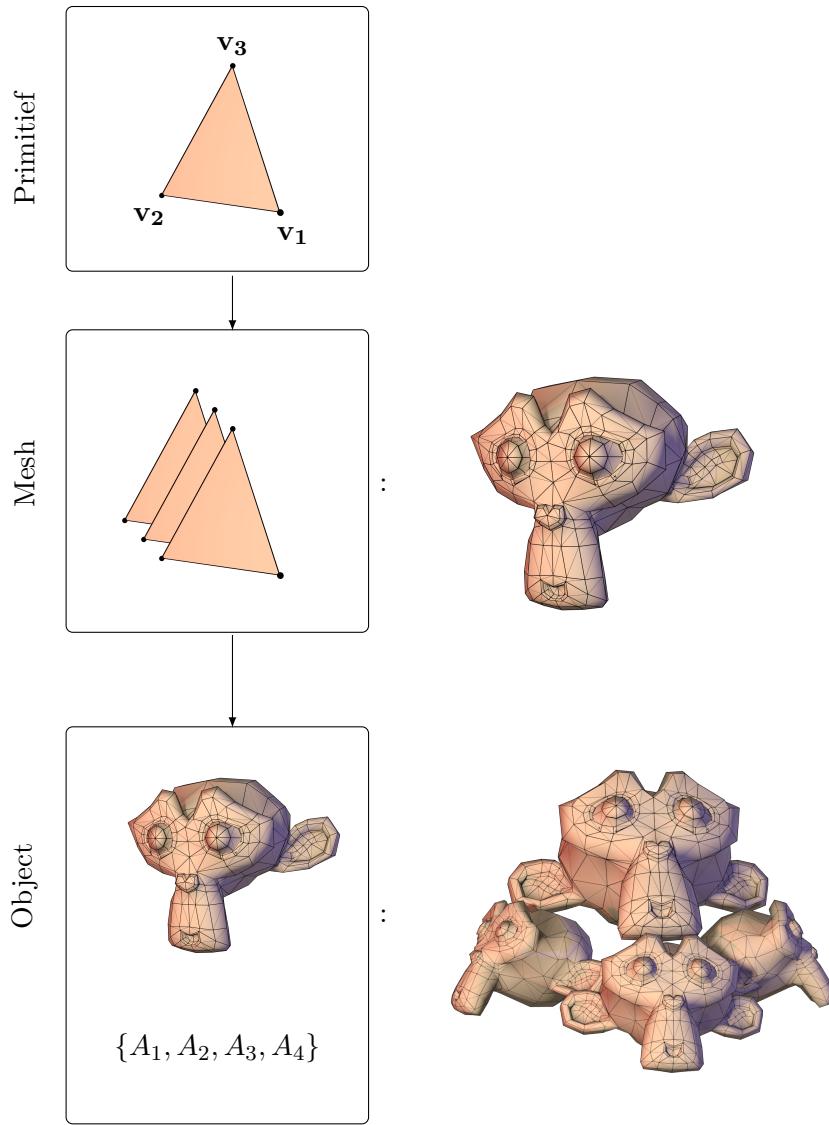
Om een omgeving voor te stellen wordt een verzameling van primitieven gedefinieerd. Deze primitieven maken het mogelijk om een beschrijving te geven van elk object binnen de scène. De basis renderprimitieven die gebruikt worden in grafische renderhardware zijn punten, lijnen en driehoeken.

Wanneer 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 scène 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. Vervolgens wordt een *object* gedefinieerd door middel 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 driedimensionale wereld. Hierdoor is het mogelijk om meerdere objecten met verschillende transformaties binnen de wereld voor te stellen dezelfde *mesh*. Een voorbeeld van een verzameling van vier objecten met dezelfde *mesh* maar verschillende transformatiematrices is weergegeven in figuur 2.4.

De volledige beschrijving van een omgeving zal een *scène* genoemd worden. Deze bevat de definities van de verschillende objecten binnen de wereld, als ook de lichtbronnen en eventuele camera's en zichtpunten.

2.2.2 Voorstelling van de camera

Om een afbeelding van een scène te genereren moet het zichtpunt en oriëntatie van het camerastandpunt gedefinieerd worden. Hiervoor wordt gebruik gemaakt van een cameramodel[CON08]. Dit cameramodel is een beschrijving van een virtuele camera die zich bevindt binnen de scène. Om deze camera te definiëren worden de volgende attributen opgesteld:



Figuur 2.4: Voorstelling van objecten door middel van driehoeken.

$\mathcal{O}_{\text{camera}}$ De positie van de camera in de scène.

up De lokale **y**-as van de camera.

kijkrichting De lokale **z**-as van de camera, waarin de camera kijkt.

Z-near De afstand van de oorsprong tot het zichtvenster.

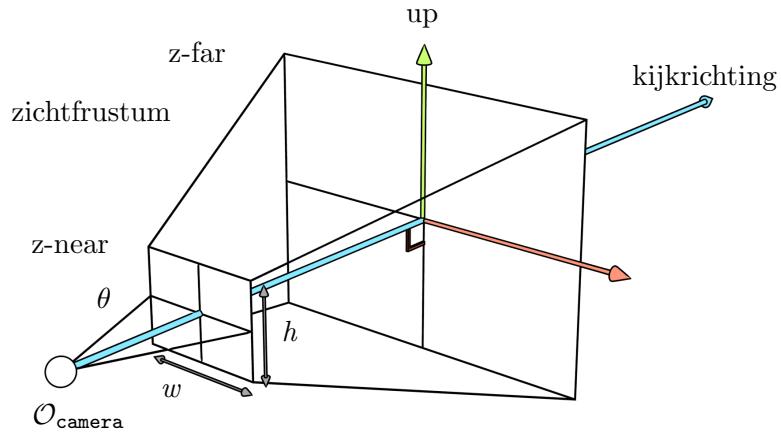
Z-far De maximale afstand tot waar objecten worden weergegeven.

gezichtsveld θ De hoek die bepaald hoeveel van de wereld zichtbaar is.

aspectratio De ratio breedte w tot hoogte h van het zichtvenster.

Dit specificeert de volledige camera, zoals weergegeven in figuur 2.5.

De $Z\text{-near}$, $Z\text{-far}$, oorsprong $\mathcal{O}_{\text{camera}}$, gezichtsveld en aspectratio definiëren het zo-



Figuur 2.5: Het Cameramodel.

genoemde zichtfrustum. Dit frustum specificeert de ruimte waarbinnen alle zichtbare objecten in de scène zich bevinden.

2.3 Perspectiefprojectie en het visibiliteitsprobleem

In sectie 2.1 zijn de twee problemen vastgesteld die opgelost dienen te worden om geloofwaardige afbeeldingen te generen. In deze sectie zal het eerste probleem geadresseerd worden: wat is zichtbaar binnen een scène vanuit het huidige zichtpunt. Om te bepalen wat zichtbaar is dient zowel het perspectief gesimuleerd te worden, als bepaald te worden welk van de objecten in dit perspectief als zichtbaar is.

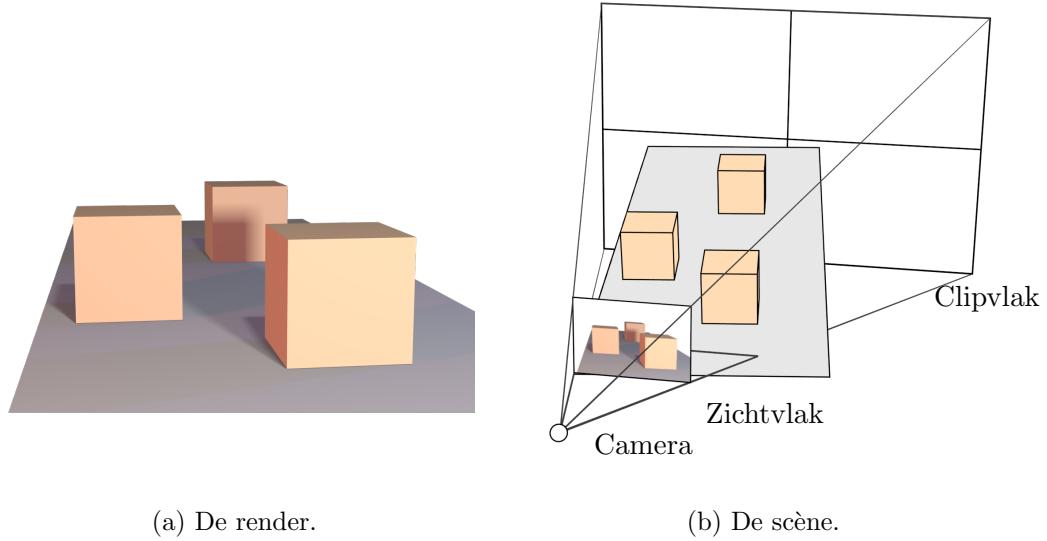
2.3.1 Perspectiefprojectie

De twee effecten van perspectief kunnen gesimuleerd worden door de driedimensionale scène te projecteren naar een oogpunt om deze zo af te beelden op een tweedimensionaal canvas[Suf07]. Dit is weergegeven in Figuur 2.6. Het cameramodel gedefinieerd in de vorige sectie wordt gebruikt om deze projectie mogelijk te maken. Dit leidt ertoe dat alle geometrie binnen het zichtfrustum geprojecteerd zal worden op de oorsprong om zo het perspectief te simuleren[Suf07, AMHH16].

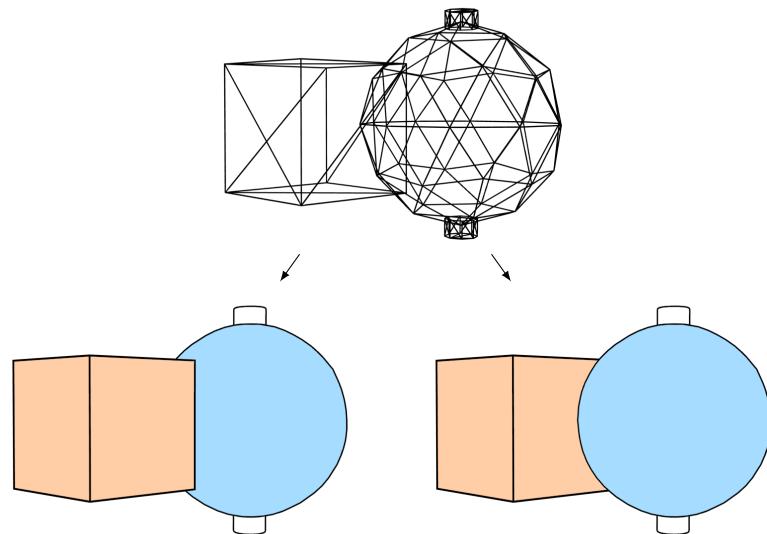
2.3.2 Visibiliteitsprobleem

Er is nu vastgesteld hoe objecten in perspectief afgebeeld kunnen worden op het canvas. Echter, hiermee is nog niet volledig vastgesteld wat daadwerkelijk zichtbaar gaat zijn op het tweedimensionale canvas. Hiervoor is het tevens nodig om te bepalen welke delen van objecten zichtbaar zijn, en welke verborgen zijn achter andere objecten. Dit probleem is weergegeven in figuur 2.7. Door alleen de objecten af te beelden kan niet worden bepaald welk van de twee afbeeldingen van de kubus en de

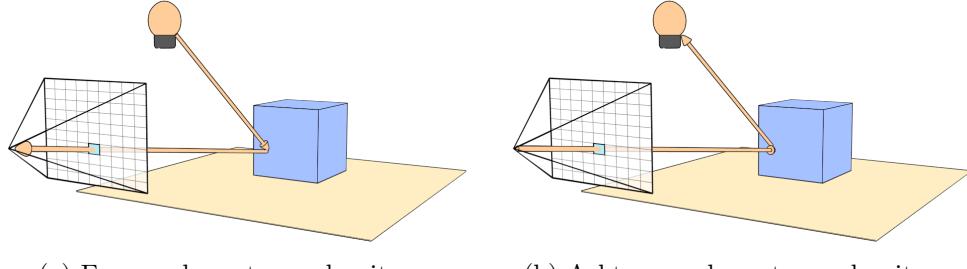
2.3. Perspectiefprojectie en het visibiliteitsprobleem



Figuur 2.6: Perspectiefprojectie.



Figuur 2.7: Visibiliteitsprobleem in een scène met meerdere primitieven.



Figuur 2.8: Mogelijkheden raytrace-algoritmes

bol correct is. Dit probleem wordt onder andere het visibiliteitsprobleem genoemd, en was een van de eerste grote problemen binnen de computergrafieken[SSS74].

De oplossing van het visibiliteitsprobleem kan gevonden worden in de realisatie dat de bepaling van de zichtbare ruimte op een tweedimensionaal canvas, neerkomt op een sorteerprobleem. Indien in een punt \mathbf{p} bepaald moet worden welk primitief zichtbaar is, kan met behulp van perspectiefprojectie de verzameling S van objecten die afgebeeld worden op dit punt, opgesteld worden. Het object dat vervolgens zichtbaar is, zal het object in de verzameling S zijn dat het dichtst bij het punt \mathbf{p} en dus het zichtpunt van de camera ligt. Indien de objecten in S gesorteerd worden op basis van hun afstand tot de camera zal het eerste object het dichtst bij de camera liggen en dus zichtbaar zijn. Gezien de afstand tot de camera overeenkomt met de \mathbf{z} -afstand in cameracoördinaten, dient dus het object met de kleinste \mathbf{z} -waarde gevonden worden. De algoritmes die dit probleem oplossen worden verborgenoppervlaktebepalingsalgoritmes genoemd[War69, SSS74].

De algoritmes die zowel de perspectiefprojectie als de visibiliteit oplossen kunnen grofweg worden ingedeeld in twee categorieën, raytracing-algoritmes en rasterisatie-algoritmes. Het resultaat met betrekking tot het bepalen van de zichtbare geometrie van deze algoritmes zou in theorie hetzelfde moeten zijn.

Raytracing werkt op basis van het trekken van zogenoemde stralen uit het zichtpunt door de pixels in de canvas. Vervolgens wordt gekeken of deze stralen snijden met primitieven. Rasterisation beeldt de vertices van primitieven af op de canvas met behulp van de perspectiefprojectie. Vervolgens wordt bepaald welke pixels overlappen met het primitief. In de volgende secties zal ingegaan worden op beide algoritmes.

2.3.3 Raytracing

Raytracing simuleert de werking van licht en het menselijk oog, en lost hiermee zowel het visibiliteitsprobleem als de perspectiefprojectie 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 gebeurt in het oog. In dit geval zouden vanuit lichten willekeurige stralen geschoten kunnen worden. Hierbij is een

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

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

    do_shading(closest, ray)
```

Listing 1: Raytracing algoritme.

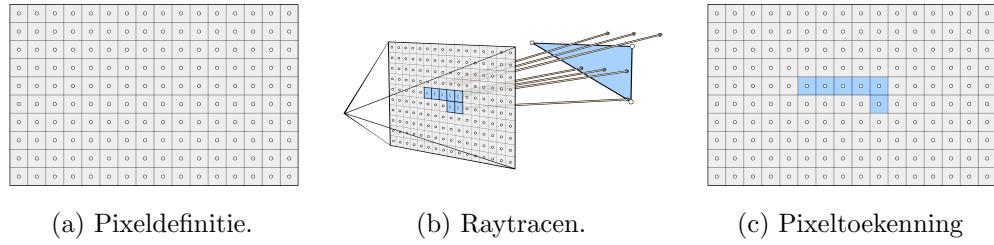
straal gedefinieerd als zijnde een vector met een beginpunt in de oorsprong van een licht. Wanneer een dergelijke straal door de canvas op het oogpunt valt, wordt deze meegeteld. Dit is geïllustreerd in figuur 2.8. Deze techniek wordt forwaards tracen genoemd. Vaak is het canvas van een camera velen malen kleiner dan de scène in kwestie. Dit zorgt ervoor dat de kans dat het 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 is in te zien dat de techniek ook omgedraaid kan worden. In plaats van willekeurige stralen te schieten vanuit lichten, wordt er door elk punt op de canvas een straal getrokken vanuit het zichtspunt. 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.

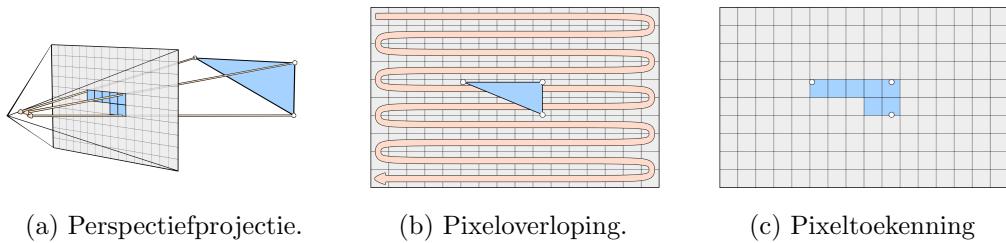
Deze techniek, waarbij gestart wordt vanuit de camera wordt, achterwaardse tracing genoemd. Belangrijk om hierbij op te merken is dat ray tracing, dus voor elk punt \mathbf{p}' een punt \mathbf{p} vindt. Wanneer gesteld wordt dat punt \mathbf{p}' een (sub)pixel is, zou een algoritme dus bestaan uit twee lussen. In de eerste plaats wordt per pixel een straal gegenereerd. Vervolgens worden per straal alle objecten overlopen, om het object te vinden dat zowel geraakt wordt en het dichtst bij de oorsprong ligt. Doordat de buitenste lus over de pixels loopt, worden raytracing-algoritmes dan ook wel beeldcentrische algoritmes genoemd. De pseudocode is weergegeven in listing 1.

Waarbij `do_shading` gebruikt wordt om de kleur te bepalen van de specifieke pixel. Dit is verder geïllustreerd in figuur 2.9. Hierbij worden eerst de punten op het canvas opgesteld. Vervolgens wordt door elk van deze punten een straal getrokken, en wordt gekeken of deze snijden met een object. Vervolgens worden alle pixels

2. THEORIE



Figuur 2.9: Raytrace-algoritme.



Figuur 2.10: Het rasterisatie-algoritme.

gemarkerd die snijden.

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

2.3.4 Rasterisatie

Rasterisatie-algoritmes lossen de perspectiefprojectie en het visibiliteitsprobleem op de volgorde van de object- en pixellussen ten opzichte van de raytracingalgoritmes om te draaien. Waar raytracing uitgaat van het punt \mathbf{p}' op de canvas en kijkt welk object hier op valt, begint een rasterisatiealgoritme met het afbeelden van punten in de ruimte 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 beeldcentrisch zijn, zijn rasterisatie-algoritmes objectcentrisch. Hierbij wordt in de buitenste lus over alle objecten gelopen. En daarna per object gekken welke pixels door dit object worden beïnvloed. Dit leidt tot de volgende pseudocode:

```

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

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

Dit is tevens afgebeeld in figuur 2.10.

Om een enkele primitief dus af te beelden op het canvas dient eerst, voor elke hoek van dit primitief de perspectiefdeling uitgevoerd te worden. Hierna dient het resultaat omgezet te worden naar rasterruimte, 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.

Hiermee is vastgesteld dat de oplossing voor de perspectiefprojectie 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[SSS74].

Het visibiliteitsprobleem wordt in grafische kaarten opgelost met behulp van een zogenoemd z-buffer algoritme[SSS74]. Zoals opgemerkt bij de besprekking van het visibiliteitsprobleem, is dit intrinsiek een sorteerprobleem, waarbij objecten geordend dienen te worden ten opzichte van de kijk-as in cameracoördinaten, de camera-z-as. Om het zichtbare object binnen een punt \mathbf{p}' te bepalen, dient dus bepaald te worden welk object de kleinste z-as waarde heeft ten opzichte van het oogpunt. De oplossing voor dit probleem is dan ook simpel. Voor elke pixel wordt de kleinste gevonden z-as waarde bijgehouden in een corresponderende twee dimensionale array. Deze array wordt een z-buffer, of een diepte-buffer genoemd. Wanneer een pixel gevonden wordt met een kleinere z-waarde, wordt zowel het object in punt \mathbf{p}' als de nieuwe diepte bijgewerkt. Wanneer alle objecten overlopen zijn zal er dus per pixel bekend zijn welke objecten gebruikt dienen te worden om de shading berekening uit te voeren.

2.4 Shading

In de voorgaande secties is besproken hoe visibiliteit opgelost kan worden. Dit is de eerste stap in het genereren van beelden. Nu vastgesteld is hoe objecten in de scène weergegeven worden op het zichtvenster kan bepaald worden hoe deze objecten er daadwerkelijk uitzien, de tweede stap van het renderproces. 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 de computergrafieken is de rendervergelijking [Kaj86]:

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

Hier weergegeven in hemisfeervorm. Deze vergelijking toont de stabiele toestand van de stralingsenergiebalans binnen een scène. Hierbij is $L_o(\mathbf{p}, \omega_o)$ de radiantie uitgezonnen vanuit punt \mathbf{p} over ω_o . Deze radiantie kan gedefinieerd worden aan de hand van

2. THEORIE

de som van gereflecteerde radiantie, en de radiantie die door het punt \mathbf{p} zelf wordt uitgestraald. De uitgestraalde radiantie is gedefinieerd als $L_e(\mathbf{p}, \omega_o)$. De reflectie van radiantie wordt beschreven door het tweede deel van de rendervergelijking:

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

Dit wordt de reflectievergelijking genoemd[AMHH16]. Hierbij wordt geïntegreerd over de gehele hemisfeer om de volledige binnenkomende radiantie te berekenen. Vervolgens specificert een zogenoemde Bidirectionele Reflectie Distributie Functie (BRDF)[Nic65, Suf07], 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 .

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 rendervergelijking. Hierbij zijn de materialen van objecten beschreven met BRDFs. Er is nog wel een groot probleem met deze voorstelling. De radiantie die uitgezonden wordt vanuit \mathbf{p} over ω_o is afhankelijk van alle radiantie binnenkomend over de gehele hemisfeer in punt \mathbf{p} . De binnenkomende radiantie is gelijk aan de radiantie uitgezonden vanuit alle punten op de hemisfeer. Dit heeft als gevolg dat een recursieve integraal opgelost moet worden. Dit is niet mogelijk, en dus zullen alle shading algoritmes pogingen te geven van de daadwerkelijke oplossing van de rendervergelijking. De kwaliteit van de benadering hangt af van meerdere aspecten. Een grote beperkende factor binnen realtime computergrafieken is de beschikbare rekentijd.

2.4.2 Directe lichtbenadering

De meest simpele benadering van de reflectievergelijking is de directe belichtingbenadering. Hierbij wordt voor elk punt slechts de lichtbijdrage berekend van lichten die door één bounce op de camerasensor vallen. Deze lichten schijnen dus direct op een punt. Indirecte belichting, waarbij lichtstralen via meerdere bounces op de camerasensor vallen, en dus eerst andere oppervlaktes raken voordat het punt bereikt wordt, worden buiten beschouwing gelaten. Dit leidt tot een grote versimpeling van de reflectievergelijking ten koste van een kleine hoeveelheid energieverlies. Deze kan nu opgesteld worden als een sommatie over de lichten in plaats van een integraal over de hemisfeer[AMHH16]:

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

hierbij is de lichtbron k gedefinieerd als l_k .

De directe-belichtingbenadering reduceert het aantal bewerkingen significant. Om deze reden zal deze benadering verder gebruikt worden binnen deze thesis.



Figuur 2.11: Lambertiaanse BRDF.

2.4.3 Lambertiaanse BRDF

Materialen kunnen gedefinieerd worden als een verzameling van BRDFs, die het gedrag van het licht beschrijven indien het in contact komt met een object. De simpelste BRDF is de lambertiaanse BRDF[Suf07]. Deze BRDF beschrijft een puur diffuus oppervlakte, wat inhoudt dat elke richting waarin een binnengekomende lichtstraal gereflecteerd kan worden evenveel kans heeft om voor te komen. Hierdoor wordt het licht uniform over de hemisfeer verdeeld. Dit is weergegeven in fig. 2.11. Deze BRDF heeft als uitkomst een constante waarde. Deze constante waarde wordt veelal gedefinieerd als de *diffuse kleur* c_{dif} van dit object. Dit leidt tot de volgende functie:

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

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

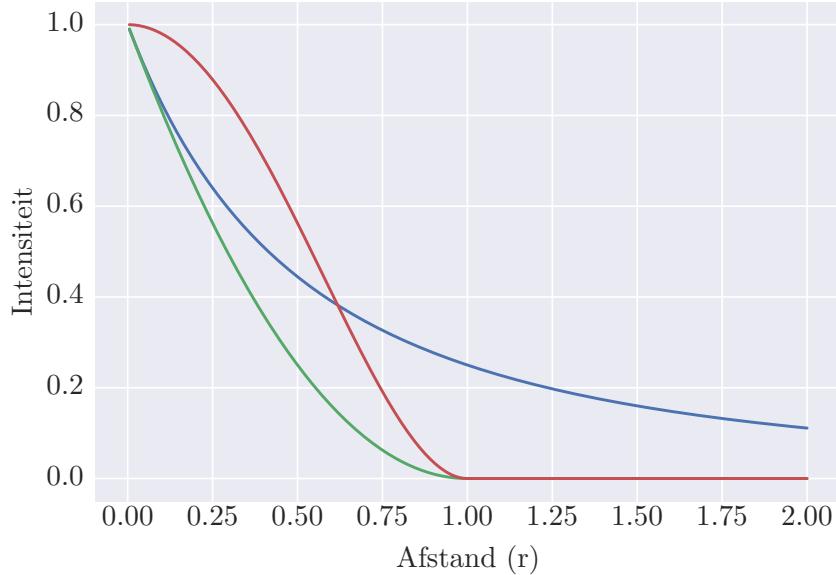
De lambertiaanse BRDF is als standaard materiaal gebruikt binnen deze thesis. Indien niet anders vermeld zullen afbeeldingen en testen gegeneerd zijn met deze functie.

2.4.4 Schaduw

Schaduw is een belangrijke visuele aanwijzing voor de plaatsing van objecten in de scène. De hersenen gebruiken schaduw om de onderlinge relatie van objecten te bepalen. In de fysische werkelijkheid is een schaduw op een oppervlak het gevolg

2. THEORIE

- referentie: $f(d) = \frac{1}{1+d^2}$
- $f(d) = \left(1 - \frac{d}{r}|_{[0,1]}\right)^2$
- $f(d) = \left(1 - \frac{d^2}{r^2}|_{[0,1]}\right)^2$



Figuur 2.12: Afstandsdiminutiescurves voor eindige omnipuntlichtbronnen.

van obstructie tussen deze oppervlakte en de lichtbron. Om schaduw te simuleren binnen computergrafieken dient dus bepaald te worden of op een positie er objecten tussen het punt en een lichtbron liggen. Dit betekent dat er extra werk uitgevoerd dient te worden om deze bepaling uit te voeren.

Zoals behandeld zal worden in de volgende sectie, zal de belichting binnen de moderne grafische pijplijn plaatsvinden per pixel. Bij deze berekening is er niet impliciet een beschrijving van de scène beschikbaar. Deze dient explicet in het geheugen te worden bijgehouden. Dit leidt tot een toename in de complexiteit. Om deze reden zijn schaduwen buiten beschouwing gelaten.

Voor een uitgebreidere beschrijving van rendering onderwerpen kan gerefereerd worden naar de boeken: Raytracing from the ground up [Suf07], en Physical Based Rendering [PJH16].

2.4.5 Definitie van lichtbron

Zoals eerder benoemd, draait de kern van deze thesis om het optimaliseren van het aantal lichtberekeningen in realtime toepassingen. Om deze reden is het belangrijk om het concept lichtbron zoals gebruikt in deze thesis te definiëren. Wanneer er gesproken wordt van een lichtbron zal binnen deze thesis altijd worden gedoeld op een eindige omnipuntlichtbron die zich bevindt op punt \mathbf{p} binnen de scène.

In de fysische wereld zijn lichtbronnen nooit eindig, echter de invloed die ze

hebben op de omringende wereld zal bij grotere afstand 0 benaderen. Dit is enerzijds het gevolg van absorptie door het medium waardoor de lichtbron zich beweegt. Binnen de fysica wordt deze relatie vastgelegd met de wet van Lambert-Beer[Bou29] gedefinieerd voor uniforme demping als:

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

waar d de padlengte van de lichtstraal door het medium is en μ de dempingscoëfficiënt. Hierbij wordt de demping van het licht gerelateerd aan het medium waardoor het zich beweegt. Wanneer de demping als functie van de afstand d wordt geplot, worden zogenoemde dempingscurves verkregen. Anderzijds is het een direct gevolg van de geometrie. Voor een puntlichtbron zal de intensiteit van een lichtbron kwadratisch afnemen met de afstand d tot de oorsprong van de lichtbron. De oppervlakte van een bol is gedefinieerd als

$$A = 4\pi r^2$$

Dit betekent dat bij een toename van afstand r tot de oorsprong, de oppervlakte kwadratisch zal toenemen. De energie die wordt uitgezonden door het puntlichtbron zal bij een toename van een afstand r op een kwadratisch groter oppervlak worden afgebeeld. Dit betekent dat de waargenomen energie kwadratisch afneemt. De demping ten gevolge van de afstand kan worden voorgesteld met een zogenoemde afstanddempingsfunctie f :

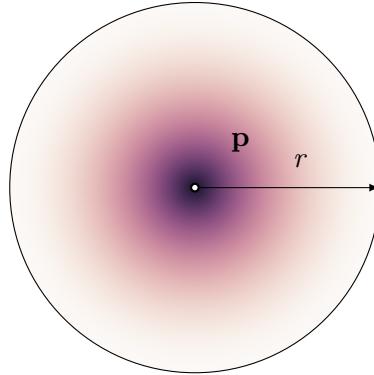
$$f(d) = \frac{1}{1 + d^2}$$

Binnen de computergrafieken wordt veelal het medium waarin de scène zich bevindt niet explicet voorgesteld, in plaats hiervan wordt aangenomen dat de wereld zich in vacuüm bevindt. Uit performantie overwegingen worden binnen veel realtime renderingtoepassingen lichtbronnen verder voorgesteld door een eindige benadering. De invloed wordt beperkt tot een eindige ruimte. Buiten dit lichtvolume zal een lichtbron geen invloed hebben. Voor puntlichtbronnen komt dit neer op een bolvolume met een radius r . Dit leidt tot een voorstelling als weergegeven in figuur 2.13.

Om de illusie te wekken dat de lichten fysiek accuraat zijn, dient tevens een benadering gemaakt te worden van de afstanddempingsfuncties. Deze functies dienen te voldoen aan de eerder gestelde voorwaarde, waarbij de invloed één is in de oorsprong, en nul op afstand r . De fysieke afstanddempingsfunctie en enkele veel gebruikte benaderingen zijn gegeven in figuur 2.12. Binnen de thesis zelf is gekozen voor de benadering:

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

Als laatste dient de intensiteit van de lichtbron vastgelegd te worden. Zoals gebruikelijk binnen de computergrafieken, is hier gekozen voor een RGB voorstelling. Waarbij de waardes zich bevinden in het bereik van 0 tot en met 1.



Figuur 2.13: Voorstelling van een puntlicht.

Dit alles leidt ertoe dat een lichtbron gedefinieerd kan worden als de verzameling van de volgende eigenschappen:

- De positie \mathbf{p} van het lichtbron ten opzichte van een coördinatenstelsel met oorsprong \mathcal{O} .
- Een afstand r die de invloed van de lichtbron bepaalt.
- Een afstanddempingsfunctie f die het verval van invloed moduleert
- Een intensiteit \mathbf{i} die de kleur en intensiteit van de lichtbron bepaalt.

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 realtime renderen. Het onderliggende gereedschap verantwoordelijk voor het renderen van de afbeeldingen is de realtime grafische pijplijn. In deze sectie zal eerst conceptueel de opbouw van de grafische pijplijn beschreven worden, zoals gegeven in Real-Time Rendering (3rd edition)[AMHH16]. Hierna zal in meer detail de OpenGL implementatie besproken worden, waarop het onderzoek binnen deze thesis is gebouwd.

2.5.1 Conceptuele architectuur

In de fysieke wereld is een pijplijn een verzameling van stappen die in parallel uitgevoerd kunnen worden. Elke stap transformeert de uitkomst van de vorige stap in een volgende uitkomst. Voor een individuele uitkomst zal dus elke stap in de pijplijn uitgevoerd moeten worden, echter doordat alle stappen per uitvoering in parallel worden uitgevoerd zal een uitkomst gegenereerd worden per uitvoering. Deze parallelisatie van stappen verhoogt dus de doorvoersnelheid. De tijd om alle stappen eenmaal uit te voeren, zal dan ook gelijk zijn aan de traagste stap. De grafische pijplijn is op een vergelijkbare manier opgebouwd en kan grofweg onderverdeeld worden in drie conceptuele stappen.

- Applicatiestap
- Geometriestap
- Rasterisatiestap

Applicatiestap

De applicatiestap beschrijft alle berekeningen die plaatsvinden binnen de applicatie van de ontwikkelaar. Belangrijke aspecten hier zijn onder andere de verwerking van invoer van gebruikers, het opzetten van datastructuren en botsingdetectie. Aan het einde van de applicatiestap dient een verzameling primitieven gestuurd te worden naar de geometriestap.

Geometriestap

De geometriestap is verantwoordelijk voor het merendeel van de per-polygon operaties. Deze stap kan verder onderverdeeld worden in de volgende sub-stappen:

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

Belangrijk hierbij is dat de conceptuele beschrijving in sommige opzichten kan verschillen van daadwerkelijke implementaties als `openGL` en `Direct3D`.

De geometriestap zorgt dat objecten geproduceerd door de applicatiestap, omgezet worden naar een verzameling van data die in de rasterisatiestap omgezet kan worden in een daadwerkelijke afbeelding. Eerst worden objecten getransformeerd opdat dat alle primitieven zich in hetzelfde coördinatensysteem bevinden. Hierna vindt een eerste shadingstap plaats die wordt uitgevoerd voor alle vertices van alle primitieven. Nadat de shading berekend is per vertex, wordt de perspectiefprojectie uitgevoerd en worden niet zichtbare objecten weggesneden uit het resultaat. Als laatste worden de primitieven die over zijn omgezet naar canvas-coördinaten. De verzameling van primitieven in canvas-coördinaten, en corresponderende shadingdata wordt vervolgens doorgestuurd naar de rasterisatiestap.

Rasterisatiestap

In de rasterisatiestap worden de daadwerkelijke kleuren van de afbeelding berekend. Hiervoor wordt een rasterisatiealgoritme uitgevoerd. De volgende sub-stappen kunnen onderscheiden worden:

- Driehoekkopzet
- Driehoekdoorkruizing

2. THEORIE

- Pixel-shading
- Samenvoeging

De eerste twee stappen komen overeen met het rasterisatie-algoritme. Hierbij wordt shadingdata uit de geometrie stap geïnterpoleerd. Dit leidt tot een verzameling van fragmenten met geassocieerde geïnterpoleerde shadingdata. Deze worden met behulp van pixel-shading verwerkt tot een specifieke kleur voor een specifieke pixel. Als laatste wordt door middel van het z-buffer algoritme en specificaties van de ontwikkelaar, elk van deze potentiele pixelwaardes samengevoegd tot een specifieke kleurwaarde die weergegeven kan worden binnen de afbeelding.

2.5.2 Moderne Grafische Pijplijn implementatie

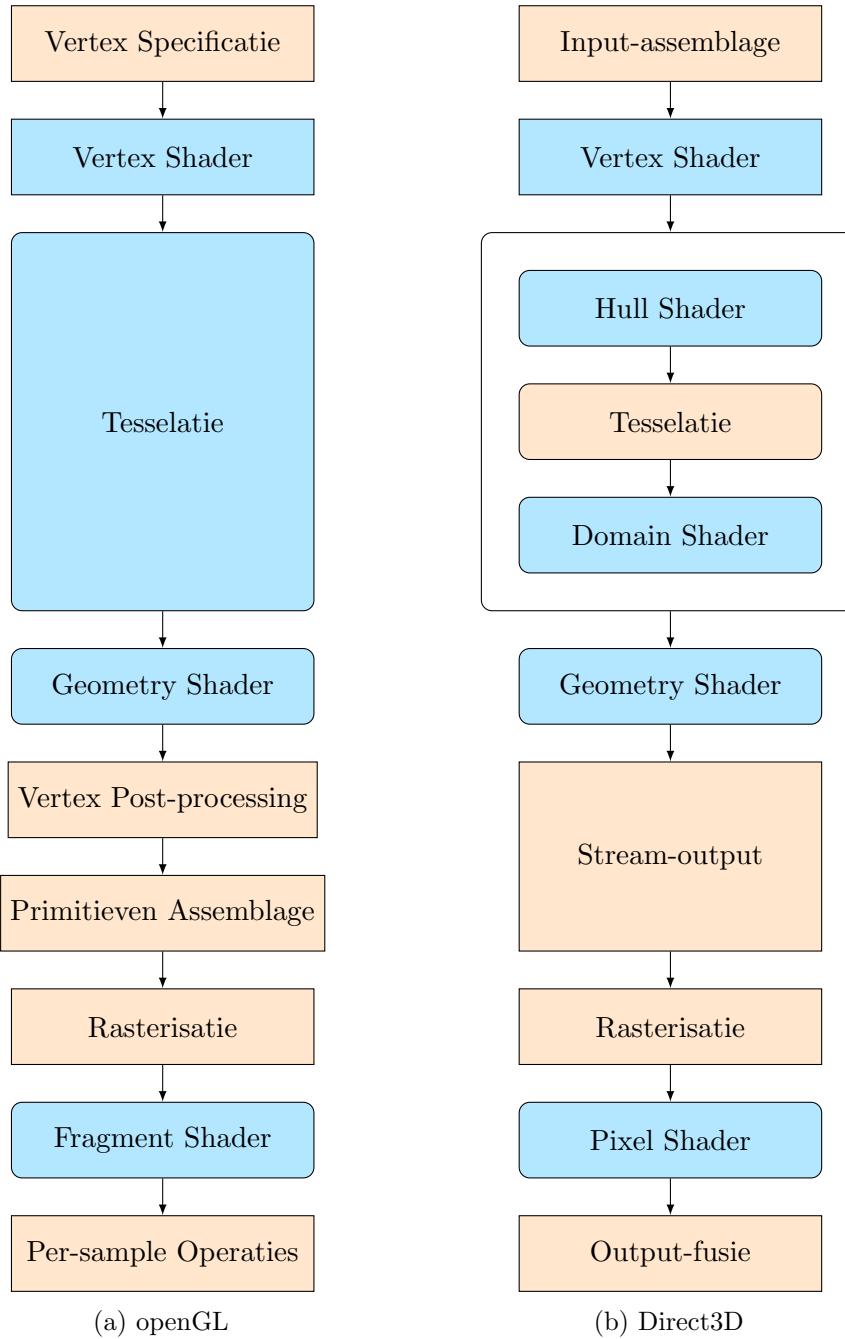
De moderne grafische pijplijn is in grote mate programmeerbaar. Dit betekent dat de ontwikkelaar in staat is om zelf algoritmes op de grafische pijplijn te implementeren. Er zijn verschillende **APIs** beschikbaar waarmee de ontwikkelaar deze algoritmes kan implementeren. De twee meest gebruikte industiestandaarden zijn **openGL** en **Direct3D**. **openGL** is een niet-platformspecifieke specificatie, terwijl **Direct3D** gebonden is aan Microsoft Windows. Om deze reden is gekozen voor het gebruik van **openGL** binnen deze thesis. **openGL** en **Direct3D** zijn vergelijkbaar, maar verschillen in nomenclatuur. In deze thesis zal gebruik gemaakt worden van de naamgeving van **openGL**.

Een overzicht van de verschillende stappen van de pijplijn voor respectievelijk **openGL** en **Direct3D** zijn weergegeven in figuur 2.14a en 2.14b. De stappen die programmeerbaar zijn, zijn weergegeven in blauw. De andere stappen zijn om redenen van efficiëntie slechts configurerbaar. De optionele stappen zijn weergegeven met ronde hoeken. In beide **APIs** zijn negen stappen terug te vinden:

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

De programmeerbaarheid van de pijplijn volgt uit zogenoemde programmeerbare shaders, de *vertex shader*, *geometrie shader*, en *fragment shader*. Deze hebben respectievelijk invloed op de 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.

Terugkijkend op de conceptuele beschrijving van de grafische pijplijn, komen de stappen als volgt overeen met de conceptuele stappen. De applicatie-stap is niet gedefinieerd binnen de **openGL** pijplijn. Deze wordt uitgevoerd voordat de **openGL**



Figuur 2.14: De stappen van zowel de OpenGL als Direct3D implementaties.

2. THEORIE

pijplijn wordt aangesproken. De applicatie stap eindigt met de vertex-specificatie. De applicatie specificert hierbij een verzameling van primitieven. Vervolgens wordt deze verzameling van vertices verwerkt door een of meerdere vertexshaders. De ontwikkelaar heeft hier volledige controle over. Veelal vinden hier de model- en zichttransformaties plaats. Ook is het mogelijk dat hier een eerste stap van de shading plaatsvindt, wat vervolgens geïnterpoleerd 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, en zullen niet gebruikt worden binnen deze thesis. 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. Aan het einde van deze stap wordt een verzameling 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 vindt dan de samenvoeging van fragmenten plaats in de per-sample-operaties. Hier wordt tevens het z-buffer algoritme uitgevoerd. Dit betekent dat per-pixel-shading wordt uitgevoerd voor alle pixels, en dus ook voor pixels die uiteindelijk helemaal geen invloed hebben op de afbeelding. In de volgende secties zal nog ingegaan worden op de drie stappen die het meest relevant zijn deze thesis.

Vertexshader

De vertexshader behandelt exclusief de punten, vertices, die gespecificeerd worden door de applicatie. De shader zelf heeft geen kennis hoe elk van de vertices zich verhoudt tot de primitieven. Veelal is de vertexshader verantwoordelijk voor het omzetten van de coördinaten van model- naar camera- of wereldruimte, afhankelijk van de specificatie van de fragmentshader. Tevens dient hier de locatie van de vertex in de canvas gezet te worden.

Fragmentshader

Nadat de primitieven omgezet zijn naar een verzameling van fragmenten, wordt op elk van de fragmenten de gespecificeerde fragmentshader toegepast. De rasterisatiestap produceert een verzameling van data, waaronder de specifieke locatie van fragmenten, en een interpolatie van berekende waarden binnen de vertexshader. Deze kunnen vervolgens gebruikt worden door de fragmentshader, om de shading van dat fragment te uit te voeren.

De fragmentshader berekent de kleur voor alle fragmenten, voor dat deze worden samengevoegd tijdens de per-sample-operaties. De berekening van de kleur is veelal de stap binnen de grafische pijplijn die de meeste berekeningsmiddelen vereist. In moderne applicaties wordt hier veelal de benadering van de rendervergelijking berekend. Hierbij is de fragmentshader niet beperkt om slechts naar één canvas data

weg te schrijven. Het is mogelijk om meerdere canvassen aan te spreken met behulp van meerder renderdoelen (*multiple render targets*).

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, met behulp 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. Deze stap zijn configurerbaar.

Zoals eerder vermeld is een belangrijke observatie dat voor de meeste implementaties van grafische pijplijnen, 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 scène bestaat uit veel primitieven. Oplossingen hiervoor zullen verder besproken worden in volgende hoofdstukken.

Voor verdere informatie over realtime rendering en de moderne grafische pijplijn wordt gerefereerd naar Real-Time Rendering[AMHH16] Meer informatie over Direct3D en openGL kan gevonden worden op hun respectievelijke documentatiewebsites.

Probleemstelling

In de introductie wordt het doel van deze thesis gesteld op het ontwikkelen van een lichttoekenningsalgoritme dat de opgestelde datastructuren hergebruikt tussen frames om zo tot een performantiewinst te komen ten opzichte van lichttoekenningsalgoritmes die datastructuren per frame opbouwen. Met de kennis van dit hoofdstuk kan dit doel onderbouwt worden.

Voor het renderen van een afbeelding met behulp van de directe-lichtbenadering van de rendervergelijking dient per pixel het geprojecteerde punt bepaald te worden. Voor elk punt dient vervolgens de fragmentshader uitgevoerd te worden, waar voor elke lichtbron in de scène de lichtbijdrage wordt berekend. Dit komt neer op het uitrekenen van de interactie tussen het materiaal van het punt en het lichtbron. Dit leidt tot een groot aantal berekeningen.

Door de optimalisatie om lichtbronnen te benaderen met een eindige voorstelling, zal niet langer elk punt beïnvloed worden door elke lichtbron. Algoritmes die voor een punt de verzameling van lichtbronnen in de scène beperken tot de deelverzameling van lichtbronnen die een bijdrage leveren aan de belichting van een punt, worden lichttoekenningsalgoritmes genoemd. Deze algoritmes reduceren het aantal lichtberekeningen in de fragmentshader.

De lichttoekenningsalgoritmes Tiled[OA11] en Clustered Shading[OBA12] bouwen de datastructuren die de lichttoekenning mogelijk maken, per frame op. Bij een hoge framerate zal er weinig verschil zijn tussen opeenvolgende afbeeldingen. Het doel van deze thesis is om tot een lichttoekenningsalgoritme te komen dat deze datastructuren hergebruikt tussen frames om zo tot een betere performantie te komen dan huidige lichttoekenningsalgoritmes. Hiervoor zullen Tiled en Clustered Shading geanalyseerd worden in volgende hoofdstukken. Daarna zal een nieuw lichttoekenningsalgoritme geïntroduceerd worden. Dit algoritme is camera-onafhankelijk waardoor de data-

2. THEORIE

structuren niet per frame opgebouwd dienen te worden, maar hergebruikt kunnen worden bij veranderingen in het zichtpunt.

2.6 Besluit

In dit hoofdstuk is een introductie gegeven tot realtime computergrafieken. De perceptie en fysische eigenschappen van licht vormen de onderliggende basis voor computergrafieken. Hierbij dienen de eigenschappen van licht en perceptie gesimuleerd te worden om geloofwaardige afbeeldingen te creëren.

Om de fysische werkelijkheid te benaderen dient de visibiliteit van objecten bepaald te worden door perspectiefprojectie en het oplossen van het visibiliteitsprobleem. Vervolgens wordt door middel van het benaderen van de rendervergelijking een afbeelding gegenereerd. Binnen realtime computergrafieken wordt dit alles gerealiseerd met behulp van de moderne grafische pijplijn. Om deze pijplijn te gebruiken wordt binnen deze thesis gebruik gemaakt `OpenGL`.

Hoofdstuk 3

Methode-overzicht

Voordat ingegaan zal worden op de verschillende lichttoekenningsalgoritmes zal eerst een overzicht gegeven worden van de gebruikte software en testscènes. Om de tijds- en geheugencomplexiteit van het nieuw geïntroduceerde Hashed Shading algoritme te evalueren en te vergelijken met andere lichttoekenningsalgoritmes, is er voor gekozen om deze algoritmes te implementeren in één nieuw programma. Vervolgens is de tijds- en geheugencomplexiteit met betrekking tot de resolutie en het aantal lichten geëvalueerd aan de hand van drie testscènes. In dit hoofdstuk zal de ontwikkelde software beschreven worden. Daarna zal ingegaan worden op de gebruikte hardware. Als laatste zullen de gebruikte testscènes en de analyse van de resultaten behandeld worden.

3.1 Software

Om de verschillende lichttoekenningsalgoritmes op een consistente manier te vergelijken, is gekozen om elk van deze algoritmes te implementeren. Hiervoor is het programma `nTiled` ontwikkeld. Naast de lichttoekenningsalgoritmes bevat dit programma alle functionaliteit die nodig is om de renderpijplijn uit te voeren, en relevante data te verzamelen.

3.1.1 Organisatie

`nTiled` kan worden onderverdeeld in de volgende modules:

camera De camera implementeert het cameramodel zoals beschreven in sectie 2.2.2.

gui De gui bevat alle componenten die nodig zijn voor het gebruikersinterface.

log De logmodule bevat de functies die gebruikt worden om relevante data te verzamelen.

main De main-module implementeert de controlerfuncties die verantwoordelijk zijn voor de uitvoering van het programma.

math De math-module bevat alle extra wiskundige functies die gebruikt worden binnen `nTiled`

3. METHODE-OVERZICHT

pipeline De pipeline-module is verantwoordelijk voor de gehele pijplijn en bevat de implementaties van de shaders en de datastructuren van de lichttoekenningsalgoritmes.

state De state-module bevat alle componenten gerelateerd aan de staat van een enkele uitvoering van **nTiled**. Het is hierbij verantwoordelijk voor het inlezen van de configuratiebestanden en het beheren van deze ingelezen attributen.

world De world-module is verantwoordelijk voor het beheren van alle geometrie en lichten.

Voor een compleet overzicht van de implementatie zie de documentatie¹ en repository² van **nTiled**.

3.1.2 Libraries

nTiled is gebouwd op de volgende libraries:

openGL 4.4 en GLAD: **openGL** met behulp van **GLAD**³ verzorgt voor de rendering pijplijn.

glfw: **glfw** is de window manager, verantwoordelijk voor het aanmaken van de applicatie in het besturingssysteem en het managen van de gebruikersinput.⁴

assimp: **assimp** is gebruikt om de geometrie-objecten in **nTiled** te laden.⁵

glm: **glm** is de wiskundige library die de vector- en matrixberekeningen aan de C++ kant verzorgt.⁶

rapidjson: **rapidjson** is verantwoordelijk voor het inlezen van de **json** configuratie bestanden en het exporteren van de verzamelde data.⁷

dear, imgui: **dear, imgui** verzorgt de GUI.⁸

De software is ontwikkeld en gecompileerd met behulp van **visual studio 2015**⁹

3.1.3 Renderpijplijn

De renderfunctionaliteit is geïmplementeerd in de pipeline-module. De pipeline-module kan worden onderverdeeld in de Forward- en Deferred-pijplijn, die verder behandeld zullen worden in hoofdstuk 4, en de lichttoekkeningsalgoritmes, die zullen worden behandeld in hoofdstuk 5 tot 7.

Voor elk lichttoekenningsalgoritme is zowel een Forward- als Deferred-shader gedefinieerd. Dit leidt tot de volgende shaders:

¹de **nTiled** documentatie kan gevonden worden op ntiled.readthedocs.io/en/latest/index.html

²de **nTiled** repository kan gevonden worden op github.com/BeardedPlatypus/nTiled

³glad project pagina: github.com/Dav1dde/glad

⁴glfw website: www.glfw.org

⁵assimp project pagina: github.com/assimp/assimp

⁶glm website: glm.g-truc.net/0.9.8/index.html

⁷rapidjson project pagina: github.com/miloyip/rapidjson

⁸dear, imgui project pagina: github.com/ocornut/imgui

⁹visual studio 2015 website: <https://www.visualstudio.com>

Naïef De shader zonder lichttoekenningsdatastructuur.

Tiled De shader met de Tiled Shading-datastructuur.

Clustered De shader met de Clustered Shading-datastructuur.

Hashed De shader met de Hashed Shading-datastructuur.

Alle belichtingsberekeningen vinden plaats in de fragmentshader. Binnen de vertexshaders worden slechts de relevante coördinatenstelseltransformaties op de positie en normaal uitgevoerd. Binnen de fragmentshaders wordt eerst de relevante set van lichten bepaald aan de hand van het corresponderende lichttoekenningsalgoritme. Vervolgens wordt voor elk van deze lichten een belichtingsberekening uitgevoerd, waarbij de resultaten gesommeerd worden.

De shading-berekening is een simpele directe-lichtbenadering van een wit lambertiaansoppervlakte, zoals beschreven in sectie 2.4. Deze functie is gedefinieerd in listing 2 en komt overeen met de functie:

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

waar

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

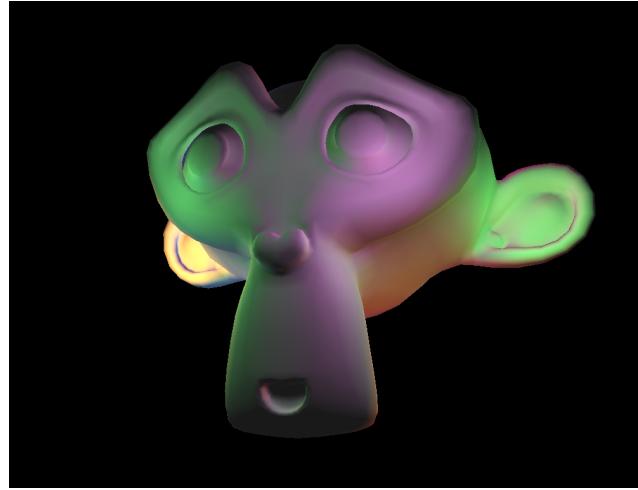
en θ de invalshoek is. Dit alles leidt tot het materiaal zoals weergegeven in figuur 3.1, waar de oppervlakte verlicht is met twaalf lichten met verschillende tinten.

```
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);
}
```

Listing 2: Kleurberekening in de fragmentshader.



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

3.1.4 Meetmethode

De uitvoeringstijdmetingen zijn verzameld met de `QueryPerformanceCounter`. Deze functionaliteit is aangeboden in `Windows.h`. De `QueryPerformanceCounter` maakt het mogelijk om de executietijd tot op μs nauwkeurig te meten. Voor zowel de shaders als de lichttoekenningsalgoritme-managers zijn klassen gedefinieerd die de `QueryPerformanceCounter` gebruiken om de uitvoeringstijd van relevante functies bij te houden.

3.2 Hardware

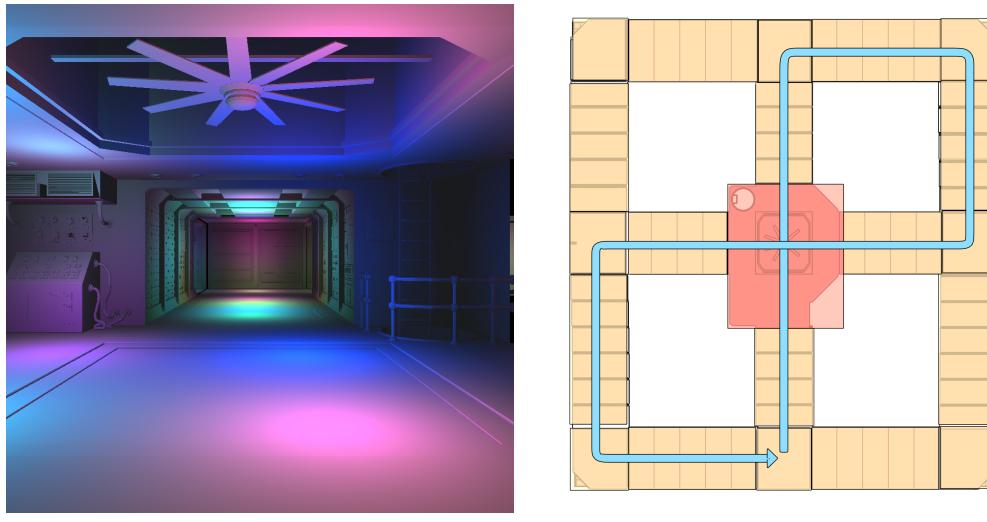
Alle testen zijn uitgevoerd op de hardware in de volgende tabel:

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

3.3 Testsuite

De testsuite bestaat uit drie verschillende scènes.

- Spaceship Indoor



Figuur 3.2: De Spaceship Indoor scène.

- Piper's Alley
- Ziggurat City

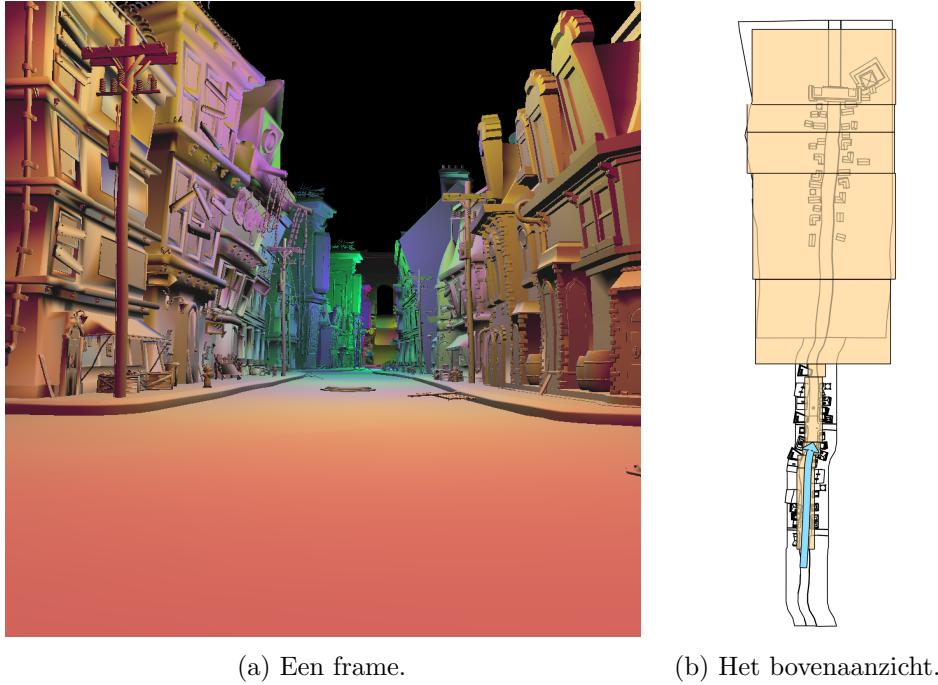
Elk van deze scènes is gedefinieerd als een set van objecten, een camerapad, en een set van lichtconfiguraties.

De scènes zijn zo geselecteerd dat zij representatief zijn voor mogelijke scènes in games: een afgesloten ruimte, een openlucht straat, en een grote openlucht scène. Hierdoor is het mogelijk om de verschillende lichttoekenningsalgoritmes te evalueren bij verschillende schalen en dieptes.

Elk van de scènes is gecreëerd in Blender. De verschillende objecten zijn geexporteerd als .obj bestand. De lichtconfiguraties zijn gegenereerd aan de hand van lichtgeneratievolumes. Een lichtgeneratievolume is een rechthoekig blok, dat per lokale as beschrijft hoeveel lichtbronnen er relatief in gegenereerd dienen te worden. Aan de hand van deze lichtgeneratievolumes worden lichtbronnen met een gespecificeerde radius en een willekeurige tint gegenereerd. Deze lichtbronnen worden uniform over de ruimte verdeeld.

Elk van de scènes zal in de volgende secties verder worden toegelicht. Een volledig overzicht van de scènes en gerelateerde data kan gevonden worden in de data-repository¹⁰

3. METHODE-OVERZICHT



(a) Een frame.

(b) Het bovenaanzicht.

Figuur 3.3: De Piper's Alley scène.

3.3.1 Indoor: Spaceship Indoor

De Spaceship Indoor scène, weergegeven in figuur 3.2a is gebaseerd op de CG Lighting Challenge #18, gemodelleerd door Juan Carlos Silva.¹¹ Deze scène staat model voor indoor-scènes. De scène bestaat uit een middenstuk en een omliggend gangenstelsel. De grootste bron van details is afkomstig van de panelen in de gangen.

Het camerapad en de lichtgeneratievolumes zijn weergegeven in figuur 3.2b. De camera, weergegeven met de blauwe pijl, beweegt zich door de gangen en kruist hierbij twee maal het middenstuk. Het pad bestaat uit 519 frames. De lichtbronnen in het middenstuk hebben een radius van 30.0, de lichtbronnen in het gangenstelsel een radius van 23.0. De lichtgeneratievolumes voor deze lichtbronnen zijn respectievelijk weergegeven met rode en gele blokken.

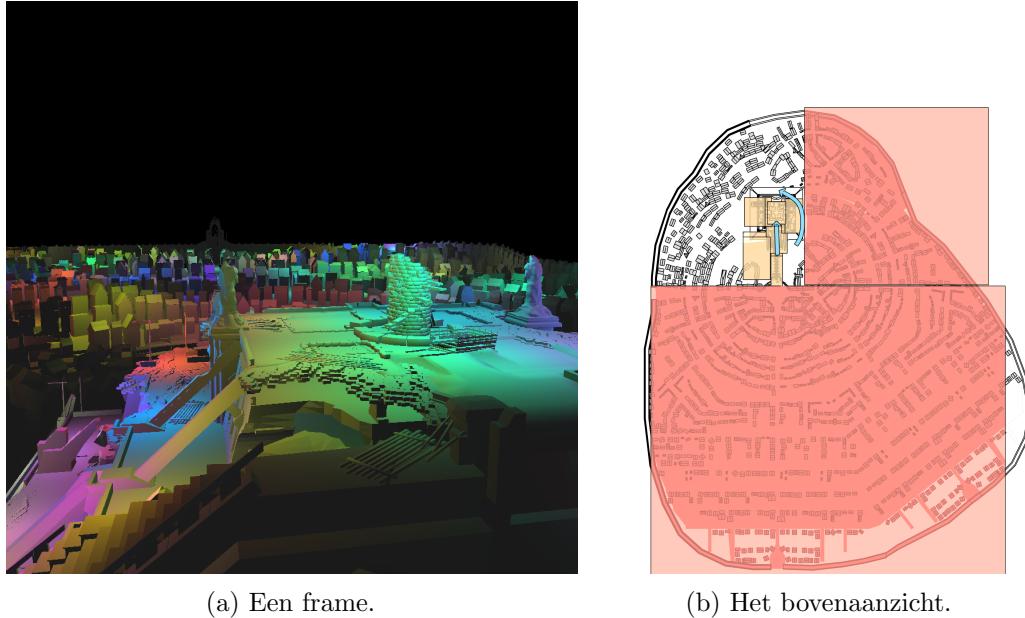
3.3.2 Straatzicht: Piper's Alley

De Piper's Alley scène, weergegeven in figuur 3.3a is gebaseerd op de CG Lighting Challenge #42, gemodelleerd door Clint Rodrigues.¹² De scène beschrijft een enkele straat met grote diepte, waar aan weerszijde huizen zijn geplaatst. In de verre diepte

¹⁰de scenes in de data-repository kunnen gevonden worden op github.com/BeardedPlatypus/thesis-data-suite/tree/master/scenes

¹¹Scene url: www.3drender.com/challenges/

¹²Lighting challenge url: forums.cgsociety.org/archive/index.php?t-1309021.html



(a) Een frame.

(b) Het bovenaanzicht.

Figuur 3.4: De Ziggoerat stadsscene.

zijn een kloktoren en triomfboog zichtbaar. De gebouwen zijn de bron van de meeste details.

Het camerapad en de lichtgeneratievolumes zijn weergegeven in figuur 3.3b. De camera, weergegeven met een blauwe pijl, beweegt zich door het eerste deel van de straat gedurende 551 frames. Alle lichtconfiguraties zijn gegenereerd met een radius van 180.0. De lichtgeneratievolumes zijn weergegeven met gele blokken.

3.3.3 Stadsscene: Ziggurat City

De Ziggurat City scène is een onderdeel van de open film Sintel¹³. De scène bestaat uit een gedetailleerde tempelberg waaromheen, in lagere resolutie, huizen en een stadsmuur zijn geplaatst. Deze scène staat model voor grote openlucht scènes.

Het camerapad en de lichtgeneratievolumes zijn weergegeven in figuur 3.4b. De camera, weergegeven met een blauwe pijl, beweegt zich eerst langs de trap omhoog, om vervolgens om de tempelberg heen te vliegen. Het camerapad is 463 frames lang. De lichtbronnen zijn onderverdeeld in twee sets. De lichtbronnen op de ziggoerat hebben een radius van 10.0 en zijn met een grotere dichtheid geplaatst. De lichtbronnen in de stad hebben een radius van 50.0, en zijn verspreid over grotere volumes. De lichtgeneratievolumes voor deze lichtbronnen zijn respectievelijk weergegeven met gele en rode blokken.



Figuur 3.5: Het Cubehelix kleurenpalet lopende van 0 tot 1.

3.4 Data-analyse

Voor de analyse van de performantie van de verschillende lichttoekenningsalgoritmes is zowel naar de uitvoeringstijd als het aantal lichtberekeningen gekeken. Hiervoor zijn voor de volgende relaties geëvalueerd:

- De uitvoeringstijd en aantal lichtberekeningen per frame gedurende een volledige uitvoering.
- Gemiddelde uitvoeringstijd en aantal lichtberekeningen per frame gemiddeld over een volledige uitvoering als functie van het aantal lichtbronnen in de scène.
- Gemiddelde uitvoeringstijd en aantal lichtberekeningen per frame gemiddeld over een volledige uitvoering als functie van de resolutie van de gegenereerde afbeeldingen.

Het aantal lichtberekeningen heeft een significante invloed op de uitvoeringstijd, echter de uitvoeringstijd wordt tevens beïnvloed door de verschillende stappen van elk lichttoekenningsalgoritme. Door beide waarde te evalueren wordt een beter inzicht verkregen in de uitvoeringstijd en performantie van de lichttoekenningsalgoritmes.

Voor elke realtime toepassing is het belangrijk dat de uitvoeringstijden consistent zijn. De uitvoeringstijd per frame gedurende een volledige uitvoering geeft hier inzicht in. De performantie van de lichttoekenningsalgoritme ten opzichte van het aantal lichtbronnen is significant omdat een belangrijk doel van de lichttoekenningsalgoritmes is om een groter aantal lichtbronnen mogelijk te maken. Het is dus van belang dat de tijdscomplexiteit als functie van de lichtbronnen zo klein mogelijk is.

De resolutie is een belangrijke factor voor de hoeveelheid fragmenten die gegenereerd worden. Verder zal deze waarde een invloed kunnen hebben op de performantie van de camera-afhankelijke lichttoekenningsalgoritmes.

Verder is per lichttoekenningsalgoritme gekeken naar de specifieke parameters van de methode, en hoe deze de tijdscomplexiteit en het geheugengedrag beïnvloeden.

Al de analyses zijn gedaan met behulp van SciPy¹⁴, Pandas¹⁵, en Seaborn¹⁶. Zowel de verzamelde data, als de analyses zijn beschikbaar in de data-repository. Voor de warmtekaarten is gebruik gemaakt van het Cubehelix kleurpalet[Gre11], met een startwaarde van 0.5, rotaties van -1.5, en een tint en gamma waarde van 1.0. De kleur overgang van dit kleurpalet is weergegeven in figuur 3.5.

¹³Open film url: durian.blender.org

¹⁴website: www.scipy.org

¹⁵website: pandas.pydata.org

¹⁶website: seaborn.pydata.org

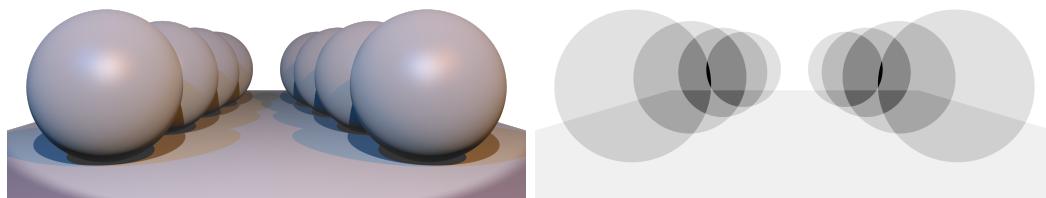
Hoofdstuk 4

Forward en Deferred Shading

Binnen sectie 2.5 is vastgesteld, dat bij standaard uitvoering pas na uitvoering van de fragmentshader 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 shadingcomplexiteit is dus direct gekoppeld aan de geometrische complexiteit van de scène. Voor simpele scènes is dit geen probleem. Echter wanneer scènes 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 gecreëerd 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. Dit concept wordt Deferred Shading genoemd[TNE⁺89].

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.



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 computergrafieken herkend als probleem. De opsplitsing van visibiliteit en belichtingsberekeningen werd toen al voorgesteld. Zo werd in de scan-line polygon renderer van Watkins [Wat70] 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[DWS⁺88, FPE⁺89, Per85, Gla88]. Tebbs, Neumann, Eyles, Turk and Ellsworth[TNE⁺89] geven een gedetailleerd overzicht van Deferred Shading in 1990.

Moderne Deferred Shading algoritmes maken veelal gebruik van GBuffers[Lau10]. Een GBuffer is een verzameling van texturen ter grootte van het zichtvenster. In elke textuur wordt een waarde geassocieerd met een pixel opgeslagen. GBuffers waren oorspronkelijk geïntroduceerd in de context van het non-fotorealistisch renderen om visuele begrijpbaarheid te verbeteren[ST90]. 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 game-engines die gebruik maken van Deferred Shading zijn: Unreal 4[Kar13], Frostbyte 2[Mag11], Unity[PD14] en CryEngine 3[Mit09].

In deze sectie zal ingegaan worden op het algoritme van Deferred Shading om geometrische complexiteit te ontkoppelen van de belichting.

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 explicet de fragmenten op diepte gefilterd, en dus zal voor elk fragment de belichtingsberekeningen uitgevoerd worden.

Deferred Shading beschrijft het algoritme waarbij explicet 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

In de sectie 2.4 is de directe belichtingsbenadering van de rendervergelijking gespecificeert als:

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

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 lichtberekening uitgevoerd in de fragmentshader. Dit betekent, dat voor elk geproduceerd fragment deze berekening uitgevoerd wordt en dat alle informatie per fragment beschikbaar is. Pas nadat de lichtberekening 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 voltooid moet zijn, voordat de kleur van het fragment wordt berekend. Wanneer deze kleur berekening echter plaatsvindt na de verwerking is de informatie over het fragment niet meer impliciet aanwezig. Het is dus nodig om deze expliciet op te slaan ten tijde van de visibiliteitsbepaling

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 reflectiecoëfficient, ruwheid etc.
- Informatie van de lichten
 - Positie
 - Intensiteit
 - Afstandsdiminishingfunctie

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 belichtingsstap beschikbaar gemaakt worden als gedaan wordt in Forward Shading. Indien deze eigenschappen worden bepaald voor elk van de fragmenten met behulp van de grafische pijplijn, 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 radiante te berekenen.

4.1.3 Gbuffer

Om het opslaan van de attributen die nodig zijn voor de lichtberekening te faciliteren wordt een datastructuur gebruik die de *GBuffer* genoemd wordt[ST90]. Dit is een object bestaande uit meerdere texturen, elk verantwoordelijk voor een attribuut

4. FORWARD EN DEFERRED SHADING



Figuur 4.2: De texturen in de GBuffer gebruikt in Killzone 2, geproduceerd door Guerrilla Games[Val09].

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 renderdoelen (Multiple Render Targets (MRT))¹ genoemd. Hiervan wordt gebruik gemaakt om de GBuffer te vullen met informatie in de geometriestap. Deze texturen zullen in het geheugen van de grafische kaart beschikbaar zijn, en opgevraagd kunnen worden gedurende de belichtingsstap.

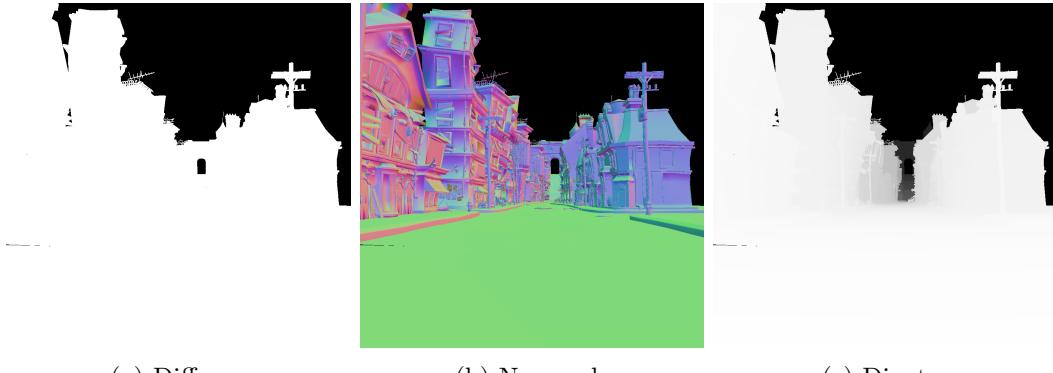
4.2 Algoritme

De voorgestelde opsplitsing van Forward Shading, weergegeven in listing 3, 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.
- Per fragment van dit vlak, bereken de lichtbijdrage van elke lichtbron met dit fragment.

Hierbij wordt de renderpijplijn dus twee maal doorlopen. Eerst om de visibiliteit te bepalen, vervolgens om de lichtberekening uit te voeren[AMHH16].

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)

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

```

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 3: Forward Shading algoritme.

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[Len02]. Voor de fragmenten die geproduceerd worden met een lichtvolume wordt de lichtberekening uitgevoerd met de corresponderende lichtbron en de geometrie attributen bepaald in de geometriestap. 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 4.

Binnen **nTiled** is deze laatste optimalisatie niet geïmplementeerd, en is gekozen om gebruik te maken van zichtveldoverdekkend vlak. De Forward en Deferred Shading algoritmes zonder extra versnelingsstructuren worden binnen **nTiled** respectievelijk **Forward Attenuated** en **Deferred Attenuated** genoemd. Voor de GBuffer binnen **nTiled** is gekozen voor drie texturen waarin drie attributen worden opgeslagen: de diffuse kleur, de normaal en de diepte. Een voorbeeld van deze texturen van de GBuffer is weergegeven in Figuur 4.3.

```
# 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 4: Deferred Shading algoritme.

4.3 Resultaten

De performantie van het Deferred Shading algoritme dat geïntroduceerd werd in de vorige sectie, is geëvalueerd aan de hand van de drie testscènes. Eerst zal het gedrag per frame gedurende een enkele uitvoering besproken worden. Vervolgens wordt gekeken naar de gemiddelde uitvoeringstijd per frame als functie van het aantal lichten, en de resolutie.

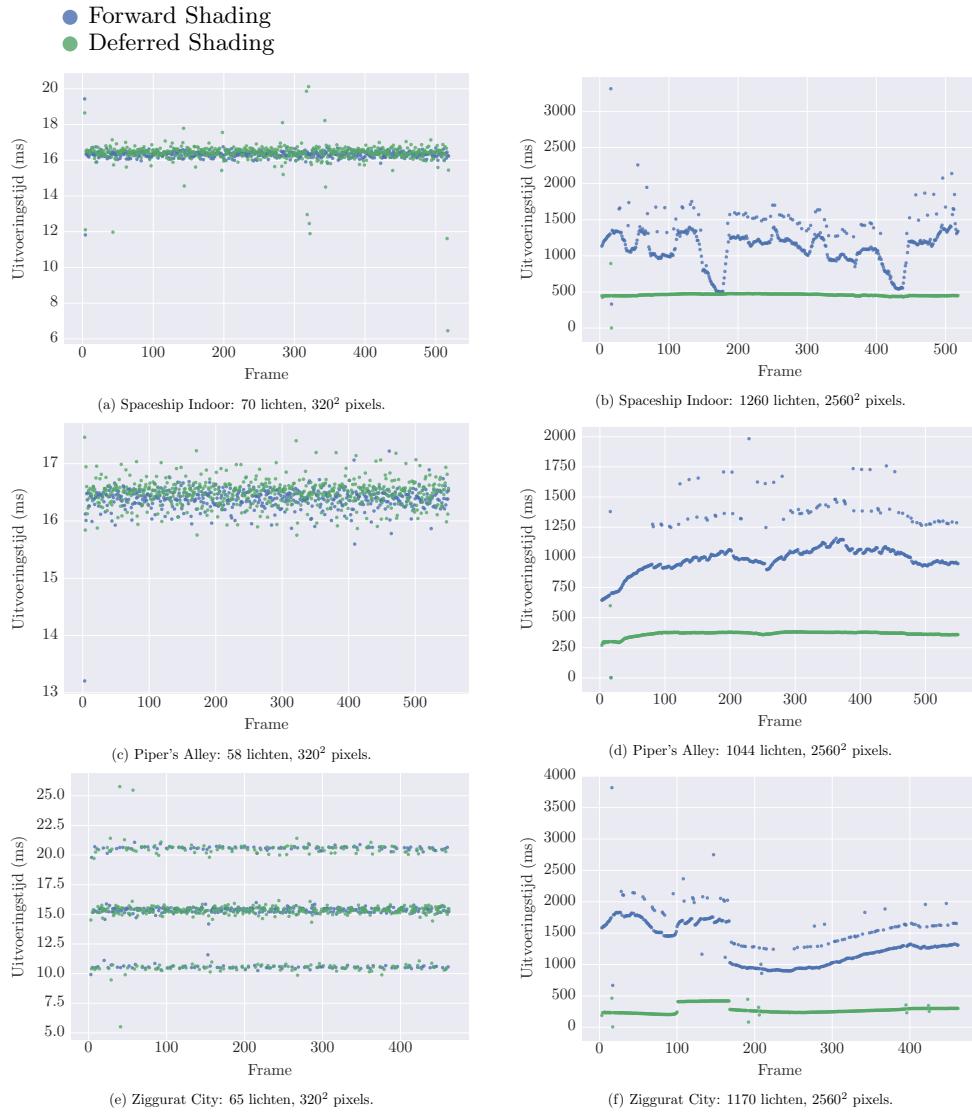
Binnen de testen refereert *Forward Shading* naar de `forward_attenuated` shader en *Deferred Shading* naar de `deferred_attenuated` shader van nTiled.

4.3.1 Frames

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

Gelijk valt hierbij op dat bij een lage resolutie en een klein aantal lichten Forward

4.3. Resultaten

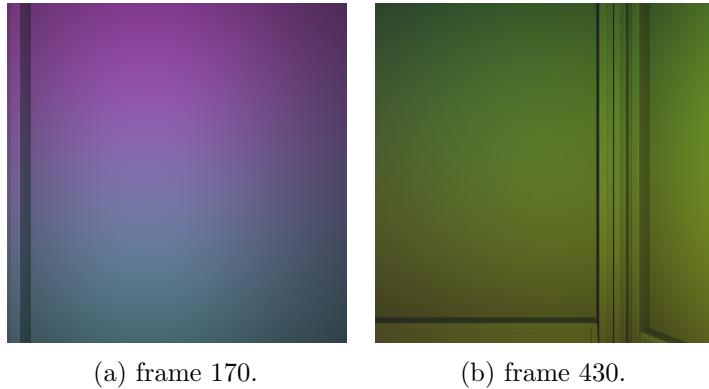


Figuur 4.4: Overzicht van de uitvoeringstijd per frame voor de drie testscènes bij verschillende resolutie en aantal lichten.

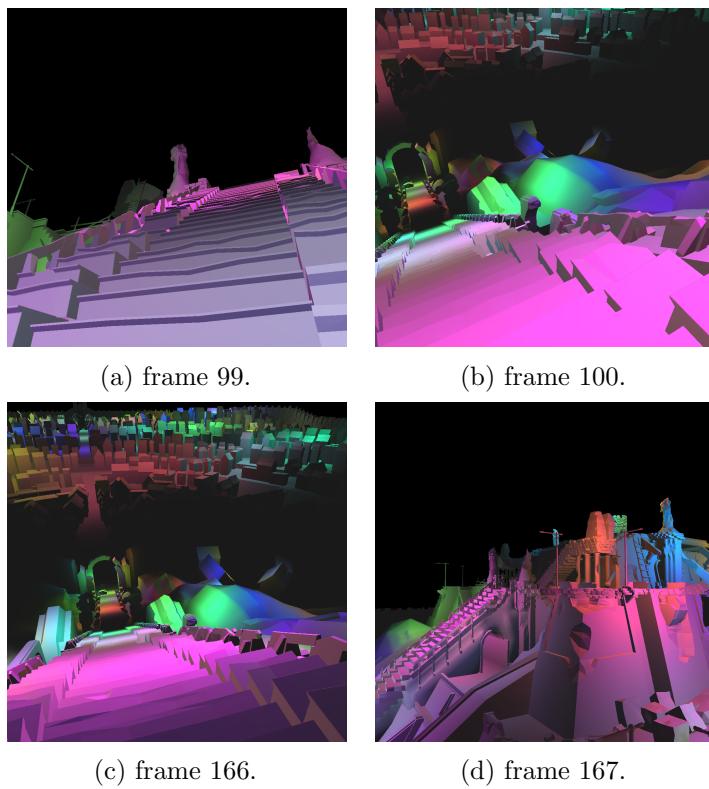
en Deferred Shading elkaar nauwelijks ontlopen. Het verschil in uitvoeringstijd is echter significant bij een hoge resolutie en een groot aantal lichten. Niet alleen is de uitvoeringstijd kleiner bij Deferred Shading, het is tevens consistent. Dit is belangrijk voor computerspellen en andere realtime toepassingen waar een consistente framerate nodig is voor een overtuigende virtuele ervaring.

De consistentie in uitvoeringstijd 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. Er zal dus weinig verschil zijn tussen het aantal uitvoeringen dat gedaan wordt. Een verschil in fragmenten zal hierdoor geen significant verschil in uitvoeringstijd veroorzaken.

4. FORWARD EN DEFERRED SHADING



Figuur 4.5: Spaceship Indoor frames waarbij het aantal fragmenten één benadert.



Figuur 4.6: Ziggurat City frames waarbij de camera van positie verspringt.

Tevens zal de lichtberekening 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 gecreeerd wordt.

Deze koppeling tussen visibiliteit en shadingcomplexiteit is dan ook direct terug te zien in het tijdsgedrag van forward shading. Zo is in de Spaceship indoor scène te zien dat de uitvoeringstijd van Forward Shading de uitvoeringstijd van Deferred

Shading benaderd rond frame 170 en frame 430. Deze frames komen overeen met een zicht op een enkele muur, zoals weergegeven in figuur 4.5a en 4.5b. Hier benaderd het aantal fragmenten per pixel één.

Binnen de Ziggurat city scène, fig. 4.4f is een sprong in uitvoeringstijd te zien bij frame 100 en frame 167. Deze sprongen komen overeen met de verandering van camera, zoals te zien is in figuur 5.8. Het verschil in uitvoeringstijd is een gevolg van het percentage lege ruimte in de drie camerastandpunten. Het zicht tussen frame 100 en frame 167 bestaat voornamelijk uit geometrie. In de frames voor 100 en na 167 is ook een lege lucht waar te nemen.

4.3.2 Lichten

In figuur 4.7 is de gemiddelde uitvoeringstijd per frame weergegeven als functie van het aantal lichten in de scène bij een resolutie van 2560×2560 . Hierbij zijn alle uitvoeringstijden 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 uitvoeringstijd per frame weergegeven als functie van de resolutie bij het grootste aantal lichten per scène. Hierbij komt een resolutie van n overeen met een gebruikte resolutie van $n \times n$. De uitvoeringstijden per frame zijn op eenzelfde manier gemiddeld als de uitvoeringstijd 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.

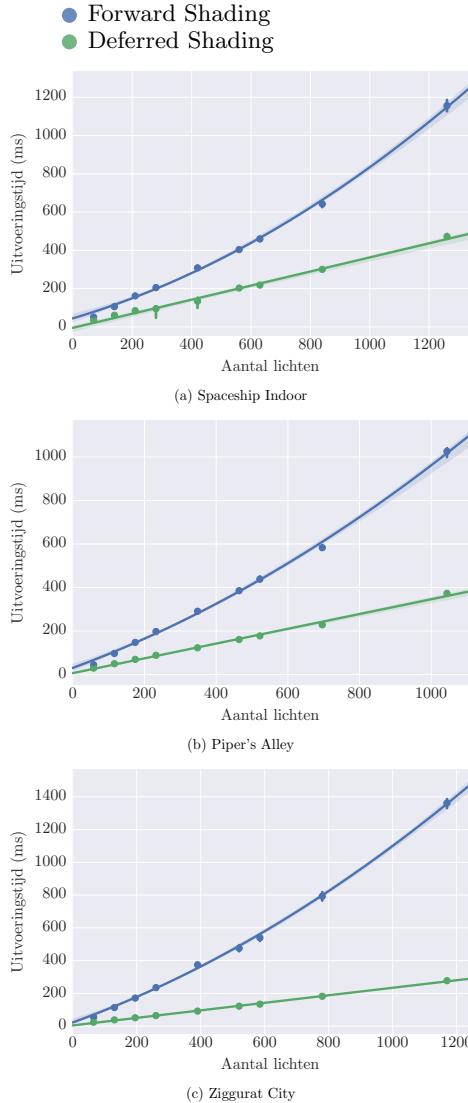
4.4 Conclusie

Uit alle grafieken blijkt dat de gemiddelde uitvoeringstijd kleiner is voor Deferred Shading dan voor Forward Shading. Echter beide technieken hebben eenzelfde tijdscomplexiteit ten opzichte van het aantal lichten.

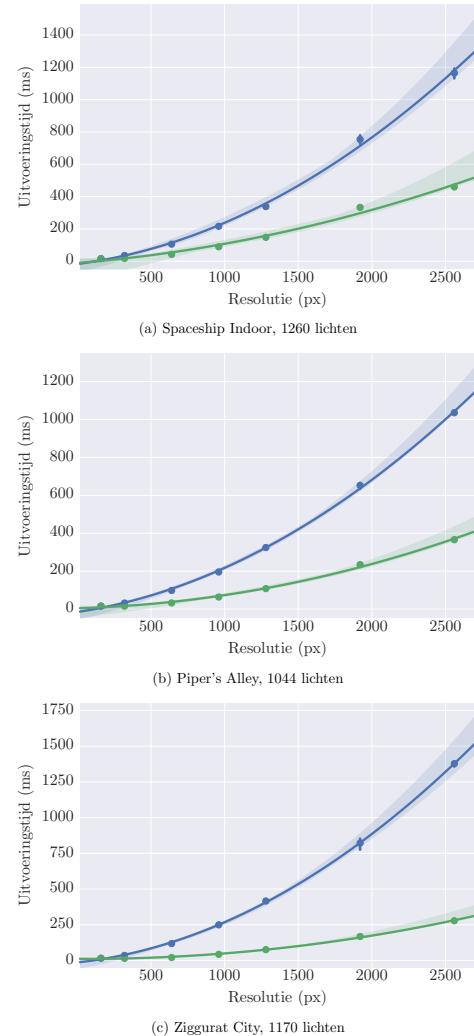
Deferred Shading heeft als bijkomend voordeel dat de uitvoeringstijd consistent is. Dit is een belangrijke eigenschap voor game-engines, waar een consistente framerrate belangrijk is voor de menselijke interpretatie, als ook voor het verdelen van de rekentijd over de verschillende subsystemen.

Tegenover de verbetering in de uitvoeringstijd staat wel dat een set van texturen met een grootte gelijk aan het zichtvenster 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. Wanneer deze steunt op meer attributen dienen meer texturen bijgehouden te worden.

4. FORWARD EN DEFERRED SHADING



Figuur 4.7: Aantal lichten.



Figuur 4.8: Resolutie.

Het gebruik van een GBuffer bij Deferred Shading maakt transparantie onmogelijk, doordat per laag een aparte GBuffer bijgehouden zou moeten worden. Dit is door geheugengebruik van de GBuffer onmogelijk. Deferred Shading is dus beperkt tot ondoorzichtige geometrie.

Het laatste minpunt is dat anti-aliasing niet meer triviaal ondersteund wordt. Binnen de belichtingsstap is het niet mogelijk om arbitrair sub-pixels te bemonsteren, gezien ook deze data expliciet opgeslagen moet worden in de geometriestap. Voor de ondersteuning van anti-aliasing dient de GBuffer uitgebreid worden om subpixels op te slaan.

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 de geheugenbandbreedte significant vergroot wordt. De texturen worden niet langer eenmalig per fragment bemonsterd, maar in het slechtste geval even vaak bemonsterd als er lichten in de scène 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 deze optimalisatie de uitvoeringstijd van Deferred Shading verlaagt 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 game-engines wordt dit ondersteund door een aparte Forward Shading renderstap uit te voeren voor de transparante objecten, nadat de kleuren van de ondoorzichtige objecten berekend zijn[OA11].

4.5.3 Ondersteunen anti-aliasing

Het bemonsteren van subpixels binnen Forward Shading is triviaal. Binnen Deferred Shading is het echter niet mogelijk om subpixels arbitrair te bemonsteren zonder dat de data hiervan opgeslagen moet worden in de GBuffer. Er zijn verschillende aanpakken voorgesteld om anti-aliasing mogelijk te maken binnen Deferred Shading[Lau10].

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[Eng09], 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.

Hoofdstuk 5

Tiled Shading

In het vorige hoofdstuk is Deferred Shading geïntroduceerd. Hiermee werd de geometrische complexiteit ontkoppeld van de shadingcomplexiteit. Er zijn enkele significante nadelen aan het gebruik van een Deferred pijplijn. Indien gebruik gemaakt wordt van een stencil optimalisatie, vereist de Deferred pijplijn dat per licht de informatie van een fragment uit de GBuffer opgehaald dient te worden. Dit leidt tot een hoge geheugenbandbreedte. Indien een dergelijke optimalisatie niet is geïmplementeerd, dient voor elk fragment, elk licht in de scene geëvalueerd te worden.

Om deze problemen tegen te gaan is de techniek Tiled Shading voorgesteld[OA11]. Het achterliggende idee is om het zichtveld onder te verdelen in tegels bestaande uit $n \times n$ pixels, voordat de belichtingsberekening wordt uitgevoerd. Voor elke tegel wordt bepaald welke lichten gedeeltelijk overlappen met de tegel. Deze lichten worden bijgehouden in een lijst geassocieerd met de tegel. Vervolgens worden tijdens de lichtberekening slechts de lichten geëvalueerd van de tegel waartoe het fragment behoort. Een voorbeeld van een dergelijke onderverdeling is weergegeven in figuur 5.1.

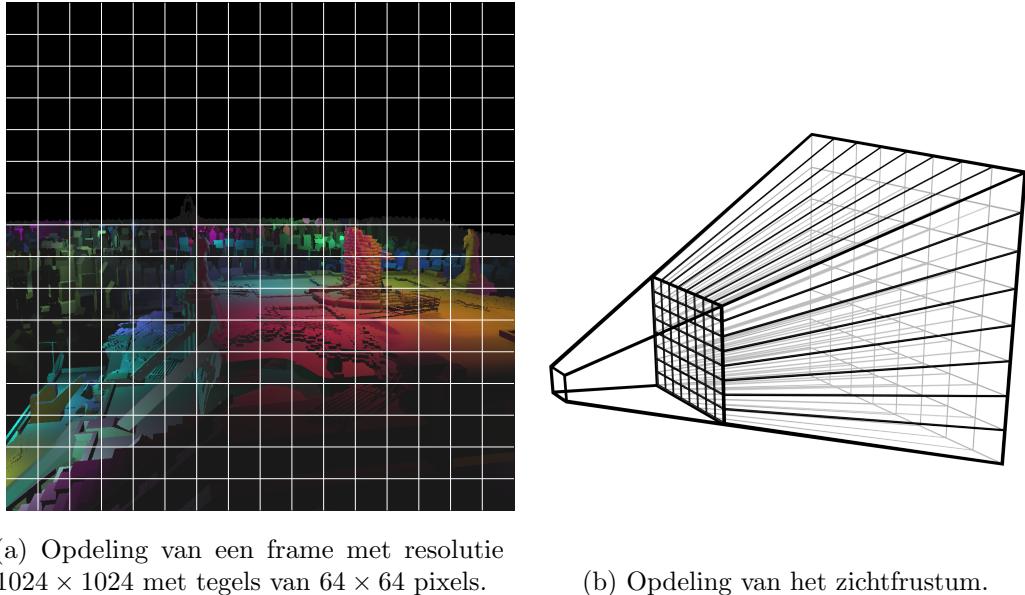
In dit hoofdstuk wordt het Tiled Shading algoritme verder toegelicht. Eerst zal de achterliggende theorie behandeld worden. Vervolgens zal ingegaan worden op het algoritme en de bijbehorende datastructuren. Als laatste wordt de implementatie geëvalueerd aan de hand van de drie testscènes en vergeleken met de naïeve shaders.

5.1 Theorie

Zoals vermeld in hoofdstuk 4, vereist Deferred Shading met een stencil-optimalisatie een hoge geheugenbandbreedte, doordat voor elk licht opnieuw de relevante data uit de GBuffer opgehaald moet worden. In het geval van Forward Shading, of als er geen stencil-optimalisatie is geïmplementeerd in de Deferred pijplijn, dient voor elk fragment de lichtberekening geëvalueerd worden voor alle lichten in de scene. Beide aanpakken hebben dus significante nadelen.

Tiled Shading is geïntroduceerd om beide problemen te verlichten[OA11]. Binnen Tiled Shading wordt het zichtveld onderverdeeld in een verzameling van tegels, zoals

5. TILED SHADING



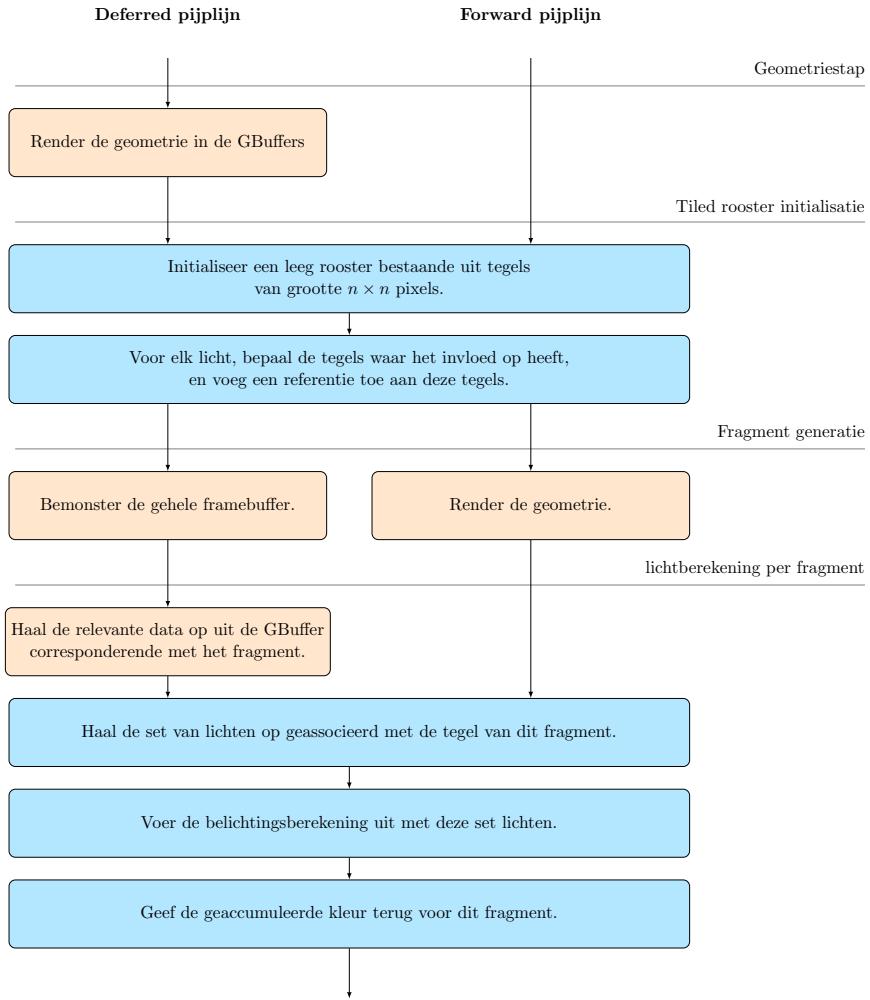
Figuur 5.1: Opdeling van het zichtveld.

weergegeven in figuur 5.1a. Hierdoor wordt het zichtfrustum opgedeeld zoals weergegeven in figuur 5.1b. Voor elk van de tegels wordt vervolgens bepaald welke lichten overlappen met de tegel. Deze verzameling van lichten kan dan opgehaald worden tijdens de belichtingsberekening, en zo het aantal te evalueren lichten beperken.

Doordat het mogelijk is om per fragment direct een verzameling van relevante lichten op te halen, kan gebruik gemaakt worden van Deferred Shading zonder stencil-optimalisatie. Hierdoor hoeft per fragment slechts eenmaal de GBuffer uitgelezen te worden. Tegelijkertijd blijft het aantal lichten dat geëvalueerd dient te worden tijdens de lichtberekening beperkt. Dit verlaagt de geheugenbandbreedte significant. Deze opdeling kan zowel toegepast worden in Deferred Shading als Forward Shading. Op deze manier kan dus ook de uitvoeringstijd van in Forward Shading beperkt worden.

Deze omzetting leidt tot een verschil in lus-structuur tussen Tiled Shading en Deferred Shading met stencil-optimalisatie[OA11]. Deferred Shading met stencil optimalisatie creëert per licht fragmenten, waardoor de binnenste lus over pixels loopt. Hierdoor zal per licht de data uit de GBuffer opgehaald worden. Tiled Shading en de naïve implementaties van Forward en Deferred Shading daarentegen creëren eerst de fragmenten, en overlopen dan de lichten. Hierdoor zal elk fragment slechts eenmalig overlopen worden. Dus wordt het de GBuffer ook maar één keer per fragment aangesproken. Dit verschil lost het geheugenbandbreedte-probleem op.

Het principe om het zichtveld op te delen, om zo de complexiteit in een sub-veld te verlagen is geen nieuw idee geïntroduceerd met Tiled Shading. Zo maakte de Pixel-planes 5 computer een vergelijkbaar concept om de geometrie op te delen, om zo per sub-vlak het aantal te evalueren polygonen te verlagen[FPE⁺89]. Verschillende Tiled Shading implementaties zijn gebruikt in games[BE08, And09, Swo09].



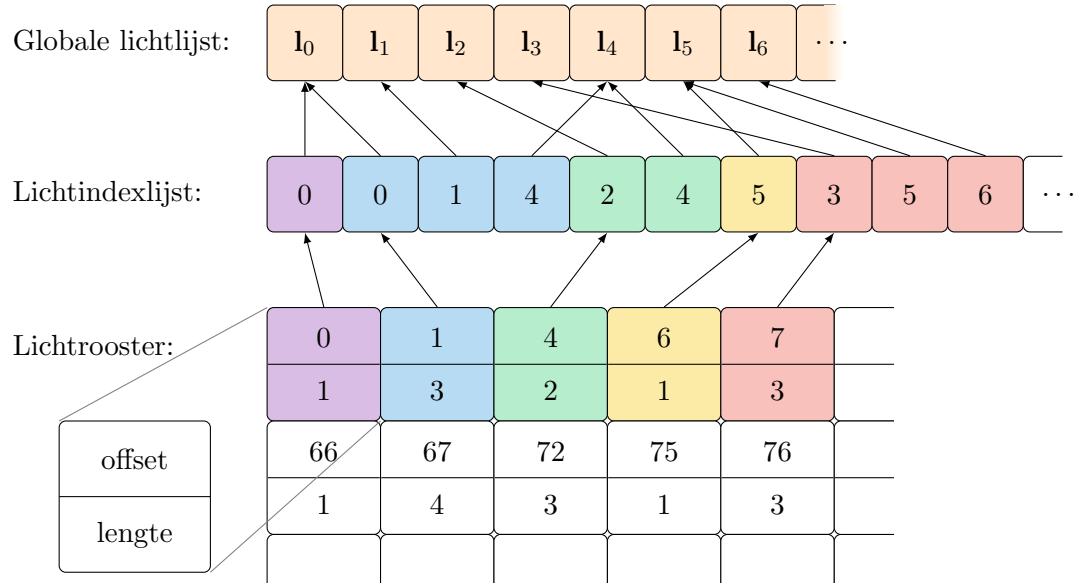
Figuur 5.2: Het Tiled Shading algoritme voor de Forward en Deferred pijplijn.

5.2 Algoritme

Het Tiled Shading algoritme[OA11] voor zowel de Forward als Deferred pijplijn is weergegeven in figuur 5.2. Hierbij zijn de pijplijn-specifieke stappen in het geel weergegeven, en de Tiled Shading stappen in het blauw. Het algoritme bevat grofweg twee extra stappen boven op het naïeve algoritme, een initialisatiestap en een lichtbepalingsstap. Deze stappen zijn in grote mate onafhankelijk van de gekozen pijplijn.

Tijdens de initialisatiestap wordt het rooster van tegels opgebouwd. Eerst worden lege tegels geïnitialiseerd. Vervolgens wordt voor elk licht bepaald met welke tegels deze overlapt. Voor de lichtberekeningsstap wordt niet meer elk lichtvolume gerasteriseerd, noch wordt elk licht geëvalueerd. In plaats hiervan wordt aan de hand van de fragmentpositie de tegel waartoe het fragment behoort, bepaald. Hierna wordt de set van lichten geassocieerd met deze tegel opgehaald.

5. TILED SHADING



Figuur 5.3: De datstructuren van Tiled Shading.

5.2.1 Datastructuren

De gebruikte datastructuur dient een rooster van tegels bij te houden. Elke tegel dient een variabel aantal referenties naar lichten bij te houden. Dit rooster dient efficiënt opgebouwd te worden. Verder dient, wanneer de tegel bepaald is, de set van lichten geassocieerd met deze tegel efficiënt op te halen zijn. Om dit rooster voor te stellen wordt gebruik gemaakt van drie arrays[OA11]:

Globale lichtlijst: bevat alle lichten. Deze array is in dezelfde vorm aanwezig in de naïeve shaders.

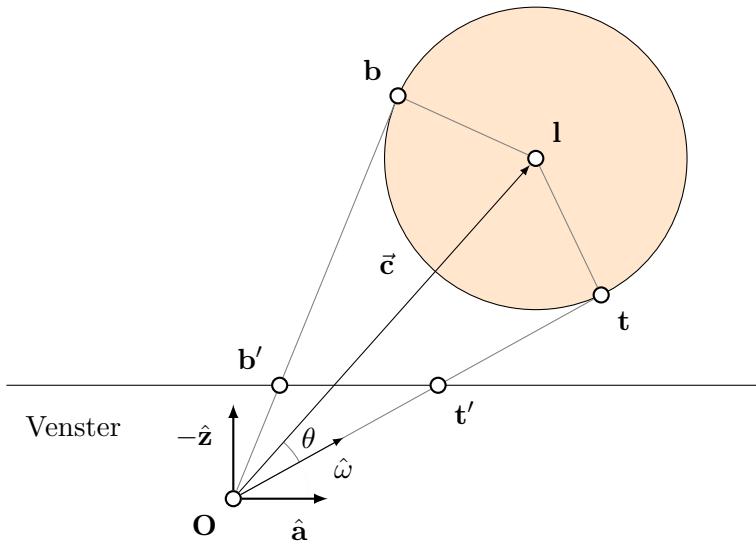
Lichtindexlijst: bevat lichtindices die verwijzen naar lichten in de globale lichtlijst. Deze array is dus een lijst van alle referenties van alle tegels.

Lichtrooster: bevat voor elke tegel één vector die een offset en aantal lichten geassocieerd met een tegel specificeert.

De relatie tussen de drie arrays is weergegeven in figuur 5.3. Deze structuren zijn op de GPU voor te stellen met behulp van bufferobjecten of texturen.

5.2.2 Lichtbepaling

Nu de voorstelling van het rooster gedefinieerd is, is het mogelijk om de lichtberekening op te stellen. Hiervoor dient voor een fragment de set van relevante lichten bepaald te worden. Om dit te bereiken, wordt eerst de tegel waartoe een fragment behoort, bepaald:

Figuur 5.4: Projectie van lichtbol op de **a**-as.

$$f : (\text{frag.x}, \text{frag.y}) \mapsto \left(\left\lfloor \frac{\text{frag.y}}{n} \right\rfloor, \left\lfloor \frac{\text{frag.y}}{n} \right\rfloor \right)$$

waar **frag** de pixelcoördinaten van het fragment zijn, en *n* de grootte van één tegel in pixels is. Op basis van deze indices kan de offset en aantal lichten opgehaald worden uit het lichtrooster. Vervolgens wordt per lichtindex in de lichtindexlijst geassocieerd met deze tegel, de lichtberekening uitgevoerd met het corresponderende licht[OA11]. De code hiervoor is gedefinieerd in listing 5

5.2.3 Lichttoekenning

Om de datastructuren op te stellen in de roosterinitialisatiestap dient voor elk licht bepaald te worden met welke tegels het lichtvolume overlapt. De volledige set van tegels komt overeen met het zichtvenster. Een simpele manier om de overlapping te bepalen is door de corresponderende lichtvolumes af te beelden op het zichtvenster. Vervolgens kan bepaald worden welke tegels bedekt worden door het geprojecteerde volume[OA11].

Wanneer de grootte van de set van lichten in de honderden is, is het mogelijk om deze berekeningen op de CPU uit te voeren. Wanneer er duizenden lichten zijn, kan deze berekening te traag zijn op de CPU, en zal deze op de GPU geïmplementeerd moeten worden.

Er zijn verschillende algoritmes om deze projectie uit te voeren[Len02, SWBG06]. Binnen **nTiled** is gekozen voor het algoritme voorgesteld door Mara en McGuire [MM12]. Hierbij wordt een compacte, omsluitende veelhoek opgesteld voor de geprojecteerde lichtvolumebollen. In **nTiled** is hier gekozen voor een omsluitend vierkant. Om dit vierkant op te stellen dienen de uiterste waarden in de **x**- en **y**-as

gevonden te worden. Deze uiterste waardes worden dan afgebeeld op het venster. De uiterste waardes kunnen als volgt gevonden worden. Eerst wordt de bol geprojecteerd op het twee-dimensionale vlak parallel aan de as in kwestie. Deze situatie is afgebeeld in figuur 5.4. Vervolgens is het doel om de waardes t_a en b_a te vinden. Deze waardes zijn gedefinieerd als:

$$\begin{aligned} t &= \sqrt{c^2 + r^2} \\ \cos \theta &= \frac{t}{c} \\ \sin \theta &= \frac{r}{c} \\ \hat{\omega}_{t_a} &= \begin{pmatrix} \cos \theta & \sin \theta \\ -\sin \theta & \cos \theta \end{pmatrix} \hat{\mathbf{c}} \\ \hat{\omega}_{b_a} &= \begin{pmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{pmatrix} \hat{\mathbf{c}} \\ t_a &= \mathbf{O} + \hat{\omega}_{t_a} t \\ b_a &= \mathbf{O} + \hat{\omega}_{b_a} t \end{aligned}$$

vervolgens kunnen de punten (t_x, t_y) en (b_x, b_y) geprojecteerd worden op het venster met behulp van de perspectiefmatrix van de camera om de uiterste waarden \mathbf{t}' en \mathbf{b}' te verkrijgen. Om het rooster op te bouwen dient aan elke tegel die overlapt met het vlak gedefinieerd door \mathbf{t}' en \mathbf{b}' een referentie naar het licht toegevoegd te worden.

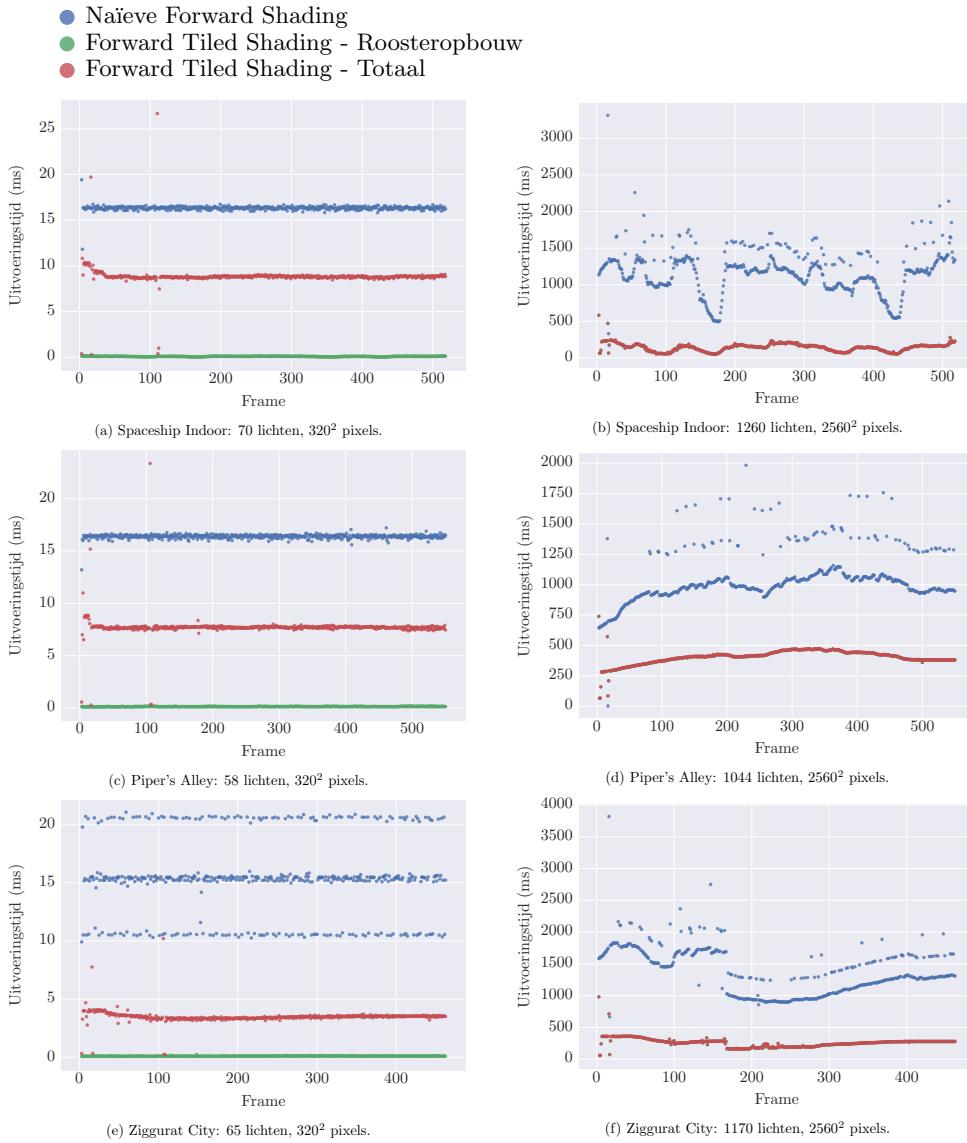
Als extra optimalisatie worden lichten die buiten de diepte van het zichtfrustum vallen niet meegenomen bij het opstellen van het rooster, gezien hier geen fragmenten kunnen vallen. Indien de berekening uitgevoerd wordt op de GPU en een dieptebuffer beschikbaar is, kunnen alle lichten die buiten de minimale en maximale diepte van fragmenten in de tegel vallen, tevens buiten beschouwing gelaten worden[OA11]. Deze optimalisatie is niet geïmplementeerd binnen **nTiled**.

5.3 Resultaten

De performantie van het Tiled Shading algoritme geïntroduceerd in de vorige sectie, is geëvalueerd aan de hand van de drie testscenes. Hiervoor zijn de uitvoeringstijden van naïef Forward Shading en Forward Tiled Shading, en naïef Deferred Shading en Deferred Tiled Shading vergeleken. Daarnaast zijn het aantal lichtberekeningen per frame van naïef Deferred Shading en Deferred Tiled Shading vergeleken.

Er zal eerst gekeken worden naar de uitvoeringstijd en aantal lichtberekeningen per frame gedurende een enkele uitvoering. Vervolgens wordt gekeken naar de uitvoeringstijd en aantal lichtberekeningen als functie van het aantal lichten en de resolutie.

5.3. Resultaten

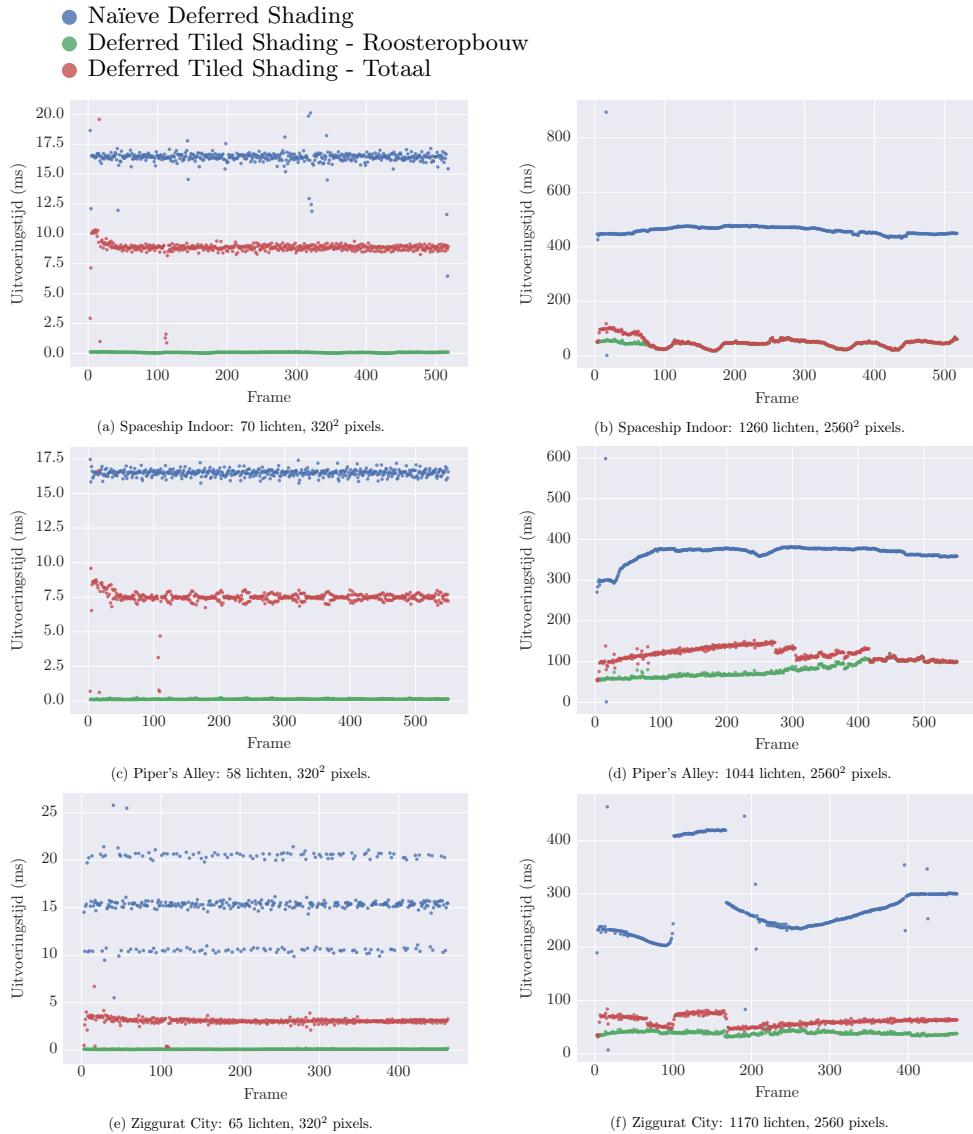


Figuur 5.5: Overzicht van de uitvoeringstijd voor Forward Shading per frame voor de drie testscénes bij verschillende resoluties en aantal lichten.

5.3.1 Frames

De uitvoeringstijden per frame voor Forward Shading zijn weergegeven in figuur 5.5, en de uitvoeringstijden per frame voor Deferred Shading in figuur 5.6. Het aantal lichtberekeningen per frame gedurende een volledige uitvoering zijn weergegeven in figuur 5.7. In elk van deze figuren geven de grafieken links de uitvoeringen weer van een klein aantal lichten bij een resolutie van 320 × 320 pixels. De rechter grafieken geven de uitvoeringen met een groot aantal lichten bij een resolutie van 2560 × 2560 pixels weer. Het Tiled Shading algoritme is bij de uitvoeringstijdmetingen uitgevoerd

5. TILED SHADING

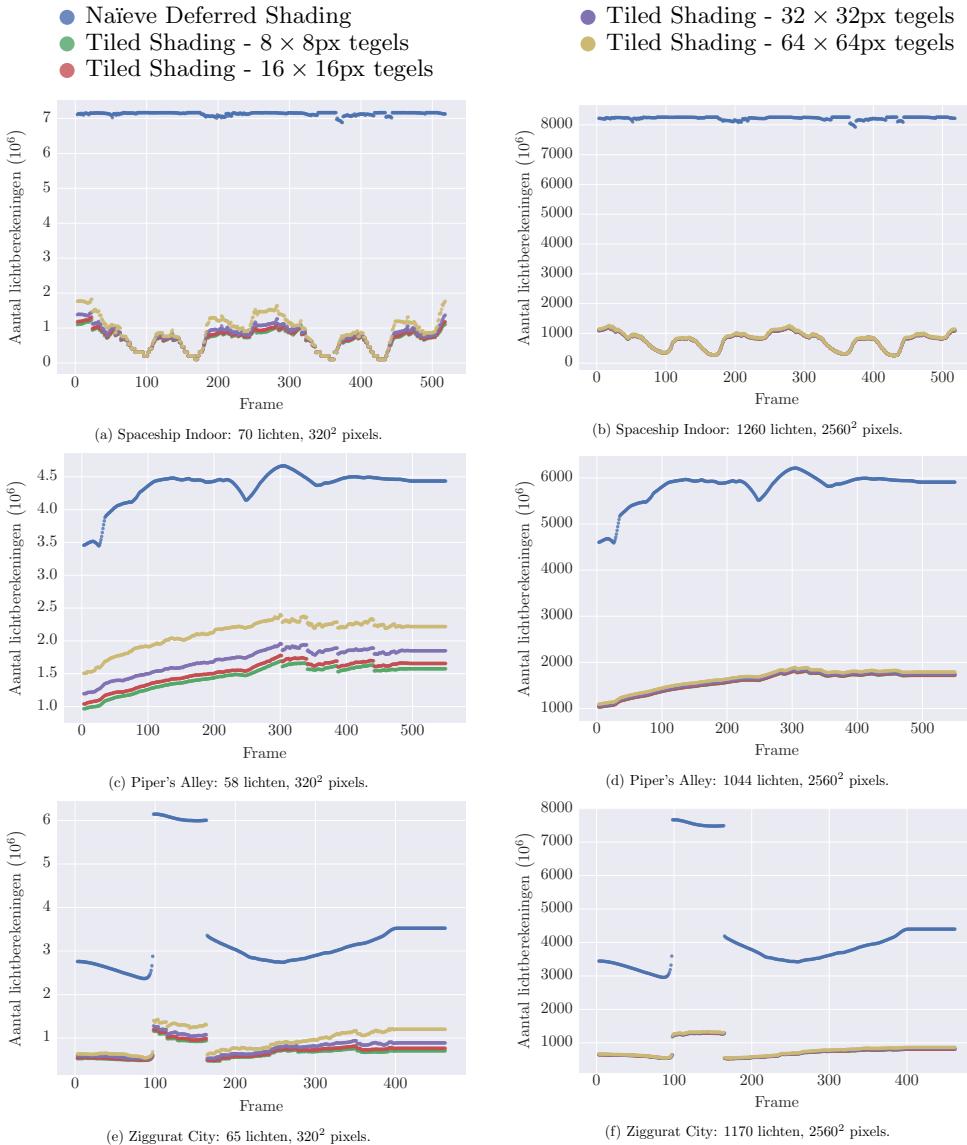


Figuur 5.6: Overzicht van de uitvoeringstijd voor Deferred Shading per frame voor de drie testscénes bij verschillende resoluties en aantal lichten.

met een tegelgrootte van 32×32 pixels.

In zowel de lage als hoge resolutie uitvoeringen presteert Tiled Shading beter dan de naïeve implementatie. De lage resolutie uitvoeringen worden tussen de anderhalf en twee maal sneller uitgevoerd met Tiled Shading, ten opzichte van de naïeve implementatie. Hierbij is geen significant verschil waar te nemen tussen Forward en Deferred Shading. Dit komt overeen met de resultaten van het vorige hoofdstuk. Bij deze lage resolutie en klein aantal lichten kost het opbouwen van het rooster praktisch geen tijd. Het reduceert echter wel significant het aantal lichtberekeningen, zoals weergegeven in figuur 5.7. De tegelgrootte heeft slechts een kleine invloed op

5.3. Resultaten

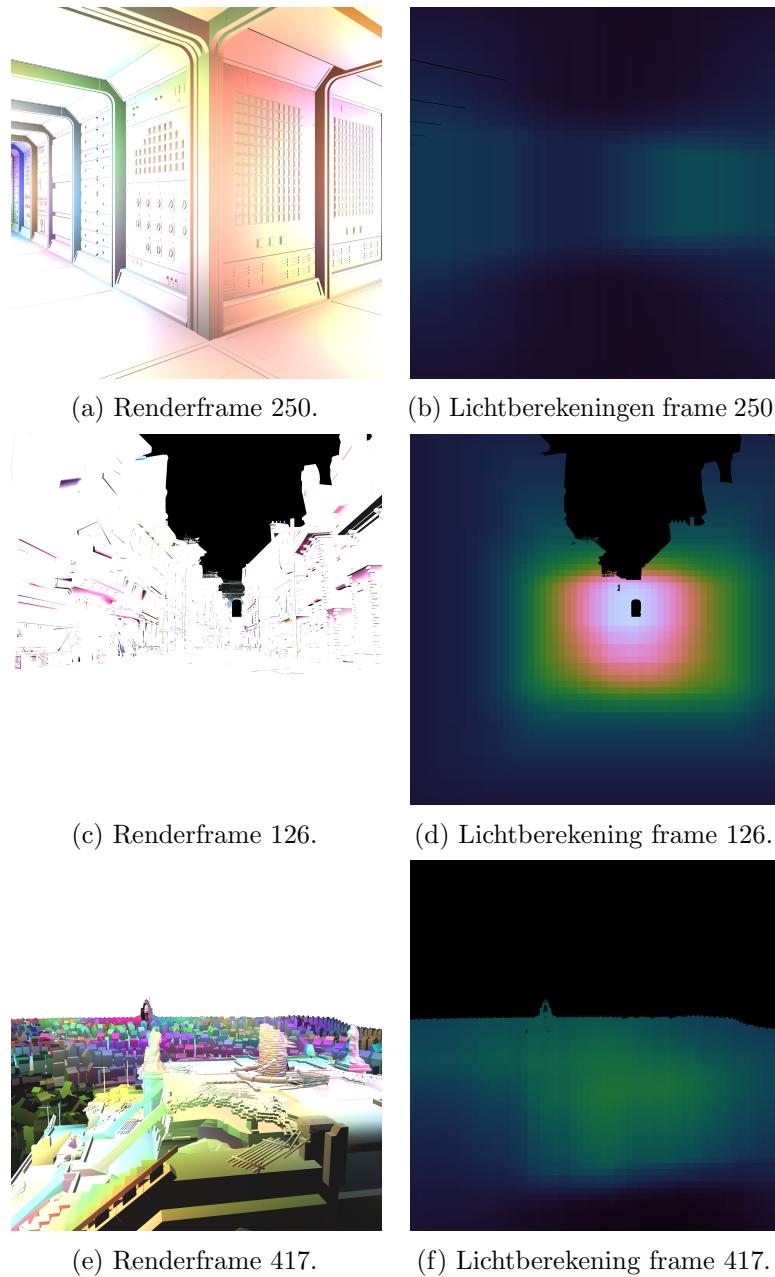


Figuur 5.7: Overzicht van het aantal lichtberekeningen per frame voor Deferred Shading voor de drie testscénes bij verschillende resoluties en aantal lichten.

het aantal lichtberekeningen per frame. Waarbij kleinere tegels leiden tot minder lichtberekeningen.

Bij een hogere resolutie en een groter aantal lichten is de invloed van het lichttoekenningsalgoritme significanter. Alle Tiled Shading uitvoeringen worden ongeveer viermaal sneller uitgevoerd dan de naïeve tegenhanger. Tevens zijn de uitvoeringstijden voor Forward Shading consistent. Dit is vooral zichtbaar in de Spaceship Indoor scène, fig. 5.5b, waar de schommelingen afhankelijk van de camerapositie, minder hevig zijn. Dit duidt er op dat er minder lichtberekeningen per fragment worden uitgevoerd, in vergelijking tot de naïeve implementatie, waardoor de totale uitvoe-

5. TILED SHADING

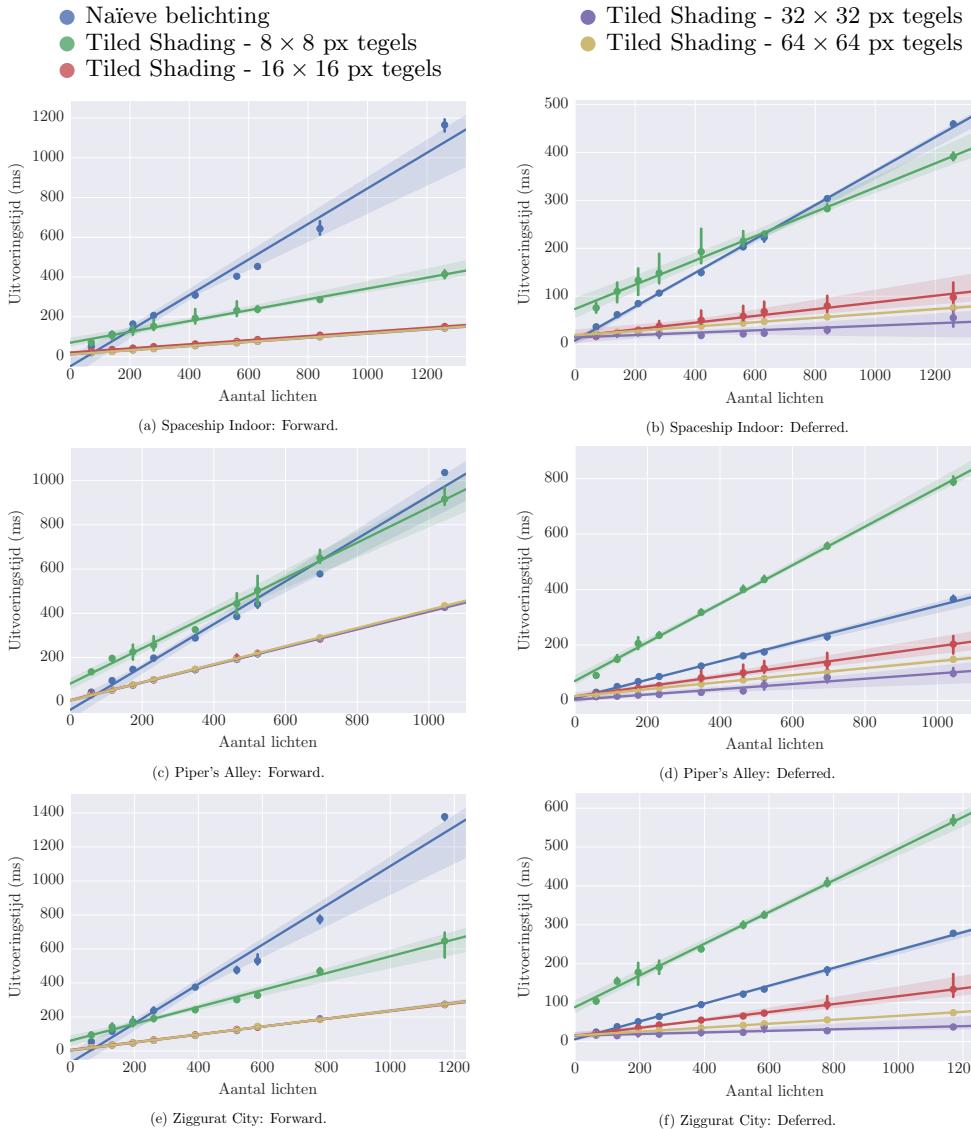


Figuur 5.8: Renders en hittekaarten van de verschillende scénes voor Tiled Shading.

ringstijd minder toeneemt wanneer een groot aantal fragmenten op één pixel vallen. Dit komt overeen met de reductie in het aantal lichtberekeningen die waargenomen wordt in figuur 5.7a en 5.7b.

Bij hogere resoluties en meer lichten is de opbouw van het rooster de tijdsbepalende stap. Dit is een direct gevolg van het groter aantal lichten, en de relatief bruteforce aanpak waarmee lichten worden toegekend aan tegels. Daarnaast neemt door de

5.3. Resultaten



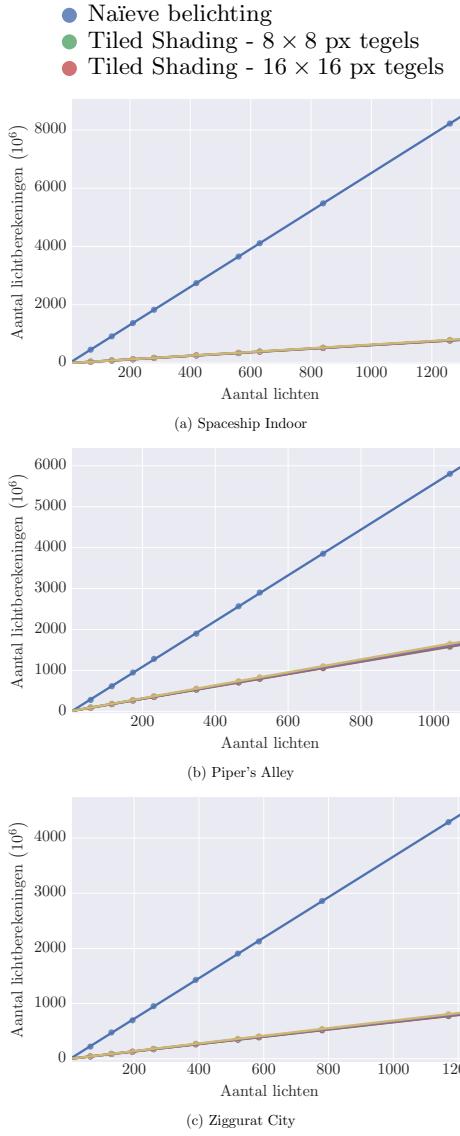
Figuur 5.9: Overzicht van de uitvoeringstijd per aantal lichten bij een resolutie van 2560^2 pixels voor de drie testscénes.

hogere resolutie het aantal tegels ook toe, waardoor een licht met meer tegels zal overlappen. Hierdoor zal tevens meer tijd nodig zijn om het licht aan alle tegels toe te wijzen. Verder is de gebruikte shader erg simpel. Bij een complexere shader zal een groter percentage van de uitvoeringstijd besteed worden aan de lichtevaluatie.

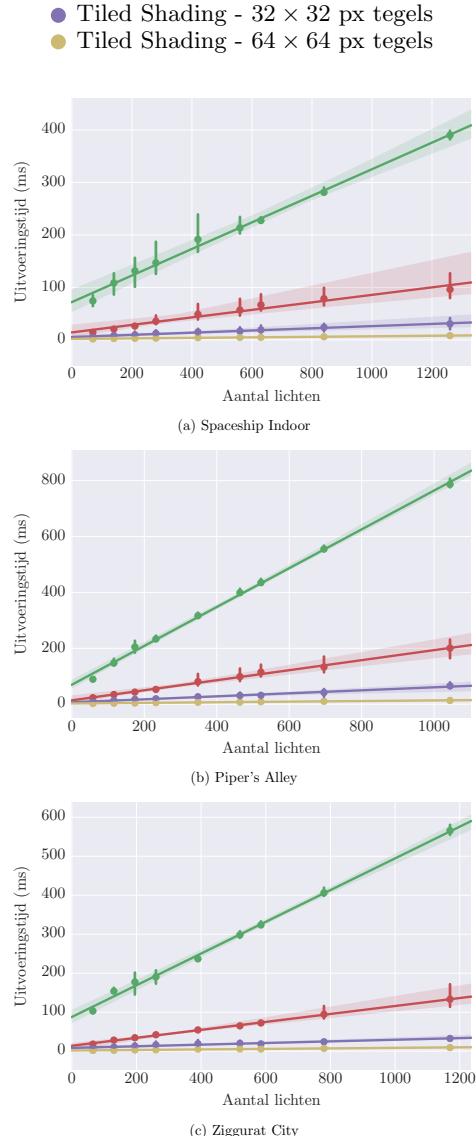
Voor alle Deferred Shading uitvoeringen reduceert het Tiled Shading algoritme het aantal berekeningen per frame met ongeveer een factor vier. Hierbij valt tevens op te merken dat voor de hoge resolutie, de grootte van de tegels geen significante invloed heeft op het aantal berekeningen.

Binnen de Ziggurat City scène, fig. 5.5f en 5.6f, is het verschil tussen de verschil-

5. TILED SHADING



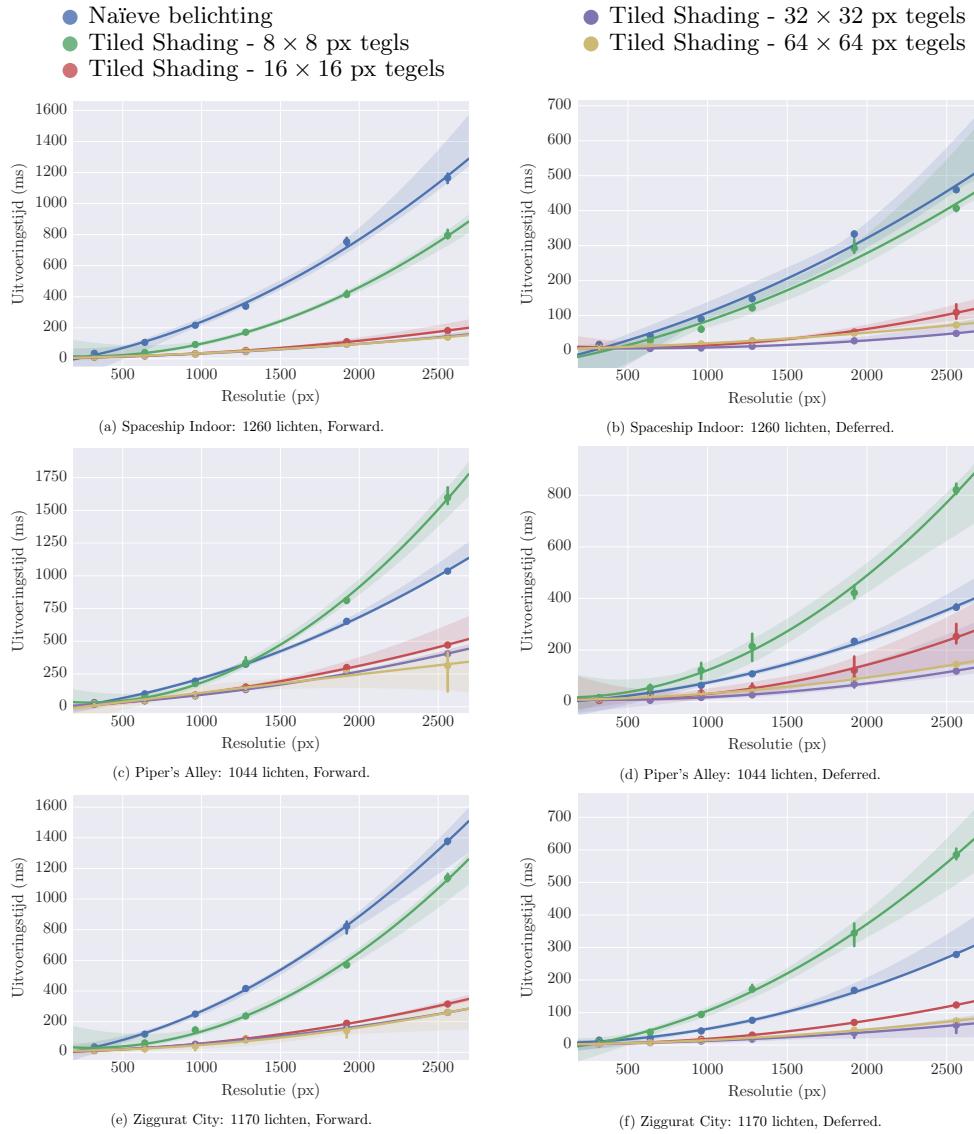
Figuur 5.10: Lichtberekeningen.



Figuur 5.11: Lichtroosteropbouw.

lende camerapunten minder nadrukkelijk aanwezig. In sectie 4.3.1 werd al vastgesteld dat het verschil in uitvoeringstijd tussen de camerapunten in grootte mate afhankelijk was van het percentage pixels zonder fragmenten. Hierbij bevat het tweede camerapunt een lager percentage lege pixels. Dit verschil is duidelijk waar te nemen in het aantal lichtberekeningen, fig. 5.7f. De fragmenten die gegenereerd worden bij de tweede camerapositie, liggen veelal in het deel van de Ziggurat City scène die verlicht wordt door enkele grote lichten, fig. 3.4b. Hierdoor zal de set van lichten geassocieerd met deze tegels kleiner zijn. Dit effect is duidelijk zichtbaar in figuren 5.7e en 5.7f waar het verschil in aantal lichtberekeningen bijna een factor acht is. Dit verschil leidt ertoe dat het verschil in uitvoeringstijd kleiner is.

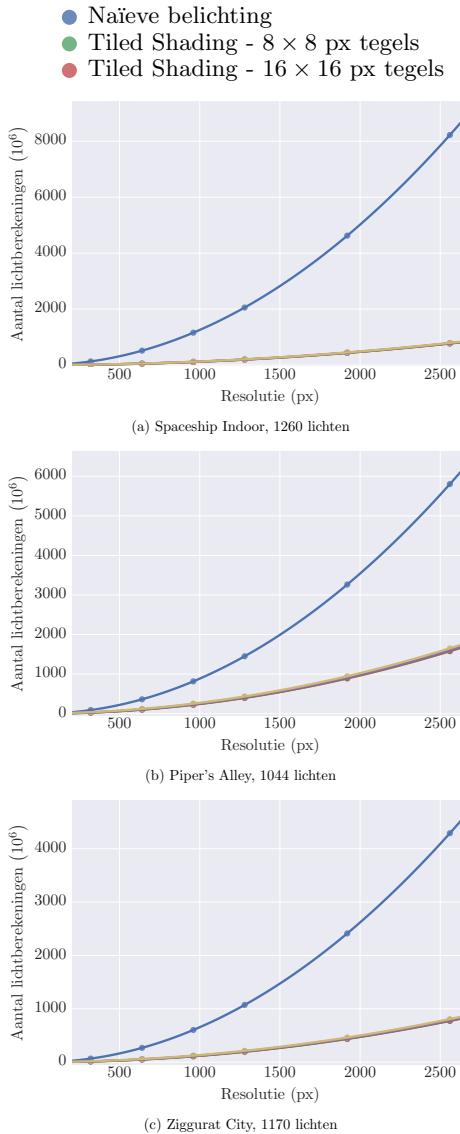
5.3. Resultaten



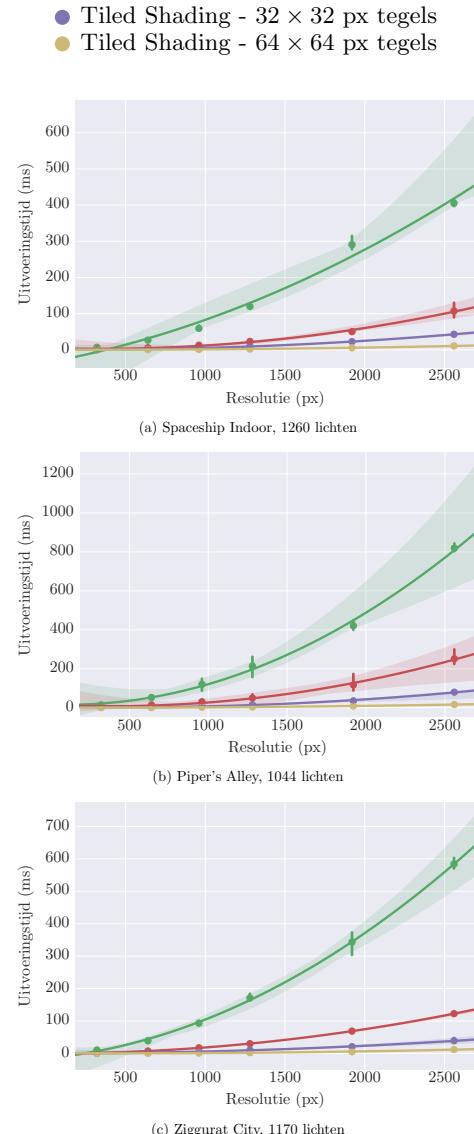
Figuur 5.12: Overzicht van de uitvoeringstijd per resolutie voor de drie testscénes.

Het verschil in efficiëntie is het kleinst in de Piper's Alley scène. Dit kan verklaard worden aan de hand van de opbouw van de scène. De Piper's Alley scène is één diepe straat, waar de lichten achter elkaar zijn geplaatst, zie fig. 3.3b. Wanneer gekeken wordt naar de onderverdeling van het zichtfrustum, fig. 5.1b is te zien dat dergelijke scènes, met veel overlappende lichtvolumes in de diepte, leidt tot de slechtst mogelijke situatie. De tegels zullen een groot aantal lichten bevatten door deze overlap. Hierdoor zal voor elk van de pixels in deze tegels een groot aantal lichtberekeningen nodig zijn. Hierdoor zullen deze tegels het tijdsgedrag van de naïeve implementatie benaderen. Een vergelijkbaar effect is niet zichtbaar in de Spaceship Indoor en Ziggurat City scènes.

5. TILED SHADING



Figuur 5.13: Lichtberekeningen.



Figuur 5.14: Lichtroosteropbouw.

5.3.2 Lichten

In figuur 5.9 zijn de gemiddelde uitvoeringstijden per frame per uitvoering geplot, als functie van het aantal lichten in de scène, voor zowel Forward Shading, links, en Deferred Shading, rechts. In figuur 5.10 zijn het gemiddeld aantal lichtberekening per frame als functie van het aantal lichten in de scène weergegeven voor Deferred Shading. Als laatste is de uitvoeringstijd voor het opbouwen van de lichtroosters weergegeven in figuur 5.11. Deze testen zijn uitgevoerd bij een resolutie van 2560×2560 pixels.

Er is voor zowel de naïeve implementatie, als de Tiled implementatie een lineair verband waar te nemen in het aantal lichtberekeningen dat uitgevoerd wordt. De keuze

van de tegelgrootte heeft geen significante invloed op het aantal lichtberekeningen.

Wanneer gekeken wordt naar de uitvoeringstijd van de verschillende Tiled Shading varianten, valt op dat de tegelgrootte van 8×8 pixels, en in mindere mate de tegelgrootte van 16×16 pixels, slecht presteren. Dit effect is toe te schrijven aan de tijd die nodig is om het lichtrooster op te stellen, zoals weergegeven in figuur 5.11. Wanneer het rooster uit meer tegels bestaat, zal de opbouw meer uitvoeringstijd vereisen. Gezien de tegelgrootte geen significante invloed heeft op het aantal lichtberekeningen, leidt dit ertoe dat kleinere tegelgroottes slechter presteren.

5.3.3 Resolutie

De gemiddelde uitvoeringstijden per frame per uitvoering als functie van de resolutie zijn weergegeven in figuur 5.12. In figuur 5.13 zijn het aantal lichtberekeningen per frame als functie van de resolutie weergegeven voor Deferred Shading. Als laatste is de uitvoeringstijd voor het opbouwen van het lichtrooster als functie van de resolutie weergegeven in figuur 5.14.

Voor zowel de naïeve implementatie als de Tiled Shading implementaties is een kwadratisch verband waar te nemen in het aantal lichtberekeningen. Dit komt overeen met de kwadratische toename in pixels ten opzichte van de resolutie waarde. Ook bij een toename van de resolutie heeft de tegelgrootte geen significante invloed op het gemiddeld aantal lichtberekeningen per frame.

Echter, zoals in figuur 5.12 is te zien, presteert Tiled Shading met een tegelgrootte van 8×8 pixels slecht. Het aantal tegels neemt tevens kwadratisch toe bij een hogere resolutie. Het opbouwen van het rooster schaalt hierdoor dus ook kwadratisch. Zoals waar te nemen in figuur 5.14

5.4 Conclusie

In alle uitvoeringen verlaagt Tiled Shading het aantal lichtberekeningen ten opzichte van de naïeve implementatie. Voor tegelgroottes van 16×16 , 32×32 , en 64×64 pixels resulteert dit in een verlaging van de uitvoeringstijd per frame. Voor 8×8 pixels is de extra constructietijd van het lichtrooster groter dan de winst die behaald wordt door de reductie in aantal lichtberekeningen.

Er is geen significant verschil in aantal lichtberekeningen waar te nemen bij verschillende tegelgroottes. Dit kan een gevolg zijn van de grootte van de gebruikte lichten. Deze zijn zodanig groot dat het merendeel overlapt met meerdere tegels. Hierdoor zullen veelal vier 32×32 tegels dezelfde set lichten bevatten, als één 64×64 tegel. Er wordt dus relatief weinig winst behaald met kleinere tegels.

Indien de gebruikte scènes veel kleine lichten zouden bevatten, zou de hogere precisie waarmee de tegels voorgesteld worden het aantal lichtberekeningen per tegel wel verlagen, doordat er minder overlap tussen tegels zou zijn bij kleinere lichten. Voor grotere lichten leidt een dergelijk kleiner rooster slechts tot meer uitvoeringstijd om het rooster op te stellen. Dit resulteert in een hogere uitvoeringstijd per frame.

Bij het gebruik van een computationeel zwaardere shaders zal Tiled Shading relatief beter presteren. De opbouw van het lichtrooster is onafhankelijk van de

5. TILED SHADING

complexiteit van de shader. De constructietijd, bij een bruteforce aanpak is lineair afhankelijk van het aantal lichten in de scène en het aantal tegels in het lichtrooster. Wanneer een complexere shader wordt gebruikt, neemt relatief de tijdsvermindering door de reductie van het aantal lichten toe. Voor de gebruikte belichtingsberekening is nu al waar te nemen dat de uitvoeringstijd van de Tiled Shading implementaties voor het overgrote deel afhankelijk zijn van de constructietijd van het rooster. Deze tijd zal niet veranderen bij het gebruik van een andere shader.

Tiled Shading presteert slechter in situaties waar een groot aantal lichten overlapt in de camera z -as, zoals in de Piper's Alley scène. In dit geval bevatten de tegels die snijden met deze overlappende lichten significant meer lichten. Hierdoor zal voor deze tegels het naïeve tijds gedrag benaderd worden.

5.5 Discussie

5.5.1 Ondersteunen van transparantie

In sectie 4.5.2 is vermeld dat Deferred Shading een aparte Forward pijplijn vereist om transparante geometrie te renderen. Tiled Shading biedt geen directe ondersteuning voor transparantie. Echter doordat de lichtberekeningen van Forward Tiled Shading en Deferred Tiled Shading gelijkaardig zijn, kan de implementatie versimpeld worden. Zowel het lichtmanagement als het opgestelde rooster kan gedeeld worden tussen de pijplijnen. Dit versimpelt de implementatie en verbetert de efficiëntie[OA11].

5.5.2 Optimalisatie met behulp van de diepte

De geteste implementatie deelt het zichtfrustum op zoals voorgesteld in figuur 5.1b. Het is echter mogelijk dat een tegelvolume slechts fragmenten tussen de dieptes z_{min} en z_{max} bevat, waar z_{min} en z_{max} tussen de uiterste waarde van het zichtfrustum liggen. Deze waarden kunnen vervolgens gebruikt worden om het volume geassocieerd met een tegel verder te reduceren[OA11]. Hierdoor is het mogelijk om de set van relevante lichten te verkleinen, doordat lichten die buiten de dieptes z_{min} en z_{max} vallen, uitgesloten kunnen worden. Om een dergelijke optimalisatie te implementeren, dient de diepte van het frame voor de opbouw van het lichtrooster bekend te zijn.

Deze optimalisatie werkt goed indien een scène weinig diepte discontinuïteiten bevat. Wanneer een scène wel fragmenten met uiteenlopende dieptes bevat, is het nog steeds mogelijk dat het slechts mogelijke scenario zich voordoet.

5.5.3 Evaluatie van verschillende tegelgroottes.

Om een beter inzicht te verkrijgen in de invloed van de tegelgrootte op de uitvoeringstijd, zou de lichtgrootte als parameter geëvalueerd dienen te worden. Hierbij is de verwachting dat kleinere tegelgroottes efficiënter zijn bij kleinere lichtvolumes. Bij grotere lichtvolumes is deze efficiëntie winst minimaal, zoals gevonden is in de resultaten. In dit geval kan beter gekozen worden voor grotere tegelgroottes, om zo het geheugenverbruik en de lichtroosterconstructietijd te beperken.

```
// determine contributing lights
vec2 screen_position = gl_FragCoord.xy - vec2(0.5f, 0.5f);
uint tile_index =
    uint(floor(screen_position.x / tile_size.x) +
        floor(screen_position.y / tile_size.y) * n_tiles_x);

uint offset = tiles[tile_index].x;
uint n_lights = tiles[tile_index].y;

// compute the contribution of each light
for (uint i = offset; i < offset + n_lights; i++) {
    light_acc += computeLight(lights[light_indices[i]],
                               geometry_param);
}

// output result
fragment_colour = vec4((vec3(0.1f) + (light_acc * 0.9)), 1.0f);
```

Listing 5: De implementatie van de lichtberekening in de GLSL shader van Tiled Shading.

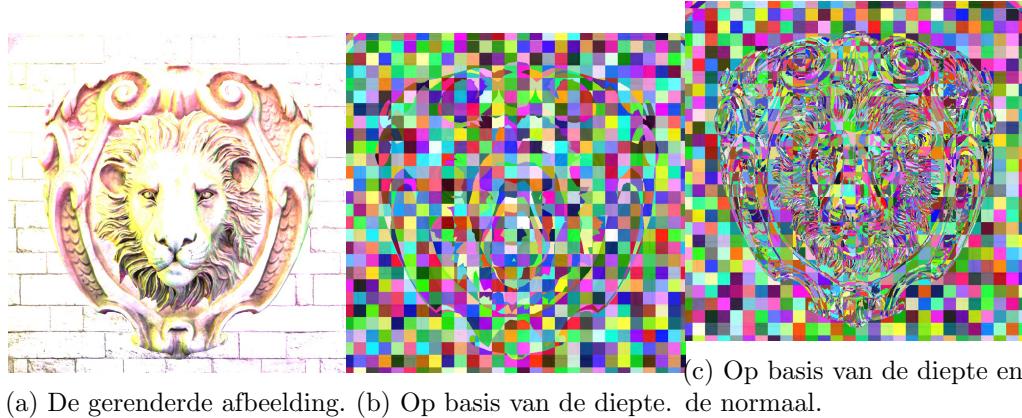
Hoofdstuk 6

Clustered Shading

Tiled Shading, zoals geïntroduceerd in het vorige hoofdstuk, is een eerste stap om efficiëntere lichttoekenning mogelijk te maken. Het doel hierbij is om de efficiëntie van Deferred Shading met stencil-optimalisatie te benaderen zonder dat dezelfde geheugenbandbreedte wordt gebruikt. Binnen Tiled Shading wordt dit bereikt door een set van tegels te creëren die het zichtveld bedekken. Voor elk van de tegels wordt bepaald, welke lichten invloed kunnen hebben op de fragmenten binnen de tegel. Bij een efficiënte implementatie worden hierbij slechts de lichten bijgehouden die overlappen met het volume waarin de fragmenten van het tegel liggen. Dit volume wordt dus beperkt door de fragmenten die het dichtst en het verst van de camera liggen. Wanneer alle fragmenten in een tegel dicht bij elkaar liggen levert dit een efficiënte voorstelling op. Echter wanneer er zich binnen een tegel grote diepte discontinuïteiten bevinden, leidt dit tot grote volumes bestaande uit grote geometrisch lege ruimtes. Hierin liggen mogelijk veel lichten die geen invloed hebben op de fragmenten in de tegel, wat leidt tot overbodige berekeningen.

Clustered Shading[OBA12] is geïntroduceerd om dit slechtste scenario tegen te gaan, en tevens een efficiëntere en consistentere lichttoekenning mogelijk te maken. Hiervoor wordt het concept van tegels uitgebreid naar hogere dimensies. Deze hogere dimensie tegels worden clusters genoemd. De meest voor de handliggende volgende dimensie is de diepte met respect tot de **z**-as van de camera. Door de diepte explicet op te delen, heeft elke cluster een maximaal volume, onafhankelijk van het zicht. Dit verhelpt het slechtste scenario van Tiled Shading, waarbij volumes geassocieerd met de tegels groter worden indien fragmenten verder uit elkaar liggen. Tegelijkertijd verkleind het de volumes van de clusters, waardoor de lichttoekenning specifieker, en dus efficiënter wordt. Naast diepte, kunnen ook andere attributen zoals de normaalinformatie van fragmenten meegenomen worden om clusters op te stellen.

In dit hoofdstuk wordt het Clustered Shading algoritme behandeld. Hiervoor zal eerst ingegaan worden op de theorie en het algoritme. Vervolgens zal de efficiëntie geëvalueerd worden aan de hand van de drie testscènes.



Figuur 6.1: Clustering op basis van diepte, en diepte en normaal[OBA12].

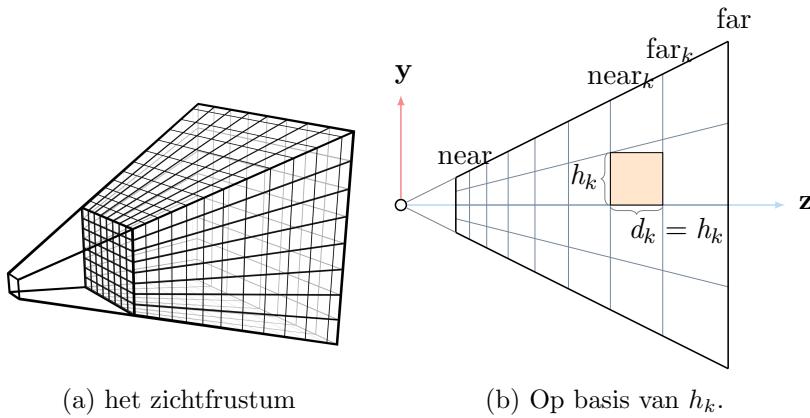
6.1 Theorie

Het doel van Clustered Shading is om de lichttoekenning zoals in Tiled Shading te verbeteren door de twee-dimensionale tegels, verder uit te breiden, zodat hogere dimensie clusters ontstaan[OBA12]. Hiertoe worden de tegelvolumes onder verdeeld in de diepte. Verdere dimensies kunnen toegevoegd worden op basis van andere attributen, zoals de normaalinformatie van fragmenten. Een voorbeeld van een dergelijke onderverdeling is te zien in figuur 6.1. Door de tegels verder op te delen worden de volumes geassocieerd met deze tegels kleiner, en dus homogener, waardoor de efficiëntie van de lichtberekening toeneemt[OBA12].

6.1.1 Onderverdeling in de ruimte

De eerste stap in het uitbreiden van tegels naar hogere dimensies is het vastleggen hoe de diepte onderverdeeld dient te worden. Voor een efficiënte lichttoekenning is het van belang dat de clusters zo klein mogelijk zijn, zodat deze zo nauwkeurig mogelijk de relevante set van lichten kunnen beschrijven. Tegelijkertijd moeten clusters zoveel mogelijk (homogene) fragmenten bevatten, zodat het geheugenverbruik laag blijft, terwijl de clusters efficiënt opgehaald kunnen worden. Tevens is het belangrijk dat de sleutel waarmee een cluster geïdentificeerd kan worden, efficiënt berekend kan worden en weinig bits vereist[OBA12].

Een belangrijk inzicht is dat fragmenten van een frame altijd in het zichtfrustum geassocieerd met het frame zullen vallen. Wanneer de lichttoekenningsdatastructuren dus per frame opgebouwd worden, zoals het geval is bij Tiled en Clustered Shading, is het niet nodig om de gehele wereld op te delen in volumes, alleen het zichtfrustum dient opgedeeld te worden. Tiled Shading deelt het zichtvenster op in de **x**- en **y**-as. Dit is impliciet een opdeling van het zichtfrustum. Binnen Clustered Shading wordt de **z**-as van de camera verder opgedeeld, om zo subfrustra te verkrijgen, zoals weergegeven in figuur 6.2a. Om deze subfrustra zo uniform mogelijk te houden, wordt de diepte van één subfrustum, d_k , gelijk gesteld aan de hoogte van het subfrustum,



Figuur 6.2: Opdeling van het zichtfrustum

h_k bij bij de diepte near_k , waar het subfrustumvolume begint[OBA12]. Dit is weergegeven in figuur 6.2b. De volledige hoogte van het zichtfrustum bij diepte near_k is

$$h_{\text{frustum}} = 2\text{near}_k \tan \theta$$

waar θ de helft van het gezichtsveld is. De hoogte h_k van één subfrustum is dan gelijk aan de hoogte van het zichtfrustum gedeeld door het aantal onderverdelingen in de y -as. De afstand van het volgende subfrustum is dan gegeven als

$$\begin{aligned} \text{near}_{k+1} &= \text{near}_k + h_k \\ &= \text{near}_k + \frac{2\text{near}_k \tan \theta}{S_y} \end{aligned}$$

Dit kan herschreven worden tot de volgende exponentiële functie:

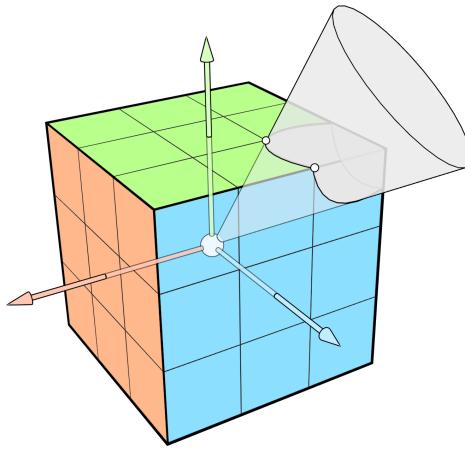
$$\text{near}_k = \text{near}_0 \left(1 + \frac{2 \tan \theta}{S_y} \right)^k$$

waarmee de opdeling van de z -as van de camera gespecificeerd wordt. Op basis hiervan is het mogelijk om elk cluster te identificeren aan de hand van een tupel van drie integers (i, j, k) , waarbij i en j respectievelijk de x en y posities van de tegel in het zichtvenster specificeert, en k de diepte waarop het cluster zich bevindt. Dit tupel vormt de sleutel van de clusters.

6.1.2 Onderverdeling op basis van de normalen

De clusters kunnen verder opgedeeld worden aan de hand van attributen waarmee fragmenten in discrete groepen onderverdeeld kunnen worden. Nuttige attributen zorgen hierbij dat de set van relevante lichten geassocieerd met elk cluster afneemt. Een voorbeeld hiervan is de normaal-informatie[OBA12]. Wanneer de fragmenten in

6. CLUSTERED SHADING



Figuur 6.3: Normaalkegel op basis van een eenheidskubus.

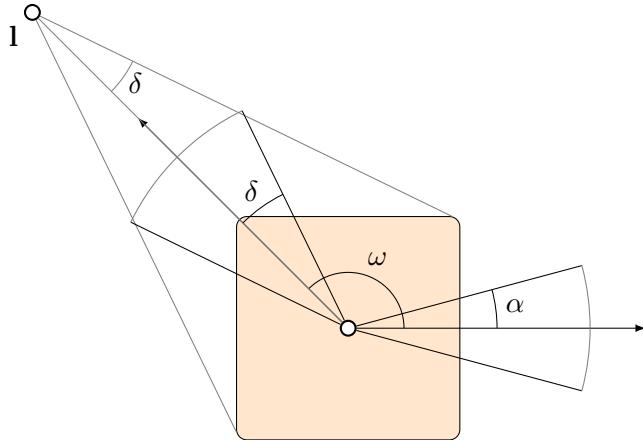
een cluster verder worden opgedeeld aan de hand van de normaal informatie, is het mogelijk om lichten uit te sluiten die een nulcontributie hebben op de fragmenten van het cluster. In het geval van Lambertiaanse materialen komt dit voor wanneer de hoek tussen het invallend licht en de normaal van een fragment groter is dan 90° . Een veel voorkomende situatie waarin dit zich voordoet, is wanneer een licht de achterkant van een primitief verlicht. Het voorkomen van een dergelijke situatie wordt achtervlak lichtruiming (Backface Light Culling) genoemd.

De normalen worden gegroepeerd aan de hand van een set van normaal kegels[OBA12]. Voor elk cluster wordt één kegel gedefinieerd. Voor de fragmenten geassocieerd met een cluster geldt dat de normaal van deze fragmenten binnen de gedefinieerde kegel van de cluster valt. Om de kegels op te stellen wordt gebruik gemaakt van een onderverdeelde kubus. Voor elk vlak van deze kubus wordt één normaalkegel opgesteld die snijdt met de vier vertexen van het vlak. Dit is weergegeven in figuur 6.3. Voor elk fragment wordt bepaald met welk vlak de normaal van het fragment snijdt, indien deze onderverdeelde kubus gecentreerd is op het fragment.

Vervolgens kunnen lichten die aan geen enkel fragment van het cluster bijdragen buiten beschouwing van dit cluster worden gelaten. Een licht heeft een nulcontributie aan een cluster als:

$$\omega > \frac{\pi}{2} + \alpha + \delta$$

waarbij ω de hoek tussen de lichtdirectie en de as van de normaalkegel is, α de halfhoek van de normaalkegel is, en δ de hoek die het licht maakt om de gehele cluster te verlichten. Dit is geïllustreerd in figuur 6.4.



Figuur 6.4: Lichtruiming op basis van de normaalkegel.

6.1.3 Bepaling van unieke clusters

Wanneer deze berekening voor alle fragmenten in de framebuffer wordt uitgevoerd, zal de resulterende textuur alle clusters relevant voor de huidige frame bevatten. Deze dienen echter nog wel gegroepeerd te worden zodat een lijst van unieke clusters verkregen wordt. Het groeperen van monsters is een veel voorkomend probleem binnen GPU rendering[OBA12].

Om de globaal unieke monsters in een textuur, of buffer, te verkrijgen, dient deze eerst gesorteerd te worden. Vervolgens moet een compact-stap uitgevoerd worden. Beide stappen zijn veel gebruikte bouwstenen in parallelle algoritmes[BOA09, SHG09]. Om deze reden bestaan er verschillende efficiënte GPU implementaties[Sch11].

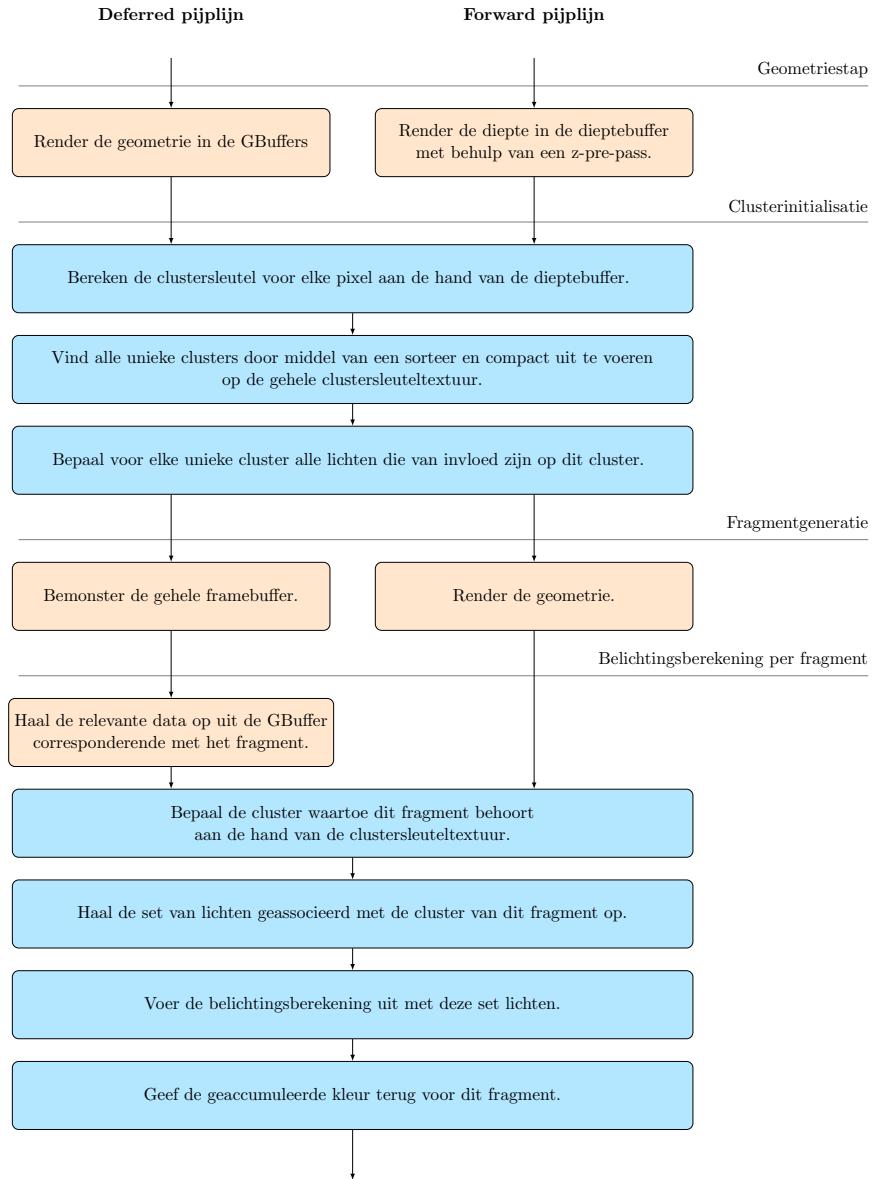
Ondanks dit blijft sorteren een computationeel dure operatie, waardoor het veelal nodig is om coherentie aanwezig in de textuur of buffer te gebruiken om de hoeveelheid data te reduceren.

Voorbeelden hiervan zijn Resolution Matched Shadow Maps[LSO07], en het Compress-Sort-Decompress algoritme[GL10]. Binnen Resolution Mapped Shadow Maps moet bepaald worden welke schaduwpagina's gebruikt worden door zichtmonsters. Om de set van data te verkleinen wordt de coherentie in het zichtveld gebruikt. Indien naburige pixels gelijke schaduwpagina's vereisen, worden deze samengenomen. In het geval van het Compress-Sort-Decompress algoritme wordt de framebuffer voorgesteld als een 1-dimensionale sequentie en wordt hierop runtime-encoding toegepast om duplicaten te voorkomen, voordat sortering plaatsvindt.

Een alternatieve oplossing is om gebruik te maken van virtuele paginatabellen. Binnen deze paginatabellen wordt expliciet per pagina bijgehouden of deze gebruikt worden. De verschillende zichtmonster zetten een bit in het pagina-adres, vervolgens kan deze bit gebruikt worden in de compactiestap. Omdat monsters die vervijzen naar hetzelfde adres allen de bit op aanzetten, zullen er geen duplicaten in de paginatable zijn[HPLVdW10].

Wanneer de unieke clusters opgesteld zijn, kan voor elk de set van relevante lichten worden bepaald.

6. CLUSTERED SHADING



Figuur 6.5: Het Clustered Shading algoritme voor de Forward en Deferred pijplijn.

Clustered Shading werd in verschillende recente games gebruikt, waaronder in de Avalanche engine, die gebruikt is voor Just Cause 2 en 3[Per13] en Doom 4[Tat16].

6.2 Algoritme

Het Clustered Shading-algoritme[OBA12] is weergegeven in figuur 6.5. Het vertoont grote overeenkomst met het Tiled Shading algoritme. Vergelijkbaar met Tiled Shading wordt per frame een set van lichttoekenningsdatastructuren opgesteld die

vervolgens tijdens de lichtberekening worden gebruikt om het aantal te evalueren lichten te reduceren.

In tegenstelling tot Tiled Shading kunnen deze clusters niet berekend worden voor de geometriestap. Dit komt, doordat de clusters afhankelijk zijn van de diepte van fragmenten, en dus dient een dieptebuffer opgebouwd te zijn, voordat de clusters bepaald kunnen worden[OBA12]. Binnen Deferred Shading wordt de dieptebuffer opgebouwd in de geometriestap. Binnen Forward Shading zonder extra uitbreidingen wordt de dieptebuffer slechts opgebouwd tijdens de renderstap. Voor Tiled Shading en de naïeve implementatie is dit geen probleem. Echter voor Clustered Shading leidt het gebrek aan een dieptebuffer ertoe dat de clusters niet opgesteld kunnen worden voor de renderstap.

Om de dieptebuffer toch beschikbaar te maken voordat de renderstap wordt uitgevoerd, dient een extra stap toegevoegd te worden, waarin de dieptebuffer al wordt opgesteld. Deze stap wordt een z-prepass[CTM13] genoemd. Tijdens de z-prepass wordt alle geometrie gerasteriseerd en gerenderd met een dummy fragmentshader. De enige taak van deze dummy shader is het wegschrijven van de diepte naar de diepte buffer. Vervolgens kan de diepte gebruikt worden om de clusters te berekenen, als ook om niet zichtbare fragmenten weg te gooien voordat de lichtberekening wordt uitgevoerd. Hier staat echter wel tegenover dat alle geometrie twee maal per frame, gerasterised dient te worden. Afhankelijk van de complexiteit van de geometrie en eventuele tesselatie kan dit performantie verslechtern.

Nadat de dieptebuffer is berekend kunnen de clusters worden opgesteld. Hier voor worden eerst de unieke clusters en hun corresponderende sleutels, berekend. Vervolgens wordt aan elk van deze clusters de lichten toegevoegd waarmee deze overlappen.

Als laatste wordt de lichtberekeningsstap uitgevoerd. Hiervoor wordt voor elk fragment aan de hand van het cluster waartoe deze behoort, de set van relevante lichten opgehaald. Vervolgens wordt de lichtberekening met elk van deze lichten uitgevoerd.

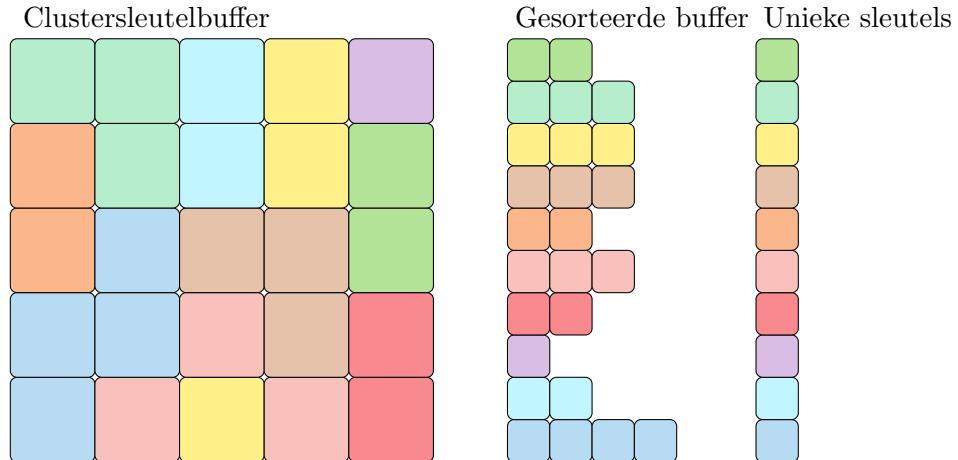
6.2.1 Clustersleutels

De eerste stap in het opstellen van de clusters is de bepaling van de relevante clusters. Omdat de clusters het zichtfrustum onderverdeelen in een discrete set van subfrusta, kan elk cluster worden aangeduid met een tupel van drie of meer integers (i, j, k, \dots) [OBA12], zoals weergegeven in figuur 6.2b. Hierbij zijn i en j gelijk aan de coördinaten voor tegels binnen Tiled Shading. De integer k specificeert de diepte-index. Een dergelijk tupel wordt een clustersleutel genoemd. In sectie 6.1.1 is de diepte van vlak near_k gedefinieerd als:

$$\text{near}_k = \text{near}_0 \left(1 + \frac{2 \tan \theta}{S_y} \right)^k$$

Hieruit kan de waarde voor k worden afgeleid als zijnde:

6. CLUSTERED SHADING



Figuur 6.6: De sorteerte- en comprimeerstap uitgevoerd over een enkel vlak.

$$k = \left\lfloor \frac{\log \left(\frac{-z}{\text{near}} \right)}{\log \left(1 + \frac{2 \tan \theta}{S_y} \right)} \right\rfloor$$

Indien deze berekening wordt uitgevoerd voor elk fragment in de dieptebuffer, wordt een k -buffer verkregen, die de **z**-coördinaat van elk subfrustum beschrijft. De k -buffer legt de clustersleutel geassocieerd met elk fragment vast door de k waarde en de positie binnen de buffer.

6.2.2 Unieke clustersleutels

De unieke clusters kunnen afgeleid worden uit de opgestelde k -buffer. Een belangrijke observatie hier, is dat fragmenten slechts tot dezelfde cluster kunnen behoren, als zij in dezelfde tegel liggen. Hierdoor beperkt het probleem om alle unieke clustersleutels te vinden zich tot het vinden van alle unieke k -waardes per tegel.

De unieke k -waardes per tegel kunnen bepaald worden door elke tegel eerst te sorteren, en vervolgens een compact-operatie uit te voeren. De verschillende stappen zijn weergegeven in figuur 6.6. Hierbij komt elke verschillend gekleurde tegel overeen met één k -waarde. Na het sorteren en de compact operatie, blijft per tegel een lijst van unieke clusters over, waar de set van relevante lichten voor bepaald kan worden.

Om vervolgens per fragment efficiënt het cluster waartoe het behoort op te zoeken, wordt tijdens deze stap ook per fragment de index overeenkomend met de cluster geassocieerd. Dit wordt gedaan door bij de sorteertap een referentie naar het fragment toe te voegen aan de k -waarde. Vervolgens wordt tijdens de compact tap deze referentie gebruikt om de index geassocieerd met de cluster toe te voegen aan een textuur op de positie van het fragment[OBA12].

Deze stappen zijn binnens **nTiled** geïmplementeerd in de vorm van een **OpenGL** compute-shader. Hierbij wordt per tegel een werkgroep gestart. De compute-shader voert de sorteerte- en compacttap uit. Hierbij wordt de sorteertap uitgevoerd

met behulp van een bottom-up merge-sort-algoritme[SW11]. Elke k -waarde wordt voorgesteld als een 16-bit integer. Bij de eerste stap van het merge-sort-algoritme wordt de k -waarde verschoven over 16-bits. Vervolgens wordt de gelineariseerde index van het fragment hierbij opgeteld.

$$k' = (k \ll 16) + p_x + p_y \times n_x$$

waar p_x en p_y respectievelijk de **x**-as en de **y**-as posities van het fragment binnen de tegel zijn en n_x het totaal aantal pixels in de **x**-as van de tegel.

Nadat de k' waardes zijn gesorteerd, wordt een compact-operatie uitgevoerd. Hierbij worden alle waardes samengenomen waar de 16 meest significante bits gelijk zijn. De 16 minst significante bits worden vervolgens gebruikt om de unieke clusterindex geassocieerd met cluster k weg te schrijven naar een clusterindextextuur op de positie geassocieerd met het fragment van k' .

Nadat deze stappen zijn uitgevoerd is er per tegel een lijst van unieke k -waardes en een totaal aantal unieke k -waardes beschikbaar. Tevens is de clusterindex voor alle fragmenten opgeslagen in de clusterindextextuur.

6.2.3 Lichttoekenning

Om de grote aantallen lichten efficiënt aan de unieke clusters toe te kennen, wordt een Bounding Volume Hierarchy (BVH) gebruikt[OBA12]. Deze BVH wordt per frame opgesteld. Hiervoor worden de lichten geordend met respect tot de **z**-as. Vervolgens worden de lichten gegroepeerd per 32, en wordt voor elke groep een Axis-Aligned Bounding Box (AABB) opgesteld. Deze groepen worden opnieuw gegroepeerd per 32, totdat er slechts 1 wortelelement over is.

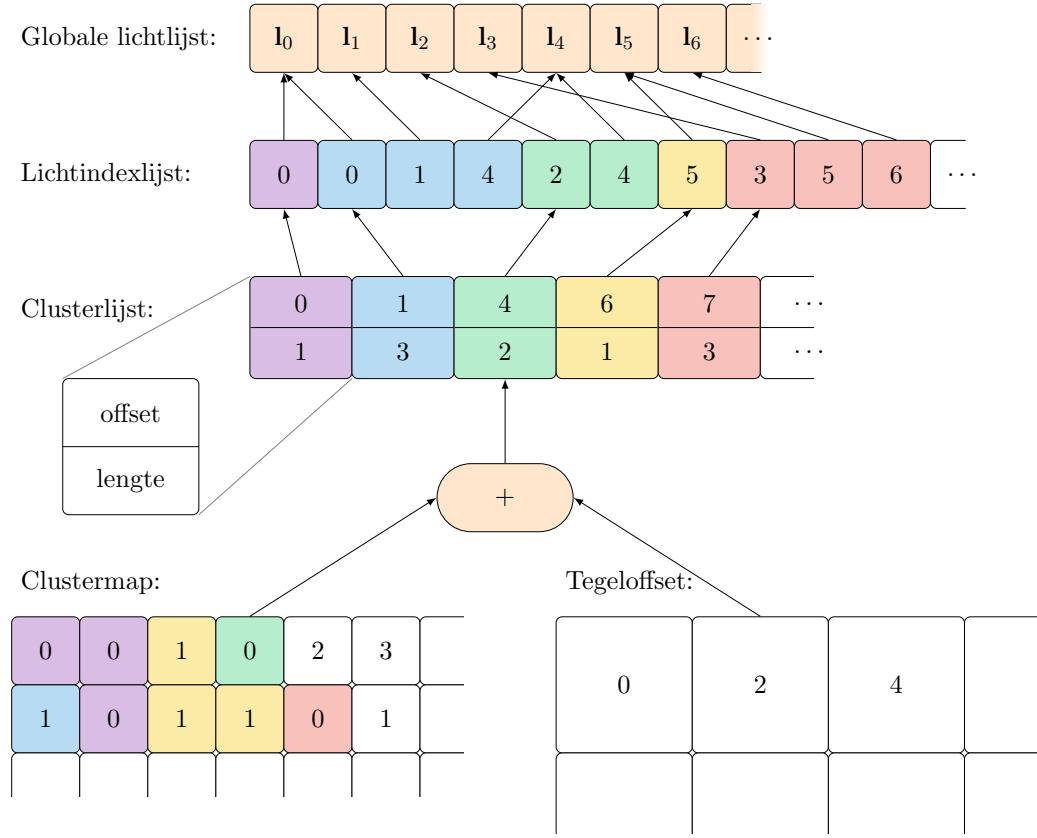
De BVH wordt vervolgens per cluster doorlopen. Hierbij wordt per niveau voor elk van de elementen getest of de AABB van het element overlapt met het volume van het cluster. Alleen van de elementen die overlappen met het cluster worden de kinderen geëvalueerd. De uiteindelijk overlappende lichten worden toegevoegd aan het cluster.

Binnen **nTiled** is deze optimalisatie niet geïmplementeerd. De clusters worden in plaats hiervan opgesteld door middel van een bruteforce methode gelijkend op de lichttoekenningsmethode van Tiled Shading. Eerst worden de lichten afgebeeld op het zichtvenster en worden de tegels waarop elk licht invloed heeft bepaald, zoals in Tiled Shading. Vervolgens wordt voor deze tegels nagegaan of het lichtvolume overlapt met de unieke clusters binnen de tegel met respect tot de **z**-as. Indien dit het geval is wordt een referentie naar het licht toegevoegd aan het unieke cluster.

6.2.4 Datastructuren

De datastructuren in Clustered Shading gelijken in grote mate op die van Tiled Shading, 5.2.1. Echter binnen Clustered Shading kan de clusterindex niet meer direct berekend worden uit de positie van het fragment. Deze associatie dient dus expliciet bijgehouden te worden. Om deze reden is de clustermaptextuur geïntroduceerd[OBA12]. Hierin wordt per fragment de index van het overeenkomstige

6. CLUSTERED SHADING



Figuur 6.7: De datastructuren gebruikt binnen Clustered Shading.

cluster bijgehouden. De clusterlijst vervangt vervolgens de functie van het lichtrooster binnen Tiled Shading.

In nTiled wordt binnen de sorteer- en compactstap slechts de lokale clusterindex binnen de tegel opgeslagen. Om de verkregen clusterindexmap om te zetten naar een textuur die de globale indices bevat, dient de offset van eerdere tegels bij elke waarde opgeteld te worden. Om niet elke waarde in de clustermap textuur explicet bij te werken na de sorteer- en compactstap, is in plaats hiervan gekozen om een tweede textuur bij te houden. Deze bevat voor elke tegel één waarde, die de offset voor alle fragmenten binnen de tegel geeft. Vervolgens kan de cluster die geassocieerd is met een fragment opgehaald worden door de lokale offset op te tellen bij de tegeloffsetwaarde. Dit leidt tot de volgende datastructuren[OBA12], waarvan de relaties zijn weergegeven in Figuur 6.7.

Globale lichtlijst: bevat alle lichten. Deze array is in dezelfde vorm aanwezig in de naïeve shaders.

```

// determine contributing lights
vec2 screen_position = gl_FragCoord.xy - vec2(0.5f, 0.5f);
uint tile_index =
    uint(floor(screen_position.x / tile_size.x) +
        floor(screen_position.y / tile_size.y) * n_tiles_x);
uint tile_offset = summed_cluster_indices[tile_index];
uint k_offset = texture(k_index_tex, tex_coords).x;

uvec2 cluster_map = clusters[tile_offset + k_offset];

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

// compute the contribution of each light
for (uint i = offset; i < offset + n_lights; i++) {
    light_acc += computeLight(lights[light_indices[i]], param);
}

// output result
fragment_colour = vec3((vec3(0.1f) + (light_acc * 0.9)));

```

Listing 6: De implementatie van de lichtberekening in de GLSL shader van Clustered Shading.

Lichtindexlijst: bevat lichtindices die verwijzen naar lichten in de globale lichtlijst.

Deze array is dus een lijst van alle referenties van alle clusters.

Clusterlijst: bevat voor elke cluster een vector die de offset en aantal lichten binnen de lichtindexlijst specificeert. Deze lijst neemt de rol van het lichtrooster binnen Tiled Shading over.

Tegeloffsettextuur: Specificeert de globale offset binnen de clusterlijst voor alle fragmenten voor elke tegel.

Clustermaptextuur: Specificeert de lokale offset binnen alle clusters van een tegel voor alle fragmenten.

6.2.5 Lichtbepaling

Op basis van de voorstelling van de clusters op de GPU kan de lichtberekening per fragment worden opgesteld. Hiervoor dient eerst de set van lichten geassocieerd met het fragment opgehaald te worden. Hiervoor wordt het cluster behorende tot het fragment bepaald. De lokale clusterindexoffset wordt opgehaald uit de clustermap, en de globale clusterindexoffset uit de tegeloffset. Vervolgens wordt met de verkregen clusterindex de offset en aantal lichten in de lichtindexlijst opgehaald uit de clusterlijst.

6. CLUSTERED SHADING

De lichtberekening wordt dan uitgevoerd met de gespecificeerde set lichten. De code hiervoor is weergegeven in lst. 6

6.3 Resultaten

De performantie van de Clustered Shading implementatie binnen `nTiled` is geëvalueerd met behulp van de drie testscènes. Hierbij is slechts gekeken naar de Deferred Shading implementatie. De Forward Clustered Shading implementatie vereist een z-prepass, zoals besproken in de vorige secties. Een dergelijke aanpassing aan de Forward Shader heeft invloed op de algemene performantie. Hierdoor wordt een eerlijke vergelijking tussen de verschillende technieken bemoeilijkt. Deferred Clustered Shading vereist geen aanpassing aan de structuur. Om deze reden kan deze variant wel accuraat vergeleken worden.

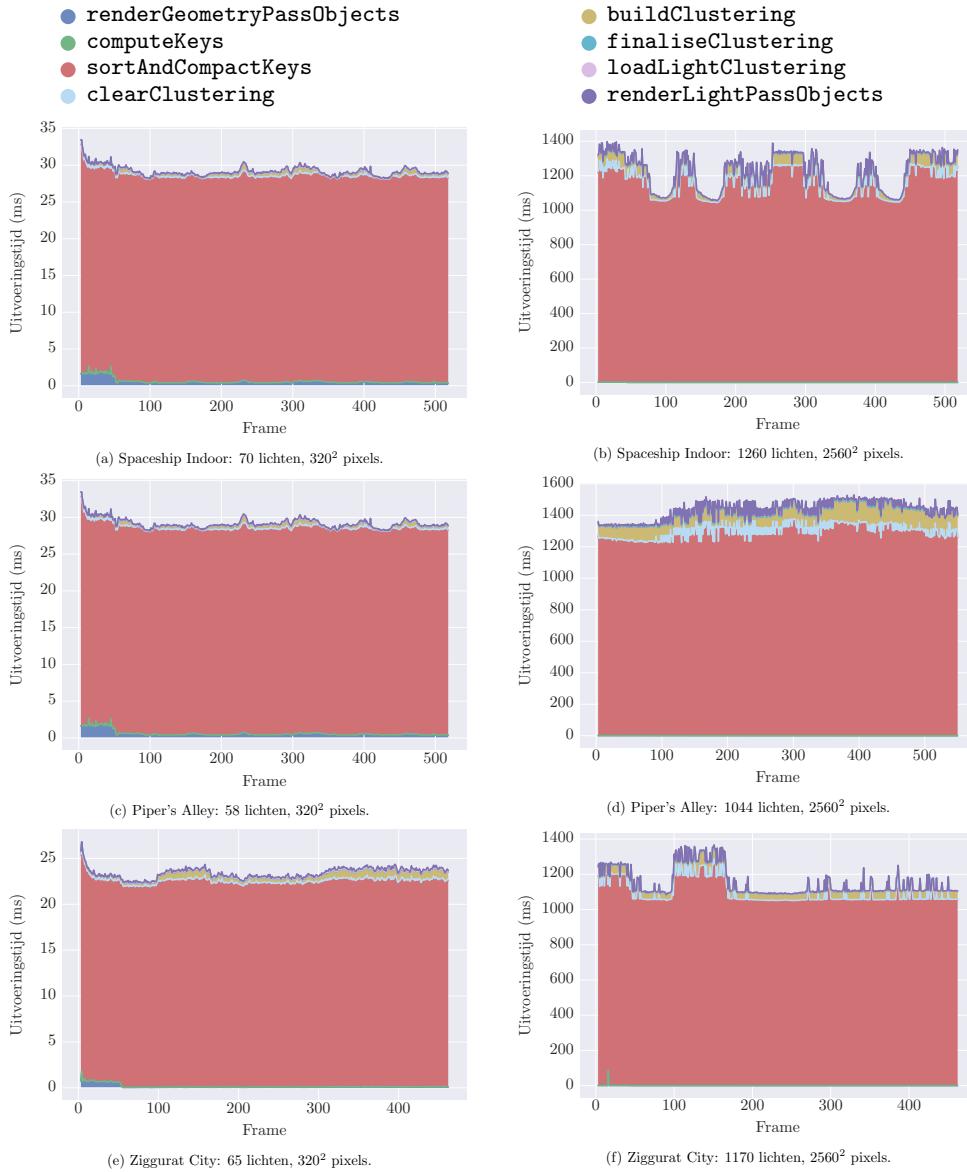
Voor de evaluatie van Clustered Shading zal eerst naar de individuele uitvoeringen gekeken worden, en vervolgens naar de invloed van lichten en resolutie op de Clustered Shading implementatie. Hiervoor zal Deferred Clustered Shading vergeleken worden met naïeve Deferred Shading en de Deferred Tiled Shading implementatie gebruikmakend van 32×32 pixels tegels. Er is gekozen voor de 32×32 pixels variant omdat het verschil in aantal lichtberekeningen tussen de verschillende tegelgroottes minimaal is en deze waarde het best presteerde.

6.3.1 Frames

In figuur 6.8 is de gesommeerde uitvoeringsstijd van de verschillende functies van Clustered Shading weergegeven. Hierin valt gelijk op dat een meerderheid van de uitvoeringsstijd gespendeerd wordt aan de functie `sortAndCompactKeys`. Dit is het gevolg van het gebruik van de `OpenGL` functie `glGetTexImage`¹. Deze functie wordt gebruikt om de berekende clusters uit het videogeheugen op te halen. Dit is een trage operatie, waarvan de tijd direct afhankelijk is van de grootte van de texturen die opgehaald dienen te worden. In het geval van deze Clustered Shading implementatie wordt een textuur ter grootte van het zichtveld en een textuur ter grootte van het aantal tegels opgehaald. Dit leidt tot een erg trage implementatie. Hierdoor is de tijdscomplexiteit van de Clustered Shading implementatie hoofdzakelijk afhankelijk van de trage implementatie van `sortAndCompactKeys`. Dit maakt de uitvoeringsstijd een weinigzeggende indicator voor de performantie van het algoritme. Om deze reden zal verder niet de uitvoeringsstijd vergeleken worden met de andere lichttoekenningsalgoritmen. Omdat de sorteert en compact stap een belangrijke en computationele zwaardere stap is binnen het Clustered Shading algoritme, zou een vergelijking waar `sortAndCompactKeys` buiten beschouwing gelaten worden ook tot een vertekend beeld leiden. De verdere vergelijkingen kijken slechts naar het aantal lichtberekeningen.

¹Documentatie van `glGetTexImage`: '<https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/glGetTexImage.xhtml>

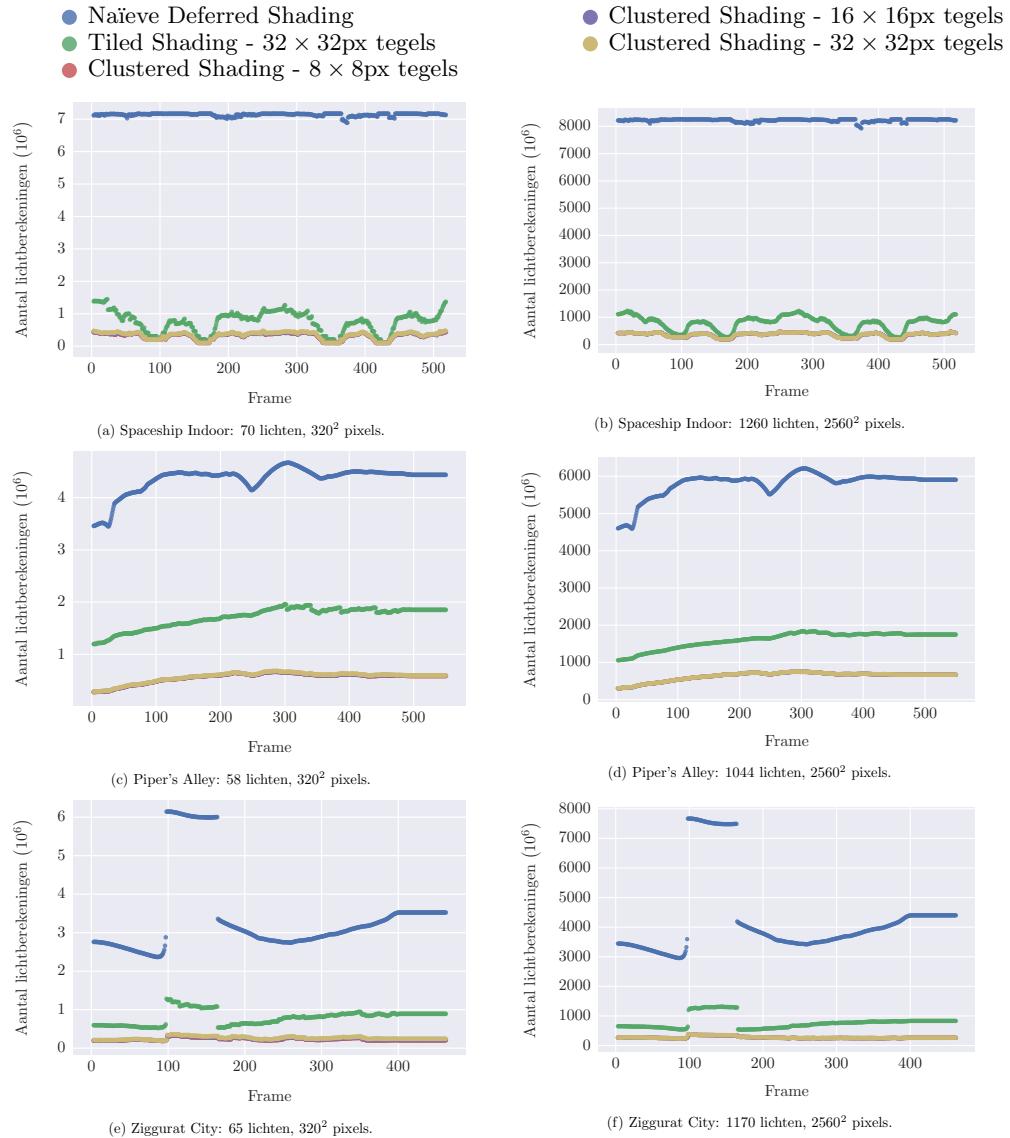
6.3. Resultaten



Figuur 6.8: Overzicht van de uitvoeringstijd voor Deferred Clustered Shading per frame voor de drie testscènes bij verschillende resoluties en aantal lichten.

Het aantal lichtberekeningen per frame gedurende een complete uitvoering van Clustered Shading zijn weergegeven in figuur 6.9. In deze grafieken is tevens het aantal lichtberekeningen voor de naïeve implementatie en voor Tiled Shading gegeven. Alle varianten van Clustered Shading vereisen minder lichtberekeningen dan zowel de naïeve implementatie als de Tiled Shading implementatie. Opnieuw lijkt de tegelgrootte geen grote invloed te hebben op het aantal lichtberekeningen. Dit is vergelijkbaar met de resultaten gevonden bij Tiled Shading. Ook hier zal gelden dat, bij de gekozen grootte van lichten, nabijgelegen clusters veelal dezelfde set van

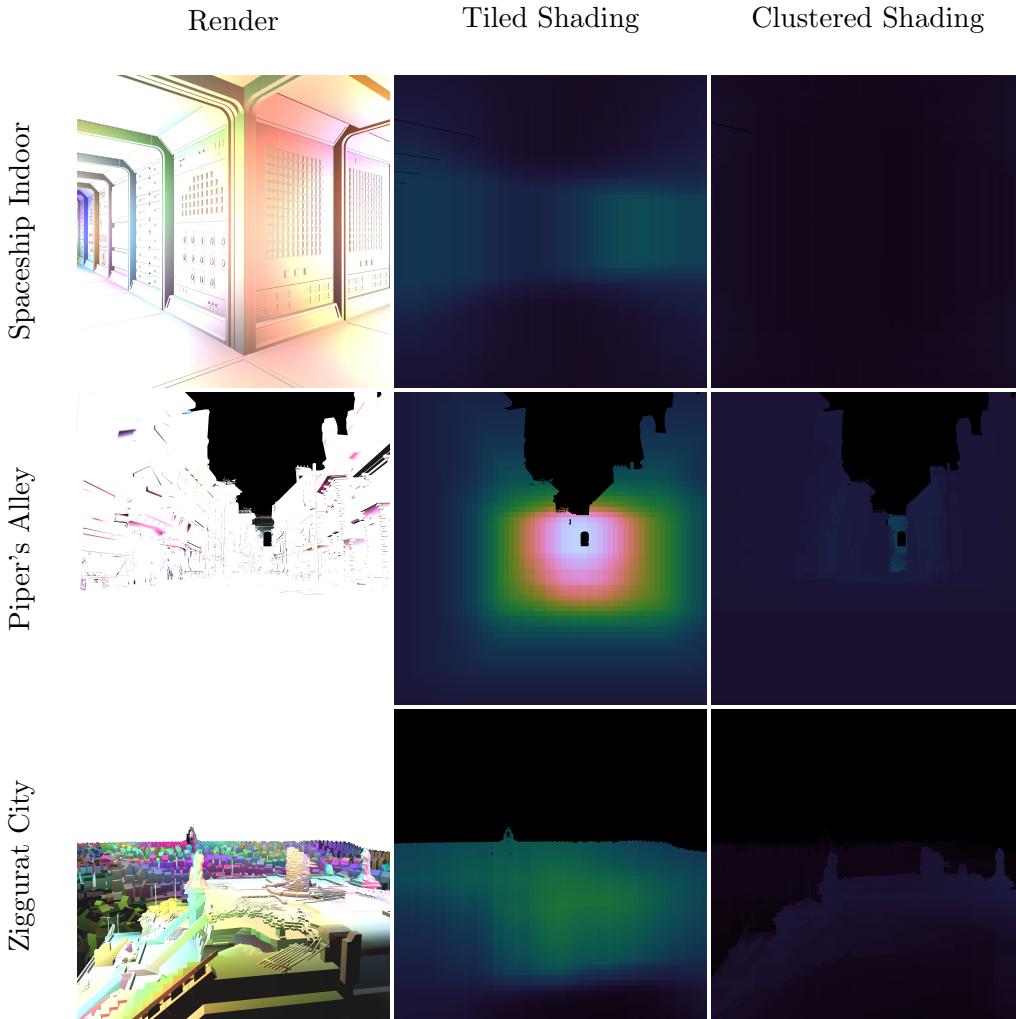
6. CLUSTERED SHADING



Figuur 6.9: Overzicht van het aantal lichtberekeningen per frame voor Deferred Shading voor de drie testscènes bij verschillende resoluties en aantal lichten.

lichten zullen bevatten. Hierdoor is er geen significant verschil tussen de clusters met een tegelgrootte van 32×32 en de clusters met tegelgrootte 8×8 . Indien de scènes meer kleinere lichten zou bevatten, zouden, net als bij Tiled Shading, de kleinere tegelgroottes waarschijnlijk wel beter presteren.

Het aantal lichtberekeningen per frame is consistentier dan bij Tiled Shading. Doordat het aantal lichten per cluster in grote mate is geminimaliseerd, is het verschil tussen pixels zonder fragmenten en pixels met fragmenten kleiner. Hierdoor is het verschil in aantal lichtberekeningen tussen de verschillende camerapunten in de Ziggurat City scène verwaarloosbaar. Dit is goed waar te nemen in figuur 6.10, waar



Figuur 6.10: Renders en hittekaarten van de verschillende scènes voor Clustered Shading.

te zien is dat het verschil in aantal lichtberekeningen tussen pixels zonder en met fragmenten klein is.

Ook in de Spaceship Indoor scène is een consistent aantal lichtberekeningen waar te nemen. Binnen de Tiled Shading implementatie is deze variatie een gevolg van de overlap van lichten. Wanneer de camera zich aan het begin van een corridor bevindt zal elk van de lichten in de corridor overlappen. Naarmate de camera zich door de corridor heen beweegt zullen de lampen achter de camera buitenbeschouwing worden gelaten, totdat uiteindelijk slechts een klein aantal lichten nog invloed heeft op de geometrie. Binnen Clustered Shading zorgt de opdeling in de camera **z**-as ervoor dat elk cluster slechts lichten bevat die ook in de **z**-as relevant zijn. Dit leidt ertoe dat voor alle fragmenten slechts een kleinere set lichten geëvalueerd dient te worden, die vergelijkbaar is met de set lichten in het minimale geval van Tiled

6. CLUSTERED SHADING

Shading. Dit heeft een lagere en consistentere hoeveelheid lichtberekeningen per frame tot gevolg.

Het grootste verschil in aantal lichtberekeningen is waar te nemen in de Piper's Alley scène. Dit is een direct gevolg van het oplossen van het slechtst mogelijke scenario van Tiled Shading. Binnen de Piper's Alley scène was de overlap van lichten het extreemst. Nu de tegelvolumes opgesplitst zijn in de camera **z**-as, worden niet alle overlappende lichten toegekend aan elk cluster. Dit is duidelijk zichtbaar in figuur 6.10. De Tiled Shading implementatie vertoont een grote hoeveelheid lichtberekeningen voor de tegels in het midden van de afbeelding. De Clustered Shading implementatie bevat geen clusters met een dergelijk grote hoeveelheid lichten. Relatief aan alle clusters bevatten slechts de verste clusters een groter aantal lichten. Dit is een gevolg van de opdeling van Clustered Shading, waar met clusters verder van de camera, grotere volumes geassocieerd zijn.

6.3.2 Lichten

In figuur 6.11 is het gemiddeld aantal lichtberekeningen per frame als functie van het aantal lichten bij een resolutie van 2560×2560 pixels, weergegeven. Clustered Shading is lineair afhankelijk van het aantal lichten, zoals ook de naïeve en de Tiled implementaties. Echter de factor waarmee het aantal lichtberekeningen schaalt is bijna een factor twee kleiner voor alle scenes. De clustergrootte lijkt geen significante invloed te hebben op het aantal lichtberekeningen, wanneer de hoeveelheid lichten wordt gevareert.

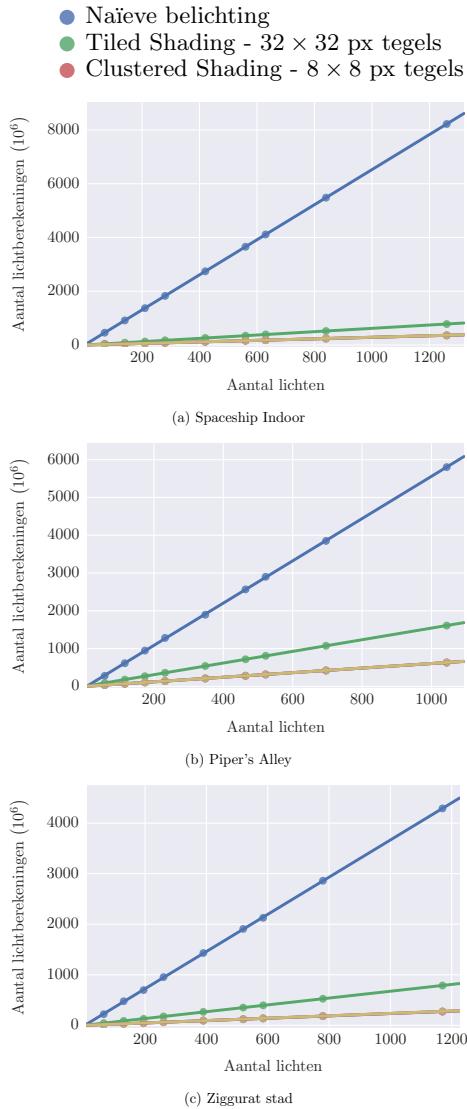
6.3.3 Resolutie

In figuur 6.12 is het gemiddeld aantal lichtberekeningen per frame als functie van de resolutie weergegeven. In deze grafieken is een kwadratisch verband zichtbaar, wat duidt op een lineaire afhankelijkheid van het aantal pixels. Vergelijkbaar met de variatie van aantal lichten, presteert Clustered Shading ook bij verschillende resoluties beter dan zowel de naïeve als de Tiled Shading implementaties. Ook bij de variërende resolutie is geen significante invloed waar te nemen voor de verschillende cluster grootten.

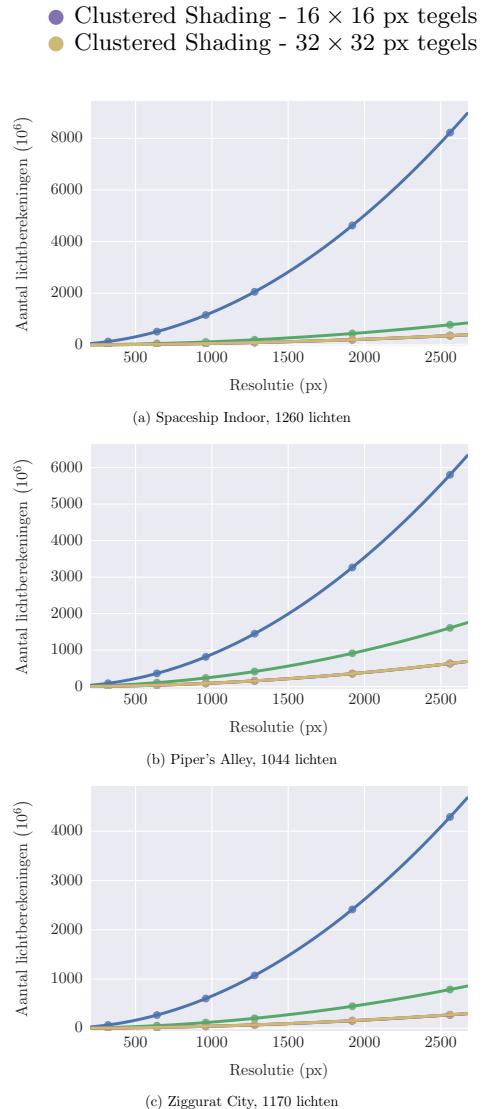
6.4 Conclusie

Op basis van het aantal lichtberekeningen presteert Clustered Shading beter dan Tiled Shading. Doordat beter onderscheid gemaakt kan worden in de camera-**z**-as bevatten clusters kleinere sets lichten. Dit leidt tot een reductie in het aantal lichtberekeningen ten opzichte van Tiled Shading. Tevens zorgt het voor een consistentere aantal lichtberekeningen tussen frames, doordat lichtoverlap met respect tot de camera-**z**-as geen significante rol meer speelt. Het aantal lichtberekeningen binnen Clustered Shading blijft echter lineair afhankelijk van zowel het aantal pixels als het aantal lichten in de scène.

6.4. Conclusie



Figuur 6.11: Lichtberekeningen.



Figuur 6.12: Resolutie.

Vergelijkbaar met Tiled Shading lijkt de tegelgrootte geen significante invloed te hebben op het aantal lichtberekeningen voor de geëvalueerde scènes. Dit is opnieuw toe te schrijven aan de grootte van de lichten binnen de scènes. Doordat de lichten overlappen met een groot aantal clusters zullen naburige clusters veelal dezelfde sets van lichten bevatten.

De uitvoeringstijden van een geheel frame, en om de set van clusters op te bouwen zijn niet explicet geëvalueerd. Hiervoor is gekozen omdat de uitvoeringstijd van het opbouwen van de clusters grotendeels wordt bepaald door de `glGetTexImage` functie. Daarnaast is de voorgestelde methode om lichten toe te kennen aan de clusters[OBA12] niet geïmplementeerd. Om deze redenen is het niet mogelijk om een

6. CLUSTERED SHADING

accurate vergelijking te maken tussen Tiled en Clustered Shading met betrekking tot de uitvoeringstijd.

Forward Clustered Shading is niet expliciet geëvalueerd omdat de toevoeging van de z-prepass een significante aanpassing aan de structuur van de shader vereist. Hierdoor wordt een vertekend beeld verkregen bij een directe vergelijking.

6.5 Discussie

6.5.1 Lichttoekenning op de GPU

In de originele implementatie van Clustered Shading wordt gebruik gemaakt van een Bounding Volume Hierarchy[OBA12] met een splitsingsfactor van 32. Een dergelijke voorstelling maakt het mogelijk om efficiënter lichten toe te kennen aan de clusters dan met de bruteforce aanpak gebruikt in `nTiled`. Hierdoor kunnen grotere aantallen lichten in een realtime setting gebruikt worden.

Deze aanpak is tevens te implementeren op de grafische kaart. Hierdoor kunnen deze lichttoekenningsoperaties parallel uitgevoerd worden, wat de constructie verder reduceert. Wanneer het opbouwen van de clusters op de grafische kaart plaatsvindt, is er geen reden meer om de clustergegevens uit het videogeheugen op te halen. Hierdoor dient de `glGetTexImage` functie niet meer uitgevoerd te worden, wat de bottleneck in de Clustered Shading implementatie binnen `nTiled` zou verhelpen. Wanneer deze twee aanpassingen gedaan zouden worden, zou de uitvoeringstijd van Clustered Shading kleiner dan de uitvoeringstijd van Tiled Shading en de naïeve implementatie moeten zijn.

Hoofdstuk 7

Hashed Shading

In de voorgaande hoofdstukken zijn Tiled en Clustered Shading geïntroduceerd. Het doel van deze lichttoekenningsalgoritmes is om voor elk fragment een verzameling van relevante lichtbronnen te bepalen, om zo 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 kunnen worden tussen frames. Tegelijkertijd behaald het nog steeds een vergelijkbare versnelling en wordt het geheugengebruik beperkt door gebruik te maken van een hiërarchische onderverdeling van de ruimte van de scène.

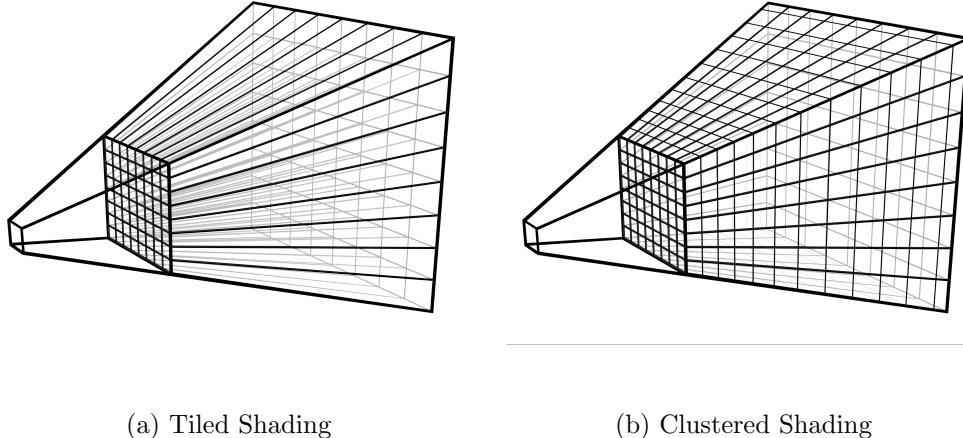
In de volgende secties zal eerst de achterliggende theorie besproken worden, waarbij ingegaan wordt op de keuze van de spatiale datastructuur, en hoe deze voorgesteld kan worden op de GPU. Vervolgens zal het het algoritme behandeld worden. Hierna zal de efficiëntie en het geheugengebruik aan de hand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

7.1 Theorie

In het algemeen kan gesteld worden dat het doel van lichttoekenning is om de ruimte zo efficiënt mogelijk op te delen, opdat voor elk punt in de scène de verzameling van relevante lichtbronnen opgehaald kan worden. De relevante lichtbronnen zijn alle lichten die invloed hebben op het punt in kwestie, i.e. voor al deze lichten geldt dat het lichtvolume overlapt met dit punt. Een datastructuur dient deze bevraging mogelijk te maken. Hiervoor dient de datastructuur 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, opdat het geheugengebruik minimaal is.

7. HASHED SHADING



Figuur 7.1: De onderverdelingen binnen Tiled en Clustered Shading

- De datastructuur moet dynamisch aan te passen zijn, indien lichten van positie of grootte veranderen.
- De datastructuur moet efficiënt geconstrueerd kunnen worden.

Als laatste dient deze datastructuur verder camera-onafhankelijk te zijn, opdat een verandering in het zichtvenster niet leidt tot een gehele heropbouw van de datastructuren.

Tijdens het renderen zal per frame voor elk fragment de verzameling van relevante lichten opgehaald worden. Deze operatie zal dus het meest uitgevoerd 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 scènes, zoals koplampen van bewegende auto's, alsook lokale lichten die veranderen in intensiteit door veranderingen in de scène, zoals uitdovende vuren, of explosies. Een datastructuur dient instaat te zijn om dergelijke effecten te modelleren, zonder dat de invloed op renderingstijd te groot 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 efficiënter kan zijn om de al opgestelde datastructuur (gedeeltelijk) te hergebruiken.

Indien een datastructuur efficiënt kan worden bijgewerkt gedurende de uitvoering, zal de efficiëntie van het initieel opstellen van de datastructuur van minder groot belang zijn, gezien deze stap eenmalig als een pre-processstap uitgevoerd kan worden. Vervolgens is het niet meer nodig om deze stap per frame opnieuw uit te voeren.

Binnen Tiled en Clustered Shading wordt de zichtvensterruimte onderverdeeld zoals weergegeven in figuur 7.1. In deze technieken 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 behandeld worden om de octree datastructuur voor te stellen op de GPU.

7.1.1 Overzicht van spatiale datastructuren

Binaire-ruimte-partitionering (BSP)

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[Nay98]. 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 punten binnen de lijst zijn. In het geval van een driedimensionale ruimte zullen de hypervlakken standaard vlakken zijn. 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. 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 toeneemt, gezien voor elke cel binnen het rooster de ruimte om een cel op te slaan gereserveerd dient te worden, ongeacht of deze daadwerkelijke data bevattet.

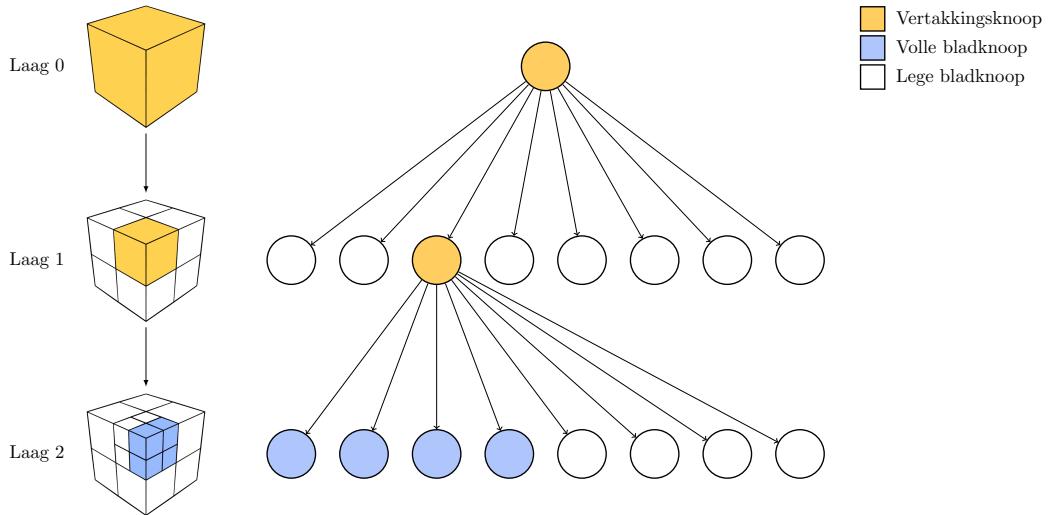
Clustered Shading maakt gebruik van een vorm van een rooster over het zichtfrustrum. Hierbij wordt de data compacter voorgesteld door slechts cellen die zowel licht als geometrie bevatten op te slaan.

Een rooster kan simpelweg gedefinieerd worden als een driedimensionale lijst. Indien dit als binaire-ruimte-partitionering wordt weergegeven, wordt recursief in elke dimensie steeds één cel gedefinieerd.

Octree

De octree is een boomdatastructuur waarbij met elke knoop een kubus is geassocieerd. Elke takknoop bezit precies acht kindknopen. De kinderen verdelen de ruimte geassocieerd met de takknoop op in acht equivalenten nieuwe balken[Mea82]. Deze

7. HASHED SHADING



Figuur 7.2: Weergave van een octree bestaande uit drie lagen, links is de 3d representatie weergegeven, rechts de pointerrepresentatie .

structuur is geïllustreerd in figuur 7.2. Data kan ofwel in elke knoop opgeslagen worden, ofwel alleen in de bladknopen. Indien een knoop de geassocieerde 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 zonder tegen dezelfde geheugenproblemen als het rooster aan te lopen, doordat grote homogene ruimtes met slechts één enkele knoop voorgesteld kunnen worden.

De octree definieert een ruimtepartitionering door per takknoop drie hypervlakken op te stellen. Deze zullen altijd het middelpunt van de octreeeknoop snijden.

kd-boom

Een kd-boom is een boomstructuur in d dimensies, waarbij k discriminatoren zijn gedefinieerd[Ben75]. De discriminator associeert een specifieke oriëntatie van een hypervlak met een punt. Alle knopen op eenzelfde niveau delen dezelfde discriminator. Een takknoop definieert vervolgens de positie van het hypervlak met oriëntatie k_i . Hierbij wordt de ruimte geassocieerd met de knoop opgedeeld in twee delen.

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 deelverzameling is van alle mogelijke hypervlakken.

R-boom

De R-boom is een n -dimensionale zelfbalancerende boom. Met elke knoop is een n -dimensionale balk geassocieerd. De data opgeslagen in de R-boom bevindt zich in de bladknopen, waar de positie van elk datapunt binnen het vierkant geassocieerd met

de bladknoop ligt[Gut84]. De n -dimensionale vierkanten hebben verder de volgende eigenschappen

- De vierkanten overlappen minimaal
- Elk vierkant bevat slechts een minimale hoeveelheid lege ruimte.

Deze eigenschappen in combinatie met het feit dat de boom gebalanceerd is, zorgen ervoor dat de spatiale data efficiënt opgehaald kan worden.

7.1.2 Keuze voor de datastructuur

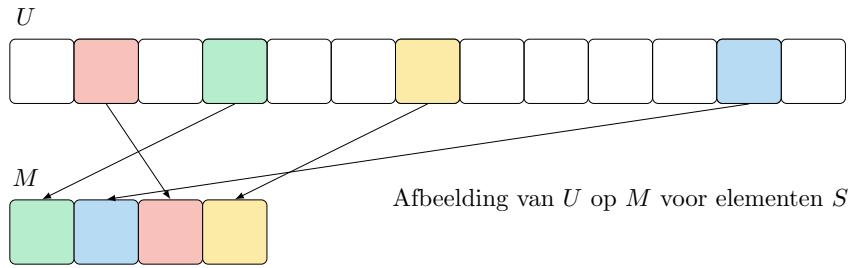
Aan het begin van deze sectie is het doel van de lichttoekenningsdatastructuur gesteld op het efficiënt onderverdelen van de ruimte, zodanig dat voor een punt \mathbf{p} in deze ruimte de verzameling van relevante lichtbronnen opgehaald kan worden. Hiervoor dient de ruimte onderverdeeld te worden, waarna vervolgens de lichten toegekend moeten worden aan de volumes van deze opdeling op basis van hun corresponderende lichtvolumes. Nadat de datastructuur opgesteld is, dient het volume van de opdeling waar punt \mathbf{p} in valt berekend te kunnen worden.

De R-boom is ontwikkeld met het oog op zoekproblemen zoals het dichtstebuurprobleem. Om deze reden is het minder geschikt als lichttoekenningsdatastructuur, waar het doel is om een beschrijving rond het punt \mathbf{p} te geven.

De rooster-datastructuur vereist dat voor de gehele scène cellen worden opgesteld met een uniforme grootte. Om een accurate voorstelling te verkrijgen dient deze grootte zo klein mogelijk gekozen te worden. Dit leidt tot een groot geheugenverbruik, gezien voor elk van deze cellen geheugen gereserveerd dient te worden. Dit geheugenverbruik gaat in tegen de eis dat de datastructuur compact voor te stellen moet zijn.

Zowel de kd-boom als de octree zijn hiërarchische datastructuren waardoor deze compacter in het geheugen voorgesteld kunnen worden. Doordat de lichtvolumes niet als geheel aan de volumes van de opdeling toegekend hoeven te worden kunnen de grenzen van de opdeling arbitrair gekozen worden. In het geval van de kd-boom kunnen deze grenzen gekozen worden op basis van de data opgeslagen in de kd-boom. Voor de octree liggen de mogelijke grenzen van de opdeling vast, ongeacht de data in de octree. Dit betekent dat voor een efficiënte opdeling van de lichten binnen de kd-boom een heuristiek voor de plaatsing van de grenzen ontwikkeld dient te worden. Hierbij dient tevens rekening gehouden te worden met het dynamische karakter van de lichten.

De octree-implementatie vereist deze extra complexiteit niet. Daarnaast zou de uniformiteit van de knopen per laag van de octree het ondersteunen van dynamische lichten moeten vergemakkelijken. Verder is de verzameling van relevante lichten voor een punt \mathbf{p} efficiënt op te halen door de octree te doorlopen. De hiërarchische structuur maakt het mogelijk om de octree compact in het geheugen voor te stellen. Om deze redenen is de octree-datastructuur gekozen als basis voor het Hashed Shading algoritme.


 Figuur 7.3: Afbeelding van U op M .

7.1.3 Hash functies

Nu vastgesteld is dat de octree-datastructuur de basis zal vormen voor het licht-toekenningsalgoritme, is het nodig om deze efficiënt op de grafische kaart voor te stellen. Hiervoor zal gebruik gemaakt worden van hashfuncties. In deze sectie zal ingegaan worden op de theorie achter hashfuncties. Vervolgens zal ingegaan worden hoe hashfuncties gebruikt kunnen worden om spatiale data op te slaan. Daarna zal een octreevoorstelling op basis van dergelijke spatiale hashfuncties geïntroduceerd worden.

Het doel van een hashfunctie is om een verzameling van waarden verspreid over een ruimte af te beelden op een compactere ruimte. Een hashfunctie kan als volgt gedefinieerd worden: gegeven een ruimte $U = \{0, 1, 2, \dots, u - 1\}$ bestaande uit u positieve integers. Binnen deze ruimte U is een verzameling S gedefinieerd van n unieke elementen waar $n \leq u$. De hashfunctie h beeldt de elementen van S af op een interval $M = [0, m - 1]$:

$$h : U \rightarrow M$$

hierbij worden de elementen van S sleutelwaarden genoemd. Deze afbeelding is geïllustreerd in figuur 7.3.

Hashfuncties kunnen gebruikt worden om dunverspreide data efficiënt op te slaan. Stel de situatie waar enkele elementen in U data bevatten, en deze data wordt opgevraagd aan de hand van deze elementen. Wanneer voor elk element in U geheugen gereserveerd dient te worden waarin de data opgeslagen kan worden, leidt dit tot een onnodig groot geheugengebruik, gezien het merendeel van de elementen leeg zullen zijn. Indien nu de elementen van U waarmee data geassocieerd is, worden genomen als de sleutelwaarden, en een hashfunctie wordt opgesteld voor deze sleutelwaarden, dan hoeft slechts voor elk element van M , waarop de sleutelwaarden worden afgebeeld, geheugen gereserveerd worden. Indien $m < u$ leidt dit tot een lager geheugengebruik. In dit geval wordt M de adresruimte genoemd, en het opslagvolume waarin de data wordt opgeslagen, een hashtabel[Cor09, KT06].

Indien twee sleutels op eenzelfde adres worden afgebeeld, wordt gesproken van een botsing. De sleutels corresponderende met dit adres worden synonymen van elkaar genoemd. Botsingen leiden er toe dat waardes geassocieerd met synonymen

niet meer in een enkele stap uit de hashtabel 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 hashfunctie, of een 1-probe hashfunctie[CHM97]. 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 adresruimte M geassocieerd moet worden met de verzameling van sleutelwaardes S , groter of gelijk aan de verzameling van sleutelwaardes dient te zijn, $m \geq n$.

Wanneer geldt dat $m = n$ wordt de gesproken van een perfecte minimale hashfunctie[CHM97]. Een dergelijke minimale hashfunctie bestaat voor elke verzameling van sleutelwaarden, doordat geldt dat voor twee eindige sets X en Y van gelijke grootte er altijd een injectieve functie bestaat zodanig dat X op Y wordt afgebeeld[CHM97]:

Het blijkt echter niet triviaal om dergelijke functies te vinden. Dit wordt verder gecompliceerd doordat het geheugengebruik en de rekentijd die het opstellen van dergelijke functies vereist ook van belang is. In de praktijk is het veelal niet doenlijk om minimale perfecte hashfuncties op te stellen. Er zal dus veelal voor gekozen worden om perfecte hashfuncties op te stellen die slechts zorgen voor een kleine toename in geheugengebruik ten opzichte van een minimale perfecte hashfunctie.

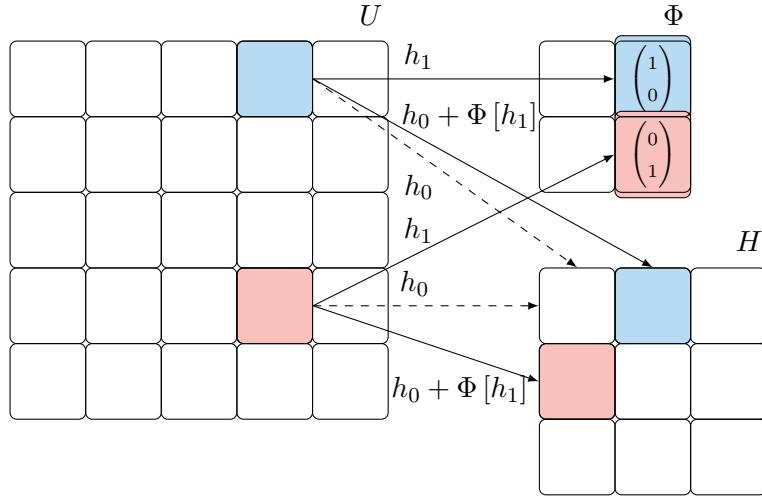
7.1.4 Spatiale hashfuncties

In Lefebvre en Hoppe[LH06] wordt een algoritme geïntroduceerd om een perfecte hashfunctie op te stellen met behulp van twee imperfecte hashfuncties en een hashtabel. Hierdoor is het mogelijk om d -dimensionale ruimtes compact op te slaan. Door het gebruik van perfecte hashfuncties kan de GPU efficiënt data ophalen uit de hashtabel, doordat er geen conditietakken nodig zijn om botsingen op te lossen. Lefebvre en Hoppe noemen deze perfecte d -dimensionale hashfuncties spatiale hashfuncties. Er zal in deze sectie eerst ingegaan worden op de terminologie en definitie van deze spatiale hashfuncties. Daarna zal de constructie en het gebruik worden toegelicht.

Terminologie

Omdat deze thesis zich richt op driedimensionale scènes zal slechts ingegaan worden op de driedimensionale spatiale hashfunctie. De ruimte U is gedefinieerd als een driedimensionaal rooster bestaande uit $u = \dot{u}^3$ knopen. Posities binnen dit rooster zijn gedefinieerd als een vector van drie integers met waarden kleiner dan \dot{u} :

$$\mathbf{p} = \mathbb{N}_{\dot{u}}^3 = \begin{pmatrix} p_x \\ p_y \\ p_z \end{pmatrix} | p_i \in [0, (\dot{u} - 1)]$$



Figuur 7.4: De perfecte spatiale hashfunctie $h(\mathbf{p})$ in twee dimensies.

Binnen dit rooster bevindt zich een verzameling $S \subseteq U$ met n elementen. Met elk element op positie $\mathbf{p} \in S$ is een data-element d geassocieerd, zodanig dat

$$D(\mathbf{p}) = d$$

Definitie

Het idee is om de perfecte hashfunctie h op te bouwen met behulp van twee simpele imperfecte hashfuncties, h_0 en h_1 [LH06]. De eerste hashfunctie, h_0 , beeldt elementen uit U af op de adresruimte M . De sleutelwaarden zijn voor deze hashfunctie mogelijk synoniemen. De tweede hashfunctie beeldt de sleutelwaarden af op de adressen van een kleine hashtabel Φ , die offset-waarden bevat. De synoniemen in de eerste hashfunctie dienen geen synoniemen in de tweede hashfunctie te zijn. Hierdoor worden synoniemen van h_0 op verschillende adressen in Φ afgebeeld door hashfunctie h_1 . Vervolgens wordt de offset-waarde opgehaald uit Φ , gebruikt om de botsingen in M door de hashfunctie h_0 te voorkomen. Doordat de synoniemen voor h_0 een verschillende offset toegewezen krijgen, zullen deze niet langer op hetzelfde adres vallen. Hierdoor wordt een perfecte hashfunctie gerealiseerd. Dit principe is geïllustreerd in figuur 7.4. De perfecte hashfunctie h wordt als volgt gedefinieerd:

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

waarbij de modulus operatie wordt uitgevoerd op de elementen van \mathbf{p} . De offset-waardetabel Φ heeft een grootte van $r = r^3$. De hashfuncties h_0 en h_1 worden als simpele modulo-operaties gedefinieerd:

$$\begin{aligned} h_0 : \mathbf{p} &\mapsto \mathbf{p} \bmod m \\ h_1 : \mathbf{p} &\mapsto \mathbf{p} \bmod r \end{aligned}$$

De offset-waardes worden voorgesteld als driedimensionale 8-bit integer vectoren.

Constructie

Voor de constructie van de spatiale hashfunctie h dienen eerst de parameters \dot{m} en \dot{r} vastgesteld te worden. De keuze voor \dot{m} legt de grootte van de hashtabel H vast en de hashfunctie h_0 . Om het geheugengebruik te minimaliseren dient de hashtabel H zo klein mogelijk gehouden te worden. Hiervoor wordt \dot{m} zo klein mogelijk gekozen, zodat nog steeds geldt:

$$m = \dot{m}^3 \geq \begin{cases} n, & \text{als } n \leq 255 \\ n \cdot 1.01, & \text{anders} \end{cases}$$

De extra ruimte voor $n > 255$ is nodig om een perfecte hashfunctie mogelijk te maken met 8-bit offset-waardes[LH06].

De grootte van \dot{r} kan op twee manieren worden vastgesteld:

- Voor een computationeel snelle constructie wordt \dot{r} zo klein mogelijk gesteld opdat nog steeds geldt

$$r = \dot{r}^3 \geq \sigma n$$

waar σ initieel wordt gesteld op $\frac{1}{6}$ [LH06]. Omdat de offset-waardes bestaan uit drie 8-bit integers leidt dit tot een toename van vier bits per sleutelwaarde. Indien geen perfecte hashfunctie kan worden opgebouwd wordt r geometrisch vergroot.

- Voor een zo laag mogelijk geheugengebruik kan gebruik gemaakt worden van een binaire zoekfunctie over $\dot{r}^{\frac{1}{6}}$ [LH06]. Hierbij wordt gebruik gemaakt van een gretig probabilistisch algoritme om \dot{r} zo klein mogelijk te definiëren. De gevonden waarde voor \dot{r} zorgt ervoor dat de spatiale hashfunctie de minimale perfecte hashfunctie zo dicht mogelijk benadert.

Nadat de waardes voor \dot{m} en \dot{r} zijn gekozen liggen de hashfuncties h_0 en h_1 vast. De tweede stap in de opbouw van de perfecte hashfunctie h is het toekennen van de waardes in de offset-hashtabel Φ , zodanig dat er geen botsingen meer voorkomen in h . Deze offset-waardes zullen iteratief worden toegevoegd.

De ruimte O definieert de mogelijk adresruimte van de offset-hashtabel Φ :

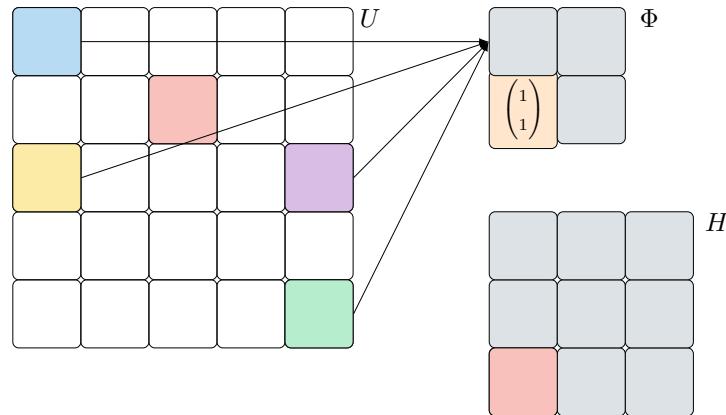
$$O : \mathbb{Z}_{\dot{r}}^3 = [0, (\dot{r} - 1)]^3$$

Indien een offset-waarde wordt toegekend aan positie $\mathbf{o} \in O$ dan ligt het adres van alle punten $\mathbf{p} \in U$ die afgebeeld worden op \mathbf{o} vast voor de hashfunctie h . Deze verzameling kan gedefinieerd worden als:

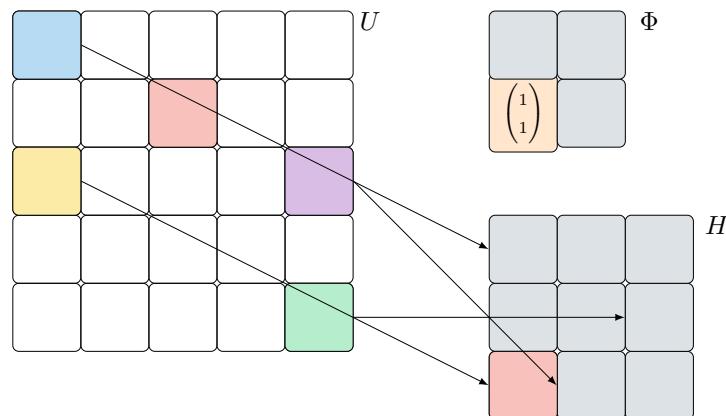
$$V = \{\mathbf{p} \in S \wedge h_1(\mathbf{p}) = \mathbf{o}\}$$

7. HASHED SHADING

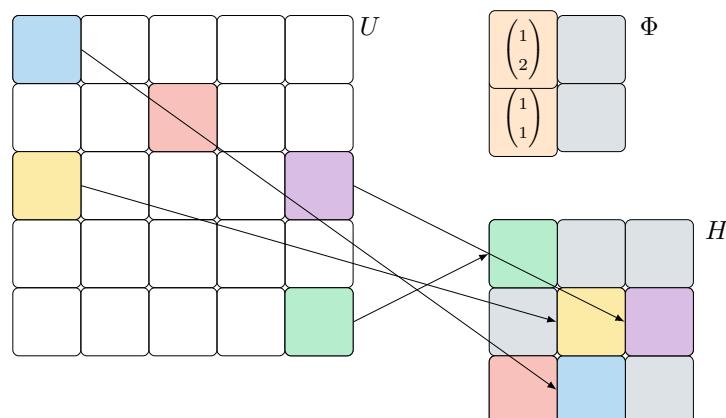
Afbeelding op Φ : $h_1(\mathbf{p})$



Afbeelding door h_0 : $h_0(\mathbf{p})$



Afbeelding door h op H $h_0(\mathbf{p}) + \Phi[h_1(\mathbf{p})]$



Figuur 7.5: De toekenning van een enkele offset-waarde om de perfecte spatiale hashfunctie $h(\mathbf{p})$ op te bouwen.

De data geassocieerd met deze sleutelwaardes kunnen vervolgens worden opgeslagen in hashtabel H zodat geldt:

$$\forall \mathbf{p} \in V : D(\mathbf{p}) = H[h_0(\mathbf{p}) + \Phi[h_1(\mathbf{p}) \bmod m]]$$

Indien de offset-waarde correct gekozen is zal geen van deze waardes een botsing veroorzaken in h .

$$\forall \mathbf{p} \in V : \#\mathbf{p}' \in S' : h(\mathbf{p}) = h(\mathbf{p}')$$

waar S' de verzameling van reeds toegekende sleutelwaarden is. Dit proces is voor een enkele stap weergegeven in figuur 7.5.

Wanneer de verzameling V van synoniemen voor \mathbf{o} groot is, zal het vinden van een correcte offset-waarde moeilijker worden[FHCD92]. Om een correcte toekenning te vergemakkelijken zullen de punten $\mathbf{o} \in O$ geordend worden op basis van de grootte van de corresponderende synoniemenverzamelingen. De waardes \mathbf{o} met de grootste synoniemenverzamelingen zullen als eerste worden toegekend.

De offset-waardes dienen verder zodanig gekozen te worden dat de ruimtelijke coherentie behouden blijft. Hierdoor zal het raadplegen van de hashtabellen ook coherent blijven, wat de performantie van de spatiale datastructuren ten goede komt. Coherentie tussen twee elementen kan gedefinieerd worden als[LH06]:

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

De coherentie binnen de hashtabel H kan vervolgens worden gedefinieerd als[LH06]:

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

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

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

De modulo-operatie van h_1 is per definitie coherent, hierdoor zal het ophalen van de offset-waardes dan ook altijd coherent zijn. Het is mogelijk om de coherentie van h te berekenen voor elke mogelijke offset-waarde. Echter dit zou computationeel te veeleisend zijn. Om deze reden wordt gebruik gemaakt van de aanname dat indien de offset-hashtabel Φ lokaal constant is, de hashfunctie h zich coherent zal gedragen. Wanneer een offset-waarde toegekend wordt aan positie \mathbf{o} , zal eerst gekeken worden of naburige adressen een offset-waarde bevatten die correct is voor \mathbf{o} . Indien geen van de omliggende toegekende offset-waardes correct is, wordt een willekeurige kandidaat gevonden.

7.1.5 Verbindingloze octree

Met behulp van deze spatiale hashfunctie is het mogelijk om de octreestructuur efficiënt op de GPU voor te stellen[CJC⁰⁹]. In de standaard CPU implementatie worden de verbindingen tussen knopen van de octree voorgesteld met behulp van pointers. Vervolgens kan een waarde uit de octree opgehaald worden door recursief af te dalen totdat een bladknoop wordt bereikt. Dergelijke implementaties zijn veelal niet efficiënt op de GPU. De ruimtelijke coherentie gaat verloren door het gebruik van pointers, en het gebruik van veel controlestructuren leidt tot performantieverlies voor meerdere-data-enkele-instructie (SIMD) operaties[HA11]. In plaats hiervan zal geen gebruik gemaakt worden van pointers, maar wordt elke laag voorgesteld met behulp van een spatiale hashfunctie. Deze datastructuur wordt de verbindingloze octree genoemd.

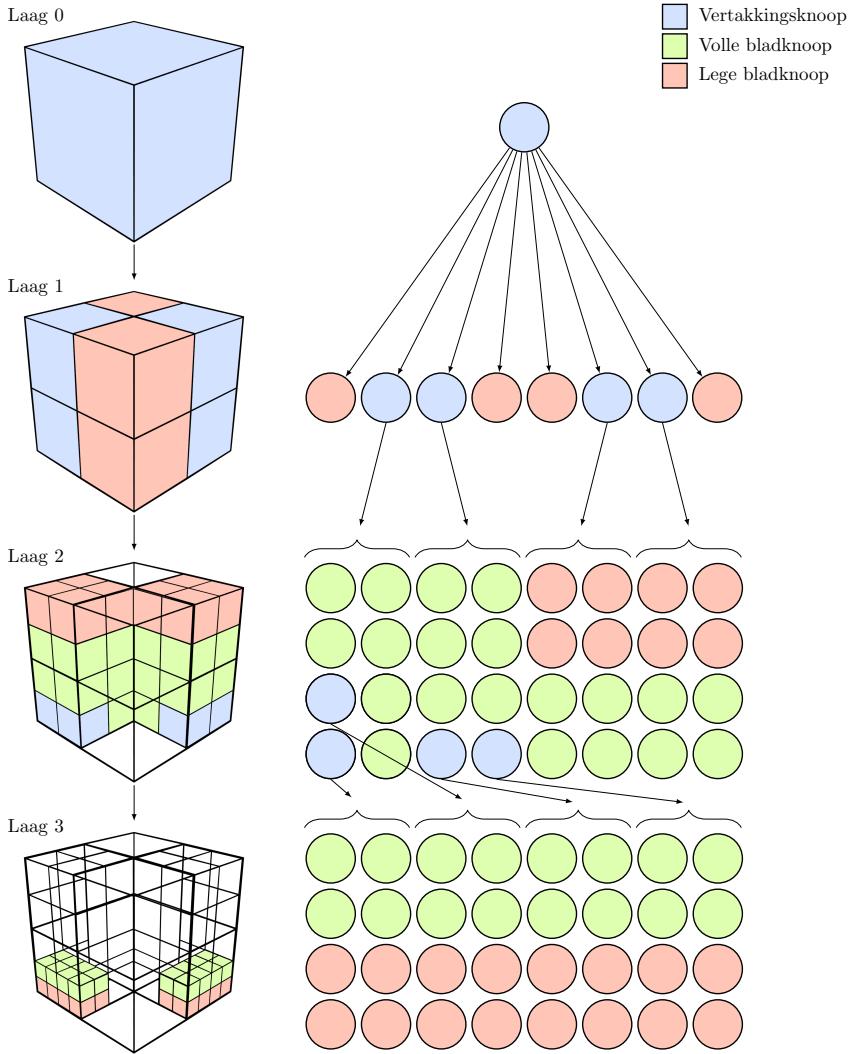
Indien elke laag van een octree apart bekeken wordt, bestaat deze uit een verzameling discrete volumes, binnen een bepaalde ruimte, zie figuur 7.6. Hierin zijn de takknopen weergegeven in blauw, de gevulde bladknopen in groen, en de lege bladknopen in rood. Omdat de grootte van elke volume vastligt per laag, is het mogelijk om de Euclidische ruimte voor te stellen als de ruimte $U_l = \mathbb{N}^3$, waarbij elk van de knopen overeenkomt met een discrete positie in deze ruimte U_l . De verzameling van knopen binnen de laag l kan dan voorgesteld worden als S_l .

Doordat de verzameling van knopen per laag S_l veelal kleiner is dan de gehele ruimte U_l , helemaal voor diepere lagen, kunnen deze efficiënt afgebeeld worden op de ruimte M_l . Om een spatiale hashfunctie op te stellen die de octreestructuur van een laag beschrijft, dient dan slechts nog de data $D_l(\mathbf{p}) | \mathbf{p} \in S_l$ opgesteld te worden. Het data-element $D_l(\mathbf{p}) = k$ geassocieerd met een punt \mathbf{p} in laag l dient de beschrijving van knoop k op punt \mathbf{p} in laag l te geven. Indien de data opgeslagen binnen de octree zich slechts bevindt in de bladknopen zijn er drie mogelijke knootypes voor knoop k . Knoop k is een takknoop, gevulde bladknoop of lege bladknoop. In dit geval kan knoop k worden voorgesteld met twee Booleaanse waardes[CJC⁰⁹]:

1. knoop k is een bladknoop
2. knoop k is gevuld

Om het geheugengebruik terug te dringen worden steeds acht knopen behorende tot dezelfde takknoop in de bovenliggende laag, samengenomen. De acht kinderen van een dergelijke takknoop kunnen dan voorgesteld worden met behulp van twee 8-bit integers[CJC⁰⁹]. De waarde in hashtabel H_l op punt \mathbf{p} beschrijft vervolgens dus alle kinderen van de takknoop in laag l op positie \mathbf{p} . Dit is weergegeven in figuur 7.7.

Naast de octreebeschrijving dient ook de data geassocieerd met knopen van de octree te worden opgeslagen. Wanneer het merendeel van de knopen data met zich geassocieerd heeft, en deze data weinig geheugen inneemt, kan deze data direct worden opgeslagen in de hashtabellen H_l . Hiervoor wordt voor elke knoop een ruimte ter grootte van de data gereserveerd naast de twee 8-bit integers[CJC⁰⁹]. Wanneer een knoop wordt opgehaald uit de hashtabel H_l wordt gelijk de geassocieerde data mee teruggeven. Wanneer de data-elementen groot zijn, of slechts een klein deel van de knopen een data-element bevat, leidt deze aanpak tot een groot onnodig

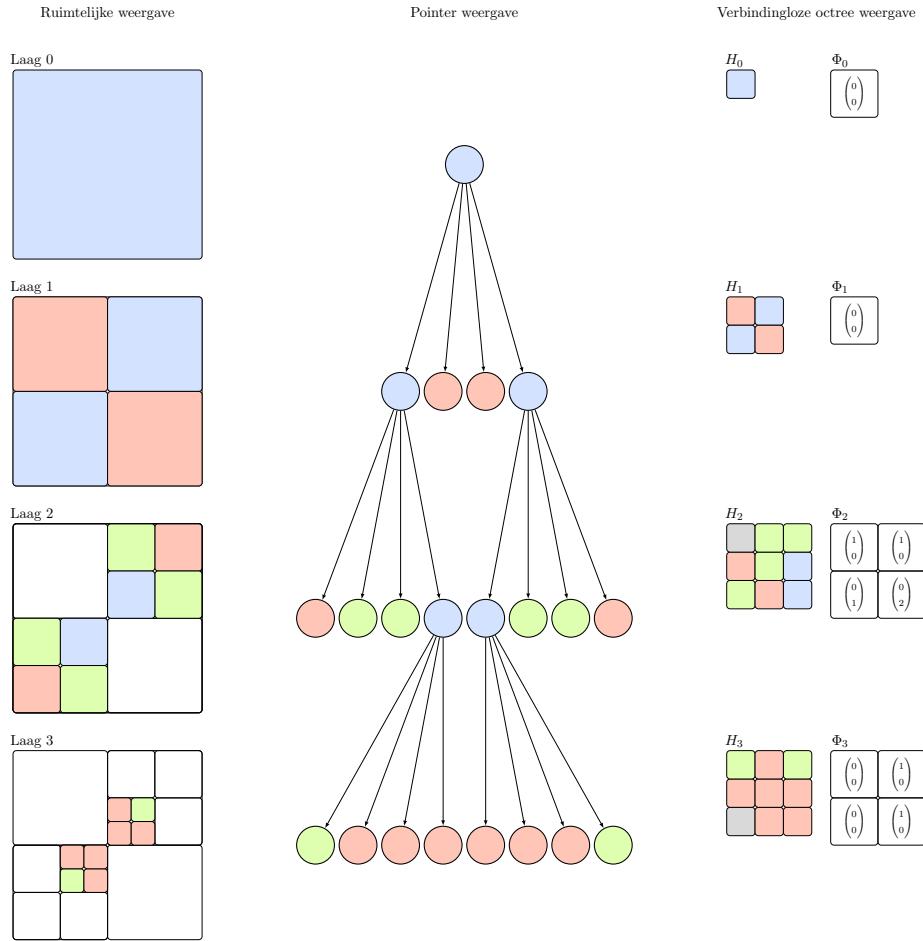


Figuur 7.6: De verschillende knopen in een octree per laag.

geheugenverbruik. In deze gevallen kan de data apart worden opgeslagen. Hiervoor wordt per laag een tweede spatiale hashfunctie opgesteld, waarin voor elke knoop met data een element wordt voorzien[CJC⁺09]. Vervolgens kan de gevuld-bit gebruikt worden om na te gaan of met een punt \mathbf{p} data geassocieerd is. Is dit het geval dan zal het punt \mathbf{p} opgehaald worden uit de tweede spatiale hashtabel.

Wanneer data voor een punt $\mathbf{p} \in \mathbb{R}^3$ opgehaald dient te worden, wordt per laag de corresponderende discrete positie bepaald. Vervolgens wordt de beschrijving van de knoop berekend. Op basis hiervan wordt of wel verder afgedaald in de verbindingloze octree en wordt de octreebeschrijving voor punt \mathbf{p} in de volgende opgehaald, ofwel wordt de data geassocieerd met punt \mathbf{p} in laag l teruggegeven[CJC⁺09]. Afhankelijk van de implementatie wordt deze data gelijk uit hashtabel H_l opgehaald, ofwel dient de tweede spatiale hashfunctie van laag l raad gepleegd te worden.

7. HASHED SHADING



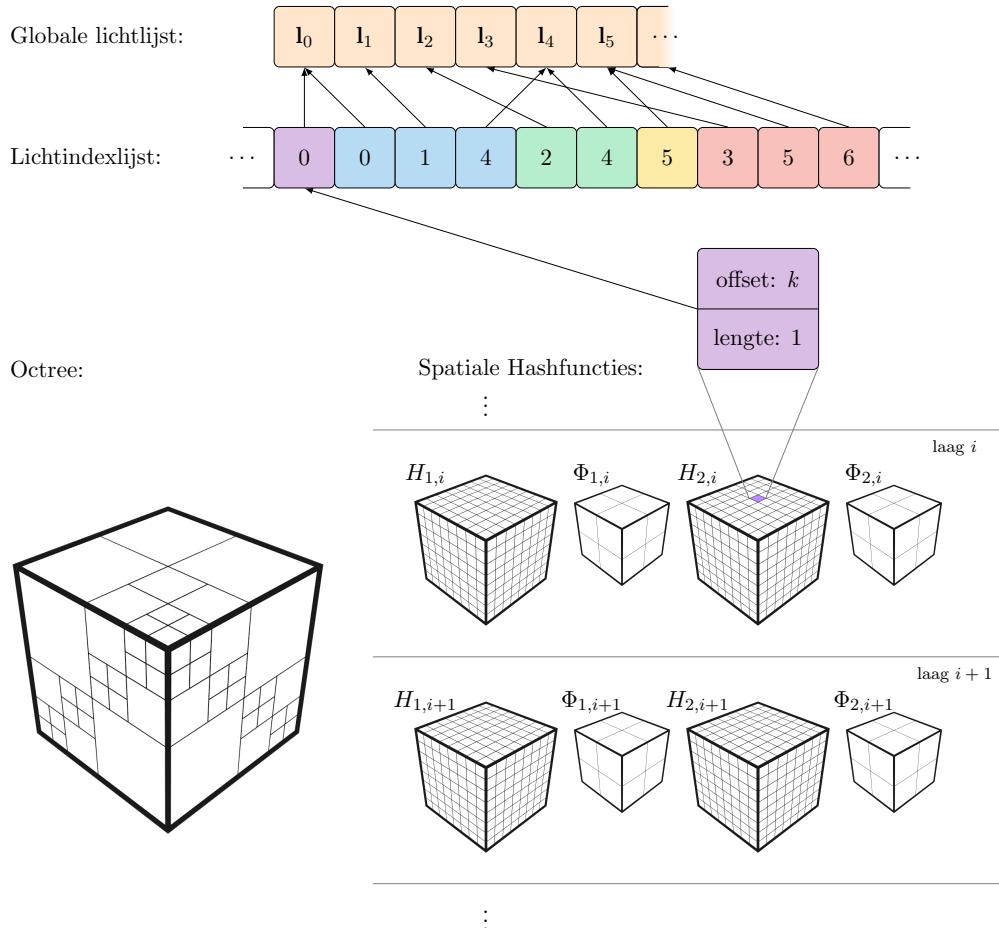
Figuur 7.7: Een voorbeeld van de representatie van een octree met behulp van een verbindingloze octree.

Doordat de verbindingloze octree alle knopen in een laag opslaat, is het mogelijk om op een diepte dieper dan 0 te beginnen. Deze begindiepte wordt de startdiepte d_s genoemd. Indien deze niet gelijk is aan nul, worden alle knopen op diepte d_s opgeslagen in de eerste spatiale hashfunctie van de verbindingloze octree[CJC⁺09].

Doordat de hashtabellen van de spatiale hashfuncties simpel voor te stellen zijn als buffers of texturen kan de verbindingloze octree efficiënt voorgesteld worden op de GPU. Hierdoor is het mogelijk om deze datastructuur als basis te gebruiken voor het Hashed Shading algoritme. Dit algoritme zal geïntroduceerd worden in de volgende sectie.

7.2 Algoritme

In deze sectie wordt het Hashed Shading algoritme toegelicht. Hiervoor zal eerst ingegaan worden op het achterliggende idee en zal een overzicht worden gegeven van



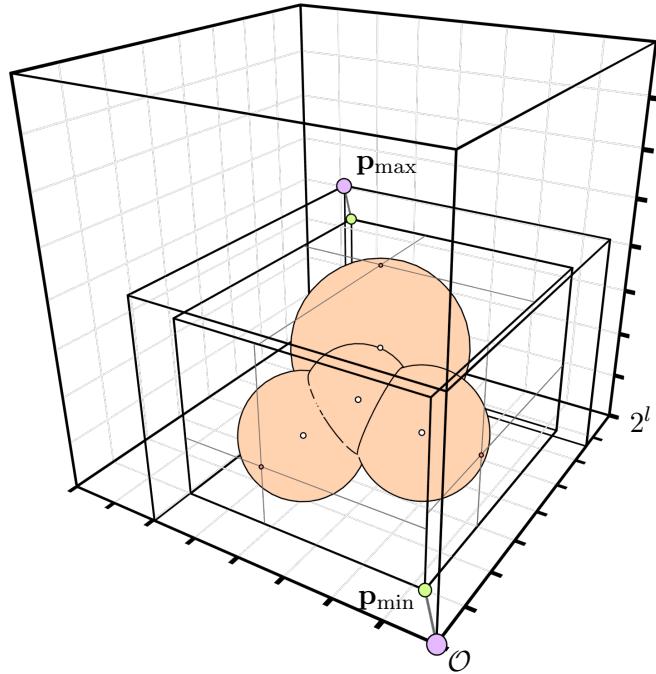
Figuur 7.8: Overzicht van de datastructuren gebruikt in Hashed Shading

de gebruikte datastructuren. Hierna zal ingegaan worden op de constructie en het gebruik van Hashed Shading binnen de fragmentshader.

7.2.1 Overzicht

Het doel van Hashed Shading is om de datastructuren gebruikt voor de lichttoekenning gedeeltelijk her te gebruiken tussen frames. Dit wordt bereikt door een camera-onafhankelijke verbindingloze octree te gebruiken om de ruimte van de scène onder te verdelen. Voor elk volume geassocieerd met een bladknoop wordt bijgehouden welke lichten overlappen. Hiervoor zal vergelijkbaar met Tiled en Clustered Shading een lichtindexlijst en een globale lichtlijst bijgehouden worden. De verbindingloze octree vervangt dan het lichtrooster van Tiled Shading, of de clusters van Clustered Shading. Dit leidt tot de datastructuren weergegeven in figuur 7.8.

Er zijn drie componenten die uitgewerkt dienen te worden om Hashed Shading te gebruiken in grafische applicaties:



Figuur 7.9: Een voorbeeld van de opdeling van de scène met behulp van een octree.

- De constructie van de datastructuren.
- Het opvragen van de lichtinformatie voor een punt \mathbf{p} .
- Het bijwerken van de datstructuren wanneer lichten aangepast worden.

Binnen nTiled zijn de eerste twee componenten geïmplementeerd. Een strategie voor het bijwerken van de datastructuren zal besproken worden in sectie 8.2.3.

Er zal eerst ingegaan worden op de constructie van de datastructuren. Vervolgens zal ingegaan worden op het opvragen van de lichtinformatie, zoals deze plaatsvindt op de grafische kaart.

7.2.2 Octree-opdeling van de scène

Om de benodigde datastructuren op te bouwen dient de scène voorgesteld te worden als een octree. Dit betekent dat eerst de opdeling van de ruimte opgesteld moet worden. Daarna wordt aan deze opdeling de lichten toegekend. Voor deze opdeling wordt een nieuwe verzameling coördinaten geïntroduceerd, de octree-coördinaten. Hierbij zijn de wereldcoördinaten getransleerd, zodanig dat de oorsprong van het octree-coördinatenstelsel overeen komt met de oorsprong van de octree. Hierdoor zullen alle relevante punten positief zijn.

De octree dient zodanig gekozen te worden dat alle lichten in de scène er binnen vallen. Hiervoor wordt de oorsprong van de octree genomen als het grootste punt \mathbf{p}_{\min} dat kleiner is dan alle lichtevolumes minus een offset \mathbf{c} . Indien L de verzameling van lichtbronnen is, kan dit gedefinieerd worden als:

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

De offset-waarde \mathbf{c} is meegenomen om te voorkomen dat door afrondingsfouten lichtvolumes buiten de octree zouden vallen.

Nadat de oorsprong van de octree is berekend, dient de grootte van de octree bepaald te worden. Op een vergelijkbare manier als de oorsprong, kan het maximale punt \mathbf{p}_{\max} bepaald worden. Dit punt \mathbf{p}_{\max} is gedefinieerd als het kleinste punt dat groter is dan alle lichtvolumes. De octree wordt voorgesteld als een kubus, dus dient de lengte van deze kubus minimaal gelijk te zijn aan de grootste dimensie tussen de oorsprong \mathbf{p}_o en het maximale punt \mathbf{p}_{\max} plus de offset-waarde \mathbf{c} :

$$\max(\mathbf{p}_{\max} - \mathbf{p}_{\min} + 2\mathbf{c})$$

De gebruiker stelt de grootte van de knopen in de diepste laag in. Dit wordt de minimale knoopgrootte d genoemd. De grootte van de octree kan vervolgens bepaald worden aan de hand van de minimale lengte en de minimale knoopgrootte. Hierbij dient de integer l zo klein mogelijk gekozen te worden zodanig dat geldt:

$$2^l * d \geq \max(\mathbf{p}_{\max} - \mathbf{p}_{\min} + 2\mathbf{c})$$

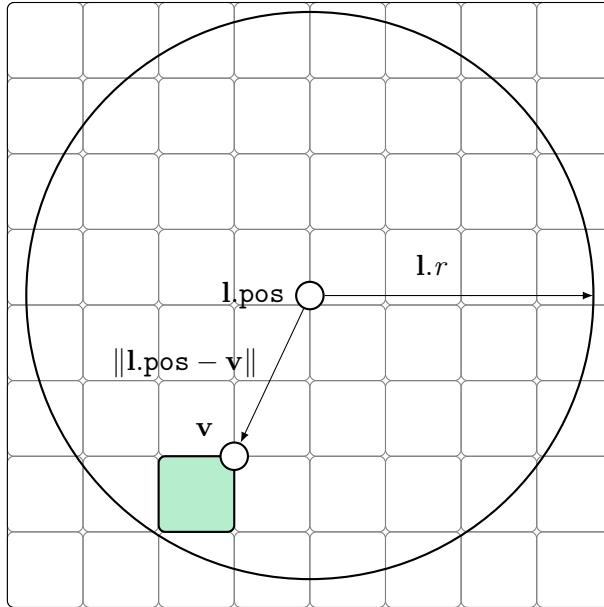
De waarde l komt overeen met het aantal lagen dat de octree bezit.

Hiermee liggen alle mogelijke opdelingen van de scène vast. Dit is geïllustreerd in figuur 7.9.

7.2.3 Voorstelling van enkele lichten

Wanneer de grootte en oorsprong van de octree berekend zijn, ligt de positie van alle mogelijke knopen vast. De volgende stap is om de lichten in de scène toe te kennen aan de opdeling. Hiervoor zal eerst voor elk van de lichten bepaald worden met welke bladknopen uit de diepste laag van de octree deze overlappen. Op basis van het lichtvolume kan het kleinste-mogelijke rooster van knopen van minimale grootte waarbinnen het licht l valt, berekend worden. Hiervoor wordt bepaald in welke knopen het kleinste en grootste punt van de omsluitende kubus van het lichtvolume vallen:

7. HASHED SHADING



Figuur 7.10: Bepaling of een knoop overlapt met het lichtvolume.

```
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 )
```

Listing 7: Functie om te evalueren of een licht overlapt met een knoop.

$$\mathbf{p}_{l,\min} = \left\lfloor \frac{l.\text{positie} - l.\text{radius}}{d} \right\rfloor$$

$$\mathbf{p}_{l,\max} = \left\lfloor \frac{l.\text{positie} + l.\text{radius}}{d} \right\rfloor$$

waarbij de punten $\mathbf{p}_{l,\min}$ en $\mathbf{p}_{l,\max}$ zich bevinden in octreecoördinaten. De waarde d is de minimale knooggrootte. De grootte van het rooster is vervolgens gedefinieerd als:

$$\max(\mathbf{p}_{l,\max} - \mathbf{p}_{l,\min}) + 1$$

De oorsprong van het rooster in wereldcoördinaten is vervolgens:

$$\mathbf{p}_{l,\min} * r + \mathbf{p}_o$$

Vervolgens dient per knoop bepaald te worden of deze overlapt met het lichtvolume. Hiervoor wordt gekeken of het punt \mathbf{v} dat binnen het knoopvolume valt en het dichtstbij de oorsprong van de lichtbron ligt, op een afstand kleiner dan de radius van het licht ligt. Is dit het geval dan overlapt de knoop met het lichtvolume. Dit is weergeven in figuur 7.10. Om het punt \mathbf{v} dat binnen het knoopvolume ligt en zich op de kleinste afstand van de oorsprong van het licht \mathbf{l} bevindt, te vinden, wordt de oorsprong van het licht per dimensie geklemd tussen de uiterste waardes van het knoopvolume:

$$\mathbf{v} = \begin{pmatrix} \mathbf{l}.position_x |_{\mathbf{k}_{x,i}, \mathbf{k}_{x,i}+d} \\ \mathbf{l}.position_y |_{\mathbf{k}_{y,i}, \mathbf{k}_{y,i}+d} \\ \mathbf{l}.position_z |_{\mathbf{k}_{z,i}, \mathbf{k}_{z,i}+d} \end{pmatrix}$$

waar \mathbf{k}_i de oorsprong van het knoopvolume is, en d de minimale knoopgrootte. Dit resulteert in de code weergegeven in listing 7.

Het is niet nodig om deze berekening voor elke knoop in het rooster uit te voeren. De lichtvolumes zullen altijd uniform zijn. Dit betekent dat een flood-fill algoritme gebruikt kan worden om ofwel de lege of volle knopen in het rooster te vullen. Het volume van een puntlicht is een bol ter grootte van:

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

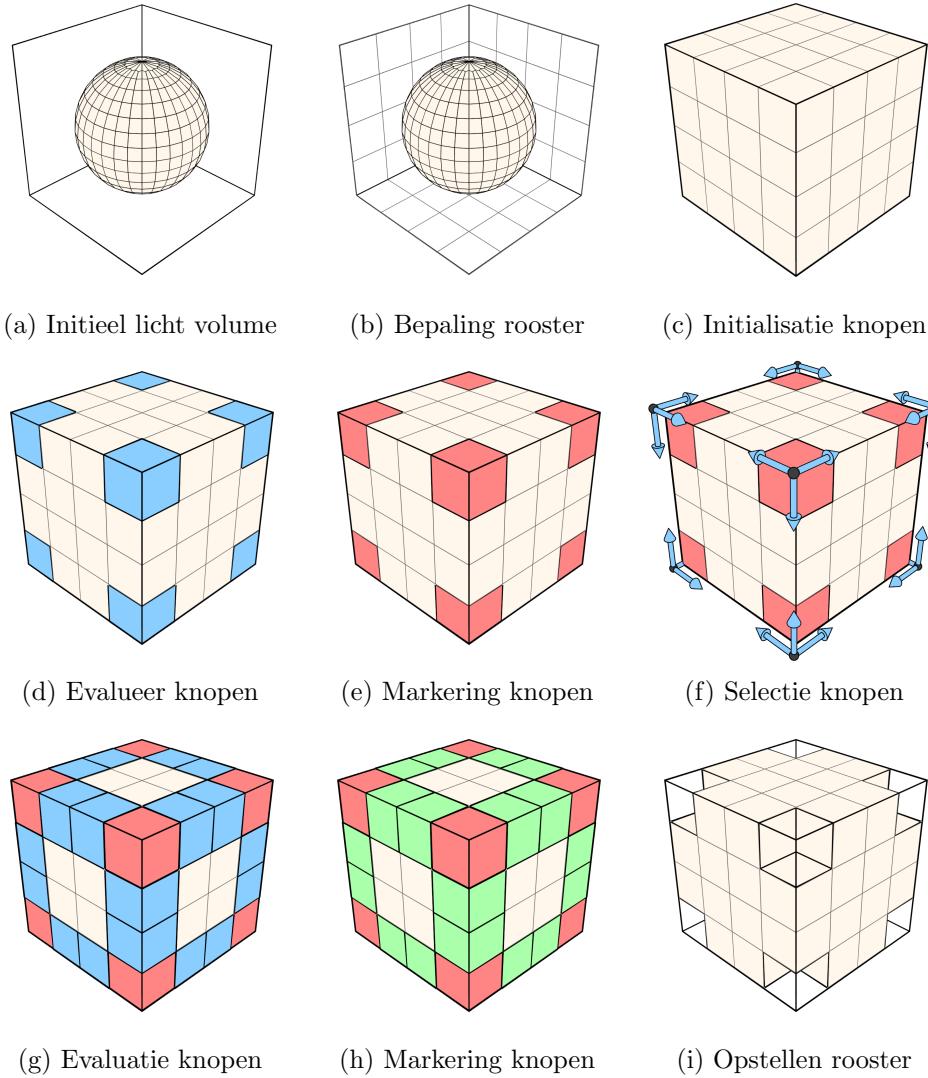
Het volume van een bol omvat iets meer dan de helft van het volume van de omsluitende kubus, om deze reden is er voor gekozen om er van uit te gaan dat alle knopen overlappen met het licht. Vervolgens worden de niet overlappende knopen als zodanig gemarkerd. Hiervoor wordt begonnen in elk van de hoekpunten van het rooster. Dit gehele proces is weergeven in figuur 7.11.

Nu vastgesteld is welke knopen van minimale grootte overlappen met een lichtvolume, zijn er drie mogelijke vervolgstappen:

- De gevulde knopen kunnen direct worden samengevoegd. Op basis van deze set knopen kan dan bottom-up de octree worden opgebouwd.
- De roosters per licht kunnen worden opgeslagen. Hierna wordt de informatie in deze roosters samengevoegd tot éénzelfde verzameling van knopen als in de eerste mogelijkheid.
- De roosters kunnen per licht omgezet worden naar een octreestructuur, die vervolgens wordt opgeslagen. Deze octrees kunnen vervolgens worden samengevoegd om zo tot een enkele octree te komen die de gehele scène beschrijft.

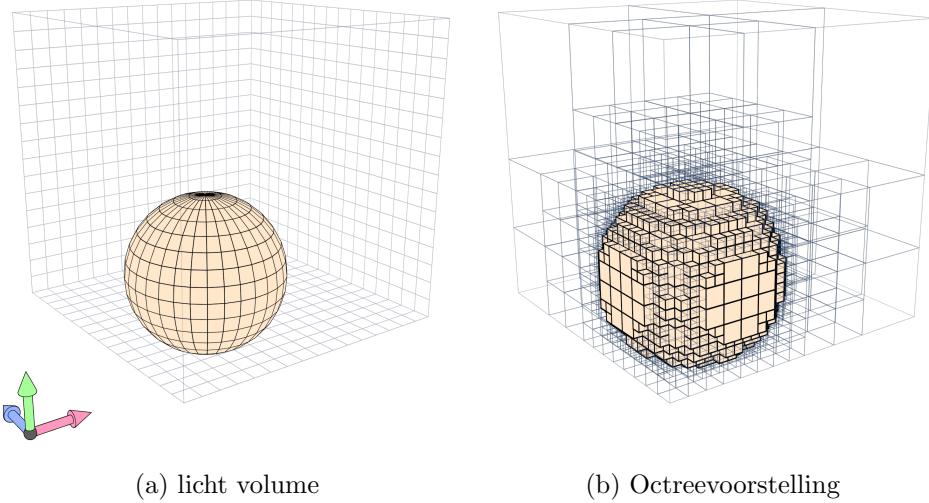
Indien slechts statische lichten worden gebruikt, is er geen reden om lichten los bij te houden, dus zal de eerste optie in dit geval de meest efficiënte keuze zijn.

7. HASHED SHADING



Figuur 7.11: Opbouw van het minimale rooster voor een enkel licht.

Indien lichten dynamisch van aard zijn, kan het nodig zijn dat de knopen waarop een licht invloed heeft, te veranderen. Om efficiënt de verandering in knopen te berekenen kan de oude verzameling knopen vergeleken worden met de verzameling van knopen die de nieuwe situatie beschrijft. Om onnodig rekenwerk te voorkomen kan vervolgens de verzameling van knopen waarop een licht invloed heeft in de huidige frame bijgehouden worden. Wanneer slechts het rooster per licht wordt bijgehouden, optie twee, is het nodig om elke knoop in het rooster met één bit voor te stellen. Indien de knoopgrootte groot is, of de lichtvolumes klein kan dit efficiënt bijgehouden worden. Echter deze verzameling knopen wordt kubiek groter bij kleinere knoopgroottes. Wanneer de knoopgrootte klein is zal een octreevoorstelling leiden tot minder geheugengebruik per licht, waardoor optie drie efficiënter zal zijn. De



Figuur 7.12: Voorstelling van een enkele licht als octree.

voorstelling van een licht als octree is geïllustreerd in figuur 7.12.

Binnen `nTiled` is er voor gekozen om de lichten op te slaan in de octreevoorstelling. Dit is gedaan met het oog op de uitbreiding om dynamische lichten te ondersteunen.

De octreevoorstelling van een enkel licht wordt top-down opgebouwd met behulp van het opgestelde rooster. Hierbij wordt de octreeknoop die het gehele rooster omvat genomen als de wortelknoop voor de octreevoorstelling van het enkele licht. Beginnend van deze knoop wordt bepaald of een knoop een takknoop of bladknoop is. In het geval dat een knoop een takknoop is, worden ook de kindknopen van deze knoop geëvalueerd. Voor knoop \mathbf{k} zijn er initieel drie mogelijke situaties:

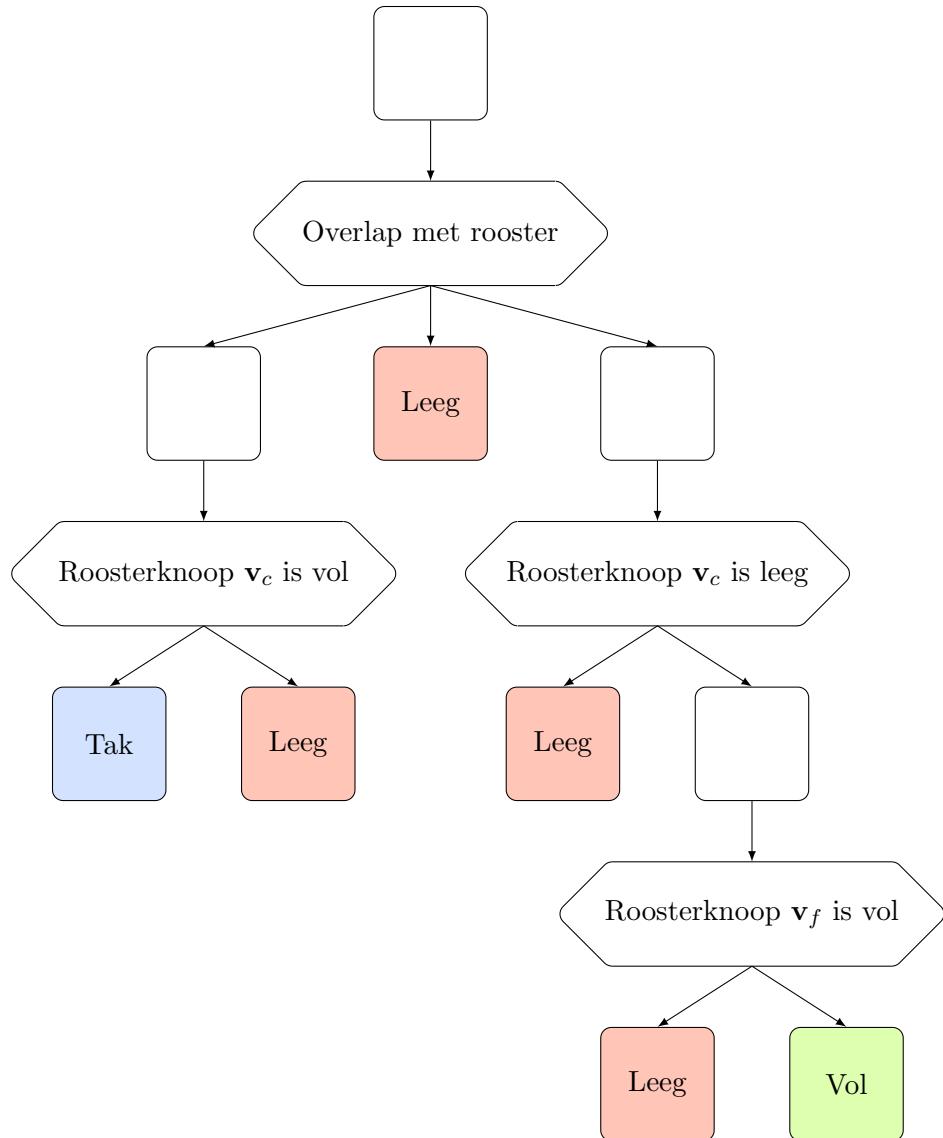
- Het volume van knoop \mathbf{k} overlapt niet met het rooster.
- Het volume van knoop \mathbf{k} overlapt gedeeltelijk met het rooster.
- Het volume van knoop \mathbf{k} valt in zijn geheel binnen het rooster.

In het eerste geval zal de knoop \mathbf{k} altijd een lege bladknoop zijn, gezien er geen gevulde knopen buiten het rooster vallen.

In het tweede geval zijn er twee mogelijkheden, of wel er liggen volle roosterknopen binnen knoop \mathbf{k} , of knoop \mathbf{k} bestaat slechts uit lege roosterknopen. Indien de roosterknoop het dichtst bij de oorsprong van het licht leeg is, zullen de andere roosterknopen tevens leeg zijn. Dit is een gevolg van het feit dat het lichtvolume bolvormig is. Er zullen geen knopen verder dan de geëvalueerde roosterknoop wel gevuld zijn als de geëvalueerde knoop leeg is. In dit geval is knoop \mathbf{k} een lege bladknoop. Is de dichtstbijzijnde roosterknoop gevuld, dan bevat knoop \mathbf{k} zowel lege als gevulde knopen, en is het een takknoop.

In het laatste geval zijn er drie mogelijke situaties:

- Knoop \mathbf{k} omvat slechts lege roosterknopen.
- Knoop \mathbf{k} omvat slechts gevulde roosterknopen.



Figuur 7.13: De beslissingsboom om het type van een octreeknop te bepalen.

- Knoop k omvat zowel lege als gevulde roosterknopen.

Knoop k is in deze situaties respectievelijk een lege bladknoop, een volle bladknoop, en een takknoop. Indien de roosterknoop het dichtst bij de oorsprong van het licht leeg is, dan omvat knoop k slechts lege roosterknopen. Indien deze roosterknoop gevuld is, en de roosterknoop het verste van de oorsprong van het licht is tevens gevuld, dan omvat knoop k slechts gevulde knopen. Anders bevat knoop k zowel een lege als een gevulde knoop, en zal deze dus een takknoop zijn. De beslissingsboom voor dit algoritme is weergegeven in figuur 7.13. Hiermee kan elk licht als octree worden voorgesteld.

scène-octreeknoop o		Lichtoctreeknoop l	
		Bladknoop	Takknoop
	Leeg	Vol	
Bladknoop	Stop	De index van knoop 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 knoop l wordt toegevoegd aan de nieuwe octreeknoop o' .
Takknoop	Stop	Knoop l wordt toegevoegd aan elk van de kinderen van octreeknoop o .	Elk van de kinderen van de knoop l wordt toegevoegd aan het overeenkomstige kind van octreeknoop o .

Tabel 7.1: De mogelijke situaties wanneer een lichtoctreeknoop wordt toegevoegd aan een scène-octreeknoop.

7.2.4 Voorstelling van de scène

Nadat vastgesteld is hoe elk licht individueel wordt toegekend aan de opdeling, dient de octree voor de gehele scène opgesteld te worden. Deze scène-octree bevat per bladknoop een verzameling referenties naar de relevante lichten.

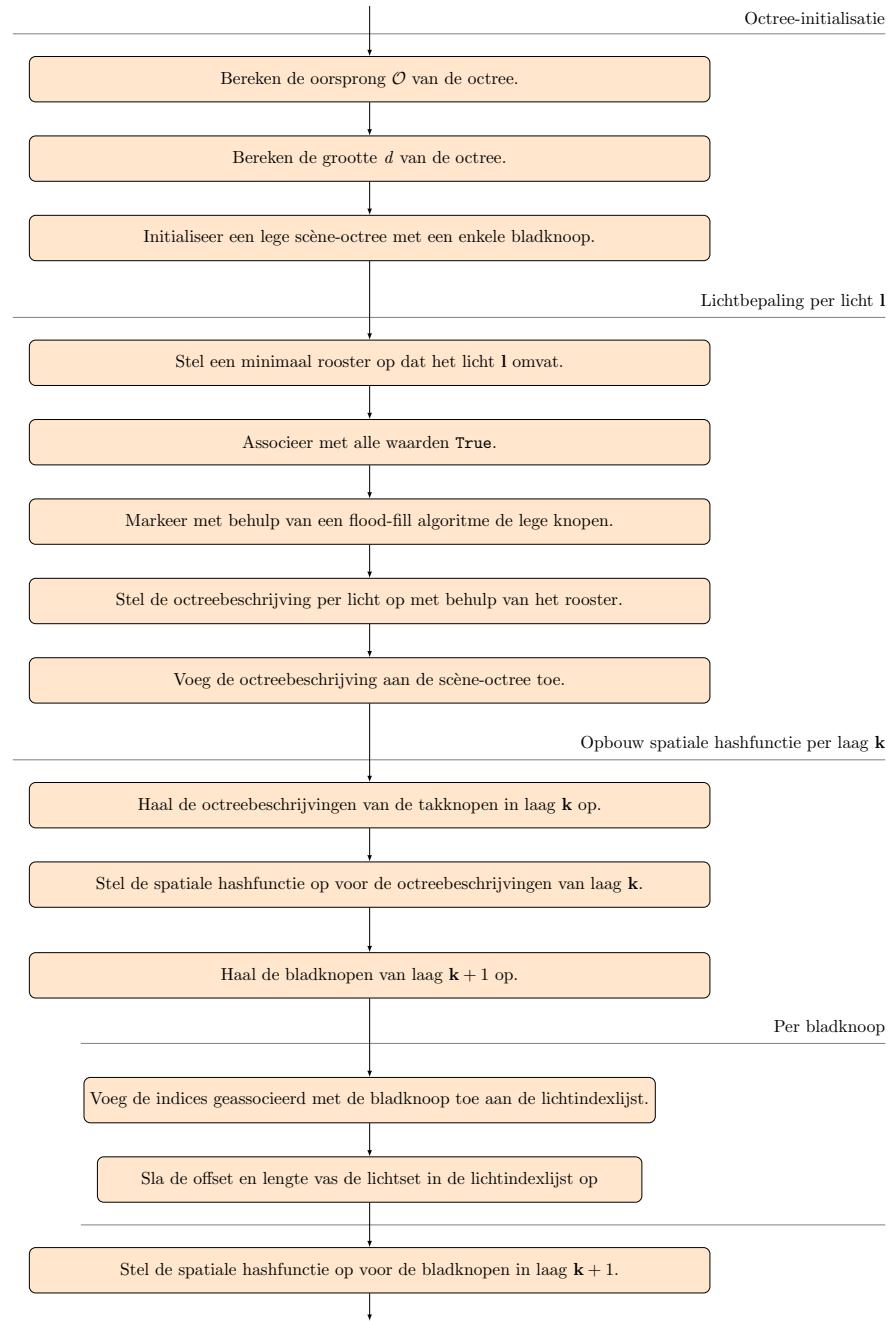
Indien de individuele lichten met een rooster voorgesteld zijn, kunnen de roosterknopen samengevoegd worden zodat een verzameling van bladknopen ontstaat. Deze kunnen vervolgens bottom-up samengevoegd worden, om zo tot een octree te komen.

Indien de individuele lichten voorgesteld zijn met een octree, kunnen elk van deze lichtoctrees top-down worden samengevoegd tot een scène-octree. Hiervoor worden de lichtoctrees iteratief toegevoegd aan een scène-octree geïnitialiseerd met een enkele lege bladknoop. Eerst wordt afgedaald tot de knoop die correspondeert met de wortel van de lichtoctree. Indien een bladknoop **k** wordt tegengekomen, wordt deze opgedeeld door deze te vervangen met een takknoop met kinderen gelijk aan bladknoop **k**. Vervolgens wordt de wortelknoop toegevoegd aan de corresponderende scène-knoop. Hierbij zijn er vijf mogelijke situaties die beschreven zijn in tabel 7.1.

7.2.5 Verbindingloze octree

Nu de scène-octree is opgesteld kan de verzameling van datastructuren die gebruikt wordt in de lichttoekenning geconstrueerd worden. Hiervoor dient de scène-octree

7. HASHED SHADING



Figuur 7.14: Een overzicht van het algoritme om de datastructuren van Hashed Shading op te bouwen.

voorgesteld te worden als een verbindingloze octree[CJC⁰⁹]. Tevens moet de lichtindexlijst opgesteld worden.

Omdat de verzameling van relevante lichten slechts gedefinieerd is voor de gevulde bladknopen, is er voor gekozen om de octreestructuur en lichtinformatie gescheiden op te slaan. Per laag l worden dus twee spatiale hashfuncties opgesteld, één die de octreebeschrijving bevat, en één die de lichtinformatie voor alle gevulde knopen in laag l bevat. De lichtinformatie wordt vergelijkbaar voorgesteld als in Tiled en Clustered Shading. Elke gevulde bladknoop krijgt twee integers toegewezen die een deelverzameling van de lichtindexlijst specificeren.

Het algoritme voor het opstellen van de datastructuren is weergeven in figuur 7.14. Na de uitvoering is een lichtindexlijst en twee sets van spatiale hashfuncties opgesteld. Deze kunnen vervolgens in het geheugen geladen en gebruikt door de shaders worden.

7.2.6 Lichttoekenning

Het tweede component van Hashed Shading is het ophalen van de relevante lichtinformatie voor punt \mathbf{p} binnen de fragmentshader. Dit komt neer op het doorlopen van de verbindingloze octree, totdat een bladknoop wordt bereikt[CJC⁰⁹]. In het geval dat deze niet leeg is, kan de corresponderende lichtinformatie worden opgehaald uit de datahashtabel.

De afdaaling komt neer op per laag het berekenen van de octreeknoop beschrijving waarin punt \mathbf{p} valt:

$$(\text{is_leaf}, \text{is_filled}) = H_l \left[h_0 \left(\left\lfloor \frac{\mathbf{p}}{d_l} \right\rfloor \right) + \Phi \left[h_1 \left(\left\lfloor \frac{\mathbf{p}}{d_l} \right\rfloor \right) \right] \right]$$

waarbij d_l de knoopgrootte van de laag l is.

`is_leaf` en `is_filled` beschrijven alle acht kinderen van de takknoop \mathbf{k} in laag l . De bits die overeenkomen met de kindknoop \mathbf{k}' van takknoop \mathbf{p} waarin punt \mathbf{p} valt, bevinden zich op positie:

$$k'_{\text{bit}} = k'_x + 2 \cdot k'_y + 4 \cdot k'_z$$

Door bitverschuivingen toe te passen, kan de bit op positie $k^p \text{prime}_{\text{bit}}$ achterhaald worden.

Er zijn drie mogelijke situaties:

- kindknoop \mathbf{k}' is een takknoop, en er wordt verder afgedaald in de octree
- kindknoop \mathbf{k}' is een lege bladknoop, en een nulvector wordt teruggegeven.
- kindknoop \mathbf{k}' is een gevulde bladknoop, en de lichtinformatie wordt opgehaald uit de tweede spatiale hashfunctie geassocieerd met laag l .

Vervolgens kan op een vergelijkbare manier als in Tiled en Clustered Shading de verzameling van relevante lichten worden overlopen om de uiteindelijke kleur van punt \mathbf{p} te berekenen. De GLSL-code voor dit algoritme is gegeven in listing 8

```

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]);

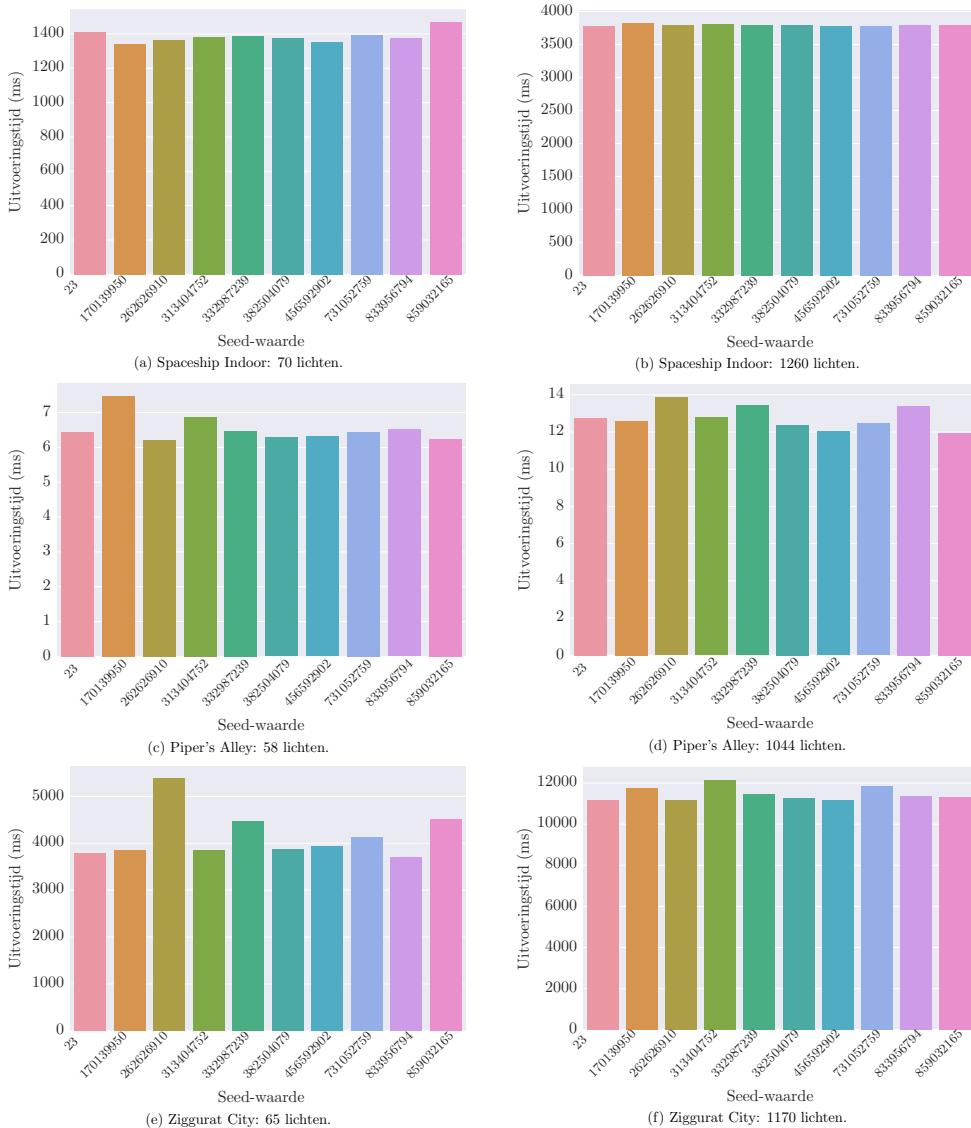
    if(extractBit(octree_data.x, index_int)) {
        if(extractBit(octree_data.y, index_int)) {
            light_data =
                obtainNodeFromSpatialHashFunction(
                    octree_coord_next,
                    light_offset_tables[layer_i],
                    light_data_tables[layer_i]);
        }
        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 8: De implementatie van de lichtberekening in de GLSL shader van Hashed Shading.

7.3. Resultaten

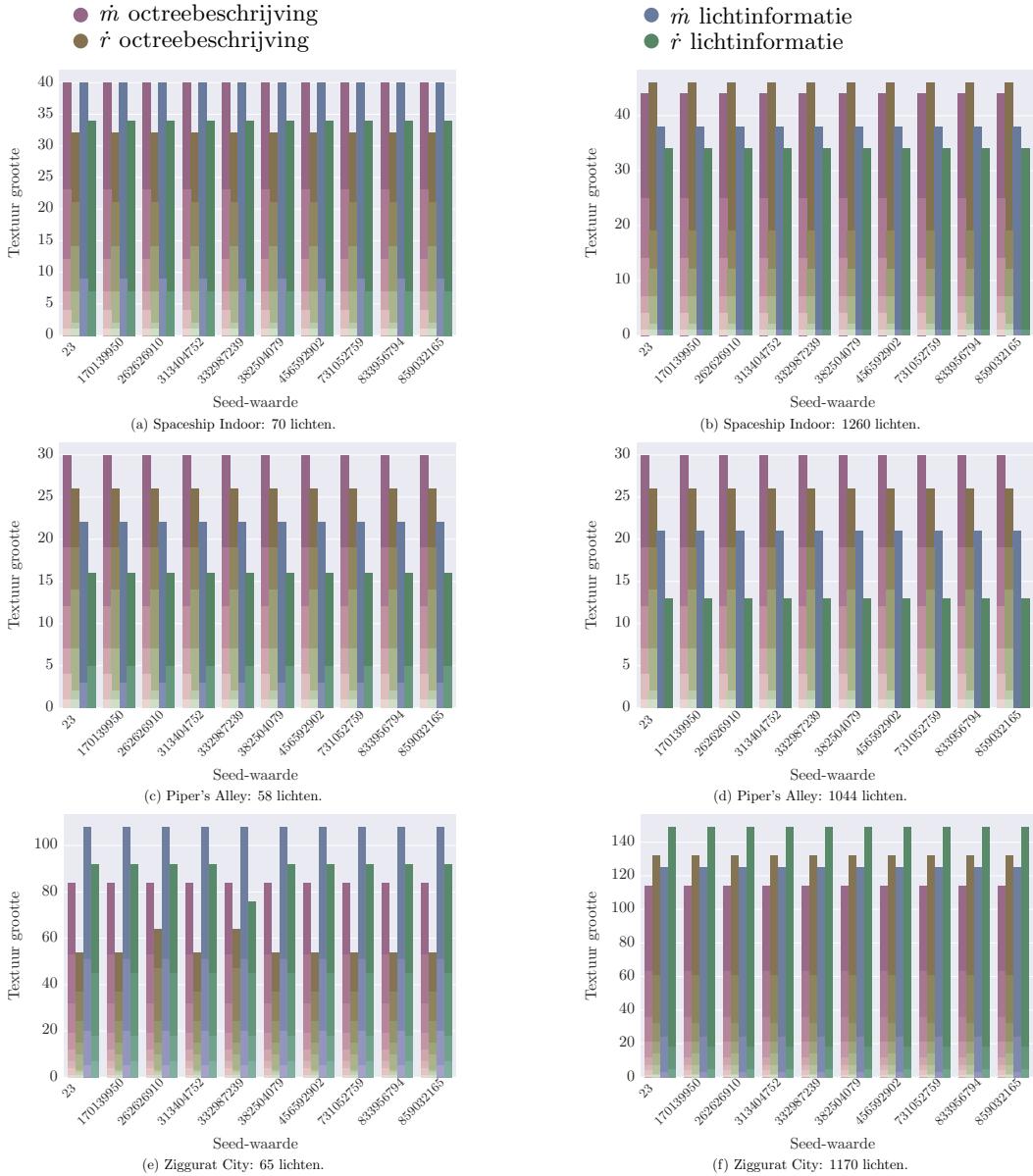


Figuur 7.15: Overzicht van de constructietijd van de verbindingloze octree bij verschillende seed-waarden.

7.3 Resultaten

In de volgende sectie zal de performantie van Hashed Shading geëvalueerd worden. hiervoor zal gekeken worden naar de invloed van de seed, minimale knoopgrootte, en begindiepte, op de constructietijd en geheugengebruik van de datastructuren, en op de uitvoeringstijd en aantal lichtberekeningen.

7. HASHED SHADING



Figuur 7.16: Overzicht van het geheugengebruik van de verbindingloze octree bij verschillende seed-waarden.

7.3.1 Seed-waarde

Hashed Shading maakt gebruik van willekeurige nummergeneratie om de offset-waarden voor de offset-tabel Φ te genereren, indien geen naburige offset-waarden correct zijn. De hypothese is dat deze seed-waarde geen invloed heeft op de performantie van de verbindingloze octree.

Indien de seed-waarde invloed zou hebben op de verbindingloze octree, zou het niet de uitvoeringstijd en het aantal lichtberekeningen beïnvloeden. De seed-waarde

heeft slechts invloed op de grootte van de gegenereerde spatiale hashfuncties, en dus op het geheugengebruik en de constructietijd. De uitvoeringstijd en het aantal lichtberekeningen worden niet beïnvloed door de grootte van de spatiale hashfunctie. Om deze reden is slechts gekeken naar de constructietijd en het geheugenverbruik, bij verschillende seed-waarden. Hierbij zijn voor de drie scènes de kleinste en grootste verzamelingen lichten geëvalueerd. Dit is gedaan bij een knoopgrootte gelijk aan de helft van de lichtradius van de lichten gebruikt in de scène, en een begindiepte van nul voor tien verschillende seed-waarden.

De resultaten voor de constructietijd zijn weergegeven in figuur 7.15. De resultaten voor het geheugengebruik zijn weergegeven in figuur 7.16. Uit de grafieken blijkt dat het geheugen niet wordt beïnvloed door de seed-waarde. Elk van de lagen verbruikt dezelfde hoeveelheid geheugen. Verder is er geen significante invloed waar te nemen op de constructietijd van de verbindingloze octree. De variatie die zichtbaar is kan verklaard worden door het feit dat bij sommige seed-waarden mogelijk meer incorrecte offset-waarden worden gegenereerd. Elk van deze offset-waarden dient gecontroleerd te worden, en er zullen offset-waarden gegenereerd worden totdat een correcte waarde wordt gevonden. Dit introduceert een kleine hoeveelheid extra werk.

Omdat de seed-waarde geen invloed heeft, zal in de verdere testen steeds een seed waarde van 23 gebruikt worden.

7.3.2 Knoopgrootte van de verbindingloze octree

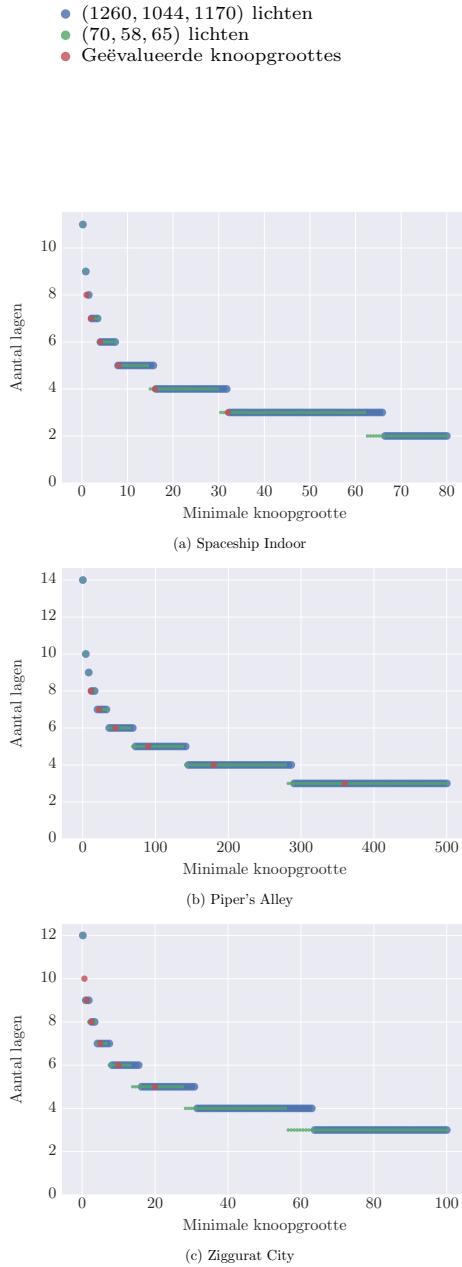
In figuur 7.17 is het aantal lagen dat de octree vereist om de gehele lichtconfiguratie te bedekken, weergegeven als functie van de minimale knoopgrootte. In het rood zijn de waarden weergegeven die verder geëvalueerd zullen worden. De scènes zijn zodanig opgesteld dat een groter aantal lichten niet leidt tot meer lagen. Elke verzameling lichten is verdeeld over hetzelfde volume ongeacht de grootte van de verzameling van lichten. Een kleinere knoopgrootte leidt tot meer lagen en meer minimale knopen. Dit heeft tot gevolg dat de verbindingloze octree zowel in constructietijd als geheugengebruik zal toenemen wanneer de knoopgrootte wordt gereduceerd.

De constructietijd als functie van de knoopgrootte voor de verschillende lichtconfiguraties is weergegeven in figuur 7.18. Hierbij zijn de lichtaantallen per scène weergegeven in de legenda. De knoopgrootte is relatief aan de gebruikte radius voor de lichten in de scène:

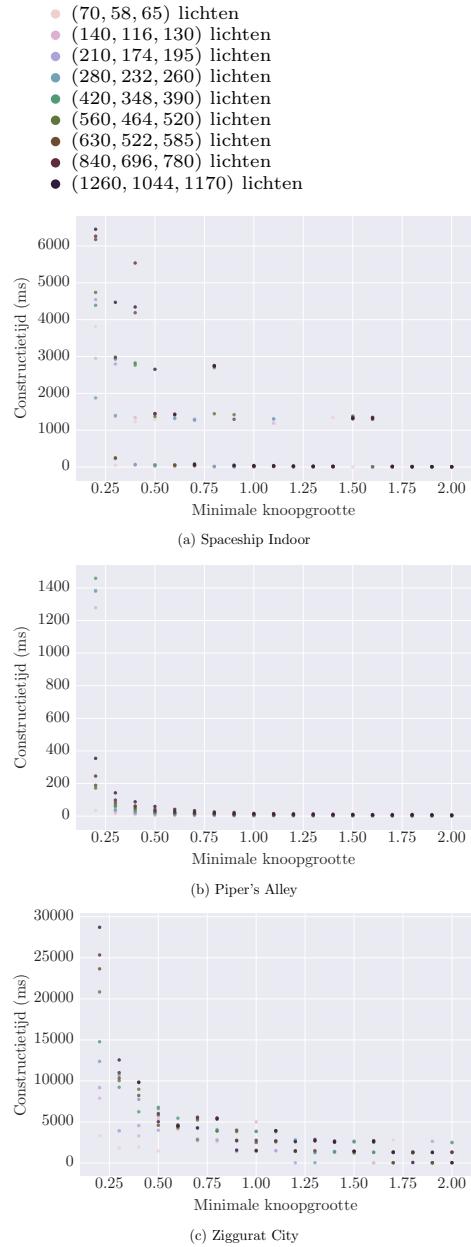
- Spaceship Indoor: 23.0
- Piper's Alley: 180.0
- Ziggurat City: 10.0

In figuur 7.19 zijn de constructietijden per functie weergegeven voor de kleinste en grootste verzameling lichten voor elke scène. In figuur 7.20 is het geheugengebruik van de verbindingloze octree als functie van de knoopgrootte relatief aan de lichtradius weergegeven. Hierbij is het aantal pixels in elke textuur van de verbindingloze octree gesommeerd. Deze waarde is als indicatie gebruikt voor het geheugengebruik. In figuur 7.21 is de grootte van de lichtindexlijst als functie van de knoopgrootte

7. HASHED SHADING



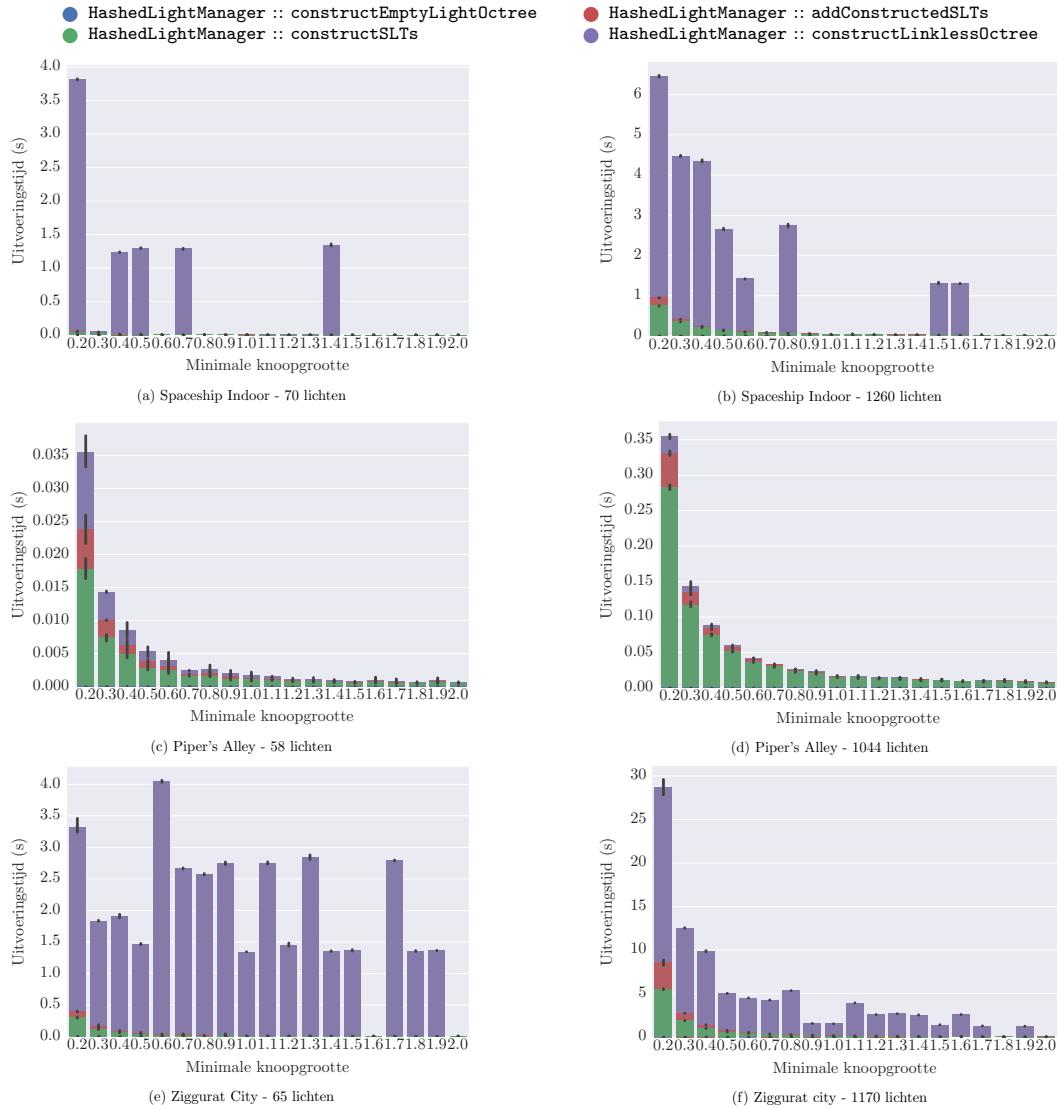
Figuur 7.17: Diepte van de octree als functie van de knoopgrootte.



Figuur 7.18: Constructietijd van de Hashed Shading datastructuren.

weergegeven. Voor elke gevulde bladknoop wordt een verzameling van relevante lichtindices toegevoegd aan de lichtindexlijst. Als laatste zijn de groottes van de hash- en offset-tabellen voor de verschillende lagen van de verbindingloze octree als functie van de knoopgrootte voor de grootste lichtconfiguraties per scène weergegeven in figuur 7.22

7.3. Resultaten



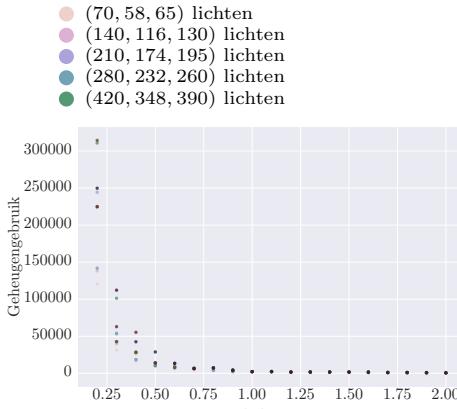
Figuur 7.19: Constructietijd per stap als functie van de knoopgrootte.

Bij alle lichtconfiguraties is een sterke toename in geheugengebruik en constructietijd waar te nemen wanneer de knoopgrootte kleiner wordt dan éénmaal de lichtradius. Dit is een direct gevolg van de toename van het aantal (minimale) knopen. Bij een kleinere knoopgrootte zal elk licht een groter aantal minimale knopen bedekken. De maximale grootte van het rooster bij een knoopgrootte van r voor een enkel licht kan gedefinieerd worden als:

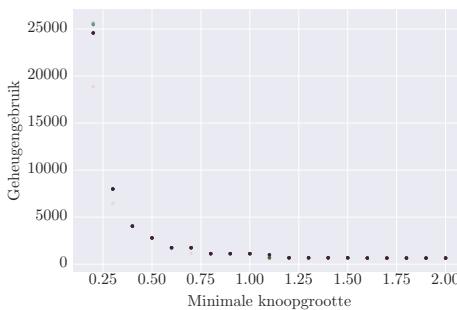
$$n = \left(\max \left(1, \frac{2l.radius}{r} \right) + 1 \right)^3$$

waar n het totaal aantal knopen in het rooster is. Uitgaande van het volume van een bol zal ongeveer 52% van de lichtvolumes gevuld zijn:

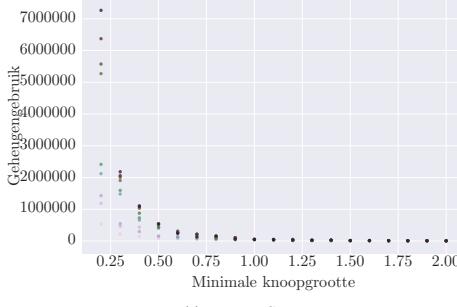
7. HASHED SHADING



(a) Spaceship Indoor

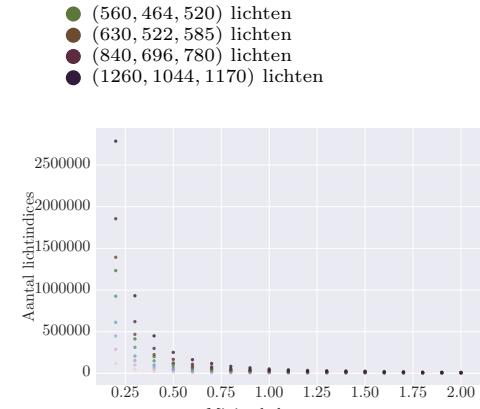


(b) Piper's Alley

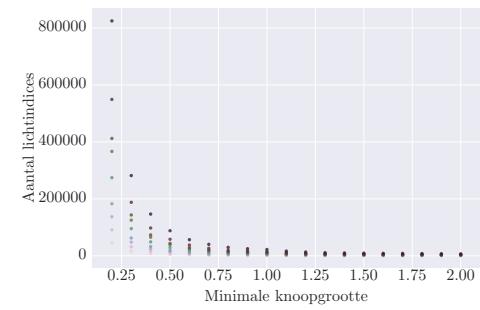


(c) Ziggurat City

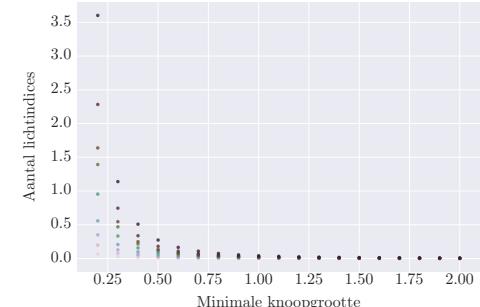
Figuur 7.20: Aantal pixels.



(a) Spaceship Indoor



(b) Piper's Alley

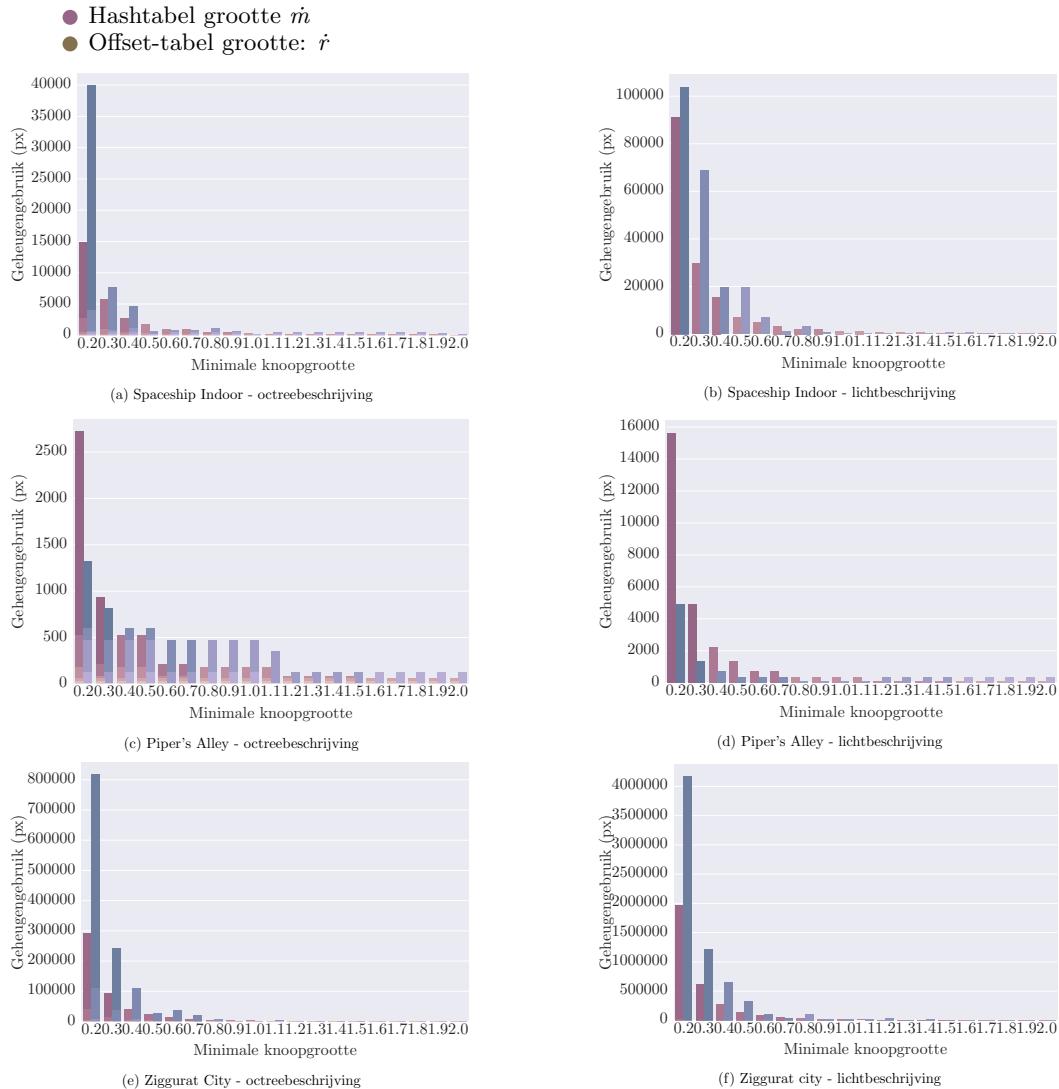


(c) Ziggurat City

Figuur 7.21: Aantal lichtindices.

$$n_{\text{licht}} \approx 0.52 \left(\max \left(1, \frac{2l.\text{radius}}{r} \right) + 1 \right)^3$$

Het aantal gevulde knopen in de diepste laag zal dus kubiek toenemen bij kleinere knoopgroottes.



Figuur 7.22: Geheugen gebruik per laag van de Hashed Shading datastructuren als functie van de knoopgrootte.

Geheugengebruik

De hiërarchische voorstelling van de lichtvolumes leidt ertoe dat de lege volumes efficiënt kunnen worden opgeslagen. Echter door de gedeeltelijk overlap van de lichten, bevindt een significant deel van de lichtinformatie zich in de diepste lagen van de octree. Dit is duidelijk waar te nemen in de grootte van de hashfuncties weergegeven in figuur 7.22. Het merendeel van de data-elementen bevindt zich in de twee diepste lagen van de octree. Doordat in de huidige implementatie met elke volle knoop een element in de lichtindexlijst is geassocieerd, leidt dit er tevens toe dat deze kubiek toeneemt bij waardes kleiner dan tweemaal de lichtradius. Dit is waarneembaar in figuur 7.21.

Doordat de knoopgrootte voor een groter aantal lagen in de octree leidt, heeft de knoopgrootte tevens een invloed op het geheugengebruik van de octreevoorstellinghashfuncties van de verbindingloze octree. Dit is zichtbaar in de grootte en het aantal lagen van de octreestructuur weergegeven in figuur 7.22.

Constructietijd

Enerzijds is de constructietijd afhankelijk van het geheugengebruik doordat een groter aantal knopen leidt tot meer berekeningen in het opstellen van de verbindingloze octree. Elke knoop per laag dient te worden gesorteerd. Vervolgens moet per groep botsende knopen een correcte offset-waarde gevonden worden.

Anderzijds leidt een kleinere knoopgrootte tot grotere roosters bij de constructie van de lichtoctree. Hierdoor neemt het aantal berekeningen per licht, en dus de gehele constructietijd toe. Doordat het aantal knopen in de diepste lagen kubiek toeneemt, neemt ook de constructietijd kubiek toe. De constructietijd is primair afhankelijk van de opbouw van de spatiale hashfuncties, zoals terug te zien is in figuur 7.19.

Lichtberekeningen en uitvoeringstijd

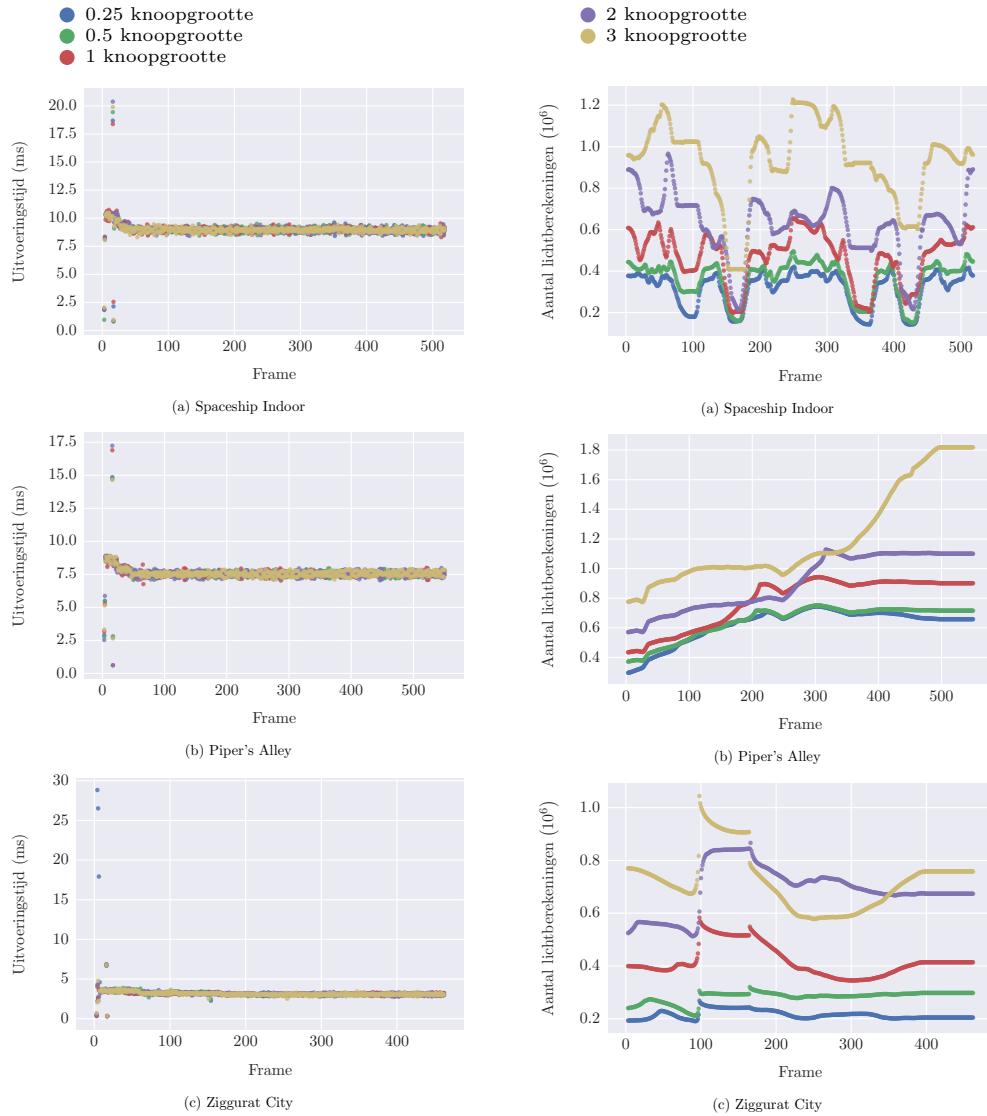
In figuren 7.23, 7.24 en 7.25, 7.26 zijn de uitvoeringstijd en aantal lichtberekeningen per frame voor respectievelijk een klein aantal lichten en een lage resolutie van 320×320 pixels, en een groot aantal lichten en een hoge resolutie van 2560×2560 pixels weergegeven. In figuren 7.27 en 7.28 zijn de gemiddelde uitvoeringstijd en het gemiddeld aantal lichtberekeningen per frame als functie van de minimale knoopgrootte weergegeven, bij een resolutie van 2560×2560 en een groot aantal lichten. Verder zijn de visualisaties van het aantal lichten bij verschillende knoopgroottes weergegeven in figuur 7.29.

Zowel bij een lagere resolutie en een kleine verzameling van lichten, als bij een hogere resolutie en een grotere verzameling van lichten leidt een kleinere minimale knoopgrootte tot minder lichtberekeningen per frame. Dit is een direct gevolg van de hogere nauwkeurigheid waarmee de lichten in de scèn voorgesteld worden. Hierdoor bevatten de bladknopen minder lichten, doordat niet relevante lichten niet meer overlappen. Deze lichten worden ook niet meer geëvalueerd in de berekening van pixels die in het volume van een dergelijke bladknoop vallen.

Wanneer de knoopgrootte groter wordt dan de diameter van de lichten leidt een grotere knoopgrootte niet noodzakelijk tot meer lichtberekeningen. Dit is een gevolg van hoe de knopen de ruimte opdelen. Het is mogelijk dat door de plaatsing van de grotere knopen, bepaalde knopen met minder lichten overlappen ondanks het groter volume dat wordt bestreken. Hierdoor neemt voor dergelijke knoopgroottes het aantal lichtberekeningen af, zoals zichtbaar is in figuur 7.26b.

De relatie tussen uitvoeringstijd en aantal lichtberekeningen is duidelijk zichtbaar in de grafieken van de uitvoeringen met een hogere resolutie en een aantal lichten. Het gedrag van de grafieken van het aantal lichtberekeningen is, in minder uitgesproken vorm, terug te zien in de uitvoeringstijd.

7.3. Resultaten

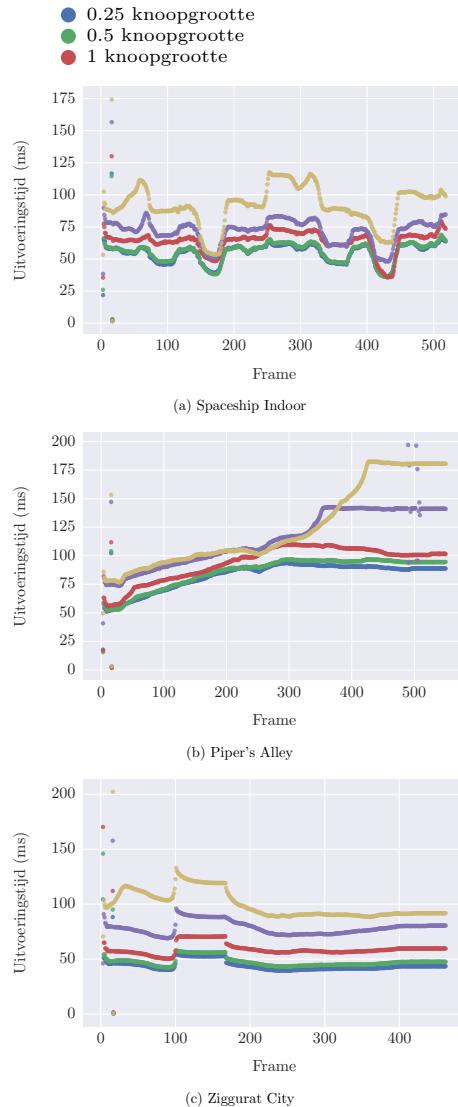


Figuur 7.23: Uitvoeringstijd: 320^2 pixels. Figuur 7.24: Lichtberekeningen: 320^2 pixels.

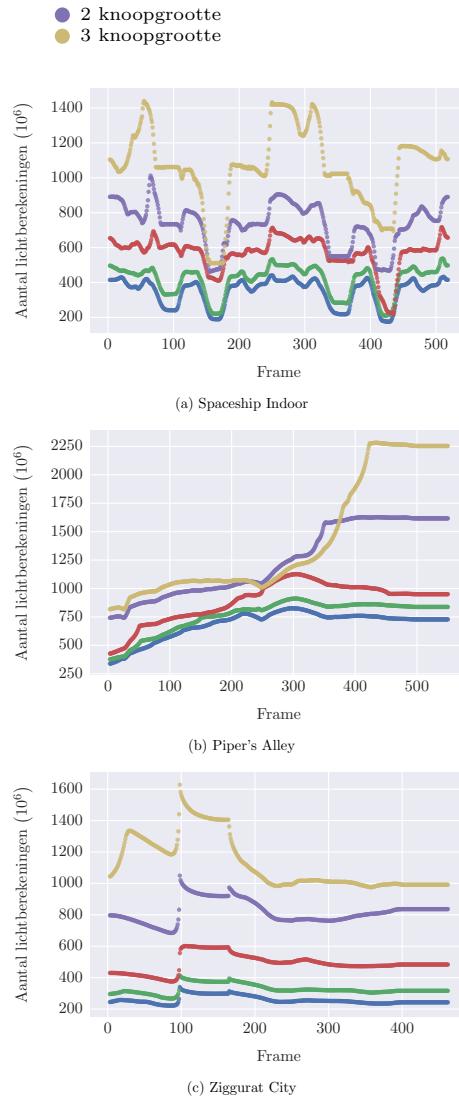
De lichtberekeningen zijn een belangrijk component van de gehele rendertijd, wat de gelijkenis in gedrag verklaard. Voordat de lichtberekening echter plaatsvindt, dient de lichtknoop waartoe een pixel behoort opgezocht te worden. Een kleinere knoopgrootte leidt tot meer lagen in de octree. Met elke laag die doorlopen dient te worden is een textuuvopvraag geassocieerd. Dit leidt tot een grotere uitvoeringstijd wanneer het aantal lagen van de octree toeneemt. Hiermee wordt de winst die verkregen wordt met de reductie van het aantal lichtberekeningen beperkt.

De invloed van het verschil in aantal lichtberekeningen op de uitvoeringstijd bij een kleine verzameling van lichten en een lage resolutie is minimaal. Hierdoor is dus ook de invloed van de knoopgrootte in dergelijke situaties, verwaarloosbaar. Deze

7. HASHED SHADING



(a) Spaceship Indoor
(b) Piper's Alley
(c) Ziggurat City

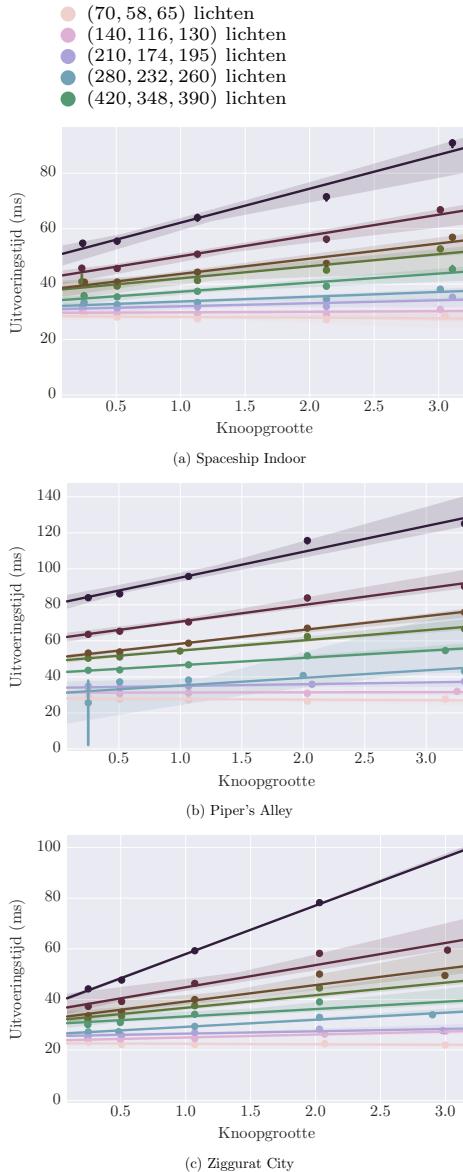


(a) Spaceship Indoor
(b) Piper's Alley
(c) Ziggurat City

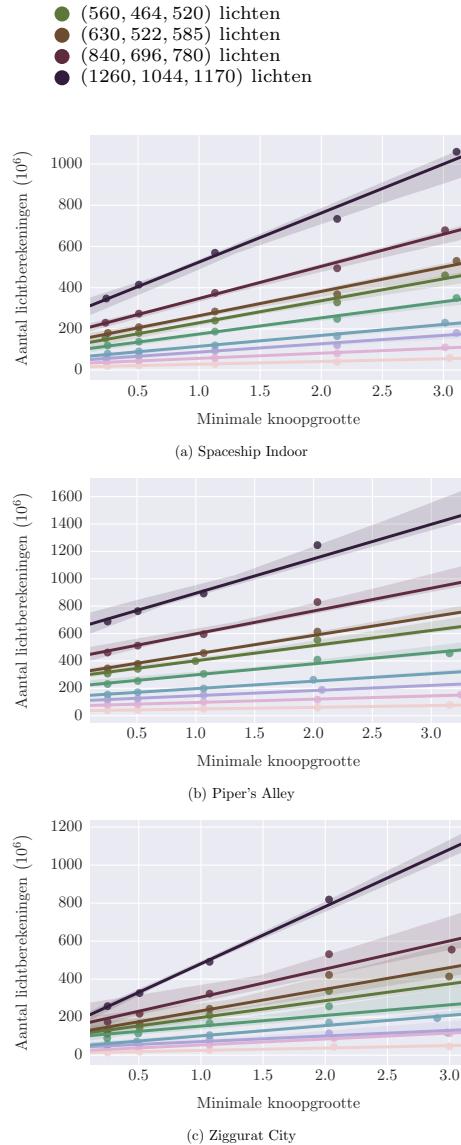
Figuur 7.25: Uitvoeringstijd: 2560^2 pixels. Figuur 7.26: Lichtberekeningen: 2560^2 pixels.

observaties zijn tevens terug te zien in figuur 7.27 en 7.28, waar de uitvoeringstijd en het gemiddeld aantal lichtberekeningen zijn uitgezet tegen de knoopgrootte voor verschillende aantallen lichten. Voor alle hoeveelheden lichten leidt een kleinere knoopgrootte tot minder lichtberekeningen. De invloed van de knoopgrootte op de reductie van het aantal lichtberekeningen is afhankelijk van het aantal lichten in de scène. Voor kleine hoeveelheden lichten is deze beperking minimaal, terwijl de invloed het grootst is wanneer het aantal lichten het grootst is. Dit leidt ertoe dat voor een klein aantal lichten de uitvoeringstijd zelfs afneemt, bij een grotere knoopgrootte. Hier valt uit af te leiden dat de uitvoeringstijd die de extra texturopvragingen vereisen, groter is dan de winst die wordt verkregen met de reductie in het aantal lichtberekeningen.

7.3. Resultaten



Figuur 7.27: Aantal pixels.



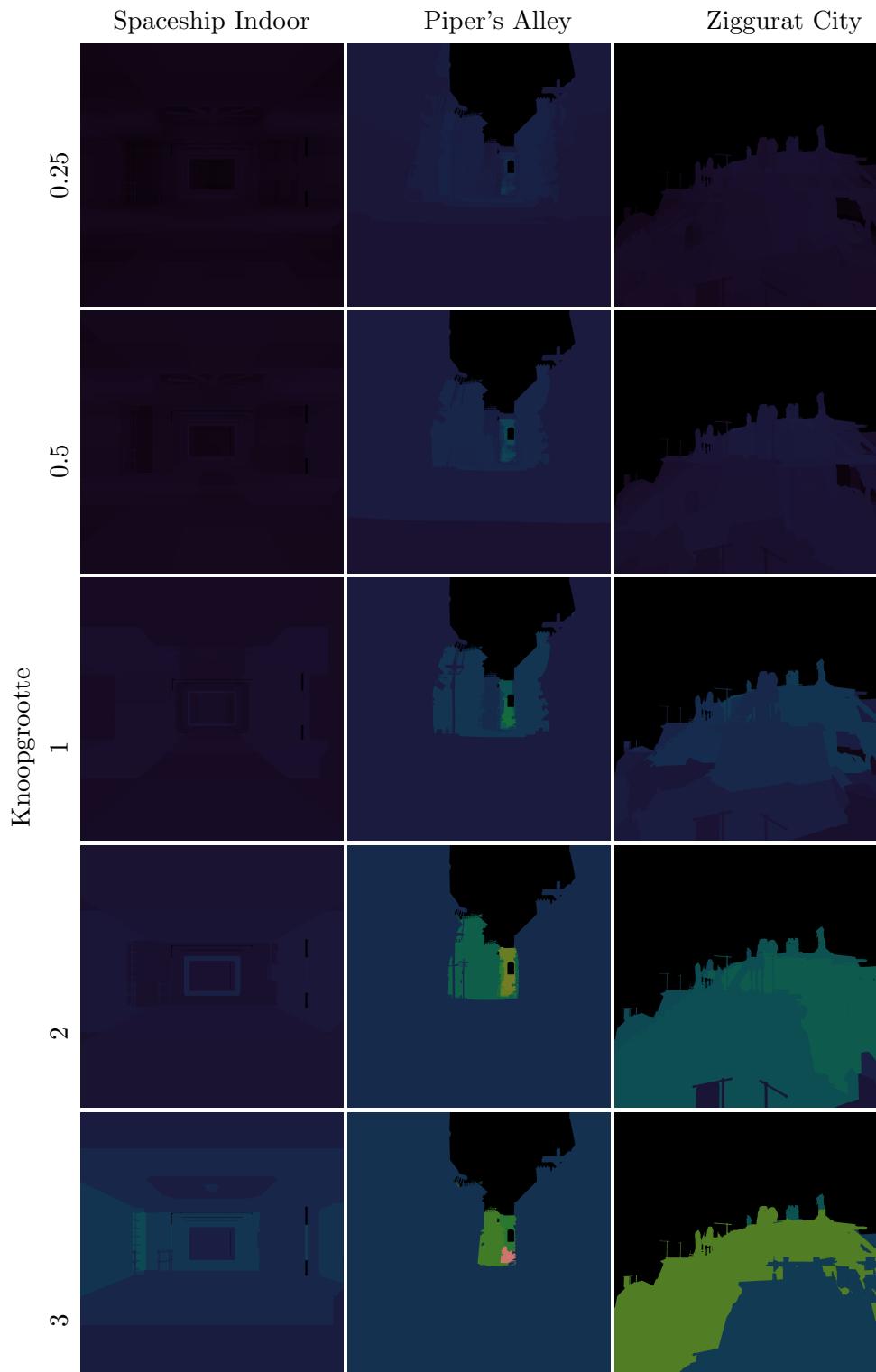
Figuur 7.28: Aantal lichtindices.

7.3.3 Begindiepte van de verbindingloze octree

Zoals vermeld in sectie 7.1.5 is het mogelijk om naast de knoopp groote de begindiepte in te stellen voor de verbindingloze octree. Hierbij komt een begindiepte van nul overeen met een beschrijving van elke laag binnen de verbindingloze octree, en een diepte van het aantal lagen minus één met een verbindingloze octree bestaande uit slechts een enkele laag.

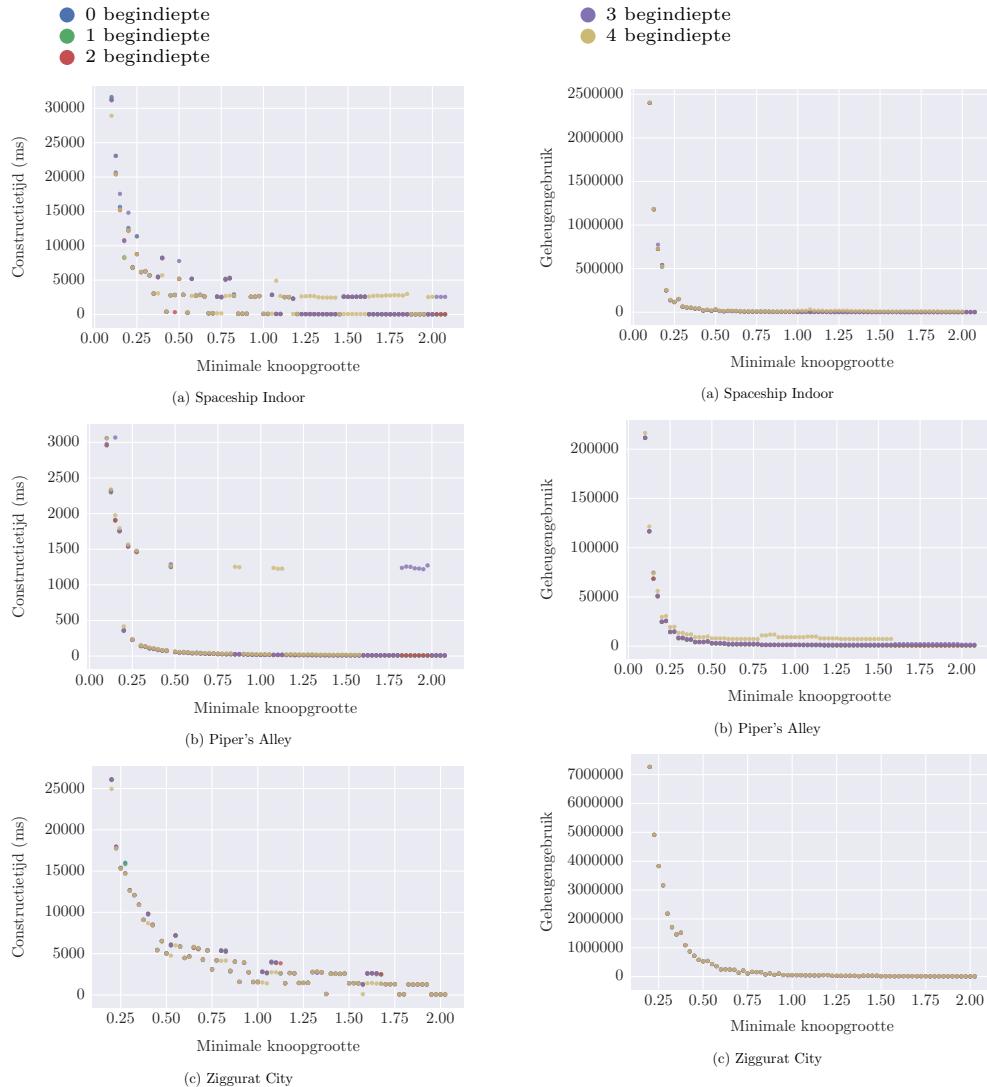
Het veranderen van deze waarde heeft geen invloed op het aantal lichtberekeningen dat wordt uitgevoerd, doordat ongeacht de begindiepte dezelfde verzameling van

7. HASHED SHADING



Figuur 7.29: Visualisatie van het aantal lichtberekeningen bij verschillende knoopp groottes relatief aan het naïeve Deferred Shading algoritme.

7.3. Resultaten



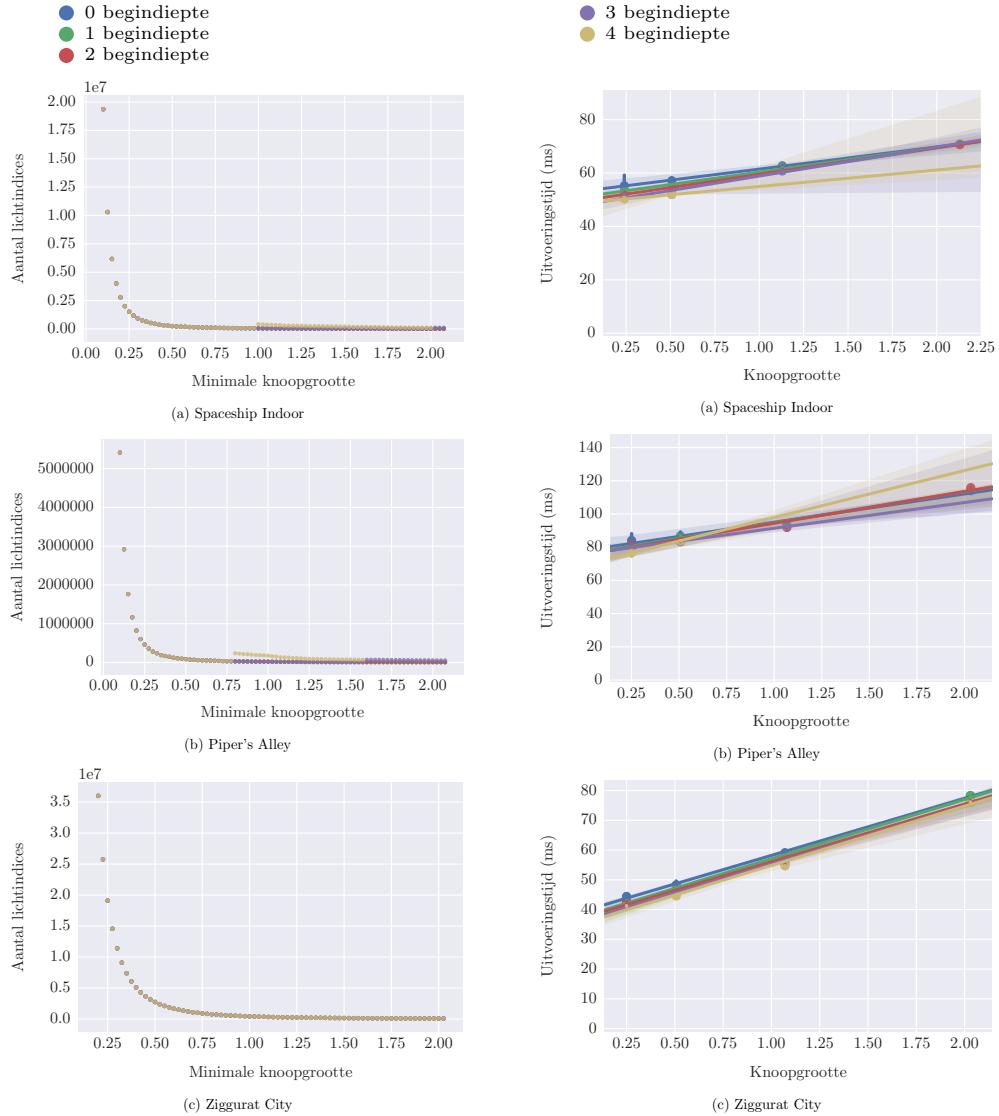
Figuur 7.30: Constructietijd als functie van de begin diepte.

Figuur 7.31: Aantal pixels als functie van de begin diepte.

lichten wordt opgehaald binnen de renderstap. Het kan echter wel van invloed zijn op de uitvoeringstijd, doordat bij een grotere begin diepte minder textuuraanspraken worden uitgevoerd.

Daarnaast zal het van invloed zijn op het geheugengebruik en de constructietijd. Het gebruik van kleine texturen, bijvoorbeeld van één pixel voor de eerste laag van de verbindingloze octree, leidt veelal tot onnodig geheugengebruik, door de overhead geassocieerd met het bijhouden van texturen. Hier staat tegenover dat een grotere begin diepte een toename van kleinere knopen vereist, die explicet bijgehouden dienen te worden. Dit leidt tot een toename in geheugengebruik. Daarnaast leidt een groter aantal knopen tot een grotere constructietijd, doordat elk van deze knopen toegekend

7. HASHED SHADING



Figuur 7.32: Aantal pixels als functie van de begindiepte.
Figuur 7.33: Aantal lichtindices als functie van de begindiepte.

dient te worden binnen de corresponderende spatiale hashfunctie.

Om deze aspecten te evalueren zijn de volgende figuren opgesteld. In figuur 7.30 is de constructietijd als functie van de knoopgrootte relatief aan de lichtradius weergegeven bij verschillende begindieptes voor de grootste gedefinieerde verzameling lichten per scène. In figuur 7.31 en 7.32 zijn respectievelijk het aantal pixels in de datastructuren en het aantal lichtindices als functie van de knoopgrootte voor verschillende begindieptes gegeven. Als laatste is de rendertijd als functie van de knoopgrootte voor de verschillende begindieptes weergegeven in figuur 7.33.

Uit alle figuren valt op te maken dat de geëvalueerde begindieptes geen significante

invloed hebben op de constructietijd, geheugenverbruik, en uitvoeringstijd. De afwezigheid van een effect op de constructietijd en het geheugenverbruik kan verklaard worden doordat een verandering van de begindiepte slechts invloed heeft op de eerste lagen van de verbindingloze octree. Uit de resultaten van het geheugenverbruik per laag laag van de verbindingloze octree, fig. 7.22 bleek dat het merendeel van de knopen zich in de diepere lagen bevindt. Een samenvoeging van de eerdere lagen zal geen invloed hebben op de verzameling knopen in de diepste lagen. Hierdoor zal het merendeel van de constructietijd en het geheugenverbruik niet veranderen.

De uitvoeringstijd voor grotere begindieptes is iets kleiner, wat een gevolg van de reductie van het aantal textuuraanspraken is. Echter deze verschillen zijn minimaal. Doordat de uitvoeringstijd primair afhankelijk is van het aantal lichtberekeningen heeft de begindiepte dus ook weinig invloed op de uitvoeringstijd.

Op basis hiervan kan geconcludeerd worden dat het voordelig is om de eerste lagen van de verbindingloze octree samen te voegen, doordat het gebruikte aantal texturen hierdoor gereduceerd wordt, zonder dat dit een negatieve invloed heeft op het geheugen en de uitvoeringstijd. Door Choi et. al. [CJC⁺09] wordt een begindiepte van drie aangehouden om de overhead die elke textuur met zich meebrengt te reduceren. Uit deze resultaten blijkt dat een begindiepte van drie geen vergroting van geheugenverbruik of uitvoeringstijd met zich meebrengt. Om deze reden zal een begindiepte van drie gebruikt worden in de vergelijking met Tiled en Clustered Shading.

7.3.4 Vergelijking met Tiled en Clustered Shading

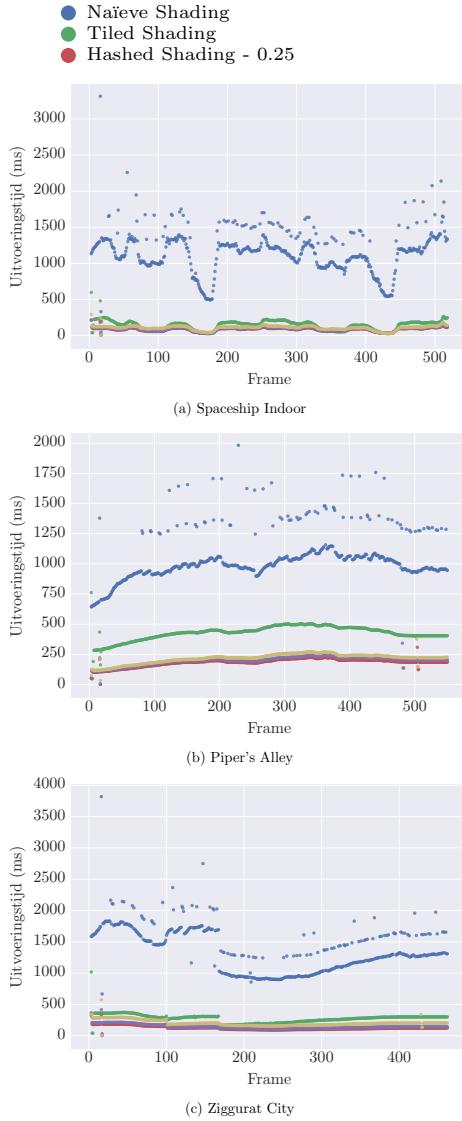
In de voorgaande secties zijn de verschillende parameters van de verbindingloze octree geëvalueerd. Als laatste zal nu gekeken worden naar de performantie van Hashed Shading in verhouding tot Tiled en Clustered Shading. Hiervoor wordt de uitvoeringstijd van Hashed Shading met die van Tiled Shading voor zowel de Forward als Deferred pijplijnen vergeleken. Tevens wordt het aantal lichtberekeningen van Hashed Shading vergeleken met het aantal lichtberekeningen van zowel Tiled en Clustered Shading voor de Deferred pijplijn. Om een duidelijk zicht op de performantie te verkrijgen, zijn de uitvoeringstijd en aantal lichtberekeningen voor gehele uitvoeringen als ook als functie van het aantal lichten en de resolutie verzameld. Deze zullen in de volgende secties verder gepresenteerd worden.

Zowel de data van Tiled Shading als Clustered Shading zijn verzameld met een tegelgrootte van 32×32 pixels. Voor Hashed Shading zijn knoopgroottes van 0.25, 0.5 en 1 maal de radius van de lichten in de scène gebruikt. De verbindingloze octree is bij elk van deze knoopgroottes geïnitialiseerd met een begindiepte van 3.

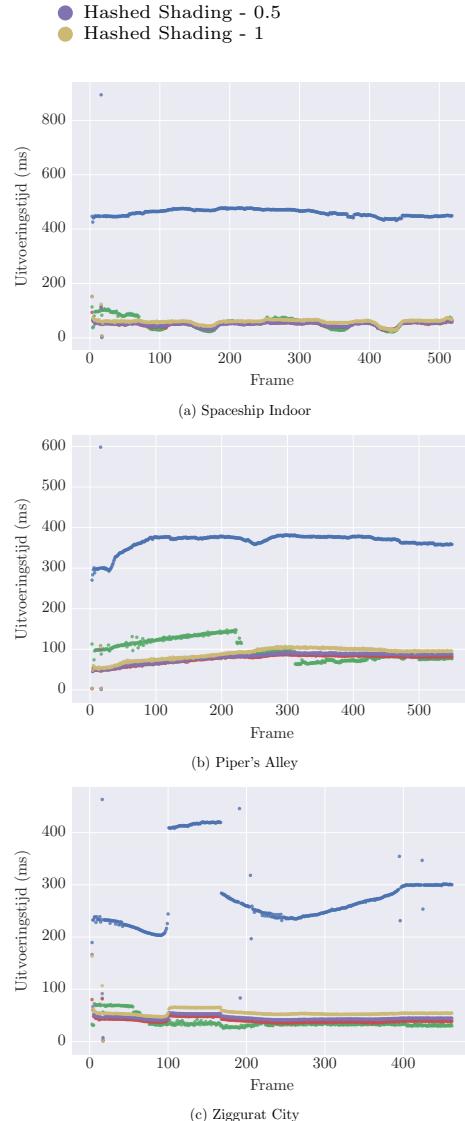
Frames

In figuur 7.34 en 7.35 zijn de uitvoeringstijden per frame weergegeven voor respectievelijk de Forward en Deferred pijplijn. In figuur 7.36 is het aantal lichtberekeningen per frame weergegeven.

7. HASHED SHADING



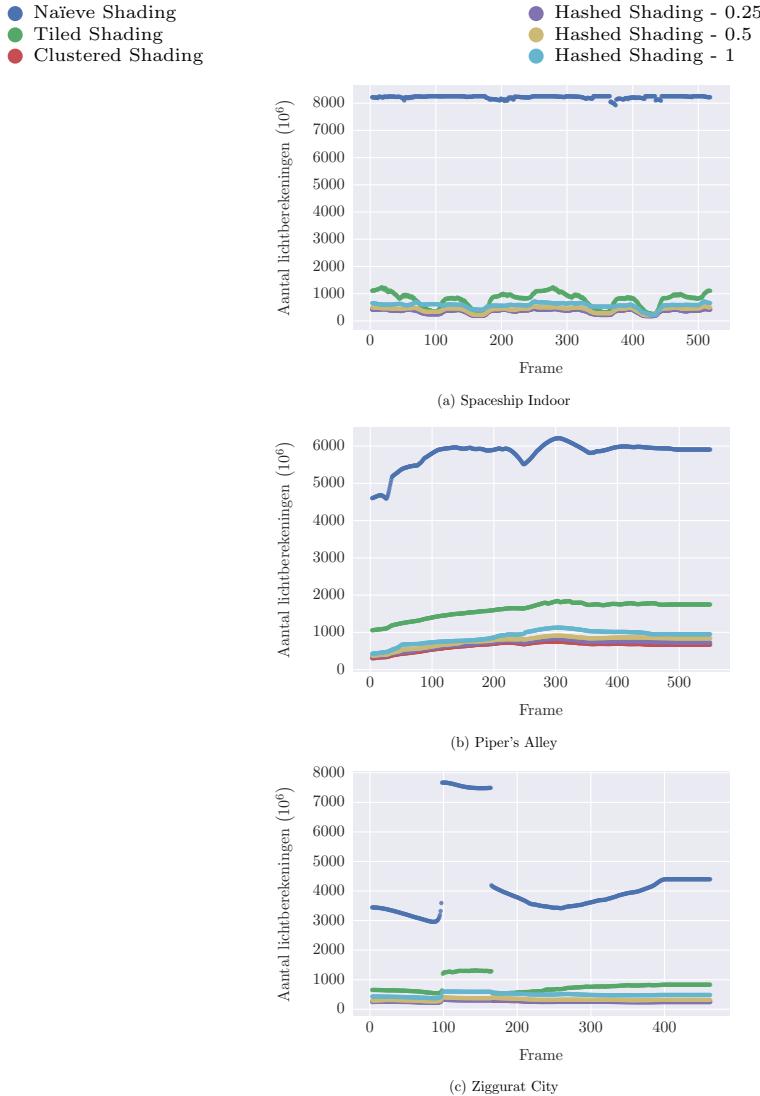
Figuur 7.34: De uitvoeringstijd per frame voor Forward Shading.



Figuur 7.35: De uitvoeringstijd per frame voor Deferred Shading.

Voor Forward Shading ligt de uitvoeringstijd van Hashed Shading onder de uitvoeringstijd van Tiled Shading. Dit verschil in uitvoeringstijd is verwaarloosbaar voor de Deferred pijplijn. Dit verschil is te verklaren aan de hand van de reductie van het aantal lichtberekeningen en de simpelheid van de lichtberekening. In het geval van Deferred Shading zal het verschil in aantal lichtberekeningen tussen Tiled en Hashed Shading kleiner zijn dan in Forward Shading, doordat bij Deferred Shading slechts één fragment per pixel gegenereerd wordt. De winst in uitvoeringstijd die behaald wordt door de reductie van het aantal lichtberekeningen wordt in het geval van Deferred Shading te niet gedaan door de extra complexiteit geassocieerd met

7.3. Resultaten

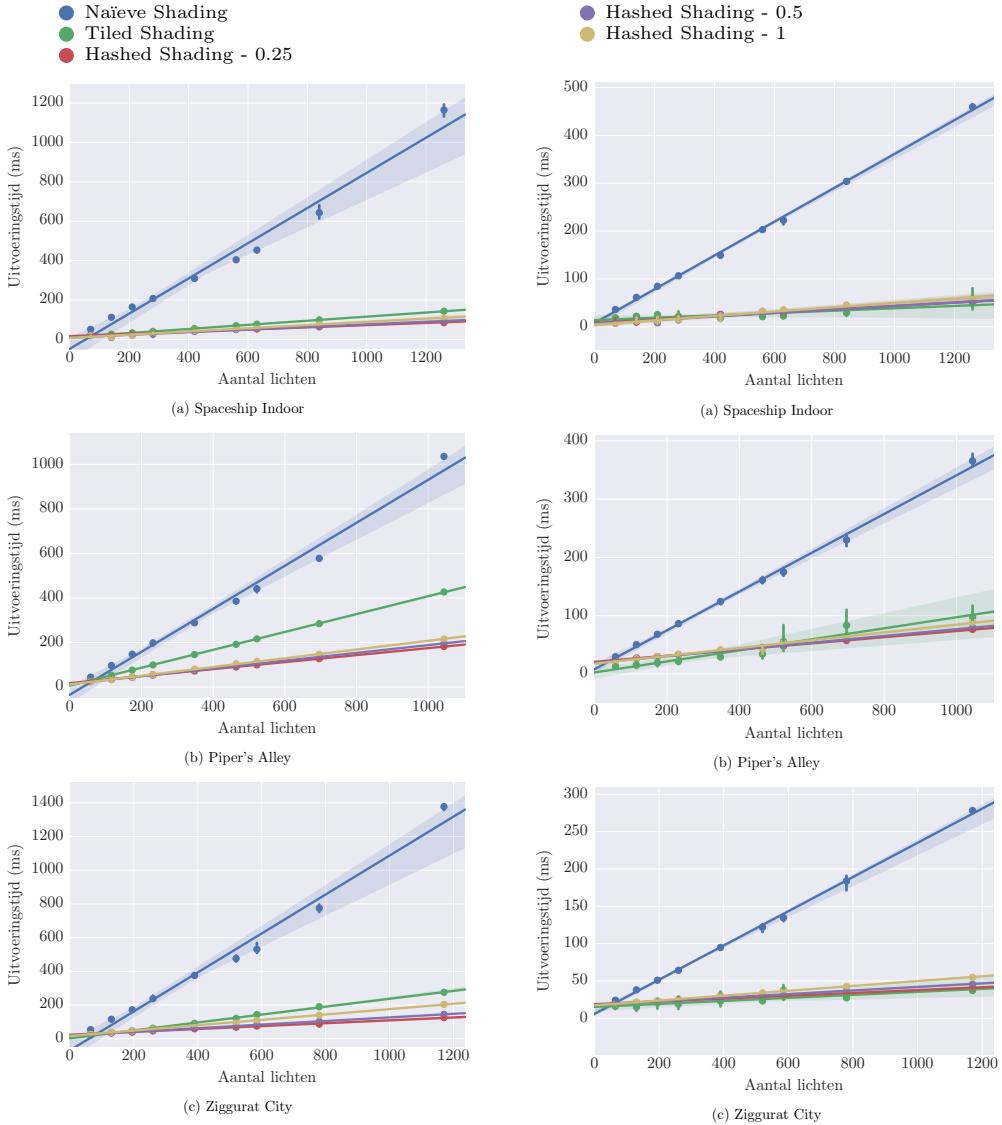


Figuur 7.36: Het aantal lichtberekeningen voor Deferred Shading.

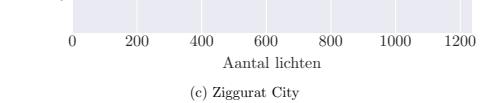
het ophalen van de relevante lichten bij Hashed Shading. Bij een complexere shader zal Hashed Shading waarschijnlijk beter presteren dan Tiled Shading. Doordat het aantal lichtberekeningen wordt beperkt per fragment, leidt dit ertoe dat de reductie in aantal lichtberekeningen wel een significante invloed heeft op de uitvoeringstijd bij Forward Shading.

De reductie in aantal lichtberekeningen is duidelijk terug te zien in figuur 7.36. Clustered Shading presteert slechts minimaal beter dan Hashed Shading met een kleine knoopgrootte. In figuur 7.36b is tevens duidelijk waar te nemen dat Hashed Shading, net als Clustered Shading de slechtst mogelijke situatie van Tiled Shading oplost.

7. HASHED SHADING



Figuur 7.37: De uitvoeringstijd per frame als functie van het aantal lichten voor Forward Shading.



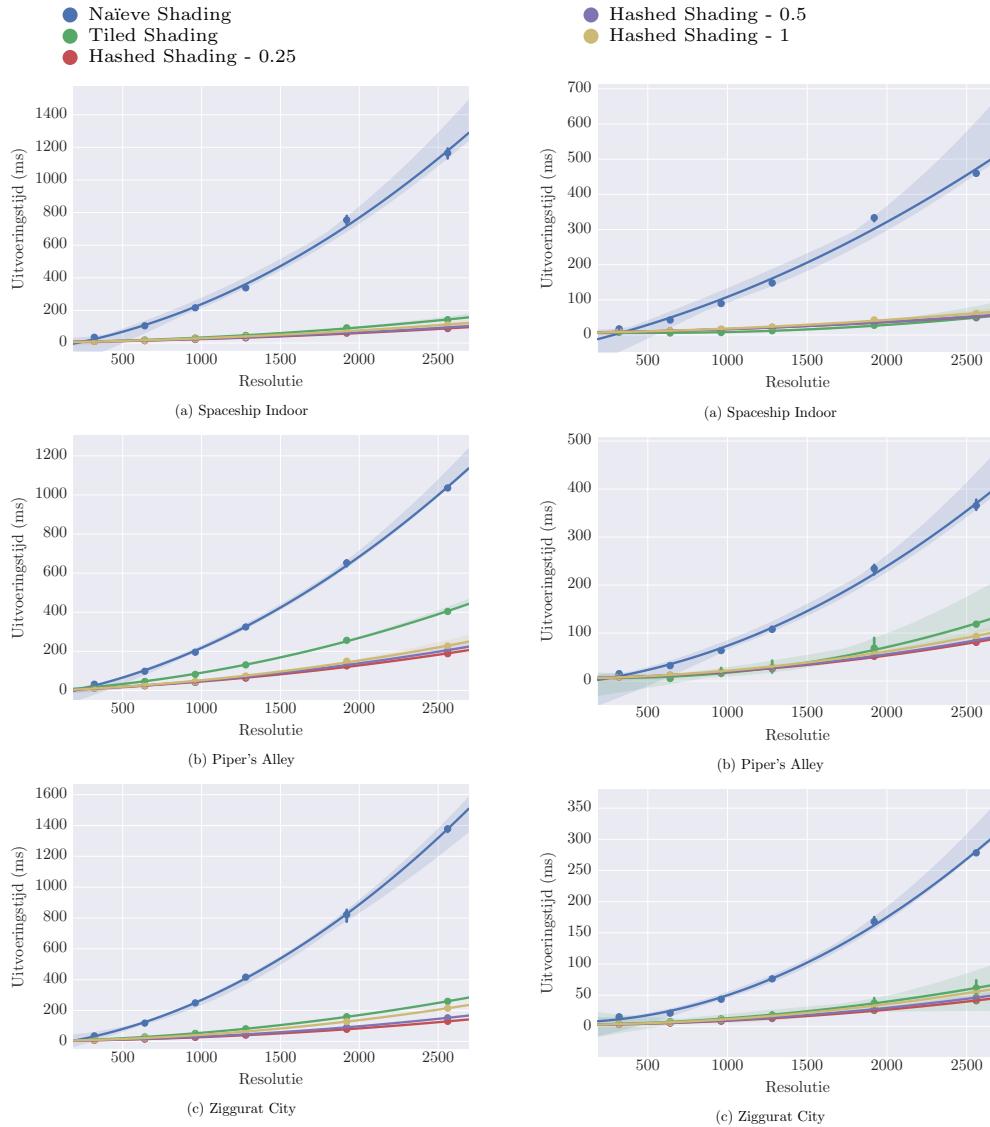
Figuur 7.38: De uitvoeringstijd per frame als functie van het aantal lichten voor Deferred Shading.

Lichten

In figuur 7.37 en 7.38 zijn de gemiddelde uitvoeringstijden per frame als functie van het aantal lichten weergegeven voor respectievelijk de Forward en Deferred pijplijn. In figuur 7.41 is het gemiddeld aantal lichtberekeningen per frame als functie van het aantal lichten weergegeven.

Hierin is duidelijk te zien dat Hashed Shading voor de gebruikte lichtberekening een vergelijkbare uitvoeringstijd heeft als Tiled Shading voor de Deferred Pijplijn. Binnen Forward Shading leidt Hashed Shading tot een factor twee verbetering.

7.3. Resultaten



Figuur 7.39: De uitvoeringstijd per frame als functie van de resolutie voor Forward Shading.

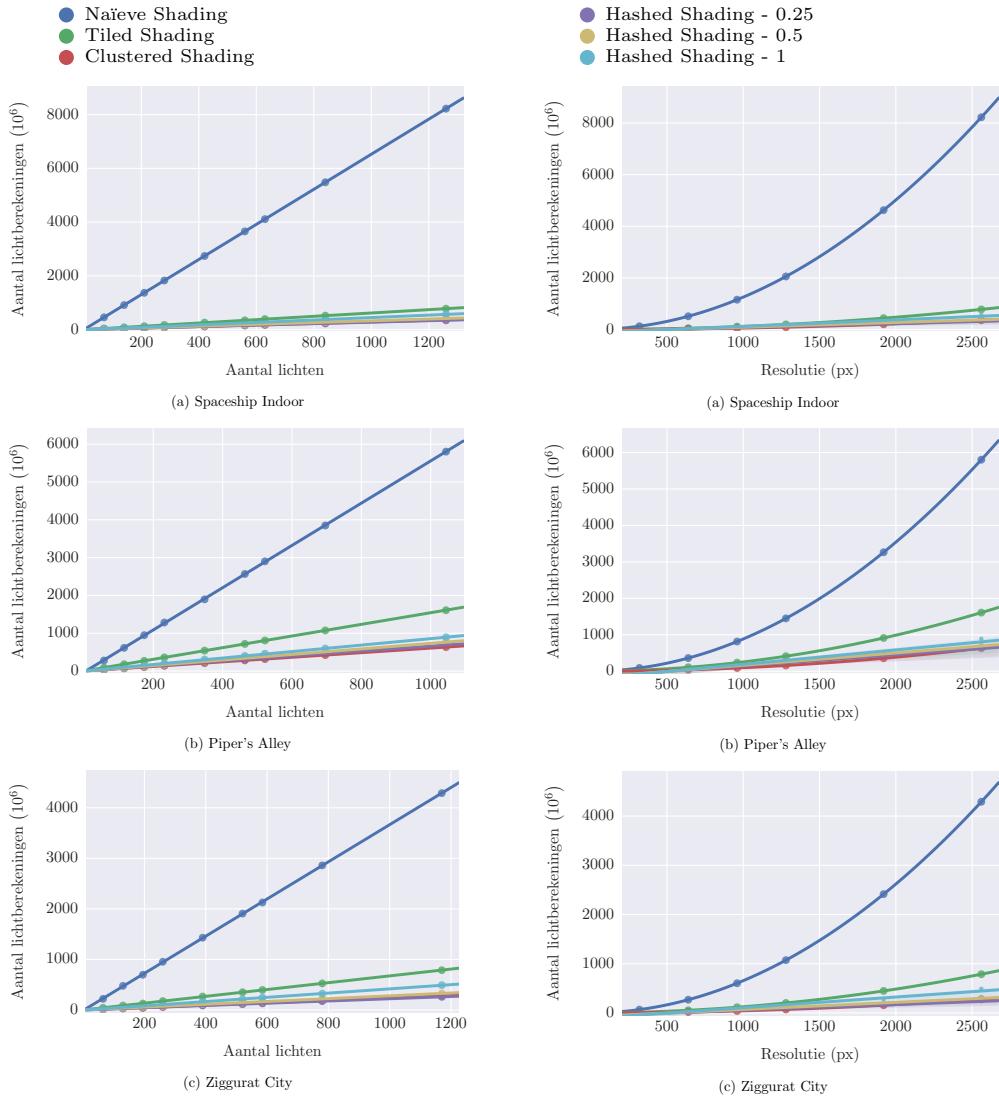
Figuur 7.40: De uitvoeringstijd per frame als functie van de resolutie voor Deferred Shading.

Wanneer gekeken wordt naar het aantal lichtberekeningen als functie van het aantal lichten, is te zien dat Hashed Shading vergelijkbaar presteert als Clustered Shading.

Resolutie

In figuur 7.39 en 7.40 zijn de gemiddelde uitvoeringstijden per frame als functie van de resolutie weergegeven voor respectievelijk de Forward en Deferred pijplijn. In figuur 7.42 is het gemiddeld aantal lichtberekeningen per frame als functie van de resolutie weergegeven.

7. HASHED SHADING



Figuur 7.41: Het aantal lichtberekeningen als functie van het aantal lichten.

Figuur 7.42: Het aantal lichtberekeningen als functie van de resolutie.

De uitvoeringsstijd en aantal lichtberekeningen als functie van de resolutie tonen dezelfde trend als de uitvoeringsstijd en aantal lichtberekeningen als functie van het aantal lichten. In het geval van Deferred Shading is de uitvoeringsstijd van Hashed Shading vergelijkbaar met die van Tiled Shading. Binnen Forward Shading stijgt de uitvoeringsstijd met ongeveer een factor twee langzamer. Het aantal lichtberekeningen van Hashed Shading stijgt minimaal langzamer dan dat van Clustered Shading.

Hierbij dient de opmerking geplaatst te worden dat het geheugengebruik van Hashed Shading onafhankelijk van de resolutie is, waar dit voor zowel Tiled als Clustered Shading direct gekoppeld is aan de resolutie door de keuze van de tegelgrootte.

7.4 Conclusie

In dit hoofdstuk is Hashed shading geïntroduceerd met als doel om tot een lichttoekenningsalgoritme te komen waarvoor de datastructuren niet per frame opgebouwd dienen te worden. Door het hergebruik van de datastructuren zou het mogelijk moeten zijn om een performantiewinst te verkrijgen ten opzichte van de lichttoekenningsalgoritmes, Tiled en Clustered Shading. Enerzijds dient het gepresenteerde algoritme (een gedeelte van) de opgestelde datastructuren her te gebruiken. Anderzijds dient het algoritme het aantal lichtberekeningen en dus de performantie met minimaal een zelfde factor te reduceren als Tiled en Clustered Shading.

Om de opgestelde datastructuren te kunnen hergebruiken tussen frames, dient de ruimteopdeling camera-onafhankelijk te zijn. Tevens dienen de datastructuren efficiënt voorgesteld te kunnen worden in het geheugen van de grafische kaart. Er is gekozen voor een verbindingloze octree datastructuur die de lichten in de scène beschrijft. Hiermee is het mogelijk om enerzijds de verzameling van relevante lichten te verkrijgen in slechts enkele texturopvragen en anderzijds niet elk minimale knoopvolume explicet voor te stellen door het gebruik van een hiërarchische structuur.

Er is aangetoond dat het mogelijk is om de lichten in een scène met behulp van een dergelijke structuur voor te stellen en op deze manier het aantal lichtberekeningen te beperken. Het geheugengebruik en de constructietijd is hierbij in de eerste plaats afhankelijk van de grootte van de kleinste knopen. Er is een kubiek verband waar te nemen tussen het geheugengebruik en constructietijd, en de grootte van de ribben van de kleinste bladknopen wanneer deze kleiner is dan de diameter van de gebruikte lichten. Het aantal lichtberekeningen en de uitvoeringstijd is lineair afhankelijk van de grootte van de ribben van de kleinste bladknopen.

In vergelijking tot Tiled en Clustered Shading blijkt dat Deferred Hashed Shading een vergelijkbare uitvoeringstijd heeft als Deferred Tiled Shading. Forward Hashed Shading is ongeveer een factor twee sneller dan Forward Tiled Shading. Hashed Shading vereist voor alle geëvalueerde knoopgroottes minder lichtberekeningen dan Tiled Shading. Clustered Shading vereist in de huidige implementatie nog steeds minder lichtberekeningen, echter het verschil tussen Clustered Shading en Hashed Shading met de kleinste geëvalueerde knoopgrootte is minimaal.

Op basis hiervan kan gesteld worden dat het inderdaad mogelijk is om met behulp van Hashed Shading een vergelijkbare performantie als Tiled en Clustered Shading te verkrijgen, zonder dat hiervoor de datastructuren opnieuw opgebouwd dienen te worden. De huidige implementatie is echter nog niet performanter dan Clustered Shading. Daarnaast is het geheugenverbruik van Hashed Shading groot, en zijn dynamische lichten nog niet ondersteund.

7.5 Discussie

7.5.1 Performantie

Op dit moment is het geheugenverbruik een beperkende factor voor de keuze van de knoopgrootte. Door het kubiek gedrag neemt het geheugenverbruik significant

toe wanneer een hogere precisie wordt vereist. De verwachting is dat bij een hogere precisie het aantal lichtberekeningen verder zal afnemen, doordat per pixel slechts relevante lichten worden berekend. Om dezelfde performantie te verkrijgen als Clustered Shading is het dus nodig om het geheugenprobleem op te lossen om zo een hogere precisie in de voorstelling van lichten in de scène te verkrijgen.

Een oplossing voor dit probleem zal niet worden geïmplementeerd binnen deze thesis. Echter een voorstel voor een mogelijke oplossing zal gedaan worden in het volgende hoofdstuk, als mogelijk verder onderzoek.

Hiernaast is nog significante optimalisatie mogelijk voor de constructietijd, door de opbouw van de octree te verplaatsen naar de grafische kaart. De opbouw van de octreevoorstelling van de enkele lichten kan triviaal worden geparallelliseerd door deze voor de verschillende lichten tegelijkertijd uit te voeren. Daarnaast kunnen de verschillende lagen van de verbindingloze octree onafhankelijk van elkaar worden opgesteld.

7.5.2 Dynamische Lichten

Doordat de datastructuren niet meer per frame worden opgesteld, worden dynamische lichten niet meer triviaal ondersteund. Dit is wel het geval voor Tiled en Clustered Shading. Voor elk van de lichten die een transformatie ondergaat dient te worden nagegaan of deze zodanig veranderd is dat de knopen waarop het licht invloed heeft anders zijn dan voor de transformatie. Is dit het geval dan dienen deze knopen aangepast te worden. Dit leidt tot extra rekentijd, die niet meegenomen is in de huidige evaluatie. De ondersteuning voor dynamische lichten zal niet verder geïmplementeerd worden binnen deze thesis, echter een theoretische beschrijving van de oplossing zal gegeven worden als voorstel voor verder onderzoek.

Hoofdstuk 8

Besluit

Het gestelde doel van deze thesis is om een lichttoekenningsalgoritme te ontwikkelen dat een deel van de datastructuren tussen frames hergebruikt om zo tot een betere performantie te komen dan verkregen wordt met moderne lichttoekenningsalgoritmes. Hiervoor is in het vorige hoofdstuk het Hashed Shading algoritme geïntroduceerd. Tevens zijn in voorgaande hoofdstukken Tiled en Clustered shading geëvalueerd, twee moderne lichttoekenningsalgoritmes.

In dit hoofdstuk zal aan de hand van de verkregen inzichten van de voorgaande hoofdstukken geëvalueerd worden of met Hashed Shading het gestelde doel bereikt is. Hierna zal ingegaan worden op enkele mogelijke oplossingen voor de problemen van Hashed Shading.

8.1 Vergelijking van de resultaten

De hypothese gesteld aan het begin van deze thesis, is dat het mogelijk moet zijn om een lichttoekenningsalgoritme te ontwerpen dat berekeningen van voorgaande frames hergebruikt om zo tot een betere performantie te komen dan huidige lichttoekenningsalgoritmes. De moderne lichttoekenningsalgoritmes die als referentie voor dit nieuwe lichttoekenningsalgoritme zijn genomen, zijn Tiled Shading [OA11] en Clustered Shading [OBA12].

Om te voldoen aan de gestelde hypothese dient het nieuw ontwikkelde lichttoekenningsalgoritme voor zowel de Forward als Deferred pijplijnen minimaal een vergelijkbare versnelling te verkrijgen als huidige lichttoekenningsalgoritmes opleveren. Om dit te evalueren is gekeken naar zowel de uitvoeringstijd per frame, als het aantal lichtberekeningen dat per frame dient te worden uitgevoerd. Deze waardes zijn geëvalueerd binnen drie scènes, die samen model staan voor de verschillende scènes die voorkomen in moderne games. Deze waardes zijn verzameld met verschillende aantallen lichten en bij verschillende resoluties.

Binnen Hashed Shading is er voor gekozen om de onderverdeling van de ruimte camera-onafhankelijk te maken. Hierdoor is het mogelijk om de opgestelde onderverdeling her te gebruiken tussen frames, indien deze niet verandert. Deze onderverdeling is geïmplementeerd met behulp van een octree. De hiërarchische structuur maakt

8. BESLUIT

het mogelijk om de onderverdeling van de ruimte waarin de lichten vallen, efficiënt en met hoge precisie voor te stellen. Om waardes efficiënt op te kunnen vragen op de grafische kaart is gebruik gemaakt van de verbindingloze octree voorstelling [CJC⁺09]. De resultaten van dit nieuwe lichttoekenningsalgoritme zijn vergeleken met die van Tiled en Clustered Shading.

Ongeacht het lichttoekenningsalgoritme is zowel het aantal lichtberekeningen als de uitvoeringstijd lineair afhankelijk van het aantal pixels, en het aantal lichten. De drie lichttoekenningsalgoritmes reduceren allen echter de factor van deze lineaire afhankelijkheid. Doordat de factor gereduceerd wordt, wordt tevens de variatie in uitvoeringstijd tussen frames gereduceerd. Hierdoor wordt een constantere framerate verkregen.

In elk van de testen presteerde de Deferred pijplijn beter dan de Forward Pijplijn. Dit is een direct gevolg van de ontkoppeling van de geometrische complexiteit van de shadingcomplexiteit. Deze verbetering in uitvoeringstijd is ten koste van het geheugengebruik. Voor Deferred Shading is het nodig om GBuffers op te stellen. Hierin wordt voor elk van de verschillende attributen die nodig zijn in de lichtberekening een textuur ter grootte van het zichtsveld bijgehouden. Een tweede nadeel is dat Deferred Shading geen transparantie ondersteunt. Hiervoor dient een aparte Forward pijplijn opgesteld te worden. Een voordeel van elk van de geëvalueerde lichttoekenningsalgoritmes is dat de opgestelde datastructuren zowel voor de Deferred pijplijn als de Forward pijplijn gebruikt kunnen worden, zonder dat deze opnieuw berekend dienen te worden.

Tiled Shading reduceert het aantal lichtberekeningen en de uitvoeringstijd ten opzichte van de naïeve implementatie. Dit wordt bereikt door het zichtsveld in tegels onder te verdelen, en voor elk van deze tegels de lichten die overlappen met deze tegel bij te houden. Hierbij wordt ongeveer een factor drie tot vier verbetering waargenomen in zowel de uitvoeringstijd als het aantal lichtberekeningen. De tegelgrootte leek hierbij geen significante invloed te hebben. In de situatie dat er een significante overlap van lichten in het zichtvenster was, presteert Tiled Shading significant slechter. Dit is het gevolg van een gebrek aan opdeling in de camera-z-as. Hierbij benadert het aantal lichtberekeningen in deze tegels dat van de naïve implementatie.

Clustered Shading verlaagt het aantal lichtberekeningen door bovenop de onderverdeling van het zichtvenster in tegels, ook het volume geassocieerd met deze tegels onder te verdelen met betrekking tot camera-z-as. Hiermee wordt het slechts mogelijke scenario van Tiled Shading voorkomen, en een betere lichttoekenning in het algemeen bereikt. Ook binnen Clustered Shading is geen significante invloed waar te nemen van de tegelgrootte op het aantal lichtberekening. Een bijkomende eis voor Clustered Shading is dat de dieptebuffers beschikbaar zijn op het moment dat de clusters opgesteld worden. Dit is geen probleem binnen de Deferred Shading pijplijn, maar vereist een extra pas om deze buffers te vullen binnen de Forward Shading pijplijn.

Hashed Shading benadert voor kleine knooppoottes het aantal lichtberekeningen van Clustered Shading. Daarnaast is de Forward Shading pijplijn implementatie ongeveer twee maal sneller dan de Forward Tiled Shading implementatie. De opgestelde datastructuren kunnen bij statische lichten direct hergebruikt worden

zonder dat een herberekening vereist is. Er zijn echter nog wel twee grote problemen te identificeren binnen de huidige implementatie van Hashed Shading. De eerste is het geheugengedrag van Hashed Shading. Bij kleinere knoopgroottes neemt het geheugengebruik kubiek toe. Hierdoor kan de huidige implementatie nog niet dezelfde precisie behalen als Clustered Shading. Het tweede probleem is het gebrek aan ondersteuning voor dynamische lichten. Doordat de datastructuren hergebruikt worden zullen deze niet automatisch veranderingen in lichten reflecteren. Doordat deze veranderingen expliciet moeten worden doorgevoerd, zal het ondersteunen van dynamische lichten altijd tot een toename in uitvoeringstijd leiden.

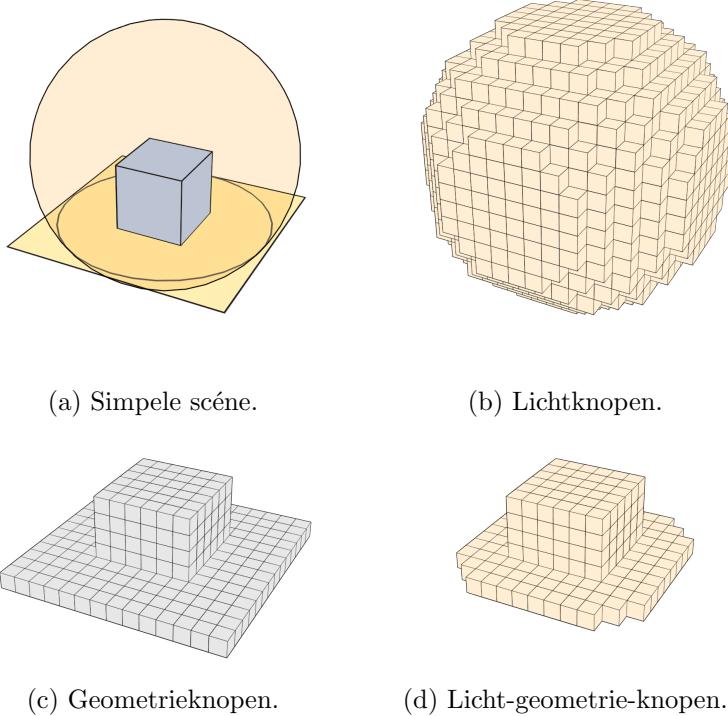
Op basis van deze resultaten kan gesteld worden dat het mogelijk is om een camera-onafhankelijk lichttoekenningsalgoritme op te stellen op basis van de verbindingloze octree [CJC⁺09]. Een dergelijke implementatie kan voor een vergelijkbare reductie van lichtberekeningen zorgen als Clustered Shading. Hiermee is het doel van deze thesis gerealiseerd. Echter de huidige implementatie van Hashed Shading is nog niet robuust genoeg om Clustered Shading te vervangen. Hiervoor zal eerst het geheugengebruik van Hashed Shading verlaagd moeten worden, en zullen dynamische lichten efficiënt ondersteund moeten worden. Hiervoor is verder onderzoek nodig. Enkele mogelijke oplossingen voor deze problemen zijn in de volgende sectie uitgewerkt.

8.2 Verder onderzoek

In de discussie van Hashed Shading zijn twee significante problemen geïdentificeerd, die zich voordoen in het Hashed Shading algoritme. Enerzijds is er de beperkende factor van het geheugenverbruik bij kleine knoopgroottes, die kubiek toeneemt voor kleinere knoopgroottes. Anderzijds is er de ondersteuning voor dynamische lichten. Dit wordt triviaal ondersteund in Tiled en Clustered Shading, maar vereist extra werk binnen Hashed Shading. Binnen deze sectie zullen eerst twee mogelijke verbeteringen ten opzichte van het geheugenverbruik voorgesteld worden, wat in verder onderzoek uitgewerkt en geïmplementeerd zou kunnen worden. Hierna zal gekeken worden hoe dynamische lichten ondersteund kunnen worden binnen Hashed Shading. Als laatste zullen nog enkele andere richtingen gegeven worden waar verder onderzoek naar gedaan kan worden.

8.2.1 Reduceren van geheugenverbruik doormiddel van geometrie

In de huidige implementatie van Hashed Shading wordt het volume van de gehele scène voorgesteld, zodat elk punt binnen deze scène opgevraagd kan worden. Een belangrijk inzicht van Clustered Shading is dat slechts punten liggend op de geometrie opgevraagd zullen worden. In Clustered Shading wordt dit gebruikt om slechts de zichtbare clusters voor te stellen en zo het geheugenverbruik te beperken. Hetzelfde inzicht kan gebruikt worden om het aantal knopen in de verbindingloze octree te reduceren. Alleen knopen die geometrie bevatten dienen voorgesteld te worden binnen de verbindingloze octree.

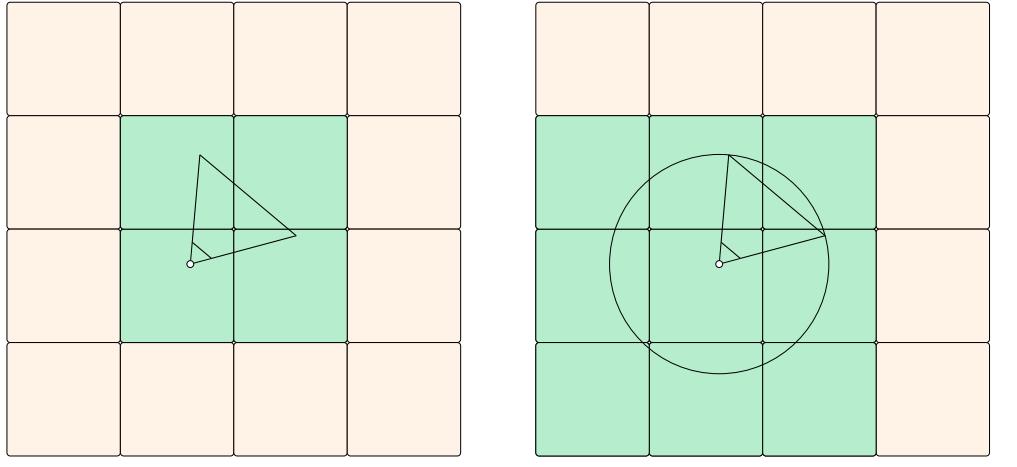


Figuur 8.1: Reductie van het aantal lichtknopen met behulp van de geometrie.

Om het geheugenverbruik op de grafische kaart te reduceren, dient dezelfde octree voorstelling gemaakt te worden van de geometrie zoals gedaan is voor de lichten. Vervolgens kunnen alle lichtoctreeknopen die niet tevens geometrie bevatten buiten beschouwing gelaten worden, gezien deze nooit opgevraagd zullen worden. Dit is geïllustreerd in Figuur 8.1 voor een simpele scène bestaande uit twee objecten en een enkel licht.

Een bijkomend voordeel van een dergelijke implementatie is dat een kleinere knoopgrootte niet alleen leidt tot een reductie in het aantal lichtberekeningen gedurende de uitvoering, maar tot een nauwkeurige voorstelling van de geometrie. Hierdoor kan een groter gedeelte van de lege ruimte buiten beschouwing gelaten worden tijdens het opstellen van de verbindingloze octree.

Als laatste optimalisatie is het mogelijk om de hiërarchische structuur van de octree uit te buiten, om zo het aantal knopen verder te reduceren. Gezien volumes die geen geometrie data bevatten nooit opgevraagd zullen worden, maakt de verzameling lichten geassocieerd met een dergelijke knoop niet uit. Dit betekent dat voor de opbouw van de verbindingloze octree gesteld kan worden dat de verzameling van lichten geassocieerd met een dergelijke knoop gelijk is aan de omliggende knopen, zonder dat dit visuele artefacten introduceert. Op basis van dit inzicht kan de eis voor het samennemen van acht bladknopen tot een enkele bladknoop in een hoger liggende laag versoept worden. In de huidige implementatie geldt dat alle acht knopen exact



(a) Opdeling met behulp van 2^3 stukken. (b) Opdeling met behulp van 3^3 stukken.

Figuur 8.2: Opdeling van de scèneruimte.

dezelfde verzameling van lichten dienen te hebben, voordat deze samengenomen kunnen worden. In de voorgestelde implementatie kan dit worden versoepeld tot de eis dat de deelverzameling van de acht knopen die geometrie bevat, de zelfde verzamelingen van lichten bevat. In dit geval zullen meer knopen samengenomen kunnen worden, dan dat het geval is voor de huidige implementatie. Dit leidt ertoe dat er in totaal minder knopen bijgehouden dienen te worden.

Deze combinatie van optimalisaties zou tot een sterke reductie in knopen moeten leiden, waardoor de constructietijd en het geheugenverbruik afneemt. Deze reductie in geheugenverbruik kan gebruikt worden om kleinere knoopgroottes te gebruiken, waardoor de performantie van Hashed Shading, de performantie van Clustered Shading nog beter zou moeten benaderen.

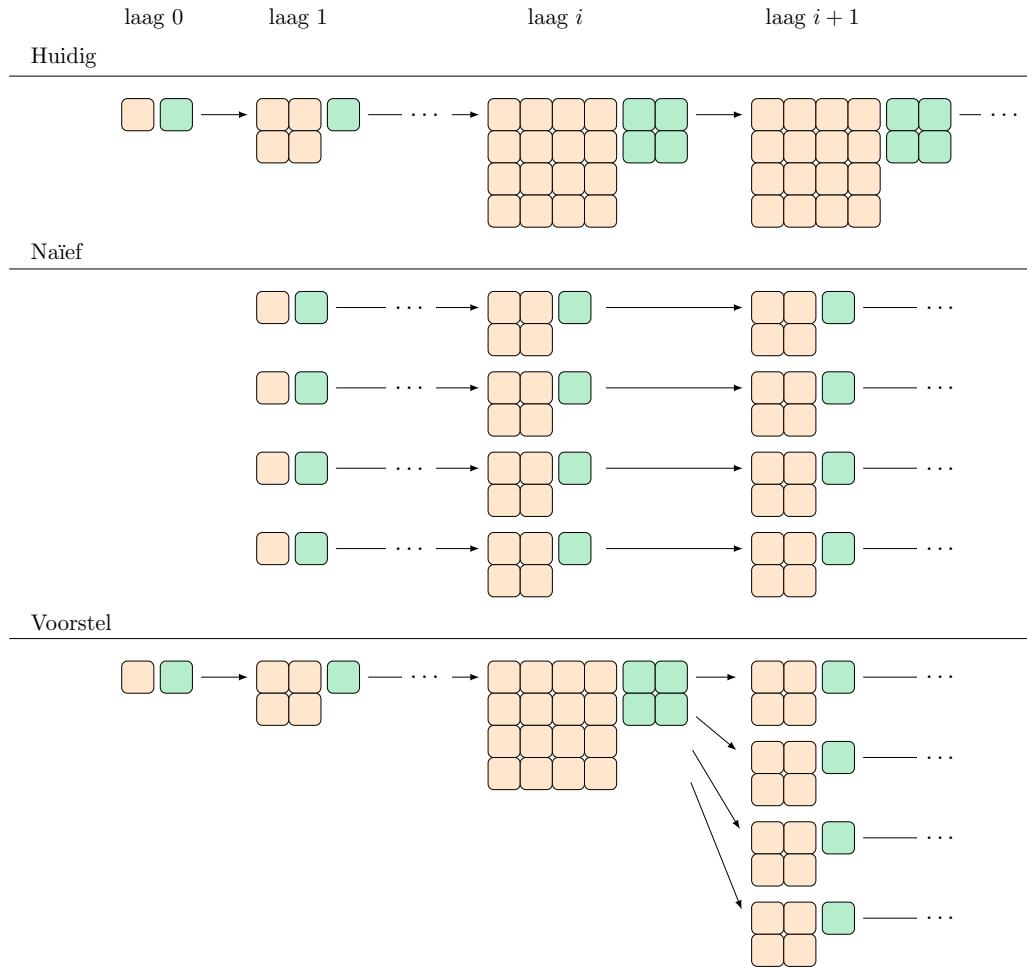
8.2.2 Reduceren van geheugengebruik doormiddel van opdeling van de scène-ruimte

Een tweede inzicht waar Clustered Shading gebruik van maakt is dat fragmenten slechts gegenereerd worden binnen het zichtfrustum. Hierdoor hoeft slechts de beschrijving van de relevante lichten voor deze deelverzameling van de ruimte aanwezig te zijn in het geheugen van de grafische kaart. Binnen Clustered Shading wordt slechts een beschrijving van het zichtfrustum gemaakt. Dit introduceert echter de camera-afhankelijkheid die Hashed Shading verhelpt.

Om toch het geheugenverbruik te reduceren kan de scène-ruimte in stukken worden onderverdeeld, zodanig dat slechts een kleine verzameling van stukken aanwezig hoeft te zijn in het geheugen. Indien deze verzameling een kleiner volume beschrijft dan de gehele scène, leidt dit tot een afname van geheugenverbruik.

In de meest simpele implementatie zijn hier 2^3 stukken nodig, indien de lengte van de stukken zodanig wordt gekozen dat de omsluitende kubus (Axis-Aligned Bounding Box) van het zichtfrustum binnen het volume van de 8 stukken valt, zoals weergegeven

8. BESLUIT



Figuur 8.3: Opdelingstrategie van diepere lagen van de verbindingloze octree.

in Figuur 8.2a. In veel applicaties zullen rotaties van de camera veel voorkomen. Dit zal er toe leiden dat de camerarotaties verantwoordelijk zullen zijn voor de meeste uitwisseling van texturen. Indien een verzameling van 3^3 stukken gebruikt wordt, zoals weergegeven in Figuur 8.2b, zullen slechts translaties van het camerapunt leiden tot veranderingen in de texturen. Hiermee kan de geheugenbandbreedte verlaagd worden ten koste van het geheugenverbruik. Als laatste kan de uitwisseling van texturen beperkt worden door de posities van inladen uit te spreiden. Hierdoor zullen er geen posities binnen de scène zijn waar een kleine transformatie constant tot herladen van texturen leidt.

Indien elk van deze stukken wordt voorgesteld als een aparte verbindingloze octree leidt dit tot een groot aantal, veelal kleine, texturen. Dit is niet ideaal, gezien de geheugenoverhead die geassocieerd is met elke textuur, en door de vaak beperkte hoeveelheid beschikbare texturen. Door terug te kijken naar het geheugenverbruik van Hashed Shading kan hier een oplossing voor gevonden worden. Uit het geheugenverbruik blijkt dat de eerste lagen relatief klein zijn, en dus weinig geheugen

verbruiken, en dat het merendeel van het geheugen wordt verbruikt door de diepste lagen van de verbindingloze octree. Gezien alle knopen in een laag expliciet in dezelfde textuur worden opgeslagen, is er geen horde om slechts de diepste lagen onder te verdelen en apart in te laden. Hierdoor hoeft de huidige implementatie slechts minimaal aangepast te worden, en blijft het textuurverbruik kleiner. Deze oplossing is visueel weergegeven in Figuur 8.3. Om een dergelijke structuur mogelijk te maken dient voor deze diepste knopen een extra referentie bijgehouden te worden, zodat, wanneer de opsplitsing van lagen in verschillende stukken wordt bereikt, er nagegaan kan worden, in welke textuur de uiteindelijk knoopinformatie gevonden kan worden.

Een laatste optimalisatie die op basis van de onderverdeling van de scène-ruimte kan worden toegevoegd is het gebruik van detailniveaus. Onder normale omstandigheden zal het merendeel van de pixels geometrie beschrijven die dicht bij de camera ligt. Dit is een gevolg van perspectief, waar objecten dicht bij de camera groter zijn, en dus veelal meer ruimte op het zichtvenster zullen innemen. Indien de gehele ruimte met dezelfde precisie wordt beschreven betekent dit dat de ruimte ver van de camera meer geheugengebruikt relatief aan de performantie die hier mee gewonnen wordt. Om deze reden is het voordelig om punten dichtbij de camera met een hogere resolutie te beschrijven, dan punten ver van de camera. Dergelijke technieken maken dan dus gebruik van zogenoemde detailniveaus (Level of Detail) [Lue03]. Octree datastructuren kunnen dergelijke detailniveaus op een natuurlijke manier ondersteunen, door gebruik te maken van de hiërarchische structuur. Indien een scène wordt opgedeeld, kunnen stukken verder van de camera voorgesteld worden met een kleinere diepte, en stukken dichtbij de camera met een hogere diepte. Hierdoor wordt de resolutie gevarieerd, waardoor het geheugenverbruik afneemt en tegelijkertijd, indien het merendeel van het zichtvenster objecten dichtbij de camera weergeeft, het totaal aantal lichtberekening verder wordt gereduceerd.

De verbindingloze octree leent zich zeer goed voor een dergelijke strategie. Doordat alle knopen in een laag in dezelfde textuur worden opgeslagen, kunnen voor bepaalde dieptes meerdere beschrijvingen van lagen worden gedefinieerd. Hierbij beschrijft één textuur dan de geassocieerde ruimte met bladknopen, alsof het de maximale diepte is, en een andere textuur de ruimte met zowel bladknopen als takknopen, zodanig dat er nog diepere lagen bestaan. Afhankelijk van de afstand waarop een stuk zich van de camera bevindt, kan gekozen worden om de maximale diepte kleiner of groter te maken, zonder dat dit leidt tot het bijhouden van meerdere verbindingloze octrees.

Deze combinatie van optimalisaties zou, vooral voor grote scènes, tot een afname in geheugenverbruik moeten leiden. Dit gaat echter ten koste van een toename in geheugenbandbreedte en berekeningstijd.

8.2.3 Ondersteuning van dynamische lichten

Het gebrek aan ondersteuning voor dynamische lichten is het tweede grote probleem van de huidige implementatie van Hashed Shading. Doordat binnen Tiled en Clustered Shading alle datastructuren per frame worden opgebouwd, worden veranderingen impliciet meegenomen. Dit is niet het geval bij Hashed Shading, waar de datastructu-

8. BESLUIT

ren worden hergebruikt. Om deze reden is het nodig om de datastructuren explicet aan te passen, zodat deze de veranderingen van de lichten worden gereflecteerd. In deze sectie zal een update-strategie uitgewerkt worden voor de puntlichten binnen Hashed Shading.

Voordat ingegaan wordt op deze update-strategie zal eerst herhaald worden hoe de datastructuren in de huidige implementatie worden opgebouwd. Om een verbindingloze octree op te stellen worden de volgende stappen uitgevoerd:

1. Per licht wordt een octreevoorstelling opgesteld.
2. Alle lichtoctree worden samengevoegd tot een scène-octree.
3. Per laag van de scène-octree worden de relevante knopen opgehaald. Voor deze knopen wordt een spatiale hashfunctie opgesteld.
4. De texturen geassocieerd met de spatiale hashfuncties worden in het geheugen ingeladen.

Uit de resultaten van de opbouw van de verbindingloze octree, sectie 7.3.2, kan afgeleid worden dat het grofweg opnieuw uitvoeren van deze stappen per frame geen acceptabele framerate zal opleveren. Er is vastgesteld dat de tijdsbepalende stap in dit algoritme de (her)opbouw van de spatiale hashfuncties is. De leiddraad voor de gepresenteerde update-strategie is dan ook om deze opbouw zoveel mogelijk te beperken.

De verwachting is dat een dergelijke herberekening niet nodig is voor elke frame. Deze aanname is gemaakt op basis van de observatie dat transformaties van lichten tussen frames veelal klein en lokaal van aard zijn. Hierdoor zullen de meeste transformaties niet een zodanige invloed hebben dat een volledige heropbouw nodig is.

Een spatiale hashfunctie, corresponderende met een laag, dient opnieuw opgebouwd te worden wanneer een knoop moet worden toegevoegd die botst met een reeds opgeslagen knoop. Gegeven een reeds gedefinieerde spatiale hashfunctie H en een knoop \mathbf{k} , dan is dit het geval als

$$H[h_0(\mathbf{k}) + \Phi[h_1(\mathbf{k})]] \neq \emptyset$$

De positie $h_0(\mathbf{k}) + \Phi[h_1(\mathbf{k})]$ is in dit geval reeds gevuld. In alle andere gevallen kan de opgestelde spatiale hashfunctie gemodificeerd worden door de geassocieerde textuur aan te passen.

Met behulp van deze observatie kan elk mogelijke aanpassing van de octree gemodelleerd worden. Het toevoegen van elementen is weergegeven in Tabel 8.1. Het verwijderen van elementen is weergegeven in Tabel 8.2. In deze tabellen zijn de volgende operaties te onderscheiden.

Toevoegen van een knoop \mathbf{k} in de lichtbeschrijving spatiale hashfunctie H

Eerst dient gekeken te worden of geldt:

$$H[h_0(\mathbf{k}) + \Phi[h_1(\mathbf{k})]] \neq \emptyset$$

8.2. Verder onderzoek

	Toevoegen licht op huidige diepte	Toevoegen licht op een lagere diepte
Lege bladknoop	<ul style="list-style-type: none"> • Toevoegen nieuwe knoop: data hashfunctie • Aanpassen: octreebeschrijving 	<ul style="list-style-type: none"> • Toevoegen opgesplitste bladknopen: data hashfunctie volgende laag • Toevoegen nieuwe knoop: octree hashfunctie volgende laag • Aanpassen: octreebeschrijving
Gevulde bladknoop	<ul style="list-style-type: none"> • Aanpassen knoop: data hashfunctie 	<ul style="list-style-type: none"> • Verwijderen knoop: data hashfunctie • Toevoegen opgesplitste bladknopen: data hashfunctie volgende laag • Toevoegen nieuwe knoop: octree hashfunctie volgende laag • Aanpassen: octreebeschrijving
Takknoop	<ul style="list-style-type: none"> • Verdere afdaling 	<ul style="list-style-type: none"> • Verdere afdaling

Tabel 8.1: De mogelijke situaties wanneer een lichtindex wordt toegevoegd aan een octreeknooppunt.

	Verwijderen licht op huidige diepte	Verwijderen licht op een lagere diepte
Bladknoop met een enkel licht	<ul style="list-style-type: none"> • Aanpassen: octreebeschrijving • Verwijderen knoop: data hashfunctie 	-
Bladknoop met twee of meer lichten	<ul style="list-style-type: none"> • Aanpassen knoop: data hashfunctie 	-
Takknoop	-	<ul style="list-style-type: none"> • Verdere afdaling

Tabel 8.2: De mogelijke situaties wanneer een lichtindex wordt verwijderd uit een octreeknooppunt.

8. BESLUIT

Indien dit het geval is kan de knoop \mathbf{k} hier geplaatst, en dient de textuur geassocieerd met H aangepast te worden. Is dit niet het geval dan dient de data spatiële hashfunctie H opnieuw opgesteld te worden. Daarnaast dienen de lichtindices geassocieerd met knoop \mathbf{k} toegevoegd te worden aan de lichtindexlijst.

Toevoegen van opgesplitste knopen in de octree spatiële hashfunctie H

Opnieuw wordt gekeken of voor elke opgesplitste knoop \mathbf{k} geldt:

$$H[h_0(\mathbf{k}) + \Phi[h_1(\mathbf{k})]] \neq \emptyset$$

Afhankelijk hiervan wordt de spatiële hashfunctie H opnieuw opgesteld of aangepast. De waardes in deze nieuwe knopen beschrijven de nieuwe octreestructuur.

Aanpassingen van de lichtbeschrijving spatiële hashfunctie Indien een lichtindex wordt toegevoegd aan een knoop dient deze te worden toegevoegd aan de verzameling van indices geassocieerd met deze knoop in de lichtindexlijst. In het geval dat een lichtindex verwijderd wordt, dient deze uit de lichtindexlijst verwijderd te worden. Vervolgens dient de textuur zodanig aangepast te worden dat alle knopen opnieuw wijzen naar de correcte deelverzameling van de lichtindexlijst.

Aanpassing van de octreebeschrijving spatiëlehashfunctie Indien de octreestructuur verandert, dienen alle relevante knopen aangepast te worden, zodanig dat de nieuwe situatie wordt beschreven. Dit kan gedaan worden door de geassocieerde texturen aan te passen

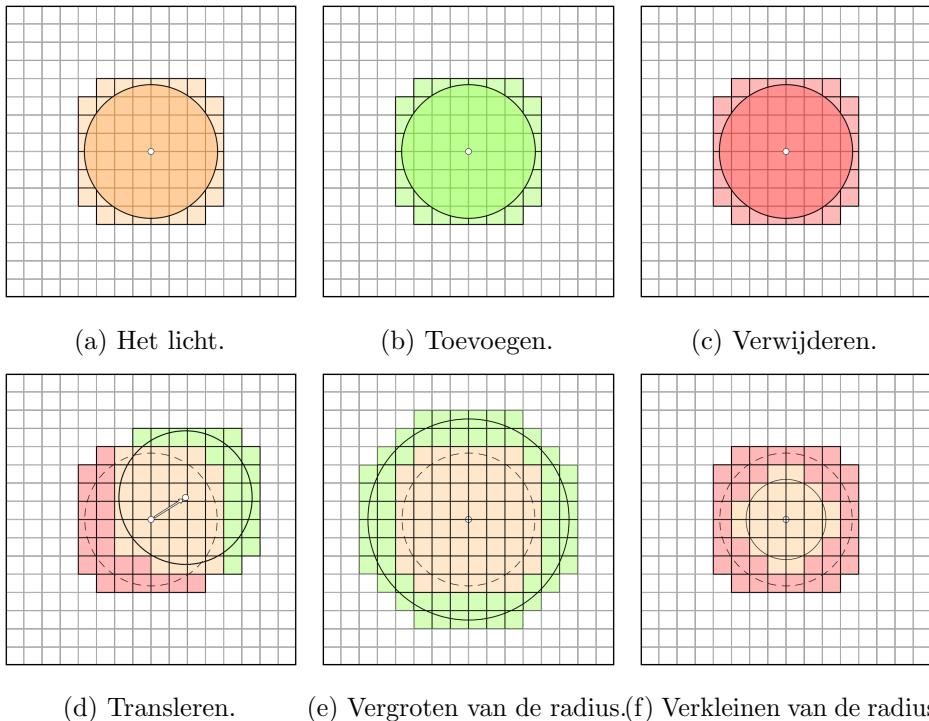
Verwijderen van knoop \mathbf{k} uit de lichtbeschrijving spatiële hashfunctie H

Gezien knoop \mathbf{k} slechts opgehaald wordt als de corresponderende octreeknoop is gedefinieerd als vol, dient de dataknop in het geheugen van de grafische kaart niet explicet verwijderd te worden. Het volstaat om knoop \mathbf{k} te markeren als leeg, \emptyset , zodat het geheugen hergebruikt kan worden tijdens het toevoegen van nieuwe knopen.

Met behulp van deze operaties kan de verbindingloze octree aangepast worden wanneer de lichten transformaties ondergaan. Voor puntlichten kunnen vijf transformaties geïdentificeerd worden.

- Translatie van het puntlicht.
- Het groter schalen van de radius.
- Het kleiner schalen van de radius.
- Het toevoegen van een puntlicht.
- Het verwijderen van een puntlicht.

Rotatie heeft geen invloed op puntlichten en kan daarom buiten beschouwing gelaten worden. Elk van deze transformaties is geïllustreerd in Figuur 8.4. Hierbij is



Figuur 8.4: 2D weergave van de mogelijke transformaties van puntlichten en hun invloed op de octree structuur.

het toevoegen van de index geassocieerd met het licht aan een knoop weergegeven in groen, en het verwijderen van een index uit een knoop weergegeven in rood.

Om vervolgens de verbindingloze octree aan te passen dienen de volgende stappen uitgevoerd worden:

1. Voor alle dynamische lichten dient de verzameling van toevoegingen en verwijderingen opgesteld te worden.
2. Alle verwijderingen en toevoegingen per knoop dienen samengevoegd te worden.
3. De aanpassingen aan de verbindingloze octree dienen te worden opgesteld aan de hand van de eerder opgestelde operaties. Hierbij wordt, indien nodig, ook geëvalueerd of knopen samengevoegd kunnen worden.
4. De aanpassingen dienen doorgevoerd te worden in het grafisch geheugen.

Wanneer de transformaties vloeind weergegeven worden bij een framerate van 30 tot 60 frames per seconde, zullen de transformaties klein en lokaal van aard zijn. Hierdoor zal het aantal aanpassingen dat doorgevoerd dient te worden in de verbindingloze octree ook klein zijn. Dit leidt ertoe dat enerzijds herberekeningen van de spatiale octree schaars zullen zijn, en anderzijds de aanpassingen die gedaan dienen te worden in redelijke tijd uitgevoerd kunnen worden.

Het aantal herberekeningen van de spatiale hashfuncties kan verder beperkt worden met het inzicht dat extra lichtindices binnen een knoop niet tot lichtartefacten

8. BESLUIT

zullen leiden. Dit houdt in dat het opsplitsen van bladknopen als gevolg van het toevoegen of verwijderen van een lichtindex aan een subknoop niet noodzakelijk hoeft worden uitgevoerd. In het geval dat een lichtindex wordt toegevoegd aan een subknoop van een bladknoop kan in plaats hiervan de lichtindex direct toegevoegd worden aan de bladknoop. In het geval dat een lichtindex verwijderd kan worden van een subknoop van de bladknoop, kan deze operatie genegeerd worden. In beide gevallen wordt de structuur van de octree behouden ten kosten van extra lichtberekeningen. Dit is voordelig wanneer de winst behaald met het reduceren van de lichtberekeningen niet opweegt tegen de extra kosten die de aanpassingen aan de octree met zich meebrengen. In plaats hiervan kan gekozen worden om deze pas uit te voeren wanneer de behaalde winst groter is, of wanneer de spatiale hashfunctie herberekend moet worden als gevolg van een andere operatie.

Een significant nadeel van de huidige implementatie van Hashed Shading is dat de gehele ruimte per laag beschreven wordt door een enkele spatiale hashfunctie. Hierdoor kunnen lokale aanpassingen leiden tot een globale herberekening. Er kan dus geen gebruik gemaakt worden van de lokaliteit van de transformaties. Een oplossing hiervoor is reeds geïntroduceerd in de vorige sectie. Door de ruimte onder te verdelen in kleinere stukken, is het mogelijk om, indien nodig, slechts de ruimte her te berekenen, waar de transformaties plaatsvinden. Hierdoor zullen de kosten van herberekeningen kleiner zijn. Daarnaast kan door de keuze van de grootte van stukken, de uitvoeringstijd van een herberekening beter beheerst worden. Verder kan er bij een dergelijke implementatie er voor gekozen worden om slechts de ingeladen stukken direct aan te passen. Voor niet zichtbare stukken kunnen de aanpassingen opgeslagen worden, en slechts berekend worden wanneer deze opnieuw ingeladen worden.

Ondanks al deze optimalisaties zal de ondersteuning voor dynamische lichten altijd extra kosten met zich meebrengen binnen Hashed Shading. Het doel van de voorgestelde strategie is om deze kosten zoveel mogelijk te beperken waardoor lichtmanagement en de lichtberekeningen uitgevoerd kunnen worden binnen een realtime applicatie.

In dit voorstel is de aannname gemaakt dat de herberekening van spatiale hashfuncties voor zal komen ten gevolge van de veranderingen van lichten binnen de scène. Verder onderzoek zal moeten uitwijzen hoe vaak een dergelijke herberekening zal voorkomen. Hierbij zou de invloed van de grootte van de hashtabellen geassocieerd met de lagen van de verbindingloze octree op het aantal herberekeningen geëvalueerd kunnen worden. Indien deze ruimte groter wordt gemaakt zal er meer lege ruimte geïntroduceerd worden. Dit leidt tot een groter geheugengebruik, maar zal het aantal herberekeningen reduceren, doordat nieuw geïntroduceerde knopen relatief vaker op een lege adres zullen vallen.

8.2.4 Andere onderzoeksrichtingen

Naast oplossingen voor de twee geïdentificeerde problemen zijn er nog andere richtingen waarop verder onderzoek zich kan richten. De huidige implementatie ondersteunt geen schaduwen. Er zou geëvalueerd kunnen worden hoe de datastructuur geïntrodu-

ceerd in Hashed Shading gebruikt kan worden om schaduwen efficiënt te berekenen. Voor Clustered Shading zijn dergelijke algoritmes ontworpen, zie [OSK⁺14, KSDA16], die mogelijk als leiddraad kunnen dienen.

Daarnaast worden slechts puntlichten gebruikt binnen de huidige implementatie van Hashed Shading. Moderne game-engines ondersteunen meer lichttypes dan alleen puntlichten. Voor de ondersteuning van andere lichtvolumes dient de opdeling efficiënt opgesteld te kunnen worden.

Bijlagen

Bijlage A

Poster

Deze bijlage bevat de poster waarop deze thesis is samengevat. Hierbij zijn enkele visualisaties van het algoritme weergeven als ook de gebruikte datastructuren. De oorspronkelijke poster is in a1-formaat.

A. POSTER

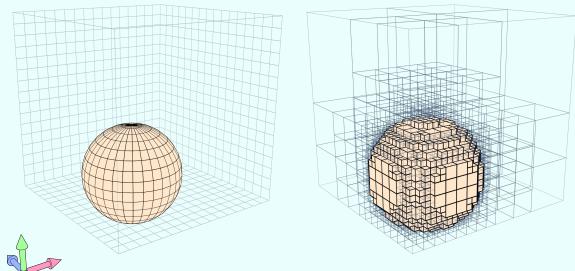


Hashed Shading

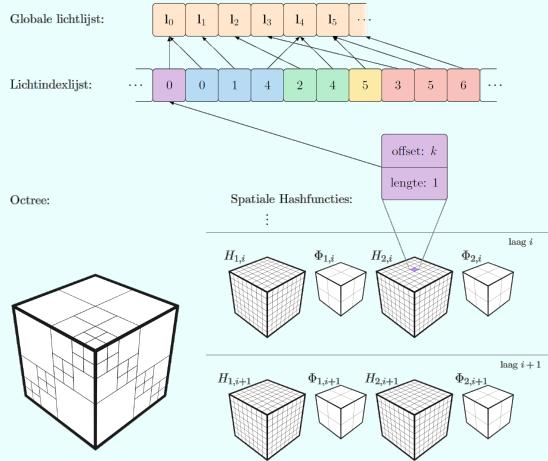
Situering

- Realtime shading
- Lichttoekenning → beperking aantal lichtberekeningen tijdens shading
- Tiled en Clustered Shading populaire lichttoekenningsalgoritmes

Opdeling ruimte



Datastructuren



Visualisatie aantal lichtberekeningen.



Doelstelling

- Evalueren van de mogelijkheid om datastructuren van lichttoekenning her te gebruiken tussen frames
- Ontwikkelen van een camera-onafhankelijk lichttoekenningsalgoritme
- Betere performantie dan Tiled en Clustered Shading

Resultaten

- Hashed Shading algoritme, camera-onafhankelijk lichttoekenningsalgoritme.
- Gebruik van Linkless Octree voor de voorstelling van de ruimte.
- Vergelijkbare lichttoekenning als Clustered Shading.
- Factor twee verbetering ten opzichte van Tiled Shading.
- Hergebruik van datastructuur tussen frames.
- Geen ondersteuning dynamische lichten, maar mogelijke oplossing.
- Nog een groot geheugengebruik, maar mogelijke oplossingen.

Bijlage B

Paper

Deze bijlage bevat het Engstalige paper waarin het ontwikkelde algoritme en de evaluatie daarvan is beschreven.

B. PAPER

Forward and Deferred Hashed Shading for Real-time Rendering of Many Lights

Martinus Wilhelmus Tegelaers

KULeuven

mwtegelaers@gmail.com

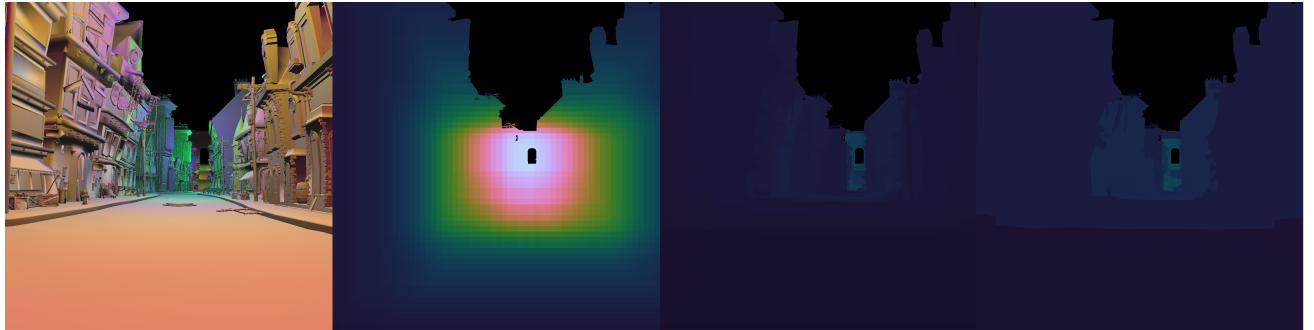


Figure 1: The number of light calculations required to shade the image left, for respectively Tiled, Clustered and Hashed Shading.

ABSTRACT

This paper introduces the Hashed Shading algorithm, a light assignment algorithm for forward and deferred shading. It uses a subdivision independent from the view frustum in order to reuse data structures between frames. The scene is subdivided into cubes of a specified size and represented using a linkless octree to store the data efficiently in memory and allow for a fast retrieval of relevant lights during shading. The performance of Hashed Shading is compared to both Tiled and Clustered Shading. Forward Hashed Shading reduces the number of light calculations and the execution time by a factor of two compared to Forward Tiled Shading. It achieves a similar reduction in the number of light calculations as Clustered Shading. Furthermore the Hashed Shading algorithm scales slightly better in regards to resolution than Tiled and Clustered Shading due to the camera-independent subdivision. Currently Hashed Shading does not support dynamic lights, and requires significant memory to store the linkless octree. Solutions for these issues are proposed but have not been implemented.

KEYWORDS

real-time graphics, light assignment, shading

1 INTRODUCTION

In order to push the limits of visual fidelity in modern games, the complexity of shaders and the number of lights keeps increasing. These complex calculations require significant processing time. To facilitate frames being rendered in real-time, the amount of unnecessary light calculations need to be kept to a minimum. There are several approaches to reduce this amount of work, while preserving the accuracy of the shading. One of such approaches is Light

Assignment. This technique reduces the number of light calculations per pixel by excluding lights that do not influence it. This is possible due to the finite representation of lights generally used within real-time applications. Ideally per pixel only lights of which the corresponding light volume contains the pixel are evaluated. This reduces the number of light calculations to the bare minimum.

In recent games two light assignment algorithms are used commonly, Tiled Shading[OA11] and Clustered Shading[OBA12]. Both algorithms subdivide the view frustum. For each sub frustum the set of lights which overlap with the volume of the sub frustum are calculated and subsequently retrieved during shading to reduce the number of lights which need to be evaluated.

This direct dependency on the view frustum requires the data structures of the two light assignment algorithms to be recalculated per frame. Due to the high frame rate at which images in real-time applications are generated, changes in the scene are generally small and local. Significant parts of the generated data structures could potentially be reused in order to reduce computation time and improve performance.

The goal of this paper is to explore whether reusing the generated data structures does indeed yield a performant algorithm. For this purpose the Hashed Shading algorithm is introduced. This algorithm makes use of a view frustum independent subdivision of the scene space, such that a change in camera position or orientation does not warrant a complete rebuild of the data structures. The subdivision itself is built upon the octree data structure, such that the scene space is represented efficiently in memory while still being precise.

In order to validate whether data structures persistent between frames can potentially achieve better performance than current light assignment data structures, the Hashed Shading algorithm

needs to achieve at least a similar execution time speed up and light calculation reduction as Tiled and Clustered Shading.

2 RELATED WORK

2.1 Deferred Shading

A first step into reducing the number of light calculations is decoupling geometry and shading complexity. In the modern graphical pipeline all geometry in the view frustum generates samples to be shaded, regardless of whether these are visible or occluded. These samples are all shaded and only after executing the light calculations the visibility of pixels is determined in the per-pixel operations. In order to shade only the visible samples, the visibility has to be determined before shading. The concept to achieve this was introduced in hardware design in 1988[DWS⁺88]. A general purpose strategy to achieve this deferred shading was introduced in 1990, making use of so called geometry buffers or GBuffers[ST90]. The idea is to execute the render pipeline twice per frame. In the first pass all attributes necessary to perform the shading computation are rendered into screen size textures, the GBuffers. In the second pass one sample per pixel is generated and for these samples the values are retrieved from the GBuffer and used to shade the pixel. This way the visibility is determined for the attributes in the in the per-pixel operations of the first pass. Then only the attributes of the visible samples are retrieved in the second pass.

A first light assignment approach is to rasterise the light volumes during the second pass. The shading contributions per light are then only calculated for the samples generated by its light volume. This reduces the total number of light calculations, as not every light will be evaluated for every pixel. This optimisation is called the stencil-optimisation[AA03]. This approach has two downsides. It requires a significant number of memory accesses, as per generated sample the shading data has to be retrieved from the GBuffers. Also, the summation of shading contributions happens in a framebuffer. This reduces the precision as the framebuffer's precision generally does not have as much bits as the GPU registers.

2.2 Tiled Shading

Tiled Shading[OA11] was designed to alleviate the memory bandwidth bottleneck of Deferred Shading with the stencil-optimisation. The underlying idea is to calculate which lights effect which pixels before shading, instead of during. This is achieved by subdividing the screen space into tiles of $n \times n$ pixels. For each tile the light indices of the lights which projected light volumes overlap with the tile, are stored. The tiles can subsequently be used to retrieve the set of relevant lights during shading. This can be done by determining the tile in which a pixel falls, and retrieving the light indices associated with this tile. Tiles can be constructed by projecting the light volumes on the view port and determining with which tiles it overlaps. To each of the tiles with which the light overlaps, the light index of the light is added.

A further optimisation can be done by calculating the minimum and maximum z-values for each tile. These can then be used to reject any light which falls outside of these boundaries, thus reducing the number of lights associated with a tile.

This algorithm has been implemented in several game engines and games[BE08, Swo09, BE08], including Unreal[Kar13], frostbite[And09, Mag11] and unity[PD14].

2.3 Clustered Shading

Clustered Shading[OBA12] extends Tiled Shading by subdividing the volumes associated with the tiles further. The tile volumes are subdivided with regards to the camera's z-axis to obtain cube-like sub frustums. These sub frustums can be further partitioned based on attributes such as the normals. These higher dimension tiles are called clusters.

The clusters are constructed by first determining the set of cluster volumes which contain visible geometry. This is done by transforming the depth buffer into a cluster buffer by binning the z-values. This cluster buffer is sorted and compacted locally per tile to extract the set of unique clusters. At the same time a mapping is build, such that for each pixel the corresponding cluster can be looked up. To each of the unique clusters the indices of overlapping lights are added.

During the shading step for each pixel the set of relevant lights is determined by first looking up the corresponding cluster associated with a pixel and then retrieving the set of lights overlapping with this cluster. The shading contributions of each of these lights are evaluated similarly to Tiled shading.

This subdivision of the view frustum is more fine-grained than that of Tiled Shading, and thus reduces the number of light calculations per pixel to a greater extend. Clustered Shading is used in several modern games, including in the Avalanche game engine, on which games as Just Cause 2 and 3[Per13], Doom 4[Tat16].

2.4 Perfect Spatial Hashing

In order to efficiently subdivide the scene space without using a large amount of memory, an octree data structure is used. A straightforward implementation using pointers would require a large number of control structures. Such control structures would degrade the performance of the data structure on the GPU[HA11]. A pointer based data structure would therefore be infeasible for a light assignment algorithm. In Hashed Shading a Linkless Octree[CJC⁺09] is used. This data structure builds upon perfect spatial hash functions[LH06].

A hash function h can be used to map sparse data D on a compact memory table H :

$$D(p) = H[h(p)]$$

When the hash function h maps none of the keys p onto the same address it is collision-free and called perfect. Perfect hash functions are ideal for GPU applications. They allow sparse data to be represented efficiently in memory and each of the elements can be retrieved from memory without needing any control structures.

Perfect spatial hashing[LH06] introduces a technique to construct memory efficient perfect hash functions for higher dimensions which are spatially coherent. This is done by using two simple imperfect hash functions h_0 and h_1 , and one offset hash table Φ . The perfect hash function h can then be defined as

$$h(p) = h_0(p) + \Phi[h_1(p)]$$

where h_0 and h_1 are simple hash functions defined as

$$h_0 : p \mapsto p \bmod m$$

$$h_1 : p \mapsto p \bmod r$$

where m is the size of a single dimension of the hash table H and r is the size of a single dimension of the offset hash table Φ .

The underlying idea is that colliding keys in h_0 do not collide in h_1 , thus by adding an offset defined in Φ to the calculated address of h_0 the collisions within h are dissolved.

This representation makes it possible to store sparse three dimensional data compactly in memory. The linkless octree data structure will be built upon these perfect spatial hash functions.

2.5 Linkless Octree

The linkless octree [CJC⁺09] is a GPU efficient octree implementation. Each layer of the linkless octree implementation is represented by a perfect spatial hash function, therefore removing the need for pointers between nodes.

An octree contains a number of layers, each containing nodes of a similar size. Because each node in a single layer has the same size, the position of a node can be represented by a three dimensional integer vector. Due to the structure of the octree, the nodes of each layer are generally sparsely distributed over the complete scene space. Each layer can thus be compactly represented by a perfect spatial hash function, where the keys are the integer positions of the nodes, and the data is a description of the node at that position. This description needs to encode whether an octree node is a branch or leaf node, and whether it contains data or not. Therefore each node can be described by two bits. Due to memory considerations eight sibling nodes in a layer $i + 1$ are represented directly by the branch node in layer i . This branch node has two 8-bit integers, which describe each of the child nodes of this branch node directly.

The data associated with the octree nodes needs to be stored as well. There are two options for doing so. The straightforward solution is to reserve the space necessary for each data element associated with a node directly in the hash table associated with a layer. This is a valid approach when the data is small or when the majority of the nodes contains data. However, when the data is large, or only a small subset of the nodes actually contains data, this solution would lead to unacceptable memory requirements. This can be remedied by constructing a second spatial hash function per layer which will hold the data elements. Each data element associated with a nonempty node in a layer is stored within this second spatial hash function. Data can then be retrieved from the octree by first descending into the octree, determining for each layer the type of octree node encountered. The leaf nodes specifies whether data is associated with a query. If it is, then this data can be retrieved from the second spatial hash function of the leaf node's layer.

3 HASHED SHADING ALGORITHM

The goal of Hashed Shading is to improve the performance compared to Tiled and Clustered Shading by reusing data structures between frames. The data structures can be reused because the subdivision is independent from the view frustum. Thus the data structures do not need to be rebuilt when the camera position or

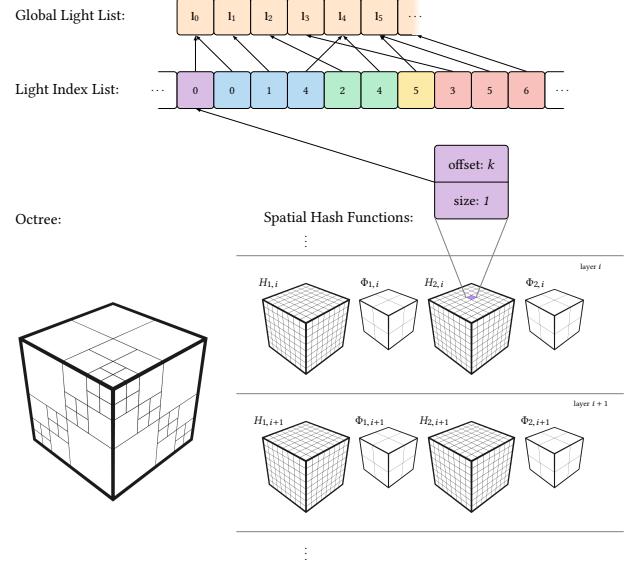


Figure 2: Overview of the data structures of Hashed Shading.

orientation changes. The subdivision of the scene space is done with an octree representation such that it is both precise and memory efficient.

The following data structures are used within Hashed Shading:

- Linkless Octree
- Light Index List
- Global Light List

These data structures are based on those of Tiled and Clustered Shading. The Global Light List contains all the lights within the scene. The Light Index List contains indices linking to the Global Light List. For each leaf node the set of indices specifying the relevant lights is added to this Light Index List. The set of relevant lights for each nonempty node within the linkless octree can thus be specified with an offset and length marking a subset within the Light Index List. This is illustrated in figure 2.

In order to construct these data structures the following steps need to be executed

- Define the subdivision of space.
- Calculate the influence of each light on this subdivision.
- Combine the influences into a single octree describing the whole scene.
- Construct the linkless octree and Light Index List based on the scene octree.

The first step requires the origin and size of the octree to be set. Once these values are specified it is possible to define the influence of each light. The combination of all these influences leads to an octree representation of the whole scene. In order to use this octree representation for light assignment, it needs to be transformed into a linkless octree and a Light Index List. Once all these steps have been completed, the constructed data structures can be used to speed up the shading step of the rendering pipeline. Each of these steps will be explained in more detail in the following sections.

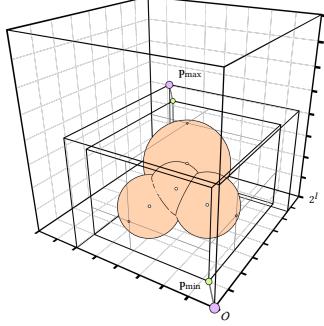


Figure 3: The subdivision of the scene.

3.1 Specification of the Octree

To construct the octree a minimum bounding box containing all samples generated at any point during the simulation needs to be defined. The minimum and maximum point that define the bounding box can either be set by the developer or be constructed based on the light volumes.

In order to ensure no values are generated outside the bounding box due to rounding errors, a small offset is added to the points.

Once the bounding box has been defined, the size of the octree can be calculated. The developer is responsible for setting the size of the nodes in the deepest layer of the octree. Based on this node size, l_o , the size of the octree can be calculated. Assuming that d_i is the length of the longest face of the bounding box, the size of the octree l_o can be defined for the smallest possible k such that holds

$$l_o = l_n \cdot 2^k \geq d_i$$

where k is the number of layers within the octree. This is illustrated in figure 3. With the origin and size of the octree defined it is possible to assign the lights to this subdivision.

3.2 Influence of Lights

Within this paper only point lights are considered. Extending this algorithm to other light types would require a similar approach to be drafted to assign the light volume to the nodes with which it overlaps.

Any light has an influence on a node if the light volume overlaps with the node volume. In the case of point lights whether a light overlaps with a node can be easily calculated by comparing the distance between the light origin, $\mathbf{l}.orig$, and the point \mathbf{p} within the node closest to the origin of the light, with the radius $\mathbf{l}.radius$ of the light. If this distance smaller than the radius, then the node overlaps with the light:

$$\|\mathbf{p} - \mathbf{l}.orig\| < \mathbf{l}.radius$$

Point \mathbf{p} within the node and closest to the origin of the light can be easily calculated by clamping the dimensions of the light origin between the extreme values of the node:

$$\mathbf{v} = \begin{pmatrix} \mathbf{l}.orig_x |_{n_x, n_x + l_n} \\ \mathbf{l}.orig_y |_{n_y, n_y + l_n} \\ \mathbf{l}.orig_z |_{n_z, n_z + l_n} \end{pmatrix}$$

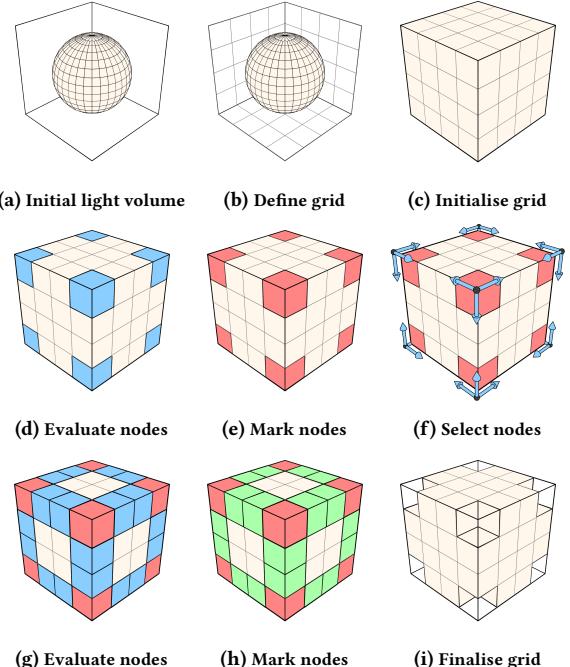


Figure 4: Algorithm to determine the overlap of a single light with a grid of minimal nodes.

where \mathbf{n} is the origin of the node.

This calculation does not need to be executed for each node within the whole space. Any node that overlaps with a light volume falls within the minimal grid of nodes containing the whole light volume. Furthermore, since each light volume is homogeneous, it is only necessary to establish the boundaries of overlapping and non-overlapping nodes in order to know of all nodes whether they overlap. This can be exploited by using a flood fill algorithm to reduce the amount of comparisons. First, all nodes are assumed to be either overlapping or non-overlapping, then the nodes which are not are marked as such. Since a sphere occupies slightly over half of its bounding box volume and partially overlapping nodes are considered to be overlapping, all nodes within the minimal grid, of nodes are initialised to be overlapping. Then starting from the corners of the grid non-overlapping nodes are marked with a breadth first flood fill algorithm. This process is illustrated in figure 4.

Lastly, the way lights are represented in memory needs to be considered. In order to efficiently calculate the change of dynamic lights the influence of lights in the previous frame can be compared to the influence of lights in the current frame. In this case it is beneficial to save the individual lights such that lights that actually change between frames can be evaluated individually. The grid constructed in the previous step can be saved directly. In this case each node requires a single bit in order to represent it. For small lights or large node sizes this is a feasible approach. In case the lights are large or a small node size is used, this approach could lead to a significant memory requirement. In these cases it is more

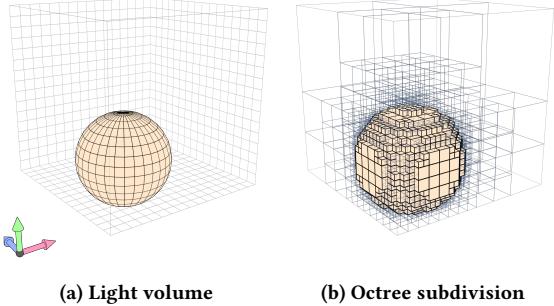


Figure 5: Octree representation of a single light volume.

beneficial to represent the lights as an octree, to reduce the memory footprint. An example of such a representation is given in figure 5.

The octree representing a light can be efficiently constructed in a top-down fashion using the constructed grid. First the octree node containing the whole grid is found. This node will serve as the root node of the light octree. Then for each octree node the type is evaluated. Initially there are three possible situations:

- The octree node volume does not overlap with the grid.
- The octree node volume overlaps partly with the grid.
- The octree node volume falls within the grid.

In the first case the octree node is an empty leaf node, as no filled nodes lie outside of the grid.

In the second case there are two possibilities. Either the node closest to the light origin is empty, or it is nonempty. The octree node is respectively an empty leaf node or a branch node. In the last case there are three possible situations. If the node closest to the light origin is empty, then the octree node is empty as well. If this is not the case, then either the furthest node from the light origin within the octree node is empty or nonempty. The octree node is then respectively a branch node or a nonempty leaf node. In the case that an octree node is a branch node, then the types of the children of this node are evaluated as well.

3.3 Construction of the Scene Octree

The scene octree combines the influences of the individual lights. Each leaf node contains a set of references to the lights which overlap with the leaf node's volume. This octree can be constructed either top-down or bottom-up, depending on the chosen representation of the single lights.

In case the lights are represented by grid nodes, these nodes need to be combined such that a set of nodes is obtained where each node contains the references to its overlapping lights. Subsequently the shallower layers can be constructed. If eight sibling nodes share the same set of indices they can be combined into a single leaf node in the proceeding layer. If they are dissimilar a branch node is added to the proceeding layer.

When the lights are represented by octrees, they can be added top-down to a scene octree initialised with an empty leaf node. First the octree node corresponding with the root node of the light is retrieved, constructing additional nodes where necessary. Then each of the nodes of the light octree is added to the scene octree.

3.4 Construction of the Linkless Octree and Light Index List

The construction of the linkless octree follows the original algorithm. The scene octree is represented by two spatial hash functions per layer, as only nonempty leaf nodes have data associated with them. The first spatial hash function describes the octree structure as described in the original algorithm. The second spatial hash function contains a data element consisting of two integers per nonempty leaf node. These integers specify the subset of light indices within the Light Index List that describe the lights which overlap with the leaf node.

In order to construct this linkless octree, the nodes of the specified starting depth are gathered. For each of these nodes the octree node encoding is calculated. If a node is a nonempty leaf node, the current length of the Light Index List and the size of the set of lights associated with the leaf node are added to the second spatial hash function, as the offset and the length values respectively. The light indices of the set of lights are then added to the Light Index list.

Once all layers have been constructed, the linkless octree and the Light Index List can be loaded into GPU memory.

3.5 Light Assignment during Shading

The final step of the algorithm is to determine the set of relevant lights for the samples generated during rendering. This step is similar to that of Tiled and Clustered Shading. First the leaf node containing the sample has to be calculated by descending into the octree until a leaf node is encountered. If this leaf node is nonempty then the offset and length can be retrieved from the corresponding data hash table. Else a length of zero is returned. Once the offset and length are retrieved the set of light indices from the Light Index List can be obtained. The sample is then shaded by summing the shading contributions of the corresponding lights. This step is equal to that of Tiled and Clustered Shading.

4 IMPLEMENTATION

In order to evaluate the Hashed Shading algorithm a new program, `nTiled`, was developed. This program was build with C++ and openGL and compiled with Visual Studio 2015. Besides the Hashed Shading algorithm, also a naive implementation which evaluates each light per sample, and the Tiled and Clustered Shading algorithms were implemented. All programs execute the same shading code, a simple white Lambertian material.

4.1 Deferred Shading

Deferred Shading is implemented by executing two render passes. In the first pass the normal, a white albedo colour, and the depth are written to a GBuffer containing the corresponding textures. In the second pass a full screen quad is rendered, and for each generated sample the shading information is retrieved from the GBuffer and the light contributions calculated.

4.2 Tiled Shading

In the forward and deferred Tiled shaders, the tiles which are influenced by a light are determined by computing the bounding box of the projected light volume in screen space[MM12]. Then to each

of the tiles overlapping with this boundary box, the light index is added. This is all done on the CPU. The implementation does not take into account the minimum and maximum z-values within tiles. Thus the subdivision is not constrained with regards to these values.

4.3 Clustered Shading

Clustered Shading was implemented without the bounding volume hierarchy to assign lights to the clusters specified in the original algorithm. The lights instead were assigned in a brute force fashion by adjusting the Tiled Shading algorithm, and evaluating whether it overlapped with any clusters which contained geometry. The sort and compact step were implemented with OpenGL compute shaders. The assignment of lights is implemented on the CPU which required the use of `glGetTexImage`¹ This leads to a significant slowdown of the algorithm which skews the execution time results.

4.4 Hashed Shading

Hashed Shading is completely implemented on the CPU. The individual lights are represented by octrees. The size of the offset hash table Φ is selected by using the $\frac{n}{6}$ approximation, and increased geometrically until a correct perfect spatial hash function is constructed. The offset and length values are represented by 32 bit unsigned integers.

5 EVALUATION

The performance of the Hashed Shading algorithm is evaluated with regards to both the construction of the data structures and the execution during shading. The construction of the data structures is evaluated with regards to both the time required to construct the data structures as the memory they take up. The total amount of pixels required by the linkless octree and the size of the Light Index List are used as indicators for this memory usage.

The performance of the shading is evaluated by timing the execution time per frame and by calculating the total number of light calculations required to shade a frame in the deferred pipeline. These values are compared to those of the Tiled and Clustered implementation. Due to the performance issues of the Clustered Shading implementation the execution time of Clustered Shading is not taken into account.

All timings are performed with the `QueryPerformanceCounter` provided on the Windows platform.

The different tests were performed for three scenes:

- Spaceship Indoor: an indoor spaceship scene based on CG Lighting Challenge #18².
- Piper's Alley: a street scene based on CG Lighting Challenge #42³.
- Ziggurat City: A shot from the open movie Sintel⁴.

These scenes were chosen to represent the different potential game environments.

The influence of both the number of lights, as the resolution on the execution time and number of lights of the different light

assignment algorithms is evaluated. For this six different resolutions ranging from 320×320 to 2560×2560 are evaluated, and number of lights ranging from less than a hundred till more than a thousand lights per scene are evaluated.

All of these simulations were executed on a Dell XPS laptop with the following hardware:

Operating system	Windows 10 64-bit
CPU	Inter Core i7 6700 HQ @ 2.60 Ghz
Memory	16 GB
GPU	NVIDIA GeForce GTX 960M
GPU drivers	372.70

6 RESULTS AND DISCUSSION

In order for Hashed Shading to be viable its performance needs to be at least on par with Tiled and Clustered Shading. This requires a low memory footprint and a similar reduction in execution time and number of light calculations.

6.1 Construction Time and Memory Usage

The results of the construction time as function of the node size relative to the radius of the lights for different number of lights can be found in figure 8. The construction time for individual functions for the largest sets of lights can be found in figure 9. A cubic relationship between the node size and the construction time can be clearly observed. This leads to a significant increase of construction time when more than 8^3 nodes are required to contain a light volume. This increase in construction time is primarily due to the construction of the spatial hash functions representing the layers of the linkless octree.

The same cubic relationship can be seen between the memory usage and the node size. Both the combined number of pixels used in the spatial hash functions of the linkless octree, fig 10, and the size of the Light Index List, fig 11, show cubic growth with regards to a smaller node size. Figure 12 and 13 show that the majority of the pixels used in the linkless octree are used within the deepest layers of the linkless octree. This indicates that a significant number of leaf nodes exist in the deepest layers of the linkless octree, when the node size decreases. This is further supported by the increase in size of the Light Index List. Each leaf node adds its set of light indices to the Light Index List. Thus the total size of the Light Index List is an indicator for the total number of leaf nodes. A smaller node size allows for a more fine-grained description of scene space. When lights partially overlap the leaf nodes can not be combined as their light index sets differ. A smaller node size will increase this effect, and more small octree nodes will be spawned as a result.

The node size effects the size in all three dimensions of the node. Thus a reduction of the node size potentially leads to a cubic increase in number of nodes. This cubic increase of nodes is the source of the increase in number of nodes which drives up the memory usage and construction time.

The influence of the starting depth on the construction time and the combined size of the textures is shown in figure 14 and 15 respectively. The construction time and memory usage are not effected by this change. However, the memory overhead is reduced

¹ `glGetTexImage`: <https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/glGetTexImage.xhtml>

² scene url: <http://www.3drender.com/challenges/>

³ scene url: <http://forums.cgsociety.org/archive/index.php?t-1309021.html>

⁴ open movie url: durian.blender.org

as the total number of required textures is reduced. Increasing the starting depth removes the shallowest layers of the linkless octree. These have a relatively small influence on the total memory usage and the construction time as the spatial hash functions are small compared to the deeper layers of the linkless octree. Thus a greater starting depth can be used to reduce the overhead of the textures and the total number of textures used, without significantly increasing the amount of nodes required. Furthermore removing the shallowest layers of the octree will reduce the number of memory accesses during the shading step, thus improving the performance during shading.

6.2 Execution Time and Light Calculations

Figure 16 and 17 show the execution time per frame for the forward and deferred pipelines during a single simulation of both Tiled and Hashed Shading. Figure 18 shows the number of light calculations per frame during a single simulation of the deferred pipeline for Tiled, Clustered and Hashed Shading.

In the deferred pipeline a similar performance with regards to the execution time can be observed for both Tiled and Hashed Shading, even though the Hashed Shading implementation requires only half of the light calculations. This is a result of the simplicity of the used fragment shader. The bottleneck in this case is not the shading pass but the geometry pass. The change in number of calculations does not influence the geometry pass, thus performance is comparable. When the complexity of the shader, and therefore the amount of work during the shading pass, is increased the difference in the number of light calculations should become noticeable. The Forward Hashed Shading implementation does perform about twice as well as the Forward Tiled Shading implementation, which is in line with the reduction of the number of light calculations achieved with Hashed Shading. The forward pipeline has significant overdraw, thus per pixel multiple fragments are evaluated. This amplifies the influence of the reduction of the number of light calculations.

For small node size the reduction of light calculations achieved by Hashed Shading approaches that of Clustered Shading. When the node size would be reduced further, it should equate and potentially surpass Clustered Shading.

In figure 19 and 20 the average execution time per frame for the Forward and Deferred pipelines as function of the number of lights is plotted. All of the algorithms scale linearly with the number of lights. The results are comparable to those observed for the single simulation. For Deferred Shading the performance increase of Hashed Shading is similar to that of Tiled Shading. In the Forward pipeline, again an improvement of a factor two is noticeable. The number of light calculations, figure 21, shows that the reduction in calculation time is a direct result of the reduction of number of light calculations. The smallest node size shows a comparable curve to that of Clustered Shading.

The behaviour with regards to the resolution is shown in figure 22, 23 and 24. Both the execution time and the number of light calculations are linearly dependent on the number of fragments. Hashed Shading performs slightly better with increasing resolutions, though the effect is almost negligible. The memory usage of both Tiled and Clustered Shading is directly linked to the size of view port. Hashed Shading has no such dependency, thus it will not

be significantly affected by changing resolutions. With the trend of growing resolutions, this might prove to be an important attribute of Hashed Shading.

7 CONCLUSION

In this paper the Hashed Shading algorithm has been presented and evaluated. Hashed Shading is a light assignment algorithm which subdivides the scene space independently of the view frustum. This subdivision is then stored in a linkless octree which can be queried during shading. Because the subdivision is independent of the view frustum, changes in the camera position and orientation do not require a complete recalculation of the data structures. Thus the data structures can be reused between frames.

The Hashed Shading algorithm performs better than the Tiled Shading algorithm and approaches the reduction of light calculations achieved by Clustered Shading. Furthermore Hashed Shading performs slightly better with increasing resolutions compared to both Tiled and Clustered Shading, because its subdivision does not depend directly on the direction.

However there are still two major problems with the current Hashed Shading implementation. The memory footprint increases cubically with a decreasing node size. This leads to unacceptable memory requirements for small node sizes. Secondly, dynamic lights are not currently supported within the Hashed Shading implementation. Adding this support would increase computation time during shading.

Overall, it can be concluded that camera-independent light assignment is a viable strategy. The decoupling of resolution could play a role with the trend of increasing screen resolutions. Future work would need to solve the memory requirements and allow for efficient support of dynamic lights in order for Hashed Shading and similar algorithms to be competitive with Clustered Shading.

8 FUTURE WORK

The two main issues with the current implementation of Hashed Shading are the large memory footprint and the lack of support for dynamic lights. These two issues have to be addressed before Hashed Shading can become a viable alternative to Clustered Shading. Proposals to solve those two issues are introduced in the next sections.

Besides these two issues, there are several other directions for future work. Currently the Hashed Shading implementation only supports point lights. In order to support other light types, strategies to convert their light volumes into octrees need to be drafted. The current implementation also does not support shadows currently. There are several algorithms which leverage the clustered shading clusters to reduce the amount of work required to support shadows within Clustered Shading[OSK⁺14, KSDA16]. Similar approaches could be used to support shadows in Hashed Shading by leveraging the octree data structure.

8.1 Memory Usage Reduction

8.1.1 Geometry. An important insight Clustered Shading uses to reduce memory usage is that only clusters which contain geometry will potentially be queried. Thus only clusters that contain potential samples need to be constructed and kept in memory. This

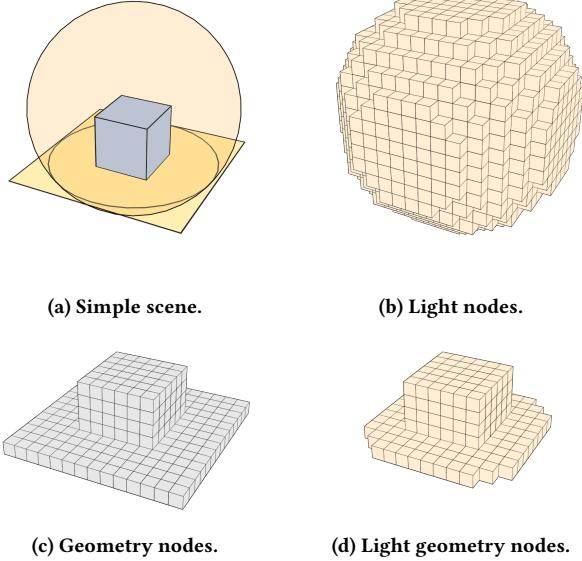


Figure 6: Reduction of the number of nodes by excluding empty space.

reduces the total number of clusters required by Clustered Shading. This insight can also be leveraged by Hashed Shading. There is no need to keep nodes which contain no geometry in memory as they will never be queried. If a second octree with the same origin and size of the screen octree is constructed, where the leaf nodes specify whether the corresponding volumes contains geometry, then this octree can be used to filter out nodes which will not be queried. Only if a node contains both geometry and light information, it needs to be stored in the linkless octree. Thus such a geometry octree could be used to greatly reduce the number of nodes in the linkless octree as is shown in figure 6.

A second advantage of such an approach would be that the number of nodes would not necessarily be cubically dependent on the node size. A smaller node size would also increase the precision with which geometry is represented, thus it would reduce the space which contains geometry and therefore reduce the increase in number of nodes which need to be represented within the linkless octree.

Lastly the conditions required for a set of sibling nodes to be combined into a single leaf node in the preceding layer could be weakened. Since nodes which do not contain any geometry data will never be queried, an arbitrary value could be assigned to them, without influencing the performance of Hashed Shading. The current condition requires all the sibling nodes to have the same set of lights, before they can be combined into a single leaf node. With this optimisation this can be weakened to requiring all sibling nodes which contain geometry to contain an equal set of lights. This would further reduce the number of nodes needed to represent the scene space.

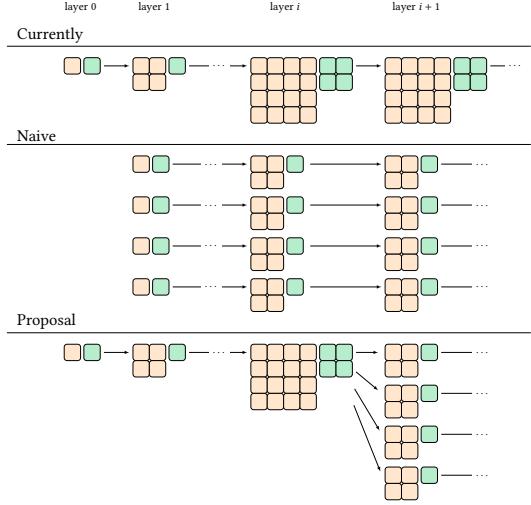


Figure 7: Subdivision strategy deeper layers of the linkless octree.

8.1.2 Subdivision of Space. A common approach in many graphical applications is to only load data into memory which is visible, i.e. only that which falls within the view frustum. In the current implementation the whole scene is loaded into memory, without any regard for the view frustum, which leads to an unnecessary large memory usage. If the view frustum is significantly smaller than the whole scene, it can be advantageous to subdivide the light assignment octree in chunks, and only load the visible chunks into memory. If the chunk size is set to be similar to the view frustum, at least eight chunks are required to cover the view frustum at any point. This could be extended to 27 chunks if camera rotations are common. In this case no new chunks will have to be loaded in when the camera orientation changes, thus reducing the texture bandwidth. When these chunks require less memory than the complete representation, the memory usage is reduced.

A potential problem of this approach is the large number of textures required to represent the octree. Each layer of each chunk would require two separate textures, four if the layer also contains leaf nodes. If 27 chunks are used, and each chunk uses several layers, this would quickly lead to an infeasible amount of textures. However looking at the results of the memory usage, it can be shown that the majority of the memory is used in the deeper layers, which contain a large number of nodes, and that the shallow layers are relatively small. An alternative would be to only subdivide the space into chunks at a certain depth. In the current implementation each layer represents the full scene space, however there is no obstacle into subdividing this space into smaller chunks. If we would subdivide layer $l + 1$ into chunks, then the branch nodes in layer l would only require a single value to identify the chunk in which the child node lays. This would make it possible to reduce the memory usage in the deeper layers, while keeping the number of changing textures to a minimum. The possible options are shown in figure 7.

Finally, the chunk approach would make it trivial to support level of detail. Objects close to the camera will generally take up

more screen space than objects further away from the camera, due to perspective. Thus if the same reduction of light calculations is created for objects far away from the camera as for objects close by the camera, the amount of memory used to reduce the number of lights for samples far away will be relatively greater. By reducing the amount of memory used for space further away from the camera, and increasing it for space close to the camera, a greater reduction of light calculations overall can be achieved. In Clustered Shading this is done by scaling the cluster volume with regards to the camera's z -axis.

The linkless octree could support this level of detail by defining multiple spatial hash functions per layer. When a chunk is loaded close to the camera, the regular spatial hash functions would be loaded. For chunks further from the camera, only layers till a certain value could be loaded in, where the last loaded layer contains only leaf nodes. Thus per layer one regular spatial hash function and one spatial hash function only containing leaf nodes would have to be defined.

8.2 Dynamic Light Support

The second issue of the current implementation of Hashed Shading is the lack of support for dynamic lights. The downside of not rebuilding data structures per frame is that changes in lights are not implicitly reflected in the data structures any more. Supporting dynamic lights would thus always increase the amount of work per frame. Games and other real-time applications generally have a high frame rate, between 30 and 60 frames per second, therefore changes in lights are generally small and local. This could be leveraged to efficiently update the Hashed Shading data structures. The bottleneck operation in the construction of the Hashed Shading data structures is the construction of the spatial hash functions of the layers of the octree. In order to efficiently support dynamic lights, these operations thus need to be kept to a minimum. A spatial hash function needs to be rebuilt if a new node is added that collides with an existing value:

$$H[h_0(\mathbf{k}) + \Phi[h_1(\mathbf{k})]] \neq \emptyset$$

where \mathbf{k} is the position of a new node. In all other cases the existing textures can be updated without a complete recalculation of the spatial hash function of a layer.

There are two situations which require the addition of nodes in layers. If a light index is added to a previously empty node, a new data element needs to be added to the data hash function of a layer. When a leaf node gets subdivided into eight child nodes, a new node is added to the first spatial hash function of the next layer describing these eight child nodes, and for each of the child nodes that contain data, the data nodes are added to the data spatial hash function of the next layer. The subdivision of nodes therefore is the most likely source of recalculations.

It might prove efficient to slightly extend the size of the hash tables to increase the amount of empty space within them. While this would increase the memory foot print, it could greatly reduce the number of times a spatial hash function needs to be recalculated. The effects of such a change would be interesting to explore.

In the current implementation the whole space of each layer is globally linked, therefore a local change of a single light might

require the complete space to be recalculated. Currently the implementation can not take advantage of the local nature of light changes. If the support of dynamic lights is combined with the chunk optimisation, only the chunks affected by a light change need to be updated, reducing the space affected by that change. Furthermore, because the construction time of a spatial hash functions is directly linked to its size, recalculating the spatial hash function of a chunk would be smaller than recalculating the spatial hash function of the complete space, further reducing the amount of time needed to update. Lastly, only chunks that are currently in memory need to be updated immediately, as changes to these chunks are directly visible. Changes to chunks that are not currently loaded on the GPU will not be visible until the chunk is loaded in GPU memory. Changes to invisible chunks can be queued, and only executed once the chunk has to be loaded into memory. This would reduce the bandwidth and potentially the execution time.

REFERENCES

- [AA03] Jukka Arva and Timo Aila. Optimized shadow mapping using the stencil buffer. *Journal of Graphics Tools*, 8(3):23–32, 2003.
- [And09] Johan Andersson. Parallel graphics in frostbite-current & future. *SIGGRAPH Course: Beyond Programmable Shading*, 2009.
- [BE08] Christophe Balestra and Pål-Kristian Engstad. The technology of uncharted: Drakes fortune. In *Game Developer Conference*, 2008.
- [CJC⁺09] Myung Geol Choi, Eunjung Ju, Jung-Woo Chang, Jehee Lee, and Young J Kim. Linkless octree using multi-level perfect hashing. In *Computer Graphics Forum*, volume 28, pages 1773–1780. Wiley Online Library, 2009.
- [DWS⁺88] Michael Deering, Stephanie Winner, Bic Schediwy, Chris Duffy, and Neil Hunt. The triangle processor and normal vector shader: a vlsi system for high performance graphics. In *ACM SIGGRAPH Computer Graphics*, volume 22, pages 21–30. ACM, 1988.
- [HA11] Tianyi David Han and Tarek S. Abdelrahman. Reducing branch divergence in gpu programs. In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-4, pages 3:1–3:8, New York, NY, USA, 2011. ACM.
- [Kar13] Brian Karis. Real shading in unreal engine 4. *Proc. ACM SIGGRAPH Courses*, page 22, 2013.
- [KSDA16] Viktor Kämpe, Erik Sintorn, Dan Dolonius, and Ulf Assarsson. Fast, memory-efficient construction of voxelized shadows. *IEEE transactions on visualization and computer graphics*, 22(10):2239–2248, 2016.
- [LH06] Sylvain Lefebvre and Hugues Hoppe. Perfect spatial hashing. In *ACM Transactions on Graphics (TOG)*, volume 25, pages 579–588. ACM, 2006.
- [Mag11] Kenny Magnusson. Lighting you up in battlefield 3. In *Proc. 25th Annual Game Developers Conference*, 2011.
- [MM12] Michael Mara and Morgan McGuire. 2d polyhedral bounds of a clipped, perspective-projected 3d sphere. *JCGT*, in submission, 5, 2012.
- [OA11] Ola Olsson and Ulf Assarsson. Tiled shading. *Journal of Graphics, GPU, and Game Tools*, 15(4):235–251, 2011.
- [OBA12] Ola Olsson, Markus Billeter, and Ulf Assarsson. Clustered deferred and forward shading. In *Proceedings of the Fourth ACM SIGGRAPH/Eurographics conference on High-Performance Graphics*, pages 87–96. Eurographics Association, 2012.
- [OSK⁺14] Ola Olsson, Erik Sintorn, Viktor Kämpe, Markus Billeter, and Ulf Assarsson. Efficient virtual shadow maps for many lights. In *Proceedings of the 18th Meeting of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D '14, pages 87–96, New York, NY, USA, 2014. ACM.
- [PD14] Aras Pranckevicius and Rendering Dude. Physically based shading in unity. In *Game Developer's Conference*, 2014.
- [Per13] Emil Persson. Practical clustered shading. *SIGGRAPH Course: Advances in Real-Time Rendering in Games*, 2013.
- [ST90] Takafumi Saito and Tokiichiro Takahashi. Comprehensible rendering of 3-d shapes. In *ACM SIGGRAPH Computer Graphics*, volume 24, pages 197–206. ACM, 1990.
- [Swo09] Matt Swoboda. Deferred lighting and post processing on playstation 3. In *Game Developer Conference*, 2009.
- [Tat16] Natalya Tatarchuk. Advances in real-time rendering, part ii. In *ACM SIGGRAPH 2016 Courses*, SIGGRAPH '16, New York, NY, USA, 2016. ACM.

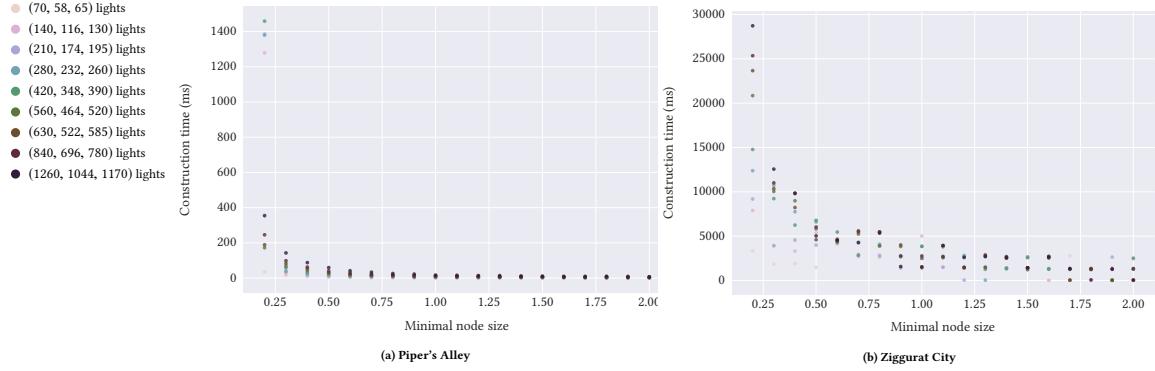


Figure 8: Construction time of the Hashed Shading data structures.

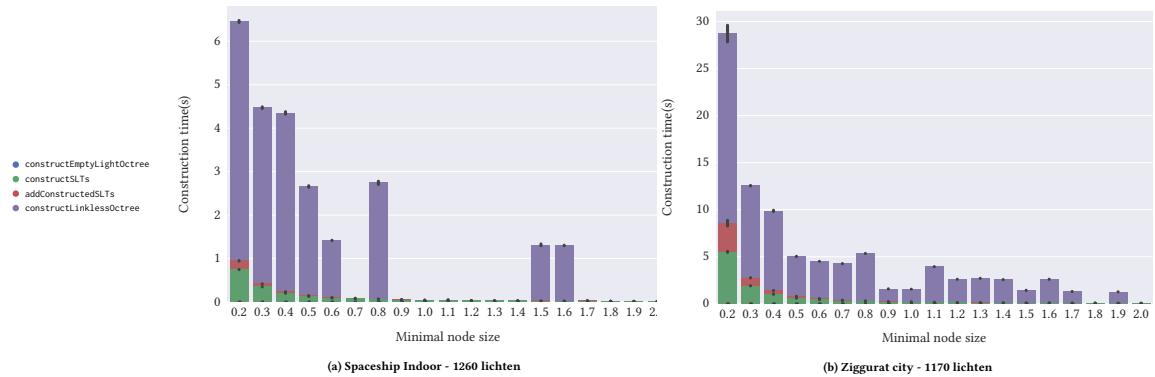


Figure 9: Construction time of the individual steps of Hashed Shading.

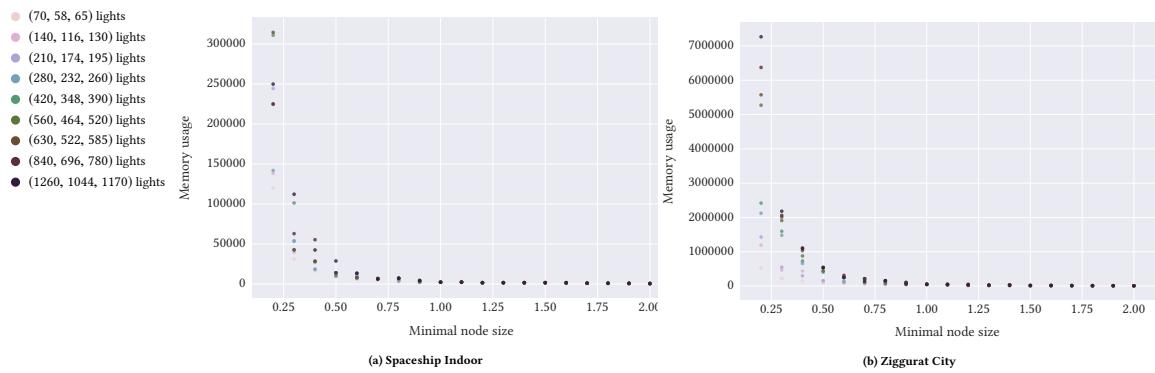


Figure 10: The combined number of pixels of the Linkless Octree.

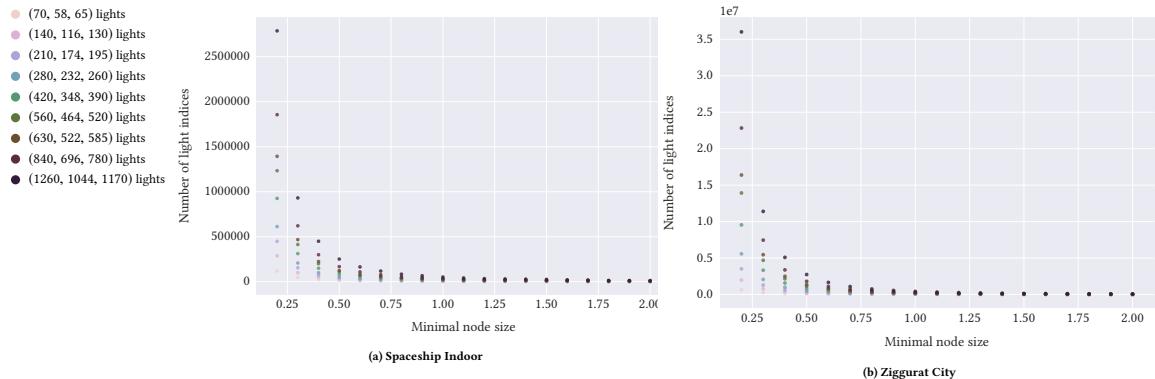


Figure 11: Size of the Light Index List.

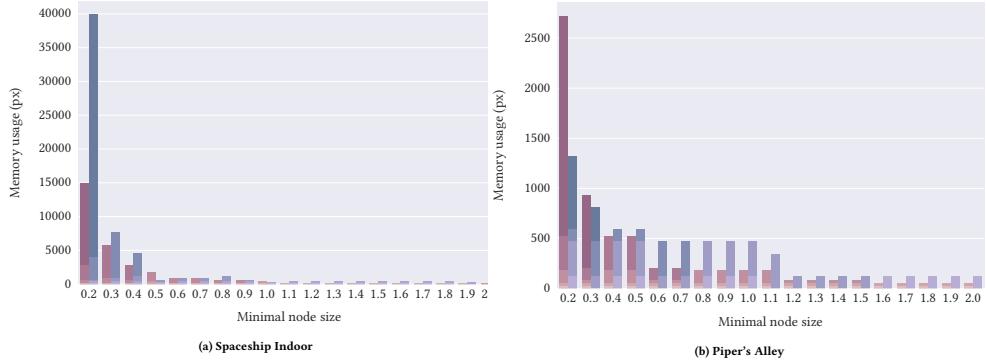


Figure 12: Number of pixels used by the octree description spatial hash functions per layer of the linkless octree.

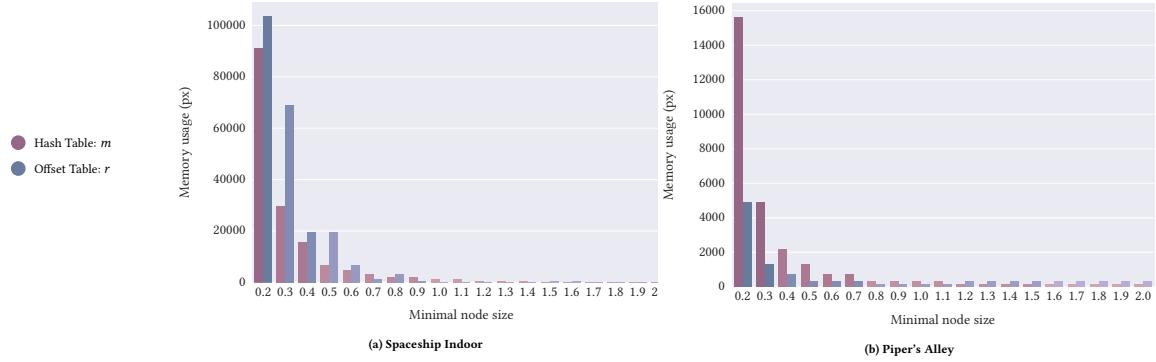


Figure 13: Number of pixels used of the data description spatial hash functions per layer of the linkless octree.

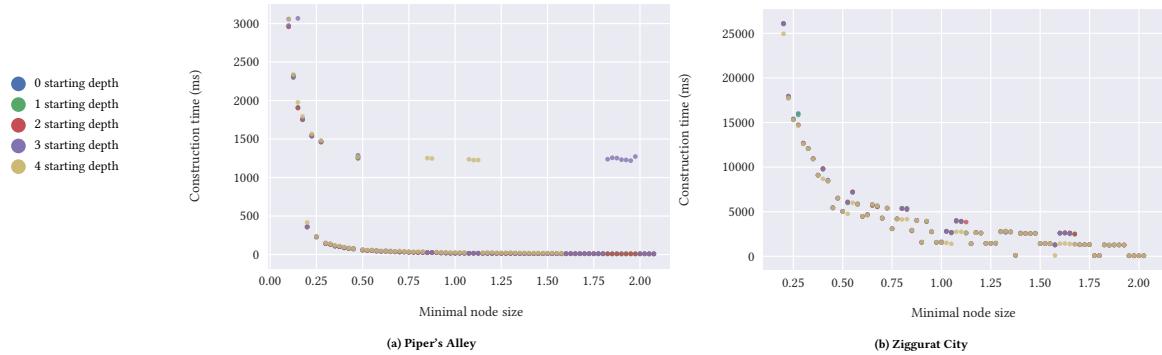


Figure 14: Construction time as function of the starting depth.

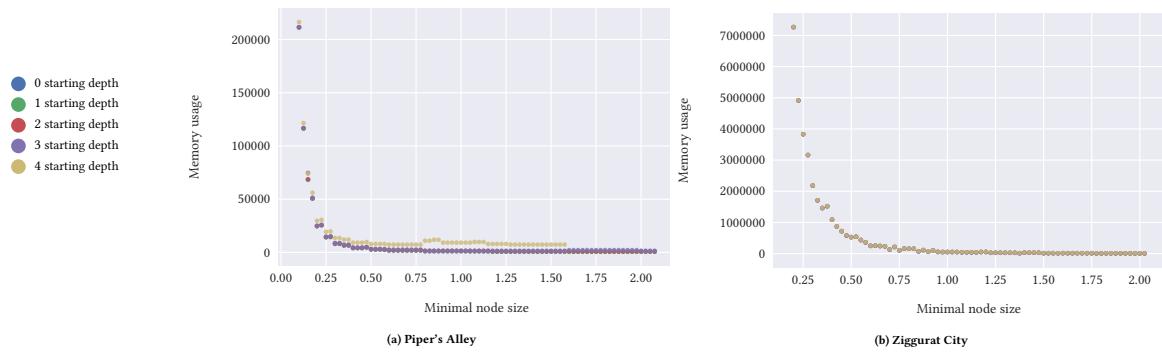


Figure 15: Number of pixels of the linkless octree as function of the starting depth.

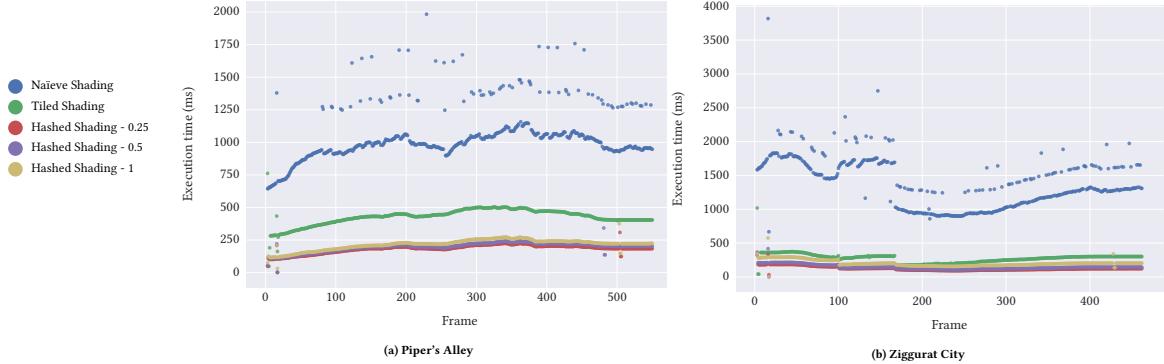


Figure 16: The execution time per frame of Forward Shading.

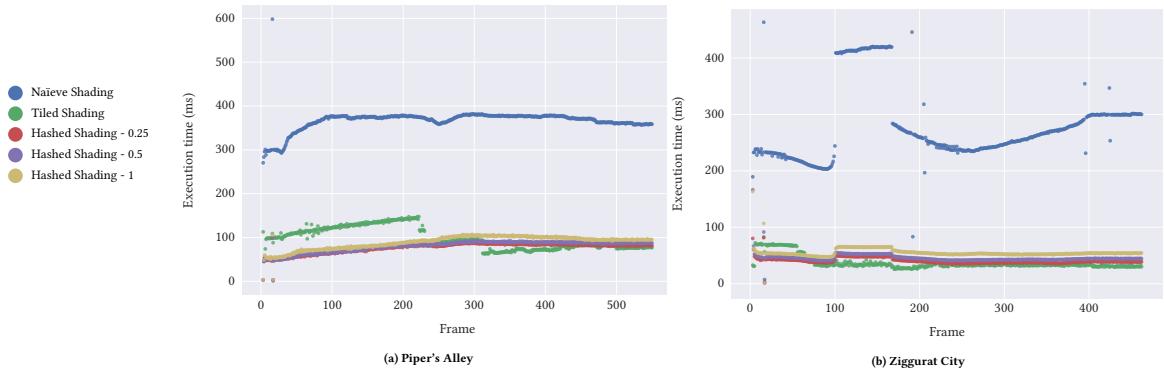


Figure 17: The execution time per frame of Deferred Shading.

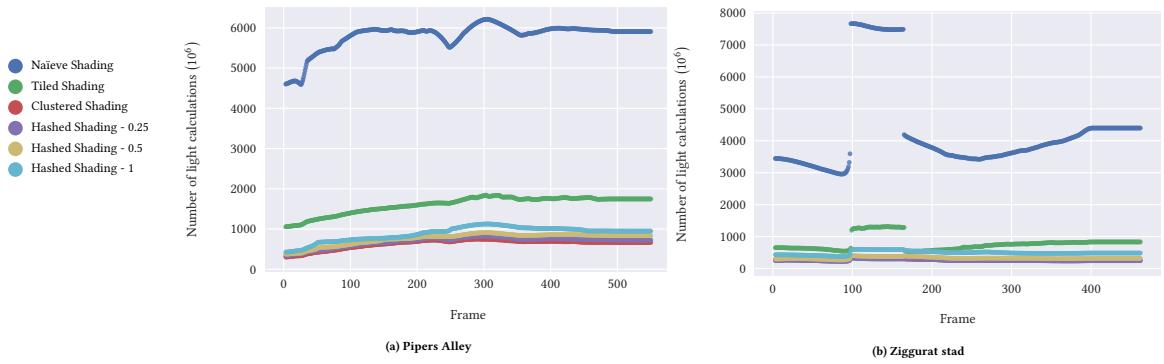


Figure 18: The number of light calculations per frame of Deferred Shading.

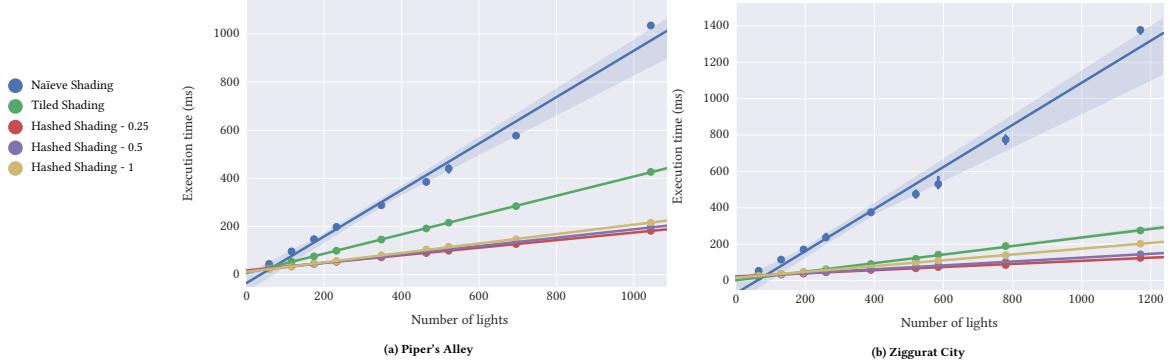


Figure 19: The execution time per frame as function of the number of lights for Forward Shading.

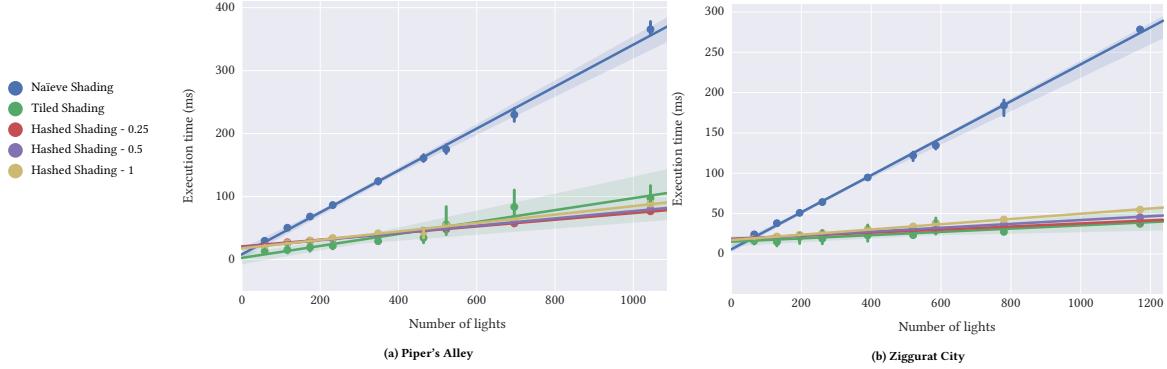


Figure 20: The execution time per frame as function of the number of lights for Deferred Shading.

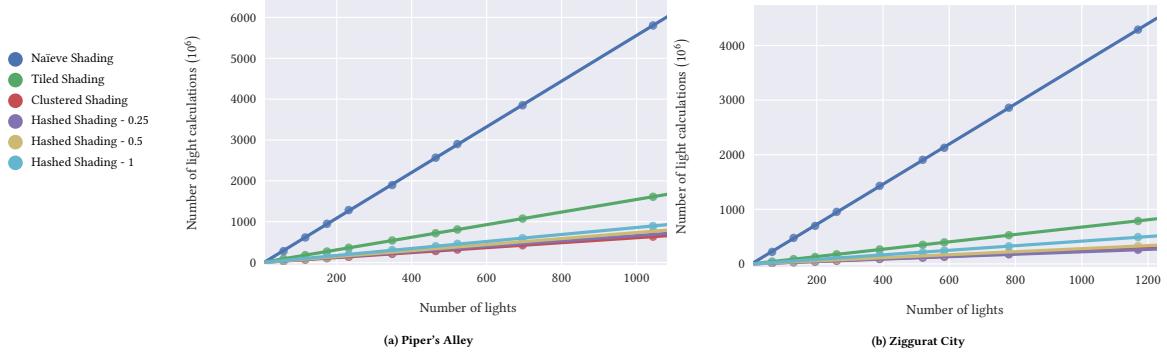


Figure 21: The number of light calculations as function of the number of lights.

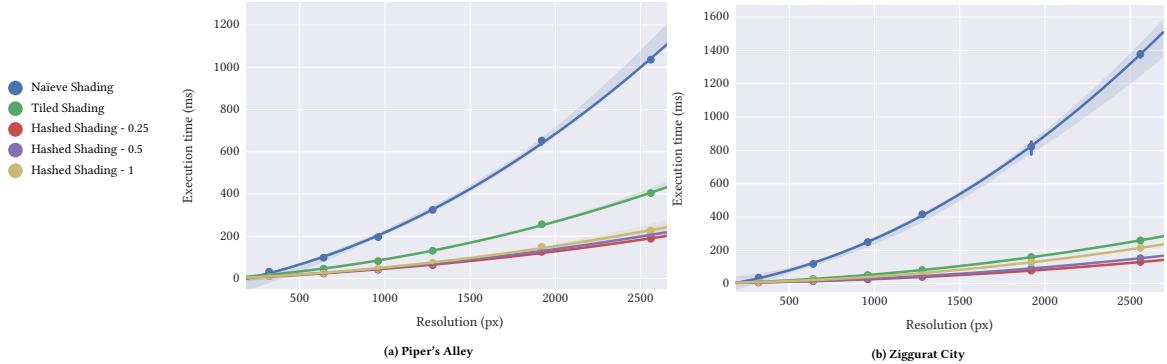


Figure 22: The execution time per frame as function of the resolution for Forward Shading.

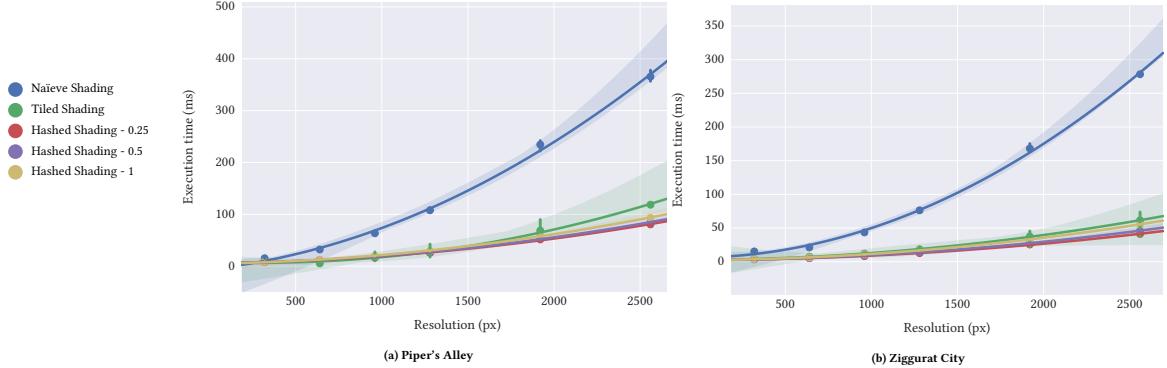


Figure 23: The execution time per frame as function of the resolution for Deferred Shading.

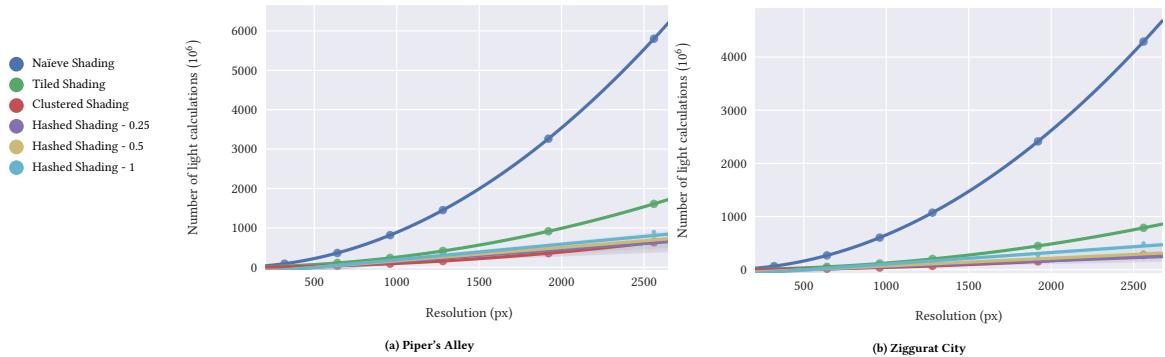


Figure 24: The number of light calculations as function of the resolution.

Bibliografie

- [AMHH16] T. Akenine-Möller, E. Haines, and N. Hoffman. *Real-Time Rendering, Third Edition*. CRC Press, 2016.
- [And09] Johan Andersson. Parallel graphics in frostbite-current & future. *SIGGRAPH Course: Beyond Programmable Shading*, 2009.
- [BE08] Christophe Balestra and Pål-Kristian Engstad. The technology of uncharted: Drakes fortune. In *Game Developer Conference*, 2008.
- [Ben75] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.
- [BOA09] Markus Billeter, Ola Olsson, and Ulf Assarsson. Efficient stream compaction on wide simd many-core architectures. In *Proceedings of the conference on high performance graphics 2009*, pages 159–166. ACM, 2009.
- [Bou29] Pierre Bouguer. *Essai d'optique sur la gradation de la lumière*. chez Claude Jombert, rue S. Jacques, au coin de la rue des Mathurins, à l'Image Notre-Dame, 1729.
- [Bun01] Ensemble Bungie. Studios and 343 industries.”. *Halo.”Microsoft Studios*, 2014, 2001.
- [CHM97] Zbigniew J Czech, George Havas, and Bohdan S Majewski. Perfect hashing. *Theoretical Computer Science*, 182(1-2):1–143, 1997.
- [CJC⁺09] Myung Geol Choi, Eunjung Ju, Jung-Woo Chang, Jehee Lee, and Young J Kim. Linkless octree using multi-level perfect hashing. In *Computer Graphics Forum*, volume 28, pages 1773–1780. Wiley Online Library, 2009.
- [CON08] Marc Christie, Patrick Olivier, and Jean-Marie Normand. Camera control in computer graphics. *Computer Graphics Forum*, 27(8):2197–2218, 2008.
- [Cor09] Thomas H Cormen. *Introduction to algorithms*. MIT press, 2009.

- [CTM13] Petrik Clarberg, Robert Toth, and Jacob Munkberg. A sort-based deferred shading architecture for decoupled sampling. *ACM Trans. Graph.*, 32(4):141:1–141:10, July 2013.
- [DWS⁺88] Michael Deering, Stephanie Winner, Bic Schediwy, Chris Duffy, and Neil Hunt. The triangle processor and normal vector shader: a vlsi system for high performance graphics. In *ACM SIGGRAPH Computer Graphics*, volume 22, pages 21–30. ACM, 1988.
- [Eng09] Wolfgang Engel. 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.
- [FHCD92] Edward A Fox, Lenwood S Heath, Qi Fan Chen, and Amjad M Daoud. Practical minimal perfect hash functions for large databases. *Communications of the ACM*, 35(1):105–121, 1992.
- [FPE⁺89] Henry Fuchs, John Poulton, John Eyles, Trey Greer, Jack Goldfeather, David Ellsworth, Steve Molnar, Greg Turk, Brice Tebbs, and Laura Israel. Pixel-planes 5: a heterogeneous multiprocessor graphics system using processor-enhanced memories. In *ACM Siggraph Computer Graphics*, volume 23, pages 79–88. ACM, 1989.
- [GL10] Kirill Garanzha and Charles Loop. Fast ray sorting and breadth-first packet traversal for gpu ray tracing. In *Computer Graphics Forum*, volume 29, pages 289–298. Wiley Online Library, 2010.
- [Gla88] Andrew S Glassner. *Algorithms for efficient image synthesis*. PhD thesis, University of North Carolina at Chapel Hill, 1988.
- [Gre11] DA Green. A colour scheme for the display of astronomical intensity images. *arXiv preprint arXiv:1108.5083*, 2011.
- [Gut84] Antonin Guttman. *R-trees: a dynamic index structure for spatial searching*, volume 14. ACM, 1984.
- [HA11] Tianyi David Han and Tarek S. Abdelrahman. Reducing branch divergence in gpu programs. In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-4, pages 3:1–3:8, New York, NY, USA, 2011. ACM.
- [Hec16] E. Hecht. *Optics, Global Edition*. Always learning. Pearson Education, Limited, 2016.
- [HPLVdW10] Charles Hollemeersch, Bart Pieters, Peter Lambert, and Rik Van de Walle. Accelerating virtual texturing using cuda. *GPU Pro: advanced rendering techniques*, 1:623–641, 2010.

- [Kaj86] James T. Kajiya. The rendering equation. In *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH '86*, pages 143–150, New York, NY, USA, 1986. ACM.
- [Kar13] Brian Karis. Real shading in unreal engine 4. *Proc. ACM SIGGRAPH Courses*, page 22, 2013.
- [KSDA16] Viktor Kämpe, Erik Sintorn, Dan Dolonius, and Ulf Assarsson. Fast, memory-efficient construction of voxelized shadows. *IEEE transactions on visualization and computer graphics*, 22(10):2239–2248, 2016.
- [KT06] Jon Kleinberg and Eva Tardos. *Algorithm design*. Pearson Education India, 2006.
- [Lau10] Andrew Lauritzen. Deferred rendering for current and future rendering pipelines. *SIGGRAPH Course: Beyond Programmable Shading*, pages 1–34, 2010.
- [Len02] Eric Lengyel. The mechanics of robust stencil shadows. <http://www.gamasutra.com>, 2002.
- [LH06] Sylvain Lefebvre and Hugues Hoppe. Perfect spatial hashing. In *ACM Transactions on Graphics (TOG)*, volume 25, pages 579–588. ACM, 2006.
- [LSO07] Aaron E. Lefohn, Shubhabrata Sengupta, and John D. Owens. Resolution-matched shadow maps. *ACM Trans. Graph.*, 26(4), October 2007.
- [Lue03] David P Luebke. *Level of detail for 3D graphics*. Morgan Kaufmann, 2003.
- [Mag11] Kenny Magnusson. Lighting you up in battlefield 3. In *Proc. 25th Annual Game Developers Conference*, 2011.
- [Mea82] Donald Meagher. Geometric modeling using octree encoding. *Computer graphics and image processing*, 19(2):129–147, 1982.
- [Mit09] Martin Mittring. A bit more deferred-cryengine 3. In *Triangle Game Conference*, volume 4, 2009.
- [MM12] Michael Mara and Morgan McGuire. 2d polyhedral bounds of a clipped, perspective-projected 3d sphere. *JCGT. in submission*, 5, 2012.
- [Nay98] Bruce F Naylor. A tutorial on binary space partitioning trees. In *Computer Games Developer Conference Proceedings*, pages 433–457, 1998.

- [Nic65] Fred E. Nicodemus. Directional reflectance and emissivity of an opaque surface. *Appl. Opt.*, 4(7):767–775, Jul 1965.
- [OA11] Ola Olsson and Ulf Assarsson. Tiled shading. *Journal of Graphics, GPU, and Game Tools*, 15(4):235–251, 2011.
- [OBA12] Ola Olsson, Markus Billeter, and Ulf Assarsson. Clustered deferred and forward shading. In *Proceedings of the Fourth ACM SIGGRAPH/Eurographics conference on High-Performance Graphics*, pages 87–96. Eurographics Association, 2012.
- [OSK⁺14] Ola Olsson, Erik Sintorn, Viktor Kämpe, Markus Billeter, and Ulf Assarsson. Efficient virtual shadow maps for many lights. In *Proceedings of the 18th Meeting of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D ’14, pages 87–96, New York, NY, USA, 2014. ACM.
- [PD14] Aras Pranckevičius and Rendering Dude. Physically based shading in unity. In *Game Developer’s Conference*, 2014.
- [Per85] Ken Perlin. An image synthesizer. *ACM Siggraph Computer Graphics*, 19(3):287–296, 1985.
- [Per13] Emil Persson. Practical clustered shading. *SIGGRAPH Course: Advances in Real-Time Rendering in Games*, 2013.
- [PJH16] M. Pharr, W. Jakob, and G. Humphreys. *Physically Based Rendering: From Theory to Implementation*. Elsevier Science, 2016.
- [Sch11] Boris Schling. *The Boost C++ Libraries*. XML Press, 2011.
- [SHG09] Nadathur Satish, Mark Harris, and Michael Garland. Designing efficient sorting algorithms for manycore gpus. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–10. IEEE, 2009.
- [SSS74] Ivan E. Sutherland, Robert F. Sproull, and Robert A. Schumacker. A characterization of ten hidden-surface algorithms. *ACM Comput. Surv.*, 6(1):1–55, March 1974.
- [ST90] Takafumi Saito and Tokiichiro Takahashi. Comprehensible rendering of 3-d shapes. In *ACM SIGGRAPH Computer Graphics*, volume 24, pages 197–206. ACM, 1990.
- [Suf07] K. Suffern. *Ray Tracing from the Ground Up*. Taylor & Francis, 2007.
- [SW11] Robert Sedgewick and Kevin Wayne. *Algorithms*. Addison-Wesley Professional, 4th edition, 2011.

- [SWBG06] Christian Sigg, Tim Weyrich, Mario Botsch, and Markus H Gross. Gpu-based ray-casting of quadratic surfaces. In *SPBG*, pages 59–65, 2006.
- [Swo09] Matt Swoboda. Deferred lighting and post processing on playstation 3. In *Game Developer Conference*, 2009.
- [Tat16] Natalya Tatarchuk. Advances in real-time rendering, part ii. In *ACM SIGGRAPH 2016 Courses*, SIGGRAPH ’16, New York, NY, USA, 2016. ACM.
- [TNE⁺89] Brice Tebbs, Ulrich Neumann, John Eyles, Greg Turk, and David Ellsworth. Parallel architectures and algorithms for real-time synthesis of high quality images using deferred shading. Technical report, DTIC Document, 1989.
- [Val09] Michal Valient. The rendering technology of killzone 2. In *Game Developers Conference*, 2009.
- [War69] John E Warnock. A hidden surface algorithm for computer generated halftone pictures. Technical report, UTAH UNIV SALT LAKE CITY DEPT OF COMPUTER SCIENCE, 1969.
- [Wat70] Gary S Watkins. A real time visible surface algorithm. Technical report, DTIC Document, 1970.
- [WKL⁺15] Jeremy M Wolfe, Keith R Kluender, Dennis M Levi, Linda M Bartoshuk, Rachel S Herz, Roberta L Klatzky, Susan J Lederman, and Daniel M Merfeld. *Sensation & perception*. Sinauer Sunderland, MA, 4 edition, 2015.

Fiche masterproef

Student: Martinus Wilhelmus Tegelaers

Titel: Forward en Deferred Hashed Shading voor het Realtime Renderen van Grote Aantallen Lichtbronnen

Engelse titel: Forward and Deferred Hashed Shading for Real-time Rendering of Many Lights

UDC: 681.3

Korte inhoud:

In deze thesis wordt een nieuw lichttoekenningsalgoritme geïntroduceerd in de context van realtime rendering op basis van de verbindingloze octree. Het doel hiervan is om een betere performantie te verkrijgen dan de huidige lichttoekenningsalgoritmes Tiled en Clustered Shading, door de opgestelde datastructuren tussen frames te hergebruiken. Hiervoor wordt de ruimte opgedeeld aan de hand van een een camera-onafhankelijke octree. Veranderingen in de scène zijn relatief klein tussen frames, waardoor de kost van het aanpassen van de datastructuur ook klein is. Om de performantie van dit nieuwe algoritme te evalueren is gekeken naar de uitvoeringstijd en het aantal lichtberekeningen per frame bij verschillende scènes, resoluties en aantallen lichtbronnen. Ter referentie zijn ook de lichttoekenningsalgoritmes Tiled en Clustered Shading geïmplementeerd binnen hetzelfde programma. De resultaten van deze implementaties zijn vergeleken. De Hashed Shading implementatie vereist de helft van het aantal lichtberekeningen per frame ten opzichte van Tiled Shading. Hierdoor is de Forward Hashed Shading implementatie een factor twee sneller dan Forward Tiled Shading. De beperking in het aantal lichtberekeningen met Hashed Shading met de kleinst geëvalueerde knopen komt overeen met de reductie van het aantal lichtberekeningen behaald binnen Clustered Shading. Daarnaast schaalt Hashed Shading in kleine mate beter met de resoluties, doordat opdeling hier niet direct afhankelijk van is. Hashed Shading vereist daarnaast geen complete herberekening van de datastructuren bij een verandering van het camerastandpunt, waardoor de kost ten opzichte van Tiled en Clustered Shading afneemt. Op basis hiervan kan gesteld worden dat het doel van deze thesis bereikt is. Er zijn echter nog wel enkele beperkingen in de huidige implementatie van Hashed Shading.

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

Promotor: Prof. dr. ir. Philip Dutré

Assessor: Ir. Tuur Stuyck

Dr. Dominique Devriese

Begeleider: Ir. Matthias Moulin

Ir. Jeroen Baert