Auteursrechterlijke overeenkomst

Opdat de Universiteit Hasselt uw eindverhandeling wereldwijd kan reproduceren, vertalen en distribueren is uw akkoord voor deze overeenkomst noodzakelijk. Gelieve de tijd te nemen om deze overeenkomst door te nemen, de gevraagde informatie in te vullen (en de overeenkomst te ondertekenen en af te geven).

Ik/wij verlenen het wereldwijde auteursrecht voor de ingediende eindverhandeling met

Titel: Frameless Rendering
Richting: master in de informatica - multimedia Jaar: 2008

in alle mogelijke mediaformaten, - bestaande en in de toekomst te ontwikkelen - , aan de Universiteit Hasselt.

Niet tegenstaand deze toekenning van het auteursrecht aan de Universiteit Hasselt behoud ik als auteur het recht om de eindverhandeling, - in zijn geheel of gedeeltelijk -, vrij te reproduceren, (her)publiceren of distribueren zonder de toelating te moeten verkrijgen van de Universiteit Hasselt.

Ik bevestig dat de eindverhandeling mijn origineel werk is, en dat ik het recht heb om de rechten te verlenen die in deze overeenkomst worden beschreven. Ik verklaar tevens dat de eindverhandeling, naar mijn weten, het auteursrecht van anderen niet overtreedt.

Ik verklaar tevens dat ik voor het materiaal in de eindverhandeling dat beschermd wordt door het auteursrecht, de nodige toelatingen heb verkregen zodat ik deze ook aan de Universiteit Hasselt kan overdragen en dat dit duidelijk in de tekst en inhoud van de eindverhandeling werd genotificeerd.

Universiteit Hasselt zal mij als auteur(s) van de eindverhandeling identificeren en zal geen wijzigingen aanbrengen aan de eindverhandeling, uitgezonderd deze toegelaten door deze overeenkomst.

Ik ga akkoord,

SEGERS, Ief

Datum: 5.11.2008

Frameless Rendering

Ief Segers

promotor :
Prof. dr. Philippe BEKAERT

Eindverhandeling voorgedragen tot het bekomen van de graad
master in de informatica multimedia

universiteit hasselt

Inhoudsopgave

Samenvatting

Interactieve visualisatie is lange tijd het domein geweest van rasterisatie, terwijl raytracingalgoritmen voornamelijk gebruikt werden om individuele hoge kwaliteitsbeelden te genereren. Rasterisatie levert slechts een sterke benadering af van het fotorealisme dat mogelijk is met raytracingalgoritmen. Het maakt daarbij gebruik van allerlei trucs of het integreert exacte resultaten door die op voorhand te berekenen met bijvoorbeeld een raytracingalgoritme. Die resultaten kloppen gewoonlijk slechts voor een aantal cameraposities. Er is meestal ook veel werk van de persoon vereist, die realistische resultaten tracht te bereiken met rasterisatie. Raytracing biedt een mooi kader, waarbinnen realisme eenvoudig te berekenen valt. Spijtig genoeg is het veel duurder om te berekenen.

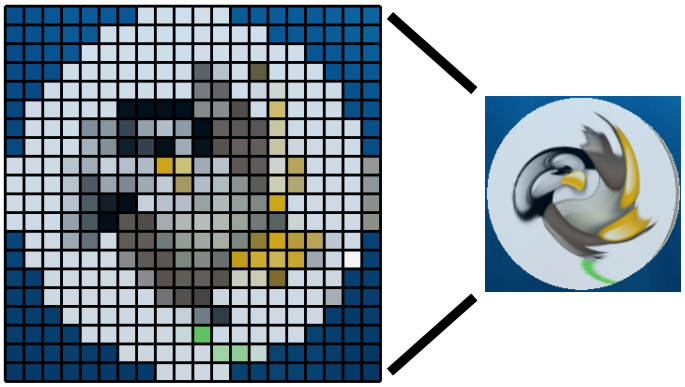
Doch begint sinds kort de mogelijkheid tot interactieve raytracing sterk dichterbij te komen. Voor eenvoudige scenes is het zelfs reeds het geval. Wanneer genoeg computers genetwerkt worden, de reflecties niet te diep gevolgd worden, de beeldresolutie niet te groot is en er niet teveel animatie in de scene aanwezig is, kan raytracing reeds interactief getoond worden. Aangezien het duidelijk nog een aantal obstakels weg te werken heeft, zijn er reeds veel technieken uitgedacht die raytracing proberen te versnellen. Ofwel versnellen ze het algoritme zelf, ofwel proberen ze het te benaderen via bijvoorbeeld frameless rendering. Dat paradigma heeft aangetoond dat het niet nodig is om telkens een heel beeld te vernieuwen. Tijdens statische scenes en camera's blijven de kleurwaarden op het beeld gelijk en het is een verspilling om dezelfde kleurwaarden herhaaldelijk te berekenen. Zelfs tijdens animaties zullen er vaak een heel deel pixels gelijk blijven. De mens heeft trouwens nood aan geleidelijke overgangen om bijvoorbeeld de kans te krijgen objecten of bewegingen te herkennen. Die geleidelijke overgangen houden temporele coherentie in, ook al is die vrij klein.

In deze thesis wordt gestart met een korte geschiedenis en een vergelijking tussen rasterisatie en raytracing. Vervolgens komt het hoofdonderwerp frameless rendering aan bod. Dat wordt uitgelegd en verschillende richtingen tot verbeteringen worden aangegeven. Daarna volgt een uitleg over enkele uitgebreide technieken, die gebruik maken van het concept frameless rendering. Enkele van deze technieken worden uitgekozen voor implementatie. Met behulp van die implementaties worden resultaten gegenereerd. Daaruit blijkt dat interactieve visualisatie mogelijk is in gevallen waar het niet mogelijk is wanneer het met de standaard techniek, namelijk dubbele buffering, gedaan wordt.

Hoofdstuk 1

Inleiding

Momenteel is er een tweestrijd tussen raytracing enerzijds en rasterisatie anderzijds [5, 93, 102, 68, 39]. Rasterisatie is tot nu toe steeds de heersende techniek geweest voor allerlei interactieve grafische toepassingen, zoals bijvoorbeeld computerspelletjes. Beide technieken dienen om drie dimensionele (3D) scenes op een twee dimensioneel (2D) computerscherm te tonen. Een dergelijk beeld bestaat uit een rechthoekig rooster van pixels. Dat zijn discrete punten, die verticaal en horizontaal gealigneerd staan en elk hun eigen lichtbron bevatten, zoals bijvoorbeeld een led bij een CRT-scherm.

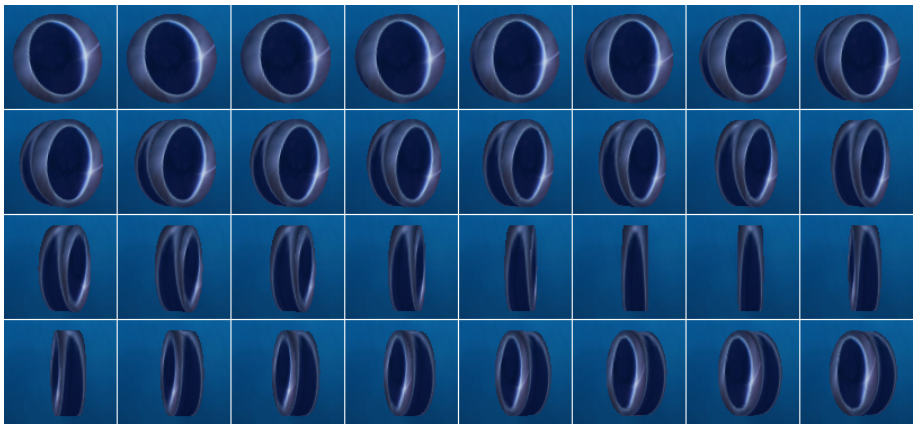


Figuur 1.1: Een beeld met een resolutie van 20×20 bevat 400 pixels.

Om nu *één* volledig beeld op te bouwen, moet aan elke pixel een bepaalde kleur toegekend worden. Dat gebeurt met het proces visualisatie, waarbij berekeningen gedaan worden op een 3D-model. Zo'n model is een beschrijving van de scene, welke informatie bevat over de geometrische objecten, texturen, illuminatie-informatie, het kijkpunt (camera) en informatie over hoe de illuminatie interageert met de geometrie. Dit wordt verder uiteengezet in de sectie over visualisatie 2.2.

Animatie kan bekomen worden door meerdere opeenvolgende beelden (frames) van de scene te genereren, waarbij de objecten of andere parameters van de beschrijving telkens aangepast worden. Indien de beelden snel genoeg getoond worden en de veranderingen bijvoorbeeld kleine opeenvolgende verschillen in de

positie van een object zijn, zal de illusie ontstaan dat het object een continue beweging ondergaat.



Figuur 1.2: Voorbeeld animatie beelden

Het is zelfs cruciaal dat de beelden snel en accuraat getoond worden. Fouten in de weergave of vertragingen leiden tot degradaties in de onderdompeling van de gebruiker in de scene. Hoe beter de visualisatie overkomt bij de gebruiker, hoe beter hij zich kan inleven in de voorgestelde omgeving en hoe minder snel hij afgeleid zal worden of gedesoriënteerd zal geraken.

Interactiviteit betekent dat mensen invoer kunnen doorgeven aan het programma, welke hierop zal reageren met een bepaalde uitvoer. Gewoonlijk zullen bepaalde eigenschappen van het 3D-model aangepast worden. Voor realtime applicaties is het cruciaal dat de applicatie snel reageert, anders zal de gebruiker het gevoel krijgen dat hij geen controle heeft over de applicatie.

Interactieve toepassingen vereisen dus realtime beeldvernieuwingen en realtime invoer verwerking. Aangezien rasterisatie in het geval van relatief kleine scenes een hoger aantal beelden per seconde kon genereren dan raytracing, werd dit gewoonlijk gebruikt in interactieve toepassingen. Het wordt daarom ook al geruime tijd standaard gebruikt op de grafische hardware die in gewone massaproductiecomputers aanwezig is. Wanneer het aantal beelden per seconde nog te laag is, kunnen allerlei benaderingen toegepast worden of kunnen minder belangrijke delen van het beeld weggelaten worden om alles tijdig berekend te krijgen.

Bij de toepassingen die geen interactieve snelheden vereisen, zoals bijvoorbeeld bij het creëren van een film, is de kwaliteit van elk individueel beeld van het grootste belang. In dergelijke gevallen werd gewoonlijk een globaal illuminatie-algoritme, zoals raytracing, toegepast. De kwaliteit van de beelden wordt dankzij deze algoritmen nogmaals versterkt, doordat ze allerlei realistische effecten kunnen tonen, zoals schaduwen, reflecties, refracties, enzovoort. Deze eigenschappen zijn een onderdeel van de fysisch correcte en fotorealistische illuminatieberekeningen van raytracing [102]. Dat is niet het geval bij rasterisatie. In sectie 2.2 wordt dit verder uiteengezet.

Buiten raytracing zijn er nog meerdere technieken die accurate en fysisch correcte beelden genereren, zoals pathtracing [52], radiosity [44], photon mapping [50, 49], metropolis light transport [94], etcetera. Dit zijn allemaal globale

illuminatie-algoritmen, welke veel tijd vragen om te berekenen.

Rasterisatie hardware is momenteel nog steeds de snelste methode voor relatief kleine scenes [103]. Dit is grotendeels te danken aan het feit, dat het geen extra berekeningen moeten doen voor het opbouwen van een versnellingsstructuur. Maar hoe groter het aantal primitieven in de scene wordt, hoe beter die kost afgelost wordt. De primitieven worden dan namelijk zo opgeslagen in een structuur, dat ze sneller teruggevonden kunnen worden. Deze zoekfunctie scaleert logaritmisch met het aantal primitieven in de scene, terwijl er zich een lineaire scalering voordoet wanneer geen structuur gebruikt wordt. Vanaf ongeveer *één* miljoen primitieven is het zelfs voordeliger om een scene te visualiseren met een raytracer, voorzien van een versnellingsstructuur. Dat overgangspunt zal steeds dichterbij komen, naarmate de hardware voortdurend sneller wordt. Ook is er een stijgende vraag naar grotere scenes. De combinatie van deze eigenschappen en andere voordelen 2.2.4, kunnen ervoor zorgen dat raytracing binnenkort wel eens de leidende techniek zal worden.

Zolang dat dit niet het geval is, moeten we genoegen nemen met benaderde resultaten [102] indien we interactiviteit wensen. Er wordt volop gezocht naar manieren om raytracing te versnellen, zodat het overgangspunt misschien eerder bereikt kan worden. Frameless Rendering [14] is een dergelijke methode om onder meer het raytracen van 3D-scenes te versnellen. Het tracht dit te bereiken door het werk van het visualisatie algoritme te verlichten en zijn inspanningen enkel te richten op plaatsen in het beeld waar veranderingen het meest noodzakelijk zijn. Op die manier wordt er niet telkens een heel beeld pixel per pixel opgebouwd, maar worden enkel de pixels, die vernieuwd moeten worden, herberekend en ingevoegd in de reeds berekende resultaten. Het probleem, eerst bestaande uit het periodiek raytracen van alle pixels in een beeld, is bijgevolg gereduceerd tot het raytracen van bijvoorbeeld een achtste van het aantal pixels in het beeld, plus de extra kosten voor de frameless rendering code. Het aantal pixels dat telkens nog berekend moet worden, hangt af van techniek tot techniek. Deze technieken worden in sectie 2.3 toegelicht.

De rest van deze thesis bestaat uit de volgende onderdelen. Hoofdstuk twee 2 geeft een korte gescheidenis over visualisatie en bevat informatie over visualisatie en frameless rendering. Het deel visualisatie 2.2 gaat dieper in op de twee belangrijkste visualisatie technieken, namelijk rasterisatie 2.2.1 en raytracing 2.2.2. Het licht hun basis algoritmen en mogelijke uitbreidingen toe, waarna het de voor- en nadelen van beide uiteenzet. In het deel frameless rendering 2.3 komt het hoofdthema aan bod. Eerst schetst het waarom de techniek onderzocht wordt, waarna het verschillende aanpakken uitlegt. In hoofdstuk drie 3 volgen dan de eigen implementaties, terwijl hoofdstuk vier 4 de verkregen resultaten behandelt. Hoofdstuk vijf 5 maakt een besluit op en geeft mogelijke richtingen voor verder onderzoek aan.

Hoofdstuk 2

Literatuurstudie

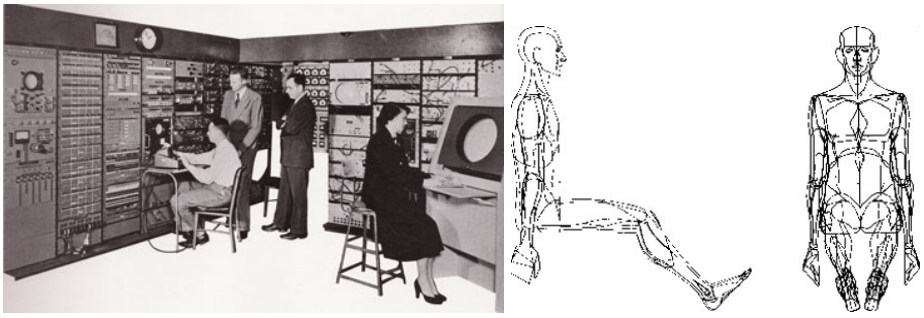
De literatuurstudie bestaat uit drie onderdelen. Het eerste onderdeel bestaat uit een korte geschiedenis van het domein visualisatie en computergrafieken 2.1. De twee belangrijke visualisatie algoritmen, rasterisatie en raytracing, komen aan bod in sectie 2.2. Tenslotte legt de laatste sectie 2.3 uit waarom frameless rendering nodig is in het domein visualisatie en licht het de techniek verder toe.

2.1 Achtergrond

Sectie 2.1.1 haalt de evolutie van visualisatie en computergrafieken kort aan. Het schetst hoe rasterisatie en raytracing ontstaan zijn en wat ze tegenwoordig te bieden hebben. De tweede sectie 2.1.2 geeft een blik op de toekomst.

2.1.1 Geschiedenis en onderdeel van computergrafieken

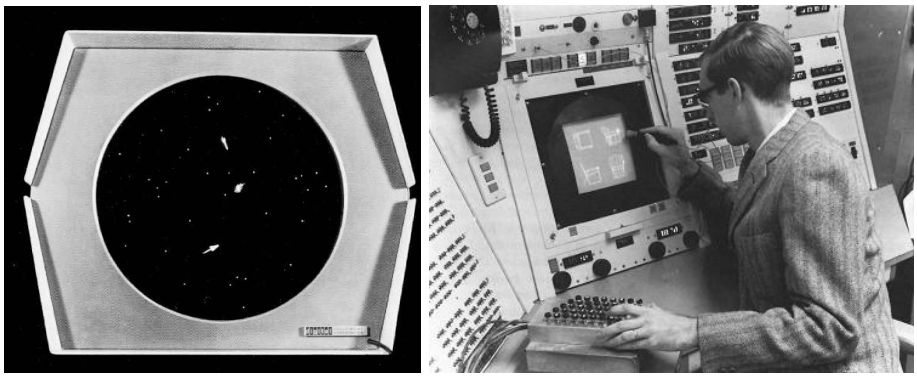
Frameless rendering is in leven geroepen om raytracing of andere pixel-gebaseerde illuminatietechnieken te versnellen. Het is een onderdeel van het domein visualisatie, wat op zijn beurt behoort tot het domein computergrafieken. Dat laatste domein omvat het samenstellen en verwerken van beelden met behulp van computers. Hieronder valt onder meer beeldverwerking en visualisatie van 3D-omgevingen.



Figuur 2.1: (a) Whirlwind computer (b) William Fetter's model van een mens

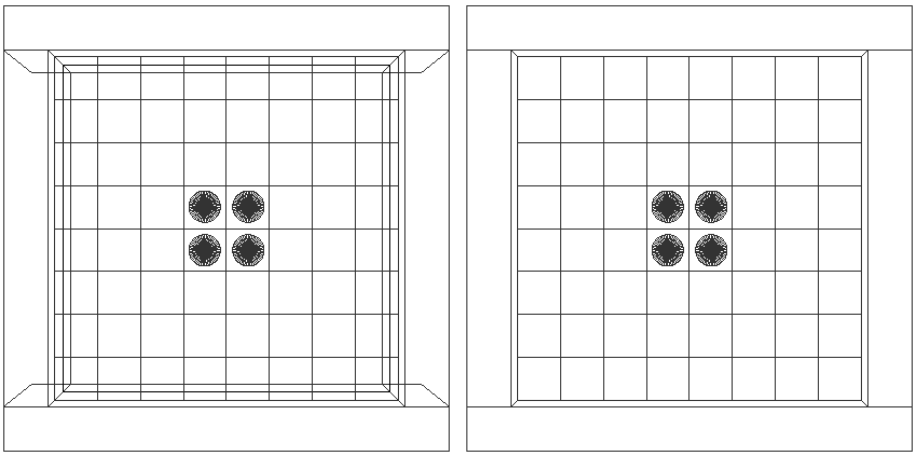
Computergrafieken bestaat al een geruime tijd, wat deze sectie kort zal schetsen. Meer informatie is te vinden op enkele sites [21, 84, 82, 108] en in het volgende boek [36]. Figuren 2.1 en 2.2 zijn komen van [21].

De Whirlwind computer was de eerste computer die realtime beelden en tekst als uitvoer op een scherm toonde. Deze werd rond 1945 ontwikkeld voor het simuleren van vluchten, luchtverkeerscontrole, en zo meer. In 1960 werd de term computergrafieken dan voor het eerst gebruikt voor schetsen van het menselijk lichaam, die gemaakt werden door William Fetter op een computerscherm. Daarop volgde de creatie van een eerste computerspel (Spacewar), een computer-geanimeerde film, enzovoort.



Figuur 2.2: (a) Spacewar (b) Sketchpad

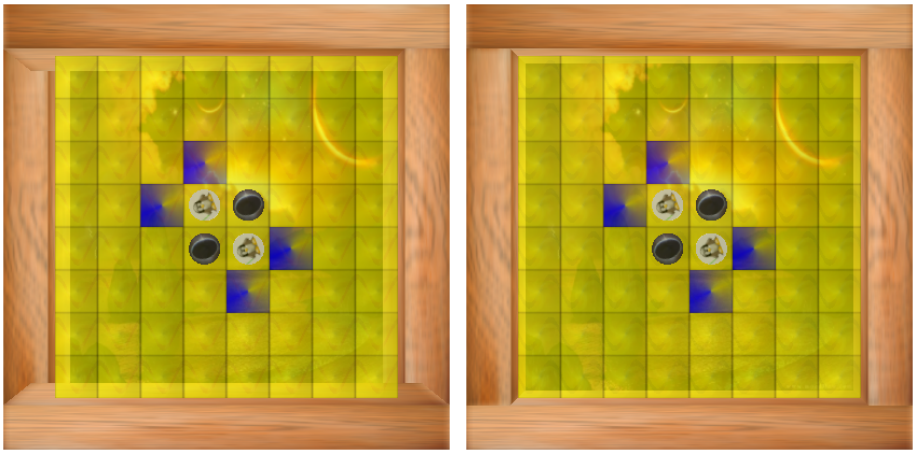
Interactieve computergrafieken vond dan zijn oorsprong, toen Ivan Sutherland in 1963 Sketchpad ontwikkelde. Met dit programma werd het mogelijk om interactief een beeld van eenvoudige 3D-objecten op een beeldscherm te creëren door middel van een lichtpen.



Figuur 2.3: Hidden line removal: (a) Niet toegepast (b) Toegepast

De eerste visualisatieprogramma's hadden de capaciteit om dergelijke objec-

ten op het scherm voor te stellen als draadmodellen. In deze draadmodellen werden de oppervlakken van objecten tussen de lijnen niet opgevuld. Dankzij hidden line removal worden lijnen, die bedekt zijn door oppervlakken tussen andere lijnen, niet getekend. Roberts was de eerste die een dergelijk algoritme ontwikkelde. Bresenham zorgde ervoor dat deze lijnen zo goed en recht mogelijk getoond werden op een discreet rooster van pixels.



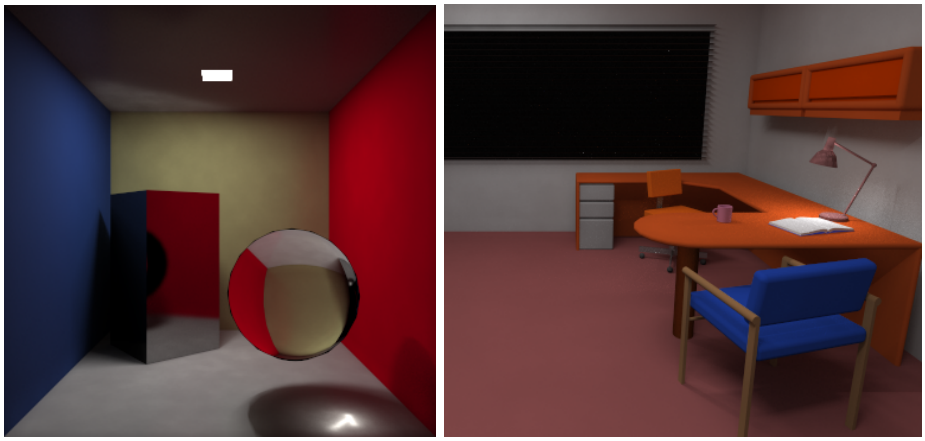
Figuur 2.4: Hidden surface removal: (a) Niet toegepast (b) Toegepast

Later, rond 1967 door onder andere Catmull, Warnock, Watkins en Appel, werden Hidden surface removal algoritmen bedacht, toen men zocht naar methoden om de objecten op te vullen met kleuren. Die algoritmen dienen om de zichtbaarheid op te lossen. Ze zorgen er namelijk voor dat objecten, die helemaal of gedeeltelijk bedekt zijn door andere objecten, op de juiste manier worden weergegeven, zodat ze niet zichtbaar zijn doorheen het object dat hen bedekt. Ook verwijderen ze de zijden van het object die weggekeerd zijn van de kijker. Dergelijke algoritmen zijn bijvoorbeeld depth sorting van Newell et al. [66], het Z-buffer algoritme van Catmull [51] en het algoritme van Warnock [109]. Verdere uitleg hierover is aanwezig in de sectie over rasterisatie 2.2.1.

Nu was het mogelijk ondoorzichtige en dus opgevulde objecten te tonen. Objecten werden in het begin gewoon volledig met *één* kleur opgevuld. Later betrok men de oriëntatie, de afstand tot de camera, en andere zaken bij het toekennen van de kleur. Zo ontstonden dan de shading methoden, zoals bijvoorbeeld Gouraud [45] of Phong shading [72]. Die methoden kunnen ook gebruik maken van reflectie modellen. Ook werd het mogelijk texturen te plaatsen op objecten via texture mapping [23] om detail toe te voegen aan een object, zonder daarvoor meer primitieven te moeten gebruiken. Er kunnen dan patronen gecreëerd worden, zoals het rooster van vakjes in figuur 2.4 of kan het object het uitzicht krijgen van een textuur, bijvoorbeeld hout.

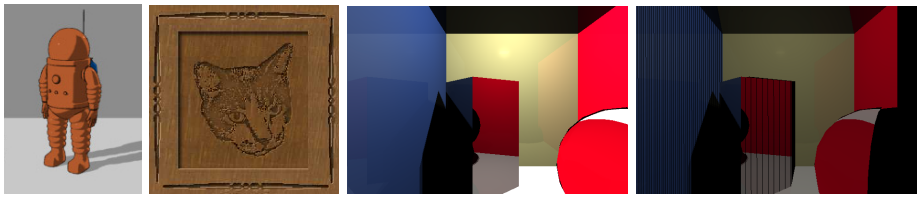
Sindsdien is het toepassingsgebied van computergrafieken blijven uitbreiden en komt visualisatie tegenwoordig voor in allerlei soorten toepassingen: videospelletjes, architecturale programma's, datavisualisatie, visuele simulatie, het creëren van animatie films, het creëren van speciale effecten, computer-aided design, e-commerce, enzoverder.

In de zoektocht naar fotorealistische beelden, werden vanaf 1980 globale illuminatietechnieken, zoals pathtracing [52] en radiosity [44], ontwikkeld. Deze houden rekening met directe en indirecte illuminatie [36]. De sectie over globale illuminatie I.4 zet dit verder uiteen.



Figuur 2.5: Voorbeelden van globale illuminatie

Buiten fotorealistische visualisatie bestaan er ook andere visualisatie domeinen, zoals niet-fotorealistische visualisatie [43] en beeld-gebaseerde visualisatie [1]. In niet-fotorealistische visualisatie is geen perfecte weergave van de realiteit nodig. Men probeert bijvoorbeeld kunststijlen te imiteren, zoals schilderen en houtgravuren. Ook het gebruik van toon shading [33] behoort tot dit domein.



Figuur 2.6: (a) Toon shader (b) Wood engravure [43] (c)-(d) Warping

Beeld-gebaseerde visualisatie gebruikt als invoer geen 3D-modellen, maar genereert de modellen zelf uit ingevoerde 2D-beelden. Ze houden bijvoorbeeld 3D-punten bij, die dan bij de visualisatie met een ander camerastandpunt geherprojecteerd worden. Image warping [63] is een voorbeeld hiervan.

2.1.2 Blik op de toekomst

Raytracing wordt steeds aantrekkelijker voor interactieve visualisatie [102]. Het kon lange tijd niet snel genoeg uitgevoerd worden op de toenmalige beschikbare hardware, waardoor het niet kon doorstoten in interactieve applicaties. Dankzij de sterk verbeterde computerhardware en verbeterde algoritmen, is het tegenwoordig reeds mogelijk interactieve globale illuminatie uit te voeren tegen interactieve snelheden op een netwerk van massaproductiecomputers [99]. Dat

maakt gebruik van een Monte Carlo algoritme en is samengesteld uit meerdere globale illuminatie-algoritmen: een soort instant radiosity, photon mapping en raytracing als kern. Bovendien vertoont raytracing heel wat andere voordelen, zoals aan bod zal komen in sectie 2.2. Dankzij al deze voordelen maakt raytracing veel kans de basis te worden van standaard algoritmen voor interactieve visualisatie of toch minstens een deel van deze basis uit te maken. Het voordeel van rasterisatie zal steeds kleiner worden, naarmate raytracing steeds hogere snelheden bereikt dankzij snellere hardware en verbeterde algoritmen.

Maar voorlopig wordt voor de veel gebruikte en relatief kleine scenes, zoals voor computerspelletjes, nog gebruik gemaakt van hardware rasterisatie, aangezien deze techniek daarvoor een hoger aantal beelden per seconden kan halen. Zodoende moeten we nog een tijdje genoegen nemen met sterk benaderde visualisaties in de meeste interactieve applicaties. Toch is er reeds veel moeite gedaan om ook bij rasterisatie exactere beelden te genereren, bijvoorbeeld met behulp van bump mapping [16], reflection mapping [15], shadow mapping [114], light mapping [48] en ambient occlusion [55]. Er worden verder ook steeds nieuwe technieken ontworpen om rasterisatie sneller te maken, zoals onder andere level-of-detail [26, 41], billboarding [83], occlusion culling [29, 120], cel-gebaseerde occlusion culling [91], hierarchische Z-buffer [46], textured depth-meshes [85, 31] en triangle decimation [81]. De bedoeling is daar het aantal primitieven te verminderen dat doorgegeven wordt aan de grafische kaart.

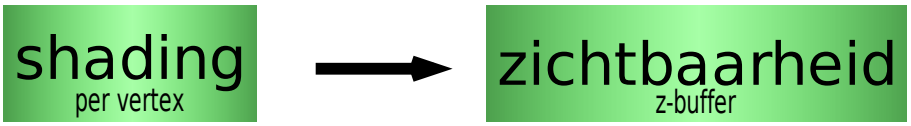
Raytracing wordt ondertussen verder onderzocht op verschillende gebieden. Dit komt aan bod in sectie II over het versnellen van het raytracing algoritme.

2.2 Basis visualisatie

Dit hoofdstuk gaat dieper in op visualisatie technieken. Het doel is 3D-scenes te tonen op een 2D-scherm. Dat zorgt voor een aantal op te lossen problemen, zoals zichtbaarheid en shading. De eerste sectie 2.2.1 gaat in detail in op de techniek, die momenteel uitgevoerd wordt op de huidige grafische kaarten, namelijk rasterisatie. Vervolgens komt raytracing 2.2.2 aan bod. Dat is de techniek die wel eens de basis kan worden van toekomstige hardware. Tenslotte trekt 2.2.4 een conclusie uit beide secties.

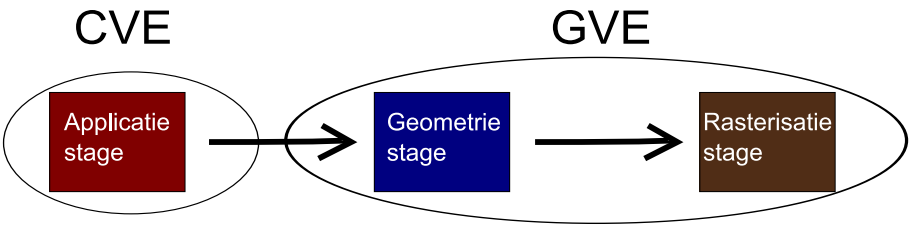
2.2.1 Rasterisatie

Dit is de regerende methode voor interactieve visualisatie van 3D-scenes [102]. Door onder andere Randy et al. [37] wordt de werking van deze methode beschreven. Rasterisatie berekent shading, vooraleer het de zichtbaarheid bepaalt.



De grafische pijplijn behandelt dit probleem door het op te splitsen in verschillende stages. Ook een scene bestaat uit meerdere delen, gewoonlijk driehoeken. Die worden *één* voor *één* door deze pijplijn gestuurd. Omdat deze pijplijn opgesplitst is, moet er niet, telkens een driehoek doorgestuurd wordt, gewacht worden totdat deze afgewerkt is door de hele pijplijn. Er kunnen meerdere

bewerkingen parallel uitvoeren op verschillende driehoeken, omdat als een volgende stage een driehoek behandelt, de voorgaande stage al in gebruik genomen kan worden voor een volgende driehoek.



Deze pijplijn bestaat nu uit drie hoofdstages, zijnde de applicatiestage I, de geometriestage II en de rasterisatiestage III. De geometriestage zorgt ervoor dat de 3D-objecten op een 2D-vlak terechtkomen en berekent voor elke driehoek in de scene per vertex een kleurwaarde met behulp van een lokaal illuminatiemodel. Daaropvolgend zet de rasterisatiestage de driehoeken om in pixels via scanconversie en bepaalt het hun zichtbaarheid met de Z-buffer. Pixels, die minder diep liggen dan andere pixels, worden behouden en de anderen worden overschreven. De centrale verwerkingseenheid (CVE) voert de applicatiestage uit, terwijl de grafische verwerkingseenheid (GVE) de andere stages uitvoert.

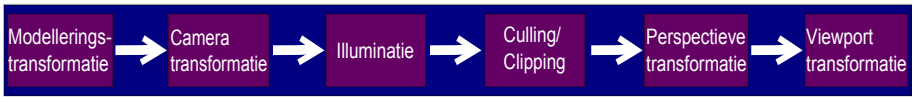
I Applicatiestage

De 3D-applicatie start met een beschrijving van de scene op hoog niveau. Die bevat informatie over primitieven, hun transformaties in de wereld en hun oppervlakeigenschappen. Het specifieert een camera, welke aangeeft waar de kijker zich in de scene bevindt, waarheen hij kijkt en waarheen zijn bovenkant gericht is. Deze camera bepaalt tevens een kijkvolume. Tenslotte bevat het nog informatie over de lichtbronnen. De objecten, camera, etcetera worden gespecifieerd in Cartesiaanse coördinaten en OpenGL maakt gebruik van het rechtshandige coördinatensysteem. Per beeld past de applicatie, indien er veranderingen gebeurd zijn, de camera- en objecteigenschappen aan, zodat animaties en interacties mogelijk worden met de applicatie. In de applicatie worden verder ook effecten berekend, die invloed hebben op deze eigenschappen, zoals bijvoorbeeld de detectie van botsingen tussen objecten, andere krachten die uitgeoefend worden op de objecten en artificiële intelligentieberekeningen. Ook alternatieve verwerking is nodig, bijvoorbeeld voor geluidsterugkoppeling, het doorsturen van gegevens over het netwerk, en zo meer. Vereenvoudigingen van de scene kunnen hier eveneens toegepast worden, zoals reeds eerder vermeld is 2.1.2. Die zorgen ervoor dat het rasterisatie algoritme minder werk zal moeten verrichten, aangezien er minder driehoeken zullen overschieten. Daardoor zal het sneller uitgevoerd worden.

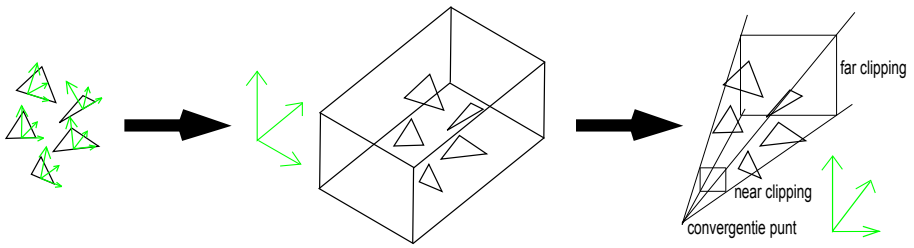
Als primitieven worden standaard driehoeken genomen en hun hoekpunten worden vertices genoemd. Driehoeken worden, in de beschrijving op hoog niveau, voorgesteld door die vertices en hun begeleidende eigenschappen. De

drie vertices hebben elk hun eigen 3D-coördinaten, rgba-waarden, materiaaleigenschappen, textuur- en normaalvectoren. Elke driehoek start in zijn lokale coördinatensysteem en wordt via de PCI-E poort doorgegeven van het systeemgeheugen aan het videogeheugen van de GVE. Dit is de verbinding tussen het moederbord en de GVE. Zodoende kan de geometriestage deze driehoeken en hun eigenschappen gebruiken voor verdere verwerking.

II Geometriestage

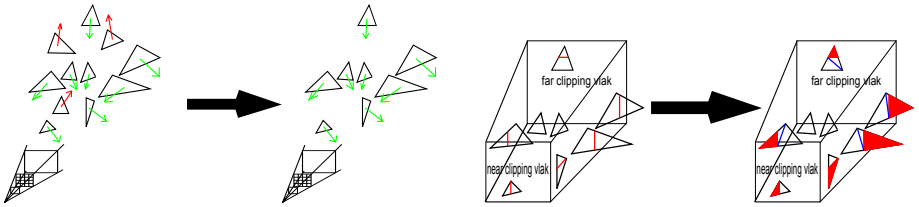


De geometriestage doet per-vertex operaties. De lokale 3D-coördinaten van elke vertex worden getransformeerd naar schermcoördinaten of 2D-coördinaten. Deze transformatie houdt een modellerings-, een camera- en een projectietransformatie in. De modelleringstransformatie dient om objecten een bepaalde positie en oriëntatie te geven in de 3D-wereldruimte. Dat gebeurt door de oorspronkelijke lokale 3D-coördinaten van de driehoek te transleren, te scaleren en/of te roteren. Ook past het de normaalvectoren en de textuurcoördinaten aan.



Figuur 2.7: Lokale coördinaten, wereldcoördinaten, cameracoördinaten

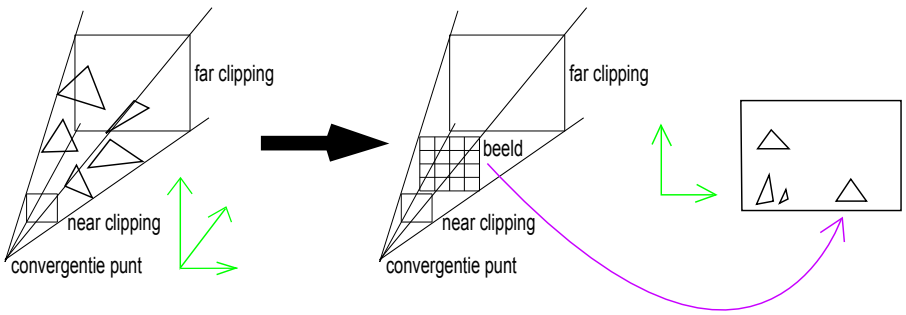
De cameratransformatie bepaalt dan hoe de camera geplaatst wordt en hoe hij gericht wordt, zodat de scene terechtkomt in cameraruimte. Dus de scene wordt gezien vanuit het standpunt van de camera. Alle driehoeken die aanwezig zijn in de scene bevatten twee zijden. Hiervan zal er telkens slechts *één* zichtbaar zijn. Daarom wordt backface culling toegepast, die de zijde wegknipt die weg van de camera wijst. Dit wordt bepaald aand de hand van de normaal van die primitieve en de richting waarin de camera kijkt.



Figuur 2.8: (a) Backface culling (b) View frustum culling

Vervolgens berekent de geometriestage de illuminatie op alle vertices van de driehoek, gebruikmakend van de aangepaste normaalvector en getransformeerde 3D-coördinaten van de vertices. Dat gebeurt in het geval van Gouraud shading. Indien flat shading gebruikt wordt, is er slechts *één* kleurwaarde nodig per driehoek. De berekeningen gebeuren volgens de aanwezige lichtbronnen, de materiaaleigenschappen van de driehoek en het illuminatiemodel. Dat is bijvoorbeeld het Phong reflectiemodel. Het materiaal bevat eigenschappen zoals een basiskleur (ambient), een speculaire reflectie en een diffuse reflectie. Deze waarden zullen later geïnterpoleerd worden tijdens de rasterisatiestage. Indien dynamische textuurcoördinaten vereist zijn, bijvoorbeeld voor environment mapping, worden deze daaropvolgend gegenereerd.

Daarna geschiedt de samenstelling van de primitieven. Hier gebeurt voornamelijk het verzamelen van zichtbare objecten en het wegknippen van punten, lijnen en primitieven bij objecten die gedeeltelijk zichtbaar zijn. De belangrijkste methode is view frustum culling. Delen, die buiten een bepaalde ruimte (het kijkvolume) liggen, worden weggeknipt aangezien deze toch niet zichtbaar zullen zijn. Het is onnodig en verspillend om hierop verdere bewerkingen te doen. Driehoeken, die gedeeltelijk weggeknipt worden, kunnen resulteren in vierhoeken. Deze worden terug gesplitst in driehoeken. Dit cullingprobleem kan aangepakt worden met behulp van een octree, in plaats van elke primitieve te testen tegen alle zes vlakken. Dat laatste betekent een lineaire complexiteit, terwijl een octree de ruimte telkens volgens elke as in acht kleinere delen onderverdeelt totdat aan een bepaald stopcriterium voldaan wordt. Deze hiërarchische onderverdeling zorgt voor een logaritmische tijdscomplexiteit. De hiërarchie wordt van boven naar onder doorlopen en wanneer een deel buiten het kijkvolume valt, worden al zijn kinderen weggegooid.



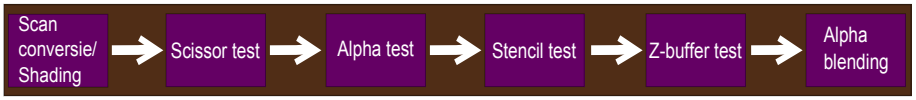
Figuur 2.9: Cameracoördinaten naar 2D-coördinaten

De projectietransformatie zet de geometrie om van cameraruimte naar 2D-beeldruimte. De 3D-scene wordt dus op een 2D-vlak geprojecteerd, waarbij dat vlak het zichtpunt van de camera aangeeft. Indien de perspectieve transformatie gebruikt wordt, volgt er nog een perspectieve deling van x, y en z door de w coördinaat om over te gaan naar Cartesiaanse genormaliseerde schermcoördinaten. Nu liggen de waarden op alle drie assen in het bereik van min *één* tot *één*.

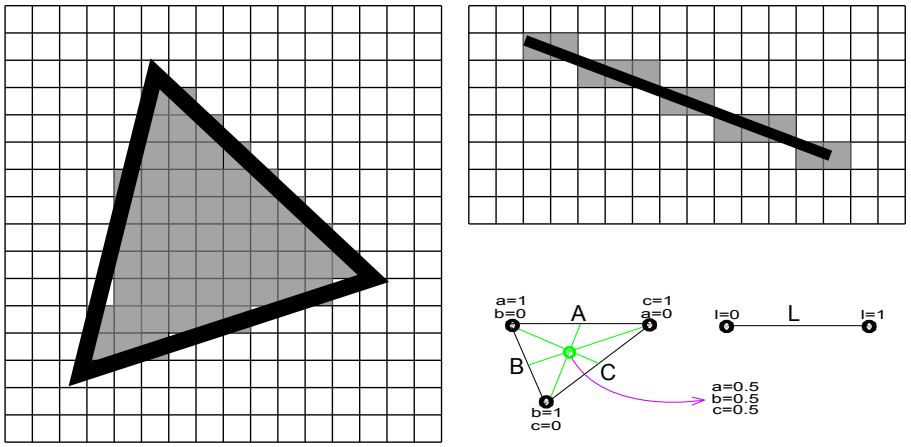
Tenslotte is er nog de viewport-transformatie. Deze staat in voor het transformeren van de 2D-schermcoördinaten naar 2D-beeldcoördinaten. Dat gebeurt met een scalering en een translatie. Het programma is zichtbaar in een ven-

ster op het scherm en het viewport geeft aan welk gebied daarvan in gebruik genomen mag worden om het kijkvolume in 2D op te visualiseren. Standaard wordt heel het venster gebruikt. De aspect ratio van het viewport kan het beste overeenkomen met die van het kijkvolume, zodat de aanwezige objecten niet uitgerokken of ingedrukt worden langs slechts *één* as.

III Rasterisatiestage



Nu al deze per-vertex operaties afgelopen zijn, kunnen de per-pixel operaties beginnen in de rasterisatiestage. Die zet elke overgebleven driehoek *één* voor *één* om in een verzameling pixels door scanconversie. Zo wordt elke pixel, die bedekt is door de driehoek, ingekleurd in een opeenvolging van boven naar onder en van links naar rechts. Een pixel (met als positie het midden van elk vierkant hier) ligt binnen de driehoek wanneer het aan de linkerkant ligt van elke zijde van de driehoek. Die zijden worden in tegenwijzerzin opgesteld.

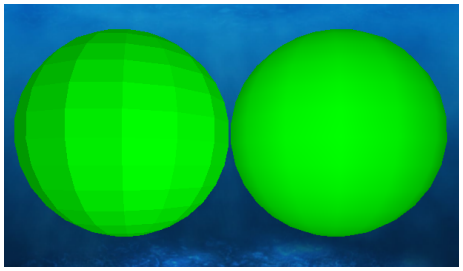


Figuur 2.10: (a) Scanconversie van driehoek (b) Scanconversie van lijn (c) Interpolatie over driehoek

Bovendien moet verzekerd worden dat er geen gaten zijn tussen aaneengrenzende driehoeken. In het geval dat ook lijnen als primitieven gebruikt worden, rijst door de discrete aard van hedendaagse computerschermen het probleem om schuine lijnen zo recht mogelijk weer te geven. De pixels, die de echte lijn het beste benaderen, moeten ingekleurd worden en daarbij moet aliasing zoveel mogelijk vermeden worden. De methode van Bresenham [19] was een van de eerste om dit probleem aan te pakken en hier zijn dan ook veel verbeteringen voor ontwikkeld [20, 78].

De uiteindelijke kleur wordt bekomen door eerst textuurcoördinaten, diepte- en alfawaarden van de vertices te interpoleren over alle punten van de driehoek. Daarna bepaalt Flat shading of Gouraud shading de kleurwaarde. Flat shading

kent dezelfde kleur toe aan elke pixel in een driehoek. Gouraud Shading [45] interpoleert de eerder berekende kleurwaarden op de vertices over alle punten op de driehoek. De rasterisatiestage mengt de bekomen kleurwaarde dan met de overeenkomende kleurwaarde van een textuur, indien die opgegeven is.



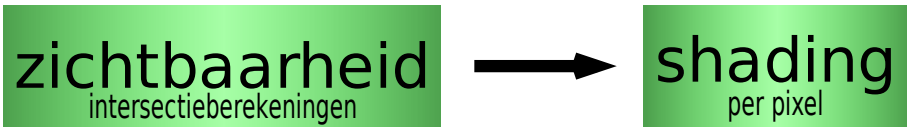
Figuur 2.11: Flat shading (links) en Gouraud shading (rechts)

Vooraleer de kleur definitief bepaald is voor een pixel en naar de framebuffer geschreven wordt, voert de stage nog enkele testen uit om te bepalen of de prepixel (fragment) getoond mag worden. Eerst wordt de scissor test uitgevoerd, gevolgd door de alpha test en daarna de stencil test. Deze vergelijken het fragment met bepaalde drempelwaarden en dat wordt weggesmeten indien het niet voldoet. Er kan bijvoorbeeld getest worden of het fragment binnen een bepaalde rechthoek ligt, of zijn alfa waarde groter is dan een bepaalde waarde, enzoverder. Daarna volgt de belangrijkste test, namelijk de Z-test. Hidden surface removal zal bepalen of het deel van het object, waartoe deze pixel behoort, wel zichtbaar is. De Z-buffer [51], het schildersalgoritme [66] en het algoritme van Warnock [109] zijn voorbeelden van algoritmen die dit behandelen. De Z-buffer wordt standaard toegepast. Telkens een pixel getekend wordt in de buffer, houdt het algoritme de z-waarde voor deze pixel bij. Indien op een gegeven moment een kleur op dezelfde pixel belandt, vergelijkt het algoritme de reeds aanwezige dieptewaarde met de nieuwe dieptewaarde. Zo kunnen pixels dus meerdere keren ingevuld worden en tenslotte zal het dichtstbijzijnde object over de rest getekend worden. Een bepaalde uitbreiding sorteert eerst de driehoeken en overloopt ze dan van dichtbij naar verder weg. Tenslotte volgt er nog alpha blending. Indien het dichtstbijzijnde object transparant is zal de bekomen kleur gemengd worden met kleurwaarden van achterliggende objecten. De rasterisatiestage zorgt ook voor anti-aliasing, anisotropische filtering, schaduwen, mist, enzovoort.

2.2.2 Raytracing

Het doel is het genereren van fotorealistische beelden, wat niet mogelijk is met rasterisatie [102]. Raytracing daarentegen steunt op fysisch gebaseerde theorieën [36], zoals de lichttransportvergelijking [52]. Het algoritme simuleert het transport van licht doorheen de omgeving en de interactie van licht met de aanwezige objecten, waardoor meerdere realistische effecten mogelijk worden, die op natuurlijke wijze geïntegreerd zijn in het algoritme. Deze effecten zijn belangrijke aanwijzingen voor het menselijk visueel systeem.

Het algoritme doet ruwweg het volgende. Eerst wordt voor een bepaalde pixel de zichtbaarheid bepaald. Dat gebeurt door een straal, die start vanaf de camera, de scene in te volgen en te intersecteren met de objecten. Het punt,



waarbij de afstand tussen de camera en het punt het kleinste is, ligt het dichtste bij de camera langs die straal en is dus het punt dat zichtbaar is op die pixel. Daarna wordt de kleur bepaald van dat punt.

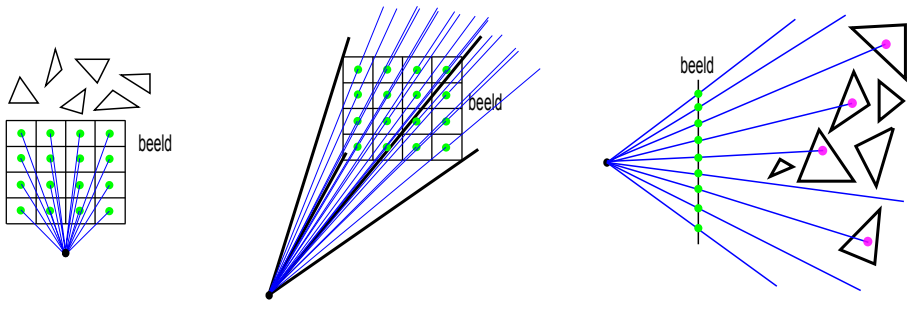
Deze techniek en zijn uitbreidingen vereisen net als rasterisatie een beschrijving op hoog niveau van een 3D-scene, die samengesteld is uit een verzameling primitieven. Er zijn hier evenzeer perspectieve en orthografische camera's mogelijk, en daarenboven ook andere camera's. Die camera bepaalt dan hoe de stralen vanaf het oog de scene in geschoten worden. Elke pixel representeert wederom een punt uit een rechthoekig rooster.

De eerste subsectie I behandelt kort de evolutie van het algoritme in accuraatheid, terwijl de tweede subsectie II mogelijke uitbreidingen beschrijft om de berekeningstijden korter te maken.

I Basis algoritme

Het raytracing algoritme wordt hier geschetst vanaf zijn prille begin en de belangrijke uitbreidingen worden kort aangehaald. Die dienen om het algoritme realistischer te maken. Ray casting, klassieke raytracing en distributed raytracing houden nog geen rekening met de lichttransportvergelijking.

I.1 Ray Casting Raytracing is een uitbreiding van ray casting, welke Appel [10] in het jaar 1968 bedacht. Hij zendt vanaf een gemeenschappelijk punt, namelijk het oog, een straal rechtlijnig de scene in door elke pixel van het scherm en indien deze inslaat op een object, test hij of er objecten aanwezig zijn tussen het object en de lichtbronnen.

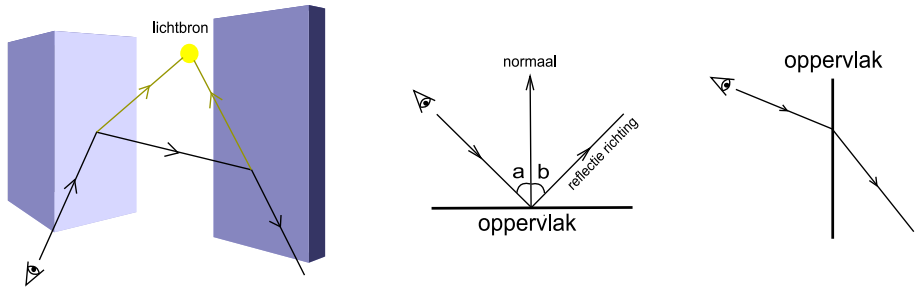


Figuur 2.12: Primaire stralen vanuit camera

Dat doet hij door stralen te schieten vanaf het object, waarop de straal inslaat, naar de lichtbronnen. Indien er zich objecten tussen bevinden, betekent dit dat er een schaduw geworpen wordt op het object, waarvan de straal start. Ray casting berekent enkel directe illuminatie. Er worden enkel primaire stralen geschoten. Na inslag op een object, wordt er niet verder gekeken, om onder andere

reflectie- en refractie-eigenschappen van objecten te benutten. Ray casting is daardoor een goedkoop algoritme, maar biedt weinig realisme. De kleur van een object op een bepaalde plaats wordt bepaald door een shading algoritme, net zoals bij rasterisatie het geval is.

I.2 Klassieke raytracing Recursieve of klassieke raytracing dateert van 1980 en is ontwikkeld door Whitted [112]. Vanaf dan worden ook secundaire stralen geschoten. Net zoals bij ray casting wordt voor elke pixel een straal geschoten. Wanneer de straal dan een oppervlakte raakt, worden er nieuwe stralen gegenereerd aan de hand van de eigenschappen van het overeenstemmende object dat geraakt is. Als het object reflecterend is, wordt een reflectie straal geschoten en indien het object doorschijnend is, wordt ook een refractiestraal geschoten. Verder wordt ook hier getest of het object niet afgedekt is van het licht door een ander object, zodat er een schaduw straal geschoten wordt vanaf het raakpunt naar elke lichtbron. Het pad dat het licht neemt, wordt op deze manier recursief gevolgd doorheen de omgeving, te beginnen van de kijker tot het punt waar het licht zijn oorsprong heeft. De generatie van nieuwe stralen stopt wanneer er geen objecten meer geïntersecteerd worden of wanneer een bepaalde recursie diepte bereikt is. De kleur van de pixel is dan een samenstelling van de kleuren op de geraakte punten. Whitted nam nog geen BSP-boom of andere versnellingsstructuur in gebruik, maar hij gebruikte bollen als bounding volumes, om sneller objecten op het pad van de straal te ontdekken. Anti-aliasing deed hij ook reeds voor regio's met sterke kleurveranderingen, voor plaatsen waar kleine objecten tussen de discrete punten vallen en voor texturen. Deze oorspronkelijke manier van raytracen berekent directe illuminatie en perfect speculaire en refractieve effecten, maar is nog geen oplossing voor het gehele illuminatieprobleem.

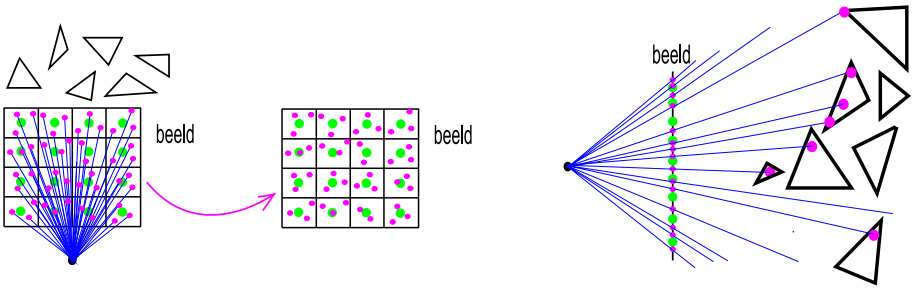


Figuur 2.13: Primaire en secundaire stralen

Stralen worden hier eigenlijk geschoten in de omgekeerde richting, want in de natuur reist het licht van de lichtbron naar het oog. Indien raytracing op de andere manier geïmplementeerd zou worden, is er weinig kans dat het oog geraakt zou worden.

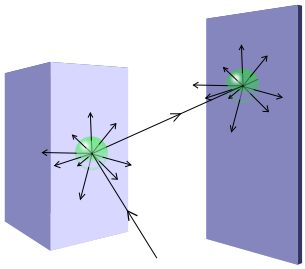
I.3 Distributed raytracing Volgens Cook et al. [28] zien beelden, gecreëerd door raytracing, er nog niet volledig echt uit in de tot nu toe besproken oplossingen. Schaduwen hebben over heel hun oppervlak dezelfde kleur, terwijl er in de realiteit zachte overgangen aanwezig zijn. Er zijn randen aanwezig, die een trapachtige vorm hebben, wat aliasing genoemd wordt en reflecties zijn

altijd scherp, terwijl in de meeste gevallen objecten eerder glanzende (glossy) oppervlakken hebben. Enkel als alle invallende stralen weerkaatsen volgens de perfecte spiegelhoek, zal een object een perfecte reflectie weergeven. Tenslotte is elk object altijd in focus.



Figuur 2.14: Random stralen

Dit alles ontstaat doordat een continue wereld bemonsterd wordt op een discreet rooster van regelmatig geplaatste pixels, wat leidt tot onnauwkeurige benaderingen. Details, aanwezig tussen de discrete monsters, worden namelijk gemist. Tot hiertoe werd de waarde van elke pixel benaderd door slechts *één* straal te schieten. Net zoals bij anti-aliasing meerdere stralen geschoten worden om getande randen te vermijden, kunnen deze andere problemen ook opgelost worden door meerdere stralen te schieten op elke plaats in de raytracer, namelijk oogstralen, schaduwstralen, reflectie- en refractiestralen. Door meer monsters,



Figuur 2.15: Over hemisfeer kunnen overal stralen geschoten worden

die random lichtjes van plaats verschillen, te nemen per discreet punt voor de primaire stralen, worden meer tussenliggende details opgenomen. Om alle details te kunnen voorstellen, moet eigenlijk al het tussenliggende opgenomen worden, maar dat is teveel werk.

Distributed raytracing [28], de eerste stochastische raytracer, gaat op die manier dus het gebied dat door *één* monster voorgesteld wordt, representeren aan de hand van een random bemonstering van dat continue gebied. Dat is het verschil met eerdere aanpakken voor anti-aliasing, zoals supersampling. Bij supersampling worden meerdere stralen geschoten op regelmatige posities. Eigenlijk wordt er bij supersampling gewoon een hogere resolutie aangenomen, waarvan dan gemiddelden genomen worden om de resolutie terug te verkleinen. Ook bij distributed raytracing wordt die ene bepaalde kleur dan samengesteld

uit deze resultaten. Op deze manier wordt aliasing verminderd. Men verkrijgt ook zachte schaduwen, motion blur, objecten in en uit focus, enzovoort.

Figuur 2.16: Voorbeeld globale illuminatie

I.4 Globale illuminatie Sinds de toepassing van de lichttransportvergelijking in 1986 door Kajiya [52] op computergrafieken is er een wiskundige formulatie voor de simulatie van lichttransport. In het boek Advanced Global Illumination [36] wordt dit uitvoerig besproken.

$$L(x \rightarrow \Theta) = L_e(x \rightarrow \Theta) + \int_{\Omega_x} fr(\Psi \leftrightarrow \Theta) L(x \leftarrow \Psi) cos(\Psi, n_x) d\omega_\Psi \qquad (2.1)$$

Deze vergelijking 2.1 steunt op fysische wetten, zodat heel wat fotorealistische effecten betrouwbaar weergeven kunnen worden. Er wordt uitgegaan van de wet van behoud van energie. De uitgestraalde energie is voor elk oppervlakpunt x en in elke richting Θ te berekenen met deze vergelijking. Die energie 2.2 is namelijk gelijk aan de zelf uitgestraalde energie plus de energie die binnenkomt vanuit alle richtingen. De inkomende energie is samengesteld uit lichtstralen die van alle andere oppervlakpunten gereflecteerd of gerefracteerd worden naar het huidige oppervlakpunt x.

$$L(x \rightarrow \Theta) = L_e(x \rightarrow \Theta) + L_r(x \rightarrow \Theta) \qquad (2.2)$$

Aangezien licht in de realiteit continu is, wordt de integraal 2.3 genomen van dit inkomend licht over de halve hemisfeer boven het oppervlakpunt. In het geval van transparante oppervlakken is dat de hele hemisfeer.

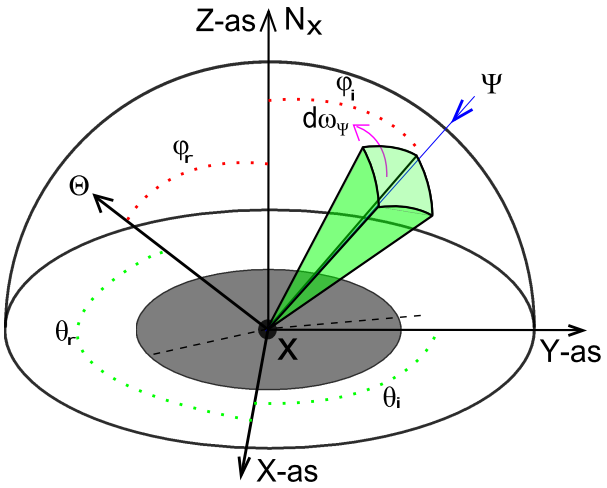
$$L_r(x \rightarrow \Theta) = \int_{\Omega_x} fr(\Psi \leftrightarrow \Theta) L(x \leftarrow \Psi) cos(\Psi, n_x) d\omega_\Psi \qquad (2.3)$$

$L(x \leftarrow \Psi)$ is het inkomende licht op het oppervlakpunt x vanuit de richting Ψ. De reflectie-eigenschappen van het oppervlak bepalen hoe deze aankomende lichtstralen terug verspreid worden vanaf het punt x. Daarom wordt dat vermenigvuldigd met de bidirectional reflectance distribution function (BRDF) $fr(\Psi \leftrightarrow \Theta)$. Ook de inkomende hoek heeft belang, waardoor er ook vermenigvuldigd wordt met de cosinus tussen de hoek van de inkomende lichstralen en de normaalvector van het oppervlakpunt x. Deze cosinus term mag vervangen worden door het dotproduct, aangezien alle vectoren in de BRDF genormaliseerd zijn.

In computergrafieken worden meestal discrete, oneindig dunne stralen genomen, waardoor deze integraal omgezet wordt in een som van alle inkomende richtingen.

De BRDF 2.4 bepaalt hoe licht interageert met de oppervlakken van objecten. Elk oppervlak verspreidt inkomend licht op een bepaalde manier en deze verspreiding kan wiskundig beschreven worden. Dit deel van de formule kan natuurlijk ook ingevuld worden met de vroegere illuminatiemodellen om de berekeningen te vereenvoudigen, maar dat houdt ook een verlies van realisme in.

$$fr(\Psi \leftrightarrow \Theta) = \frac{DL(x \rightarrow \Theta)}{L(x \leftarrow \Psi)cos(\Psi, n_x)d\omega_\Psi} \tag{2.4}$$



Figuur 2.17: BRDF

De BRDF op een oppervlakpunt x zorgt voor de relatie tussen de gereflecteerde energie in de uitgaande richting Θ ($dL(x \rightarrow \Theta)$) en de inkomende energie op het oppervlakpunt x via de ruimtehoek $d\omega_\Psi$ ($dE(x \leftarrow \Psi)$). Het levert de fractie van de inkomende energie, die gereflecteerd wordt door het oppervlakpunt x in de richting Θ.

De lichttransportvergelijking is een benadering van de geometrische optica [36], welke reeds een eenvoudig lichtmodel is. Er wordt onder meer geen rekening gehouden met participerende media, diffractie, interferentie, polarisatie, iridescence, dispersie, lichtsnelheid, zwaartekracht en magnetische velden. Dispersie zorgt ervoor dat de delen van een lichtstraal, die samengesteld is uit

meerdere frequenties, elk weerkaatsen in een andere richting. Diffractie buigt het licht af omheen hoeken en participerende media zijn bijvoorbeeld mist, rook en vuur. Deze zorgen ervoor dat lichtstralen verspreid worden in verschillende richtingen ofwel geabsorbeerd worden. Polarisatie kan voorvallen bij reflectie of refractie van het licht. Dit betekent dat sommige trilrichtingen van de golf afwezig zijn. Normaal trilt een lichtgolf in alle richtingen die loodrecht staan op de voortplantingsrichting. Door polarisatie kan men er bijvoorbeeld voor zorgen dat de lichtgolf slechts trilt in *één* vlak. Enkele van deze fenomenen kunnen ook berekend worden door niet enkel de radiantie in aanmerking te nemen, maar ook door de fase, de golflengte en de polarisatie in de vergelijking te betrekken.

Radiantie beschrijft de hoeveelheid licht dat aankomt op een bepaald gebied of ervan vertrekt. Ingewikkeldere en diepgaandere fysische modellen, die nog meer effecten kunnen weergeven, zijn onder andere de golftheorie en de kwantumoptica. Globale illuminatie is dankzij deze lichttransportvergelijking mogelijk geworden, waardoor ook indirecte illuminatie en zachte schaduwen berekend kunnen worden.

Kajiya [52] bedacht ineens een variatie op raytracing, namelijk pathtracing, waarbij een straal gevolgd wordt tot aan de lichtbron, zonder deze op te splitsen in meerdere stralen bij intersecties. Dit steunt op stochastische of Monte Carlo raytracing. De stralen moeten volgens een waarschijnlijkheidsverdeling geschoten worden, zodat enkel de stralen die het meeste bijdragen gekozen worden. Verder moet verzekerd worden dat voor elke pixel een goed aandeel van reflectie-, refractie- en schaduwstralen geschoten worden. Dit laatste lost Kajiya op door het aantal reeds geschoten stralen bij te houden bij elk van deze delen en de waarschijnlijkheid om een van hen te kiezen te variëren.

Bidirectional pathtracing [54] kan de resultaten nog verbeteren doordat het de combinatie is van pathtracing met zijn duale vorm lighttracing, die de stralen in de omgekeerde richting volgt, te beginnen vanaf de lichtbronnen. Sindsdien werden nog meer Monte Carlo methoden ontwikkeld, zoals photon tracing [50] en metropolis light transport [94].

II Methoden om raytracing sneller te maken

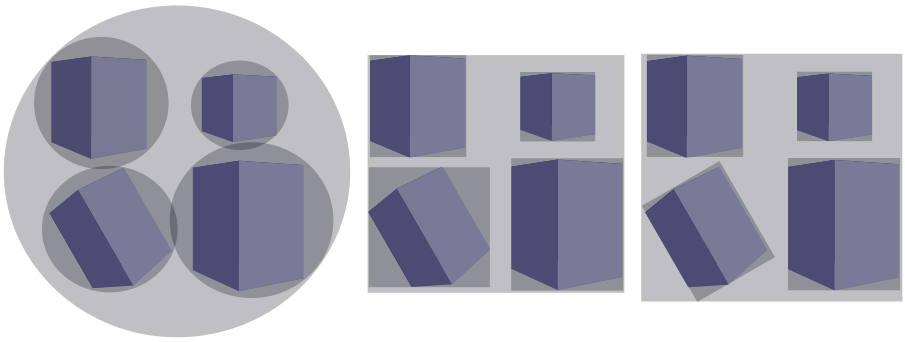
Er zijn verschillende manieren onderzocht om raytracing sneller te maken, met als uiteindelijk doel scenes realtime en interactief weer te geven. De achterliggende algoritmen kunnen verbeterd worden II.1 of ze kunnen aangepast worden om benaderingen te doen van berekeningsintensieve delen II.2. Verder kan de code geoptimaliseerd worden om efficiënter gebruik te maken van beschikbare hardware, of kan de hardware aangepast worden aan de algoritmen II.3.

II.1 Raytracing algoritme zelf versnellen

De achterliggende algoritmen kunnen aangepast worden zodat de intersectieberekeningen goedkoper zijn [64, 6, 113] of stralen kunnen gebundeld worden om voordeel te halen uit hun spatiale coherentie [76, 103]. Bovendien kunnen versnellingsstructuren toegepast worden om snel een bepaald object in de scene te vinden. Verder bestaat er instantiatie, waarbij meerdere instanties van eenzelfde object gebruikt worden [34], zodat het slechts *één* keer in het geheugen moet aanwezig zijn.

Versnellingsstructuren Het volgen van stralen doorheen een omgeving, zonder versnellingsstructuur, betekent een lineaire complexiteit. De oorsprong en richting van de stralen kan namelijk willekeurig zijn. Daarom moet voor elke straal telkens een intersectie gedaan worden met alle objecten. Het is efficiënter om de scene op te delen in een bepaalde structuur, zodat per straal slechts een miniem gedeelte van de objecten getest moet worden en meerdere objecten in *één* keer getest kunnen worden. Bij statische scenes moet deze structuur slechts *één* keer opgebouwd worden per beeld, wat ongeveer een lineaire complexiteit vereist. Het doorlopen van de structuur is gelijkaardig aan een binaire boomstructuur. Raytracing heeft zijn average-case logaritmische complexiteit hieraan te danken.

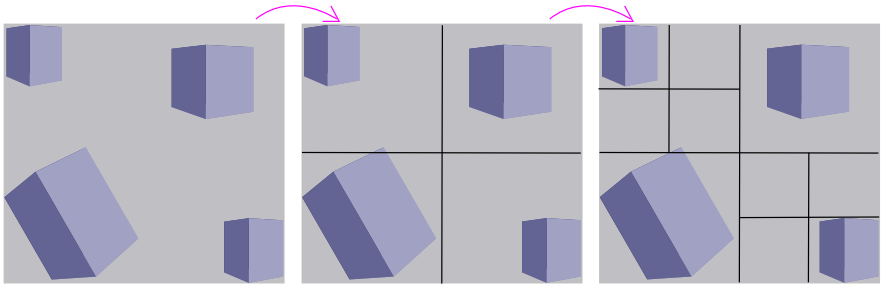
Gewoonlijk worden objecten gebundeld in object hierarchieën, zoals bounding volume hierarchies (BVH) [79] of spatiale ruimte onderverdelingen, zoals uniforme grids [8], octrees of BSP-bomen [40]. Er bestaan natuurlijk ook hybriden. Ze hebben elk hun voor- en nadelen. Bij BVHs kunnen delen van de ruimte tot meerdere volumes behoren, terwijl de objecten mooi binnen de volumes passen. Dat maakt hen efficiënter voor dun bezaaide scenes, waar zo weinig mogelijk overlap voorkomt. Overlap leidt tot de noodzaak aan meerdere testen voor een bepaalde positie, aangezien de straal beide kan raken. Bij spatiale onderverdelingsmethoden kunnen sommige objecten tot meerdere volumes behoren, aangezien ze niet altijd mooi binnen de volumes passen, maar de spatiale ruimte blijft disjunct. Daardoor zijn ze efficiënter voor dicht bezaaide scenes. Vaak kan voor twee disjuncte delen slechts *één* intersectie gedaan worden, waarna rechtstreeks verder gegaan kan worden naar de kinderen.



Figuur 2.18: Voorbeeld van bounding volume hierarchies

Een **BVH** is een boom, waarbij kinderen steeds kleinere bounding volumes bevatten, die telkens omsloten worden door de ouders. Op die manier wordt eerst geïntersecteerd met de volumes, die het hoogste gelegen zijn in de hiërarchie. Het berekenen van de intersectie van een straal met een driehoek of een gecompliceerd object, vraagt meer tijd dan het berekenen van de intersectie van een straal met een eenvoudig volume, zoals bijvoorbeeld bollen, axis-aligned bounding boxes (AABB) of oriented bounding boxes (OBB). De keuze tussen deze volumes is een afweging tussen de benodigde intersectieberekeningstijd en hoe goed dat volume past rond het object, zodat zo weinig mogelijk overbodige berekeningen gedaan moeten worden. Een bol en een AABB bieden eenvoudige

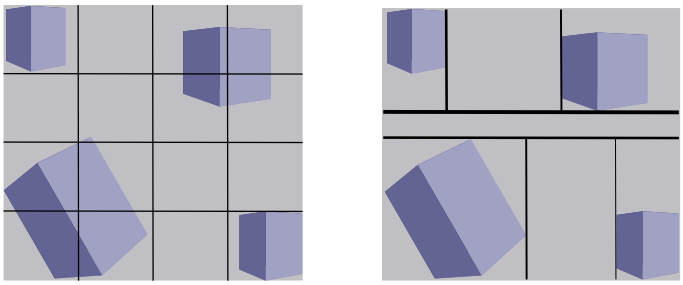
intersectieberekeningen, terwijl een OBB duurder is. Maar een OBB sluit beter aan rond het object. Een AABB is een balk, waarbij de drie paren vlakken gealigneerd zijn met de respectievelijke assen, terwijl de vlakken bij een OBB daar niet mee gealigneerd moeten zijn. Bij het doorlopen van de hiërarchie, wordt steeds dieper getest tot men bij de primitieve aankomt, waarmee dan wel een duurdere intersectieberekening gedaan moet worden.



Figuur 2.19: Voorbeeld van een quadtree (zelfde principe als een octree)

Een **octree** splitst de bounding box van de scene in acht gelijke delen (kubussen). Delen, die objecten bevatten, worden telkens recursief verder gesplitst. Wanneer een bepaalde diepte bereikt is of er slechts weinig objecten aanwezig zijn in een bepaald deel, wordt de opdeling daar stopgezet. Interne knopen van de boom representeren spatiale onderverdelingen. Bladeren stellen kubussen voor die objecten bevatten. Het doorlopen van deze octree is vrij duur.

Een **uniform grid** verdeelt de bounding box van de scene in een rooster van cellen met gelijke grootte. De resolutie wordt vaak bepaald aan de hand van het aantal objecten in de scene. Bij intersectieberekeningen worden enkel testen gedaan met objecten die volledig of gedeeltelijk gelegen zijn in de voxels, die doorboord worden door de straal. Via een soort scanconversie van de primitieven op een discreet rooster van pixels wordt het grid doorlopen. Uniforme grids kunnen zich niet goed aanpassen aan scenes, waarin de objecten niet goed verdeeld zijn. Hiërarchische grids bieden daar een oplossing voor.



Figuur 2.20: (a) Een uniform grid (b) Een mogelijke kd-boom

Een **BSP-boom** verdeelt de ruimte, initieel de bounding box van de hele scene, telkens in twee disjuncte delen. Indien de splitsvlakken een willekeurige oriëntering mogen hebben, spreken we van BSP-bomen. Wanneer er gesplitst

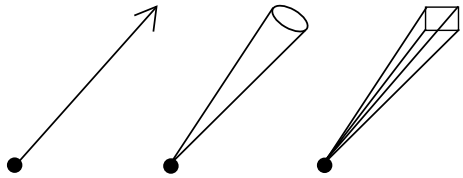
wordt volgens vlakken, die gealigneerd zijn met de assen, spreken we van kd-bomen. Het vinden van de optimale splitsvlakken is een zwaar probleem, waardoor vaak de surface area heuristic (SAH) toegepast wordt. Van alle objecten in de scene worden AABB's berekend. Dan kan eenvoudig de som van de oppervlakken genomen worden van de zes vlakken van de AABB. Telkens wordt het splitsingsvlak genomen, dat de kleinste som oplevert voor beide kinderen, aangezien dat volgens de heuristiek de meest optimale is. De opsplitsing gebeurt het beste door grote eenvoudige regio's ineens in *één* groep in te delen. Daarom krijgt de kans om die opsplitsing te kiezen ook vaak een groter gewicht. Verder kunnen delen, waarin weinig objecten overschieten, best niet verder opgesplitst worden. Ook wanneer een bepaalde diepte bereikt is, wordt de splitsing vaak stopgezet. Anders wordt de boom te diep en dat leidt tot duurdere zoekbewerkingen. De interne knopen stellen hier ook onderverdelingen voor en de bladeren van de boom stellen delen voor die objecten bevatten. Bij het doorlopen van de boom, wordt nu geïntersecteerd met AABB's, wat snel te berekenen is, totdat er bladeren bereikt worden.

In **dynamische scenes** veranderen objecten van positie, waardoor versnellingsstructuren niet correct blijven. Maar voor elk afzonderlijk beeld is er slechts weinig tijd beschikbaar. Daarom moeten versnellingsstructuren realtime herbouwd kunnen worden. Het volledig herbouwen van een dergelijke structuur vraagt minstens lineaire tijd. Daarom werd dat vroeger gewoonlijk op voorhand gedaan. Ondertussen is dit probleem onderzocht door onder andere Parker et al. [70], Lext et al. [60, 58] en Wald et al. [98]. Lext et al. en Wald et al. herbouwen telkens enkel het deel van de structuur dat beïnvloed wordt door veranderingen. Dynamische scenes worden opgedeeld in aparte objecten, die bevat zijn in een globale versnellingsstructuur, en lokaal hun eigen versnellingsstructuur bezitten. Bij veranderingen aan een object, wordt enkel de lokale versnellingsstructuur van dat object aangepast of herbouwd. Bovendien wordt bij elke beweging ook de globale structuur herbouwd. Indien er sprake is van ongestructureerde bewegingen zal de lokale en de globale structuur herbouwd worden. Bij hiërarchische bewegingen is het mogelijk de stralen te transformeren naar de lokale cöordinaten van de versnellingsstructuur van het object, waardoor die niet herbouwd moet worden.

Coherentie van stralen uitbuiten Veel stralen volgen een gelijkaardig pad en komen via dezelfde oppervlakken terecht op de lichtbronnen. Toch worden deze allen apart geschoten en berekend. Deze coherentie tussen buurstralen kan uitgebuit worden door pakketten stralen te schieten en zo kunnen de berekeningskosten onder hen verdeeld worden [103]. In beeldruimte vertrekken bijvoorbeeld verschillende stralen in ongeveer dezelfde richting van het oog de scene in. Cone tracing en beam tracing zijn de eerste methoden die deze coherentie aanwenden. Reshetov et al. [76] en Wald et al. [103] gebruiken deze coherentie expliciet voor snellere berekeningen.

Cone Tracing en Beam Tracing buiten coherentie tussen stralen uit om aliasingartefacten te verminderen. Bij raytracing wordt een straal normaal gedefinieerd als een halve rechte die bestaat uit een oorsprong en een richting. Cone tracing definieert een straal als een kegel met een oorsprong, een richting en een straal voor de cirkel op het einde van de kegel. In het geval van reflectie of refractie wordt telkens een nieuwe kegel gecreëerd. Het aantal intersectieberekeningen wordt verkleind door deze methode, maar ze zijn spijtig

genoeg vrij complex, omdat er ook berekend moet worden welk deel van de kegel intersecteert met het object.



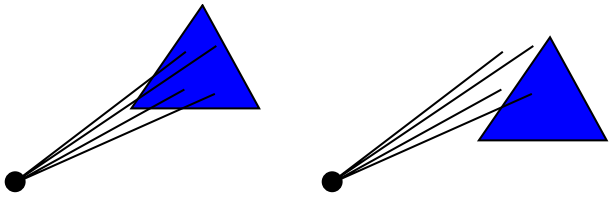
Figuur 2.21: Gewone stralen, kegels, beams

Per pixel moet slechts *één* straal geschoten worden, om direct anti-aliasing te verkrijgen. Deze techniek leidt ook tot zachtere schaduwen en minder gedetailleerde reflecties, maar het levert geen exacte resultaten. Beam tracing stelt stralen voor door onbegrensde pyramide vormen met een oorsprong en een richting. Deze worden door heel het kijkvolume gezonden en worden geïntersecteerd met elke polygoon in de juiste diepte volgorde. Telkens wordt dat deel van de beam, dat het object raakt, weggelaten uit de beam.

Multi Level Ray Tracing [76] behoudt in tegenstelling tot de bovenstaande methoden de geometrische correctheid volledig. Normaal wordt een versnellingsstructuur doorlopen vanaf wortel tot bladeren. Dit vereist intersectiekosten om gewoon te ontdekken welk deel van de boom verder te doorzoeken. Deze techniek probeert bundels stralen in *één* keer doorheen de boom te laten gaan met behulp van een inverse frustum culling methode. Primaire stralen worden gebundeld volgens een beeldruimtetegeling, die aangeeft waar de scene complex is en waar de scene vrij gelijkaardig blijft. De bundel wordt samengesteld uit stralen die gemeenschappelijke eigenschappen bezitten, zoals eenzelfde richting, om zo weinig mogelijk te moeten berekenen. Deze stralen vormen samen een volume, dat bestaat uit een oorsprong en vier vlakken. Voor elk as-gealigneerd splitsingsvlak wordt een rechthoek binnen dat vlak berekend die alle intersecties omvat van de bundel stralen en het vlak. Bundels stralen, die dan te breed zijn en dus geen diep ingangspunt delen, worden snel ontdekt en opgesplitst. Op deze manier worden ingangspunten in de boom gevonden op plaatsen die dichter bij de bladeren liggen, zodat overbodige testen gespaard worden. Vanaf daar wordt dan verder gezocht naar intersecties op de normale manier. Dit wordt gedaan voor primaire stralen, secundaire stralen en schaduwstralen.

Interactive Rendering with Coherent Ray Tracing [103] herformuleert de vroegere raytracing algoritmen om coherentie bloot te leggen. Het algoritme werkt bijna volledig binnen de caches dankzij deze herformulatie. De verbinding met het hoofdgeheugen van de computer is veel trager dan die met de caches. Data, die samen nodig is, wordt aldus zoveel mogelijk samengehouden en gealigneerd volgens cachelijnen van vaste grootte. Verder wordt er data op voorhand gehaald, zodat vertragingen minder invloed hebben. De coherentie is een gevolg van het feit dat naburige stralen vaak op dezelfde data werken. Door stralen te bundelen kan die coherentie uitgebuit worden. Primaire stralen en schaduw stralen worden samen doorheen de kd-boom gevolgd, worden samen geïntersecteerd en tenslotte ondergaan ze samen shading. Daarbij wordt gebruik gemaakt van SSE voor parallelle verwerking. Ook hier wordt minder toegang vereist tot geheugen door data per pakket eenmaal op te vragen. Ze beweren

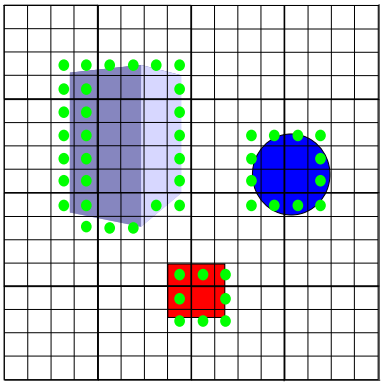
interactiviteit te bereiken op gewone massaproductiecomputers.



Figuur 2.22: Twee mogelijke gevallen bij een bundeling van 4 stralen

II.2 Benaderingen

Raytracing kan ook sneller gemaakt worden door bepaalde dure berekeningen te benaderen met goedkopere methoden. Het aantal te schieten stralen kan verminderd worden met behulp van adaptieve controle van de diepte van de boom [47]. Er wordt dan verzekerd dat het recursief dieper volgen van stralen gestopt wordt wanneer de bijdrage niet meer voldoende groot is. Het is ook mogelijk minder stralen te schieten op plaatsen waar weinig kleurveranderingen voorkomen. Daar kan bijvoorbeeld beeldruimte interpolatie gedaan worden, terwijl op complexe plaatsen meer stralen geschoten worden [7].



Figuur 2.23: Posities met hogere complexiteit

Ook in frameless rendering 2.3 worden benaderingen toegepast. Elk afzonderlijk beeld wordt bemonsterd door een beperkt aantal primaire stralen. De rest van het beeld wordt opgevuld via heuristieken zoals bijvoorbeeld met behulp van filters of door te veronderstellen dat er veel temporele coherentie aanwezig is, waardoor veel van de overige pixels dezelfde kleurwaarde zullen behouden ten opzichte van een vorig beeld.

Perceptuele visualisatie methoden zorgen ervoor dat verspillende berekeningen zoveel mogelijk vermeden worden in het spatiale en het temporele domein door rekening te houden met de gevoeligheid van het menselijk visueel waarnemingssysteem. Myszkowski et al. [65] gebruiken daar een model van, dat een drempelwaarde biedt voor spatiale en temporele kleurvariaties. Het modelleert tevens visuele maskering om benaderingen te verbergen en de hoeveelheid werk te berekenen dat daarvoor nodig is. Bepaalde zaken in de scene worden namelijk onderdrukt door interferenties met bijvoorbeeld de achtergrond. Aangezien meerdere beelden snel na elkaar getoond worden, zullen bovendien niet alle kleurvariaties opvallen en bevatten ze temporele coherentie. Informatie van vorige beelden kan dan hergebruikt worden, bijvoorbeeld voor indirecte illuminatie dat gewoonlijk vrij traag verandert. Zodoende kan meer aandacht besteed worden aan onder andere het berekenen van veranderingen in de directe illuminatie, wat gewoonlijk hard opvalt. Tevens ontdekt het algoritme fouten, die veroorzaakt worden door het weglaten van belangrijkere delen

en dus wel gevoelig zijn voor het oog. Ze sturen de visualisatie dan naar de overeenkomende gebieden, volgens lokale variaties van de illuminatie in ruimte en tijd.

Frameless rendering is geen perceptuele visualisatie techniek, maar er wordt ook bijvoorbeeld rekening gehouden met het feit dat de menselijke waarneming meer verstoord wordt door gaten in solide objecten, dan door fouten in de weergave van bepaalde kleuren. Deze techniek hergebruikt ook informatie van vorige beelden.

II.3 Aanpassingen aan hardware en gespecialiseerde hardware

De huidige beschikbare hardware is niet optimaal voor raytracing [89]. Wald et al. [103] passen het raytracing algoritme aan om beter gebruik te maken van het beschikbare geheugen, ervoor te zorgen dat zo weinig mogelijk data opgezocht moet worden en indien wel op een zo efficiënt mogelijke manier. Ze herordenen stralen om de caches van de hardware efficiënter te gebruiken. De verbindingen tussen hardware delen worden dan beter benut.

Het is ook mogelijk parallellisme uit te buiten door meerdere processoren gelijktijdig werk te laten verrichten. Wald et al. [103] verbinden meerdere massaproductiecomputers in een netwerk, terwijl Parker et al. [70] gebruik maken van parallelle supercomputers met gedeeld geheugen. Het schieten van stralen wordt nu verdeeld over meerdere processoren.

Daarnaast gebruikt men programmeerbare rasterisatie hardware om raytracing te implementeren [74, 53, 38, 22]. Intersectieberekeningen of andere delen van het algoritme kunnen er afzonderlijk op geïmplementeerd worden, evenals het gehele algoritme.

Er zijn verder ook technieken die raytracing gebruiken om rasterisatie resultaten te verbeteren, zoals bijvoorbeeld corrective textures [90]. Het berekent alle niet speculaire illuminatie-effecten met rasterisatie hardware. Een raytracer zal daarna speculaire effecten en dergelijke berekenen en opslaan in texture maps. Die worden dan toegevoegd op plaatsen waar dit nodig is. Er ontstaan artefacten wanneer veel gecorrigeerd moet worden, aangezien de raytracer traag is. Bij een statische scene leidt dat, net zoals bij frameless rendering 2.3, zo snel mogelijk tot convergentie met een beeld van de kwaliteit van de raytracer.

Tenslotte worden speciaal ontwikkelde grafische processoren voor realtime raytracing ontwikkeld. Zij kunnen hoge performantie leveren. Het nadeel van hardware raytracing is dat het minder flexiebel is dan software raytracing, maar software raytracing maakt dan weer minder gebruik van parallellisme dankzij de huidige processoren. Voorlopig is het in de meeste gevallen bij massaproductiecomputers enkel mogelijk bijvoorbeeld SIMD uitbreidingen (SSE) te gebruiken [103], waarbij vier operaties tegelijkertijd uitgevoerd kunnen worden.

2.2.3 Interactieve globale illuminatie

Het interactief tonen van fotorealistische beelden is het uiteindelijke doel. Gebruikers moeten de hele scene kunnen manipuleren, terwijl ze toch terugkoppeling krijgen in de vorm van voldoende snel getoonde beelden met een fotorealistische kwaliteit. Het is belangrijk dat hierbij een goede scalering van de berekeningstijd met de scenecomplexiteit bekomen wordt. Zoals in subsectie II vermeld is, bestaan er verschillende methoden die dit trachten te bereiken. Meer informatie is te vinden in Wald et al. [99, 101].

Radiosity systemen werden als eerste gebruikt in de context van interactieve globale illuminatie. Dankzij zijn camera-onafhankelijke eigenschap werd het reeds vroeg gebruikt voor interactieve walkthroughs in statische, diffuse omgevingen. Spijtig genoeg is het toevoegen van extra effecten zoals speculaire reflecties lastig en scaleert deze techniek niet goed met scenecomplexiteit. Het vereist te lange voorverwerkingstijden en interactieve veranderingen verlangen dure aanpassingen van de globale datastructuren. Ze gebruiken enkel polygonen als primitieven en lijden ook aan tesselatie-artefacten.

Sindsdien zijn ook heel wat hybride systemen ontwikkeld, die bijvoorbeeld radiosity gebruiken om de diffuse interacties te berekenen en raytracing gebruiken voor reflecties [90, 104, 86, 25]. Aangezien interactieve globale illuminatie nogal hoog gegrepen was en snelle visuele terugkoppeling belangrijker is dan correctheid, werden vaak benaderingen toegepast. Er bestaan daarnaast bijvoorbeeld ook systemen die raytracing gebruiken voor specifieke effecten bovenop rasterisatie systemen [24, 30, 77].

Raytracing levert een handig, doch traag, kader om interactieve globale illuminatie in te verwezenlijken [101]. Dankzij zijn goede scalering met scenecomplexiteit en zijn parallellisme kan het sterk versneld worden. Er kunnen makkelijk verschillende methoden ingevoegd worden en het kan potentieel alle effecten aan. Bijgevolg steunen de meeste interactieve globale illuminatie-algoritmen tegenwoordig op deze techniek. Een nadeel is dat globale illuminatie gebaseerd is op het schieten van random stralen, zodat coherentie moeilijker uit te buiten lijkt. Volgens Boulos et al. [17, 18] kan het desalniettemin efficiënt gecombineerd worden in pakketten. Raytracing doet zijn berekeningen gebaseerd op wat zichtbaar is vanaf de camera. Daardoor worden enkel berekeningen gebruikt die nuttig zijn voor het huidige beeld. Per beeld moet dit, indien er veranderingen gebeurd zijn, wel herberekend worden.

Wald et al. [99, 100] gebruiken Monte Carlo raytracing als kern in hun interactief globaal illuminatie-algoritme. Er worden bovendien meerdere globale illuminatie-algoritmen en een aantal andere technieken gebruikt. Een soort instant radiosity zorgt voor diffuse, directe en indirecte illuminatie en photon mapping verzorgt de directe caustics. Raytracing bepaalt zichtbaarheid, reflecties en refracties. Interleaved sampling vermindert aliasing en een discontiuïteitsbuffer zorgt ervoor dat ruis verwijderd wordt door filtering in beeldruimte. Dankzij deze laatste twee technieken moeten minder stralen geschoten worden. Quasi-Monte Carlo integratie zorgt voor een goede verdeling van de monsters. Dit algoritme levert fysiek correcte resultaten die per beeld herberekend worden. Ze bereiken interactieve snelheden op een netwerk van meerdere massaproductiecomputers.

Aangezien er bij interactieve visualisatie een nieuwe dimensie bijkomt, namelijk de tijd, wordt het mogelijk de coherentie tussen opeenvolgende beelden uit te buiten. Dat gebeurt dus in beeldruimte. Zo kan er een systeem opgebouwd worden dat zich gescheiden van het visualisatie algoritme bezig houdt met het opvolgen van de berekende monsters en aan de hand daarvan bepaalt welke pixels het visualisatie algoritme opnieuw moet berekenen. Het aantal te berekenen stralen wordt zo gereduceerd in het beeldvlak, zodat in plaats daarvan meerdere beelden getoond kunnen worden per tijdseenheid. De niet herberekende pixels worden op dergelijke manier behandeld, dat de benadering zo min mogelijk opvalt. Deze laatste klasse methoden vallen onder het gebied frameless rendering 2.3, waarin deze thesis verder zal verdiepen. Deze methoden zijn

nuttig voor interactieve globale illuminatie, omdat zo interactiviteit behouden wordt, terwijl beelden te verkrijgen die zullen convergeren tot de volledige kwaliteit van het globale illuminatie-algoritme. Tijdens interactie zullen tijdelijk beelden getoond worden, waarvan de kwaliteit verlaagd is, maar die aangepast zijn om zo min mogelijk storend te zijn voor de gebruiker.

2.2.4 Vergelijking van raytracing en rasterisatie

Deze sectie maakt een vergelijking tussen rasterisatie en raytracing. Uitgebreide informatie is hierover te vinden in de volgende werken [103, 102, 97, 101, 96, 5, 93, 68, 39]. Hieruit blijkt dat raytracing heel wat voordelen heeft ten opzichte van rasterisatie, zodat deze het nuttigste lijkt als basis voor toekomstige toepassingen.

I Rasterisatie

Hardware rasterisatie is snel genoeg voor realtime visualisatie van relatief kleine scenes, die bewegende objecten bevatten. Dynamische scenes worden eenvoudig behandeld, door per beeld de hele scene te verwerken. Voor relatief kleine scenes werkt het sneller dan raytracing, welke eerst nog een versnellingsstructuur moet opbouwen. Grote scenes daarentegen moeten benaderd worden door een kleiner aantal polygonen te tonen per beeld. Level-of-detail technieken [26, 41] zorgen zo bijvoorbeeld voor vereenvoudigingen van de geometrische modellen. Die worden ofwel op voorhand berekend ofwel tijdens de uitvoering. Geometrische vereenvoudiging vraagt enorm veel voorverwerkingstijd voor heel grote modellen. Indien het berekend wordt tijdens de uitvoering, worden extra verspillende berekeningen gedaan die niet nodig zijn bij raytracing. Occlusion culling [29, 120], cel-gebaseerde occlusion culling [91] en de hiërarchische Z-buffer [46] zorgen ervoor dat minder primitieven doorgezonden worden naar de hardware door een deel van de bedekte primitieven weg te laten.

Grotere beeldresoluties zijn voordeliger voor hardware rasterisatie, aangezien beide raytracing en rasterisatie lineair scaleren met de resolutie. Rasterisatie is geïmplementeerd in goedkope en sterk parallelle hardware, waarvan de performantie sterk toeneemt, en het is aanwezig in elke massaproductiecomputer. Deze hardware wordt ook steeds makkelijker te programmeren, zodat realistischere beelden gegenereerd kunnen worden door de toevoeging van benaderingen van steeds duurdere berekeningen van natuurlijke illuminatie-effecten. Maar het wordt steeds moeilijker om in dit rasterisatie kader dergelijke verbeteringen toe te voegen, terwijl die natuurlijk en correct berekend worden door globale illuminatie-algoritmen. Elke polygoon wordt individueel beschouwd en tijdens de verwerking ervan is er geen toegang mogelijk tot de andere polygonen in de scene. Er zijn dus geen interacties mogelijk met de andere objecten, tenzij er meerdere malen geïtereerd wordt over alle objecten [97].

Hardware rasterisatie berekent enkel directe illuminatie. Dat betekent dat alleen licht, dat direct van de lichtbron komt, gebruikt wordt in de berekeningen. Daarom wordt het vaak gemengd met globale illuminatie-algoritmen om hun effecten te integreren. De indirecte illuminatie wordt benaderd door allerlei methoden, die snel zijn, maar niet correct. Voorbeelden zijn bump mapping [16], reflection mapping [15] en shadow mapping [114]. Rasterisatie levert relatief zwakke benaderingen van fotorealistische beelden. Wanneer globale illuminatie-

algoritmen toegepast worden voor bepaalde effecten, worden deze meestal op voorhand uitgevoerd. De resultaten worden dan opgeslagen in texturen, die doorgegeven worden aan de rasterisatie hardware. Dit en andere voorverwerking leidt tot tijdrovende manuele aanpassingen, wat de dynamiek van het visualisatie algoritme sterk schaadt.

II Raytracing

Een goede vergelijking wordt geleverd door Ingo Wald in verschillende documenten [103, 96, 102]. Raytracing scaleert beter met het aantal aanwezige objecten in de scene dan rasterisatie hardware. Voor enorm grote modellen wordt de extra kost zelfs zeer miniem. Volgens Wald et al. [103] presteert raytracing beter vanaf een scene met ongeveer *één* miljoen driehoeken. Daardoor maakt raytracing meer kans om echt gebruikt te worden in de toekomstige interactieve globale illuminatie-algoritmen. Deze scalering heeft een logaritmische aard dankzij de gebruikte versnellingsstructuren, zoals eerder vermeld in II.1. Door deze structuur wordt het zoeken naar een bepaalde intersectie met een object gereduceerd tot een binaire zoektocht. Heel wat delen van de scene, die geen belang hebben voor een bepaald kijkpunt, worden op dergelijke manier gewoonweg links gelaten, terwijl deze bij hardware rasterisatie toch getest worden. View frustum en occlusion culling zijn dus automatisch ingebouwd op deze manier. De data wordt bovendien enkel ingeladen wanneer het nodig is, zodat niet constant directe toegang tot de gehele scene nodig is. In de caches moet dan telkens slechts het vereiste deel aanwezig zijn. Bij interactieve applicaties met dynamische scenes moet deze structuur voortdurend onderhouden worden.

Radiosity en photon mapping kunnen minder goed dienen als een globaal kader. Ze scaleren minder goed met het aantal objecten en vereisen meer communicatie en synchronisatie zodat ze parallellisme minder goed kunnen uitbuiten. Beide technieken vereisen ook veel voorberekeningen. Verder is radiosity voornamelijk goed voor diffuse interreflecties, maar niet voor reflecties. Photon mapping vereist dure opzoekingen in kd-bomen van dichtste buren.

Raytracing doet geen overbodige shading berekeningen zoals bij rasterisatie het geval is. Dat is te danken aan het feit dat elke volgende stap in het algoritme pas uitgevoerd wordt, nadat de vorige stap toestemming geeft. Voor een primaire straal wordt eerst de versnellingsstructuur doorlopen, waarna een intersectie gedaan wordt met enkel de overblijvende objecten, zodat het zichtbare oppervlakpunt bepaald wordt. Daarna wordt dat pas ingekleurd.

Daarenboven is er de mogelijkheid om aparte stralen te schieten op gewenste plaatsen. Dit laat toe adaptieve anti-aliasing tewerk te stellen of adaptief de kwaliteit van het beeld aan te passen aan de hand van bepaalde informatie, om bijvoorbeeld details te verduidelijken of om belangrijkere delen sterker te bemonsteren.

Raytracing kan sterker parallellisme en coherentie uitbuiten. Het kan eenvoudig de te visualiseren pixels van het beeld verdelen over meerdere processoren, coherente stralen bundelen en die vervolgens in parallel berekenen, etcetera. Rasterisatie scaleert heel wat minder goed. De coherentie wordt sterker wanneer de beeldresolutie stijgt, aangezien de inhoud van de pixels dan dichter bij elkaar ligt in objectruimte. De lineaire scalering met de beeldresolutie wordt op die manier verzacht.

Het is mogelijk meer primitieve objecten te ondersteunen dan enkel driehoe-

ken. Er kunnen eenvoudig volumedataverzamelingen, splines, onderverdelingsoppervlakken, puntenwolken, lightfields, en zo meer geïntegreerd worden en dat daarenboven tegelijkertijd in dezelfde scene. Dit is helemaal niet zo eenvoudig bij rasterisatie.

Globale illuminatie-algoritmen bieden fotorealistische beelden, wat veel voordelen heeft ten opzichte van de benaderde beelden van rasterisatie. Het steunt op de fysische wetten van lichttransport, waardoor de illuminatie overeenkomt met de realiteit en tevens de effecten, die waargenomen worden in de natuur, inherent zijn in de berekeningen. Aldus kunnen schaduwen, reflecties en refracties weergegeven worden op willekeurige objecten. Er moeten dan geen extra stappen gedaan worden, waarin meer berekend wordt dan noodzakelijk is of waarin objecten nog gesorteerd moeten worden en die dan toch nog leiden tot slechts benaderingen van de echte effecten. Fysische correctheid is trouwens bij veel applicaties, voornamelijk in de industrie, van cruciaal belang om daarop belangrijke beslissingen te vestigen. Aangezien deze wetten slechts een eenvoudig model zijn van het lichttransport ten opzichte van de wetten die men in de fysica reeds kent, zijn er toch nog een deel effecten niet beschikbaar. Hoe sneller de hardware en software implementaties worden, hoe meer dergelijke effecten kunnen bijgevoegd worden. Dankzij dit fotorealisme zullen mensen zich beter kunnen associëren met de gevisualiseerde omgeving, waardoor de gebruikerservaring zal stijgen.

Bij rasterisatie kunnen tegenwoordig ook meerdere effecten ingevoegd worden door de mogelijkheid shaders te programmeren die een deel van de vaste grafische pijplijn vervangen. Deze zijn beperkt tot lokale illuminatiemodellen, terwijl de shaders bij raytracing dat niet zijn. Het is bovendien lastiger om dezelfde effecten via shaders in te voegen bij rasterisatie dan bij raytracing. Ze eisen bij rasterisatie dan ook meerdere stappen en bieden vaak slechts een benadering van het effect. Reflecties worden bijvoorbeeld benaderd door een reflection map. Die is enkel correct voor *één* bepaald punt, terwijl de programmeur vaak zelf de beste resultaten moet uitzoeken en instellen voor elk effect apart. Reflectie van het object op zichzelf is bovendien niet mogelijk. Bij raytracing zijn de effecten correct en automatisch bepaald door de berekeningen.

Raytracing wordt spijtig genoeg nog niet goed ondersteund door de huidige hardware, hoewel er toch al redelijk wat onderzoek naar gedaan is en er reeds speciale hardware voor ontwikkeld is.

2.3 Frameless Rendering

Fotorealistische visualisatie is vrij duur en kan gewoonlijk niet interactief getoond worden op bijvoorbeeld individuele massaproductiecomputers. Allerlei technieken II trachten dit toch te verwezenlijken. De meeste technieken proberen ofwel raytracing te versnellen, met als doel continue fotorealistische visualisatie, ofwel raytracing te benaderen, met het doel interactiviteit te bereiken. Frameless rendering is een visualisatie strategie, die beide doelen combineert. Het levert een trade-off tussen beeldkwaliteit en berekeningstijd, waarbij het fotorealisme mogelijk maakt in interactieve toepassingen. Bishop et al. [14] beschreven dit idee voor het eerst.

Een animatie, zoals in figuur 1.2, toont meerdere opeenvolgende beelden snel na elkaar. Vaak kan een raytracer niet snel genoeg vernieuwde beelden genere-

ren om interactiviteit en de illusie van continuïteit te onderhouden. Daarom zal frameless rendering niet telkens een volledig beeld berekenen, maar zal het herhaaldelijk slechts een fractie van het aantal pixels herberekenen. De kwaliteit van het beeld daalt dan tijdelijk gedurende interacties of animaties, maar de snelheid van beeldopeenvolgingen en de kwaliteit van de interactie zal stijgen.

Binnen dit frameless rendering kader kunnen diverse concepten aangewend worden om de resulterende snelheid op te drijven en de kwaliteit te verbeteren. Zo benut het principes als stochastische bemonstering, importance sampling, progressieve verfijning, asynchrone beeldvernieuwing, herprojectie van monsters, uitbuiting van spatio-temporele beeldcoherentie en het gebruik van beeldruimtefilters. Die principes zullen nader verklaard worden.

In sectie 2.3.1 wordt de reden voor het bestaan van frameless rendering verder uitgelegd. Er zijn reeds meerdere strategieën uitgedacht om dit idee toe te passen. Daarvan komen er enkele aan bod in sectie 2.3.5. Het volgende hoofdstuk 3 bevat dan een uiteenzetting van de implementatie van de belangrijkste technieken, gevolgd door een bespreking van de resultaten 4.

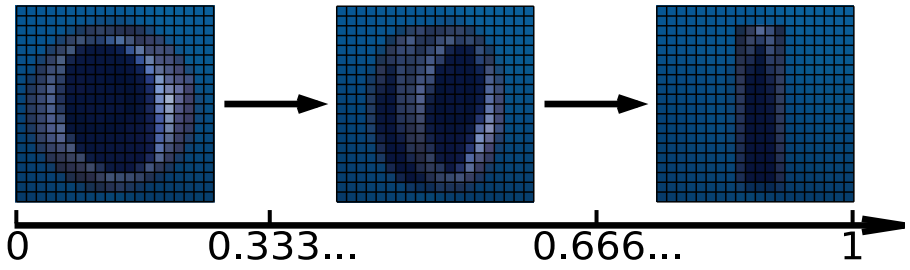
2.3.1 Motivatie voor frameless rendering

Om een 3D-omgeving weer te geven op een 2D-beeld wordt normaal door een visualisatie algoritme een kleur berekend voor elke pixel. Indien een enkel statisch beeld bereikt moet worden, is de kwaliteit van dit beeld meestal de belangrijkste factor. Het heeft dan minder belang dat er een tijdje gewacht moet worden vooraleer het volledig berekend is. Indien nu meerdere opeenvolgende beelden gegenereerd moeten worden van dezelfde scene, waarin veranderingen kunnen gebeuren, is er een visualisatie algoritme nodig dat genoeg beelden per seconde kan genereren om de gebruiker vloeiende beelden te doen waarnemen.

Dit aantal beelden per seconde mag niet verward worden met de critical fusion flicker (CFF) [69, 117, 9, 2, 67]. De mens neemt namelijk geen individuele beelden waar, wanneer er meer opeenvolgende beelden per seconde getoond worden dan de CFF. Dit geldt voor beelden waarin geen veranderingen voorkomen, net als voor beelden waarin wel veranderingen voorkomen. Het oog integreert immers licht over een bepaalde tijd, en indien discrete beelden elkaar snel genoeg opvolgen, zullen de momenten, waarop de pixels zwart zijn bij bijvoorbeeld CRT's, tussen de beelden niet waargenomen worden door het menselijk visueel systeem. De inkomende informatie lijkt dan continu te zijn. De CFF wordt door meerdere factoren bepaald, zoals de resolutie van het scherm, de afstand van de kijker tot het scherm, de vermoeidheid van de gebruiker, enzovoort. Dat aantal bevindt zich tussen de dertig en vijftig beelden per seconde [73]. Met deze CFF moet frameless rendering geen rekening houden, aangezien het beeldscherm ervoor zorgt dat we daar geen last van hebben. Dat toont bijvoorbeeld beelden aan 75 Hertz (75 beelden per seconde).

Het aantal beelden dat per seconde getoond moet worden zodat mensen bewegingen vloeiend waarnemen, ligt rond 15 *à* 18 beelden per seconde [67, 116, 73]. Dit is wel relevant voor een visualisatie algoritme. Ons visueel systeem ervaart opeenvolgende beelden met kleine verschillen in de toestand van objecten als echte vloeiende bewegingen, terwijl er eigenlijk toch kleine overgangen tussen de toestanden afwezig zijn. Dat heeft dus niets te maken met de integratie van het licht, maar is een illusie. Onze hersenen proberen afzonderlijke

toestanden te relateren. Voor films bijvoorbeeld is het standaard aantal beelden vastgelegd op 24 beelden per seconde. Indien men een film maakt is het natuurlijk minder belangrijk hoeveel berekeningstijd er geïnvesteerd moet worden in een enkel beeld, aangezien films geen realtime beelden vereisen. Verder is er ook een verschil tussen beelden die gemaakt zijn met een echte camera en beelden die gegenereerd zijn door een computer. Een echte camera genereert namelijk automatisch motion blur, doordat zijn lens een korte tijd openblijft en licht opneemt, terwijl bij computergegenereerde beelden vaak een enkel monster genomen wordt op een bepaald moment, in plaats van over een bepaald interval. Het gevolg is dat voor dit laatste geval een hogere frequentie van beelden nodig is om hetzelfde resultaat te bereiken voor de mens.



Figuur 2.24: Indien 3 beelden per seconde getoond worden met elk een resolutie van 20×20, is er 0.0008333... seconden beschikbaar per pixel.

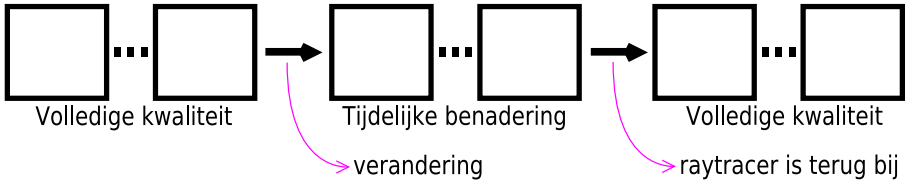
Indien interactiviteit gewenst is, worden realtime beelden verwacht. Aangezien men dan bijvoorbeeld 24 beelden per seconde verlangt, is er per beeld slechts twee en een halve seconde beschikbaar om het te creëren. Wanneer een beeld dan bestaat uit 800×600 pixels, is er ongeveer 0.000005 seconde beschikbaar voor elke pixel. Deze snelheden worden momenteel bereikt met behulp van rasterisatie hardware van massaproductiecomputers voor niet al te complexe scenes. Het behaalt dat ten koste van sterk benaderde beelden, waarbij het beeldruimte coherentie uitbuit door bijvoorbeeld interpolatie van shading waarden over de pixels in het beeld. Verder bezit rasterisatie heel wat nadelen, zoals in de vorige sectie 2.2.4 uiteengezet is.

Fotorealistische beelden kunnen gewoonlijk niet interactief getoond worden. Toch zou dit zeer interessant kunnen zijn voor een aantal applicaties, zoals eerder aangehaald werd in 2.2.4. Ook bij het opbouwen van de scenes, die achteraf fotorealistisch weergegeven zullen worden, is interactiviteit gewenst. Wanneer er genavigeerd wordt doorheen een scene is de kwaliteit van minder belang dan wanneer men een object wil inspecteren, aangezien men dan gaat stilstaan voor dat object. Nu moet er vaak gewisseld worden van visualisatie algoritme om tijdens interactie snel terugkoppeling te krijgen en bij stilstaande inspectie volle kwaliteit te verkrijgen. Zodoende zijn er veel ongemakken. Indien mogelijk, is het dus wenselijker om fotorealistische beelden te kunnen tonen, terwijl interactiviteit ook mogelijk blijft.

Frameless rendering technieken ruilen beeldkwaliteit in voor interactiviteit door beide denkbeelden te combineren. Interactiviteit is dan mogelijk in veel gevallen waarin het eerder niet mogelijk was. Volgens Bishop et al. [14] worden

de interacties zelfs aanzienlijk vloeiender. De reden is dat pixels onafhankelijk van elkaar berekend worden, waarbij ze telkens een zo recent mogelijk tijdstip voorstellen. Frameless rendering is een benadering van de "golden thread", die besproken wordt door Bergman et al. [12]. Indien opeenvolgende beelden veranderen door aanpassingen aan de scene of doordat de camera bewogen wordt, zal het benaderde beelden voortbrengen. De kwaliteit van het beeld gaat dus tijdelijk achteruit, maar het beeld zal hier blijven verbeteren richting de juiste waarden voor de huidige moment. De gebruiker zal bijgevolg slechts kort een gedeeltelijke vernieuwing waarnemen. Wanneer de scene statisch wordt, zal deze kwaliteit geleidelijk aan verbeteren tot het convergeert met de beeldkwaliteit van een fotorealistisch visualisatie algoritme of van eender welk onderliggend pixel-gebaseerd visualisatie algoritme. De coherentie in de tijd is dan sterker, waardoor monsters gedurende langere periodes geldig blijven. Zodoende is er meer tijd beschikbaar om exacte waarden te genereren en zal het beeld zo snel mogelijk convergeren naar dit correcte beeld.

Op deze manier wordt de interactieve manipulatie van scenes, die gevisualiseerd worden met globale illuminatie-algoritmen, dus beter benaderd dan via rasterisatie hardware, tenminste voor beeldopeenvolgingen met genoeg temporele coherentie. In veel applicaties zijn vloeiende interacties ten koste van lichte degradaties in de beeldkwaliteit doeltreffender dan volle kwaliteitsbeelden die schokkende interacties veroorzaken. Bovendien is berekeningstijd van monsters een minder belangrijke factor dankzij frameless rendering.



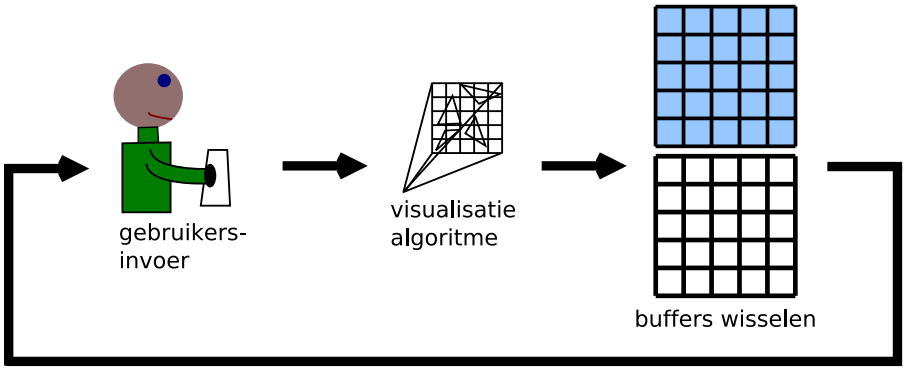
Figuur 2.25: Het getoonde beeld heeft volledige kwaliteit tot er een verandering optreedt in de scene. Dan zullen de gedeeltelijke vernieuwingen tijdelijk zichtbaar zijn totdat de raytracer tijd genoeg gehad heeft om de achterstand in te halen. Tenslotte wordt terug een beeld met de volledige kwaliteit getoond.

Volgens David Luebke [13, 111] zal de vraag naar hogere schermresoluties blijven toenemen. Indien rekening gehouden moet worden met het gehele bereik van de gevoeligheid van het menselijk visueel systeem, zal het nog lang duren vooraleer de benodigde resolutie behaald kan worden. De grootte van de resolutie zal daarbij sterker toenemen dan de beschikbare bandbreedte tussen de computer (GVE) en het beeldscherm. Daarenboven stellen andere technieken om beelden weer te geven, zoals holografieën, er nog grotere eisen aan. Het wordt dan veel moeilijker om telkens een heel nieuw beeld door te sturen naar het beeldscherm. Dit probleem tracht men onder andere op te lossen door zoveel mogelijk parallel te doen, maar ook ondanks dat parallellisme zal het nog lang duren eer die resoluties bereikt kunnen worden. Parallelle oplossingen verdelen de kost voor het visualiseren van de hele resolutie over meerdere processoren en zullen de resultaten achteraf moeten combineren op het beeld. Maar is het wel nodig om de hele resolutie te herberekenen voor elk beeld? Een andere mogelijkheid is het beeld selectief te vernieuwen. Interactieve visualisatie bezit

namelijk een sterke temporele coherentie. Frameless rendering buit die coherentie uit, waardoor het de beschikbare bandbreedte efficiënter benut. Het wordt dan immers mogelijk om enkel de veranderingen in het beeld door te sturen. Daarmee wordt de kost sterk verlaagd, omdat niet elke pixel van die resolutie telkens herberekend moet worden. Adaptive frameless rendering, ontwikkeld door Luebke et al. [32] 3.4, kan de bandbreedtevereisten reduceren met *één* tot twee grootte-ordes.

2.3.2 Werking

Normaal gezien wordt dubbele buffering gebruikt om beelden weer te geven. Dat betekent dat elke pixel van een bepaald beeld berekend wordt voordat het getoond zal worden. Die pixels worden dan opgeslagen in een buffer, totdat het beeld afgewerkt is. Daarna wordt de buffer, die het huidig getoonde beeld bevat, omgewisseld met het nieuw berekende beeld, dat dan zichtbaar wordt.

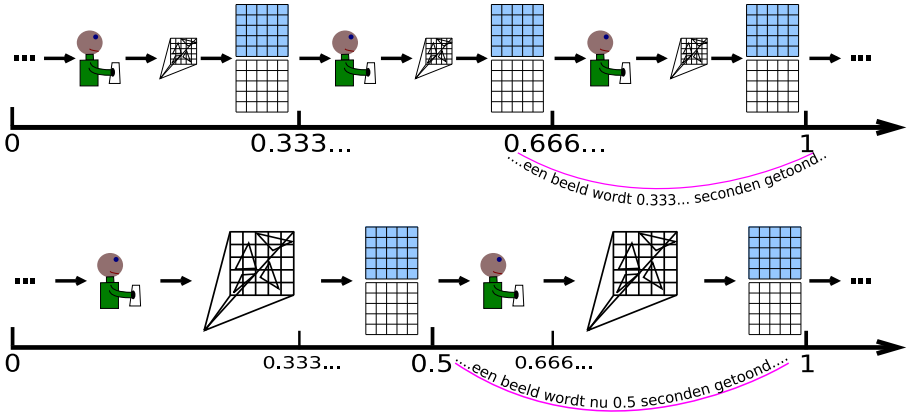


Figuur 2.26: Dubbele buffering: schets van de operatie volgorde. Telkens wordt eerst de gebruikersinvoer ingelezen, indien die er is. Vervolgens wordt elke pixel van het beeld gevisualiseerd en in de momenteel niet getoonde buffer opgeslagen. Pas wanneer dat klaar is, worden de buffers omgewisseld en verschijnt de nieuw gevulde buffer op het scherm.

Elk beeld zal getoond worden op het scherm vanaf het moment dat het reeds een bepaalde ouderdom heeft, namelijk de opgetelde berekeningstijd van alle pixels, tot het moment dat het ongeveer dubbel zo oud zal zijn, afhangende van de benodigde berekeningstijd voor de pixels van het volgende beeld. Voor het berekenen van alle pixels in *één* beeld worden de camera- en objectparameters van eenzelfde tijdstip gebruikt. Dat is namelijk het tijdstip dat zich vlak voor de berekening van de eerste pixel van dat beeld bevindt.

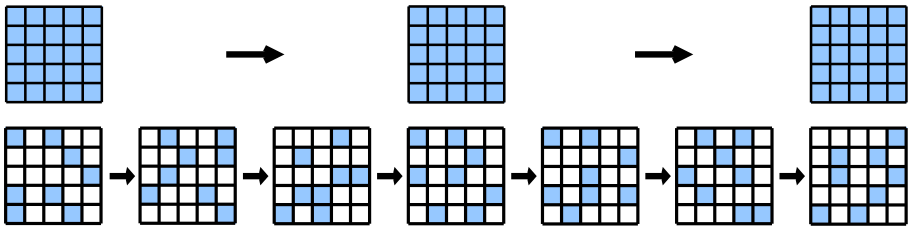
Indien het visualisatie algoritme de benodigde snelheid niet kan halen, wordt het tonen van dat nieuwe beeld tegengehouden totdat het volledig berekend is, aangezien die code pas uitgevoerd wordt nadat de code, die elke pixel berekent, uitgevoerd is. Het vorige beeld zal dan nog ouder worden, terwijl het al die tijd zichtbaar zal blijven op het scherm. Op die manier ontstaan er haperingen en gaat de interactiviteit aldus achteruit. De gebruiker gaat zich bijgevolg ook minder ondergedompeld voelen. In veel toepassingen is het derhalve wenselijker om traag opeenvolgende hoge kwaliteitsbeelden te vervangen door sneller

opeenvolgende beelden met een lagere kwaliteit.



Figuur 2.27: (a) Dit visualisatie algoritme kan in $\frac{1}{3}$de seconde een nieuw beeld genereren. Elk beeld wordt dan $\frac{1}{3}$de seconde getoond. Het beeld zal dus $\frac{2}{3}$de seconde oud worden. (b) Dit algoritme is trager en genereert om de halve seconde een nieuw beeld. Elk beeld zal een halve seconde getoond worden en zal een hele seconde oud worden.

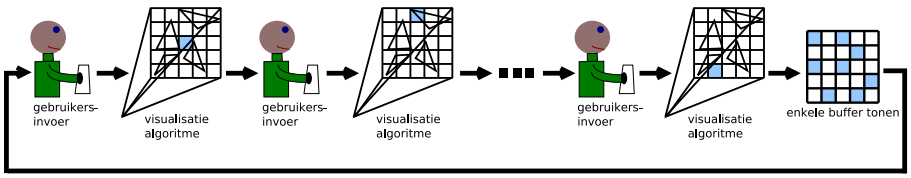
Frameless rendering kan dat verwezenlijken. Daarbij wordt aangenomen dat het berekenen van elke pixel onafhankelijk kan gebeuren van zijn buurpixels. In dit kader zal er niet meer gewacht moeten worden met het tonen van de huidige informatie van pixels tot andere pixels berekend zijn. Pixel-gebaseerde visualisatie algoritmen zoals raytracing zijn daarom geschikt, terwijl rasterisatie niet geschikt is. Doch is er ook een frameless rendering techniek geïmplementeerd voor rasterisatie [115] I. Het visualisatie algoritme zelf wordt niet veranderd om frameless rendering te integreren. Telkens wordt er opgedragen voor welke pixels het stralen zal moeten schieten.



Figuur 2.28: Bij dubbele buffering (bovenste rij) wordt voor elk nieuw beeld elke pixel vernieuwd. Wanneer frameless rendering met behulp van een tabel (onderste rij) telkens $\frac{1}{3}$de van het aantal pixels vernieuwt, kunnen er 3 nieuwe beelden gemaakt en getoond worden op de tijd dat er met dubbele buffering 1 nieuw beeld aangemaakt is. De blauwe vakjes geven aan welke pixels vernieuwd worden.

In het oorspronkelijke idee van frameless rendering [14] wordt voorgesteld om niet telkens te wachten tot een volledig beeld berekend is om het pas daarna te tonen. Zij zullen telkens met regelmatige tussenpozen een nieuw beeld tonen wanneer een deel (bijvoorbeeld een vijfde) van het aantal pixels herberekend is.

Elke pixel wordt berekend met de laatste camera- en objectparameters, in tegenstelling tot dubbele buffering. Verder wordt hier slechts *één* buffer gebruikt, waarbij nieuwe monsters gewoon ingevuld worden op de corresponderende plaatsen. Ze overschrijven dus de vroegere kleurwaarden op de respectievelijke posities, terwijl de kleurwaarden van de andere pixels behouden worden. Op die manier wordt er in de tijd een supersampling gedaan en in de ruimte een subsampling. Supersampling in de tijd betekent dat er na kortere tijdsperiodes opnieuw monsters genomen worden. Het verloop over de tijd wordt dan beter voorgesteld en er moet minder lang gewacht worden op nieuwe beelden. Ruimtelijke subsampling betekent dat telkens wanneer monsters genomen worden, er minder dan het benodigd aantal monsters zullen zijn om het hele beeld (elke pixel) op een bepaald moment voor te stellen. De ruimtelijke voorstelling of de kwaliteit en de integriteit van het beeld worden zo ingeruild voor het beter opvolgen van veranderingen doorheen de tijd en voor een toegenomen interactiviteit. Uiteindelijk zullen er evenveel pixels berekend worden in het basis algoritme van frameless rendering dan bij dubbele buffering, alleen zullen ze anders verspreid zijn over tijd en ruimte. Het resultaat is een hoger aantal beelden per seconde, die recentere informatie bevatten, in vergelijking met dubbele buffering. Het getoonde beeld representeert voortdurend de recentste toestand die bereikt kan worden met de beschikbare verwerkingskracht. Indien de gebruiker een bepaalde manipulatie doet, zal zeer snel een deel van de pixels de resulterende verandering in de scene voorstellen.



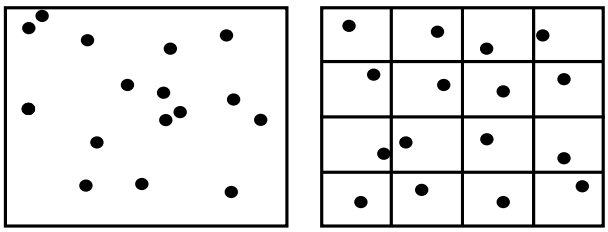
Figuur 2.29: Frameless rendering: schets van de operatie volgorde. Vooraleer elke nieuwe pixel te visualiseren, wordt gekeken of de gebruiker iets ingevoerd heeft. Indien dat het geval is, worden de corresponderende parameters aangepast. Daarna wordt random een pixel gekozen en die wordt gevisualiseerd met die parameters. Het resultaat overschrijft de corresponderende plaats in de enkele buffer. Indien een voldoende aantal pixels (bijvoorbeeld een derde van de beeldresolutie) berekend zijn, wordt de enkele buffer op het scherm getoond. De witte vakjes stellen oudere kleurwaarden voor. De blauwe vakjes stellen de laatst berekende pixels voor.

De pixels die herberekend moeten worden, zullen random in het beeldvlak uitgekozen worden. Aangezien ze *één* voor *één* random gekozen worden en telkens op het tijdstip dat ze berekend worden de laatste camera- en objectparameters gebruiken, zullen de pixels op een random tijdstip, binnen het interval van de volledige herberekening van alle pixels, vernieuwd worden. Op die manier wordt een stochastische bemonstering benaderd. Temporele aliasingartefacten, eigen aan een uniforme verdeling, worden dan omgezet in temporele ruisartefacten. Dat betekent hier dus dat bewegende objecten vloeiender waargenomen zullen worden in plaats van dat er afzonderlijke beelden waargenomen worden van de nieuwe toestanden van de objecten. Aangezien per nieuw beeld niet tel-

kens alle pixels vernieuwd worden, is er ruis zichtbaar. Dat zijn pixels die een verouderde kleurwaarde bevatten, welke op dat tijdstip vervangen zou moeten worden door de huidige kleurwaarde. Ruis is voor het menselijk visueel systeem minder storend dan aliasing. In de bovenstaande figuur 2.29 zullen op elk tijdstip, dat een beeld getoond wordt, de corresponderende blauwe vakjes (de nieuw berekende pixels) het recentste zijn. Die sluiten dus het dichtst aan bij de huidige moment en ze zullen tevens recenter zijn dan bij dubbele buffering. De andere pixels bevatten ruis wanneer de kleurwaarden, die daar zichtbaar zouden moeten zijn, veranderd zijn sinds de laatste bemonstering op die pixel.

De random bemonstering is beter dan wanneer bijvoorbeeld telkens van voor af aan elke pixel wordt vernieuwd 3.2.2. Dan zal het lijken alsof het beeld verscheurt, wat veel minder aangenaam is voor het menselijk visueel systeem.

Indien pure random posities genomen worden, zullen monsters op bepaalde plaatsen bij elkaar gaan liggen [27]. Het is beter een bepaalde bemonstering te nemen waarbij monsters elkaar een beetje mijden, zoals in Poisson disk sampling of jittered sampling. Zo zullen nieuwe kleurwaarden verschijnen op goed verspreide plaatsen in het hele beeld. Het lijkt dan alsof er een gelijktijdige vernieuwing gebeurt in het heel beeld.



Figuur 2.30: (a) Voorbeeld van pure random posities (b) Voorbeeld van random posities binnen bepaalde intervallen

Om te verzekeren dat uiteindelijk alle pixels vernieuwd zullen zijn voordat men herbegint met de vernieuwing wordt er een tabel gebruikt 3.2.2. Daarin wordt bijgehouden welke pixels reeds herberekend zijn. Recent herberekende pixels zullen niet opnieuw berekend worden voordat andere pixels, die sindsdien nog niet vernieuwd zijn, herberekend zijn.

Normaal krijgt de CVE per nieuw beeld *één* keer de kans om de gebruikersinvoer te verwerken. Hier zal dat frequenter gebeuren, waardoor kleinere veranderingen doorgevoerd worden, in plaats van dat telkens ineens een heel deel aan opgestapelde bewegingen doorgevoerd worden. Zo zal bij de berekening van elke pixel rekening gehouden worden met de recentste veranderingen. Ze stellen dan het exacte tijdstip voor waarop ze berekend werden, terwijl ze anders het tijdstip voorstellen waarop de eerste pixel van het beeld berekend werd. Op deze manier krijgen gebruikers sneller terugkoppeling, aangezien veranderingen in de scene en bewegingen van de camera sneller voorgesteld zullen worden, al is het maar door een gedeelte van de pixels.

Tenslotte is het ook mogelijk elke individuele pixel rechtstreeks op het scherm te tonen, nadat hij berekend is. Elke verandering komt dan rechtstreeks op het scherm terecht. Het nadeel is dan dat pixels lijken te fonkelen. Hier zijn

trouwens aangepaste beeldschermen voor nodig, waarbij elke pixel onafhankelijk kan gemanipuleerd worden.

2.3.3 Eigenschappen

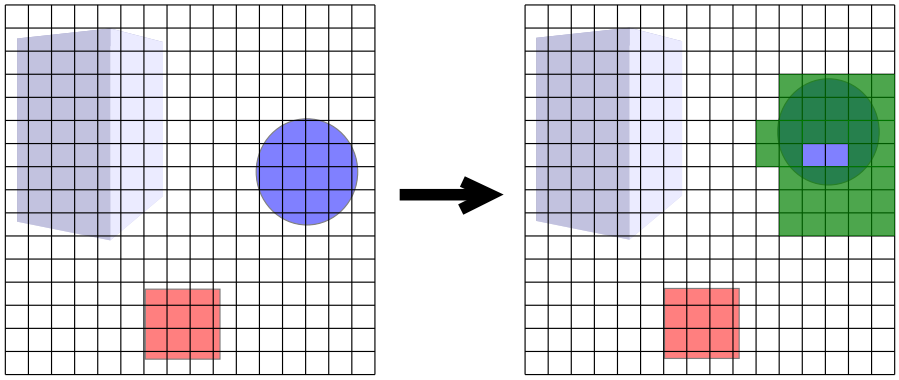
Dankzij frameless rendering worden tussentijdse resultaten getoond. Daardoor zullen de veranderingen in bewegingen sneller opeenvolgen en dus vloeiender lijken. De overgangen tussen opeenvolgende beelden zijn dan namelijk minder hard, ook al wordt telkens slechts een gedeelte van de gehele resolutie herberekend. De mens kan bewegingen duidelijker onderscheiden bij sneller opeenvolgende gedeeltelijke vernieuwingen dan bij trager opeenvolgende volledige vernieuwingen. Indien ingewikkelde bewegingen voorvallen, moet het menselijk visueel systeem meer werk verrichten, zodat resterende artefacten minder hard opvallen. Beweging weergeven is dus belangrijker bij interactie dan het weergeven van volledig correcte spatiale beelden. Monsters, die bedoeld zijn voor de correcte spatiale weergave, kunnen dan geruild worden voor monsters, die bedoeld zijn voor correctere temporele weergave.

Aangezien de gebruiker nu minder lang moet wachten op terugkoppeling, in de vorm van zichtbare veranderingen in het beeld, kan hij reeds vroeger een begrip opbouwen van de huidige situatie, zodat hij sneller kan reageren en doorwerken.

Door de slechts gedeeltelijke vernieuwing zullen ook oudere kleurwaarden van een groot deel pixels zichtbaar blijven. Indien vernieuwingen niet snel genoeg gebeuren, wordt dat waargenomen als **ruis**. Maar dat is minder storend dan aliasing. Deze ruis bepaalt de waargenomen kwaliteit van de beelden en wordt beïnvloed op een aantal manieren, die hieronder aan bod zullen komen. Meer informatie hierover wordt gegeven door Ellen J. Scher Zagier [117].

Ten eerste zorgt de hoeveelheid temporele coherentie voor een verschil in waarneming van deze ruis. Wanneer opeenvolgende beelden sterke temporele coherentie vertonen, zullen weinig van deze oudere kleurwaarden fout zijn. Dat komt omdat er dan weinig veranderd is ten opzichte van het vorige beeld. Er hebben bijvoorbeeld slechts weinig objecten of kleine objecten een geringe afstand bewogen, het kijkpunt is bijvoorbeeld slechts lichtjes aangepast of de illuminatie in de scene is amper veranderd. Elk monster heeft dan een grotere kans dezelfde kleur te behouden. Dit leidt tot minder verspilling van berekeningen, doordat een deel van de pixels die toch niet veranderd zijn in opeenvolgende beelden, niet herberekend worden. Ze kunnen gewoon hergebruikt worden in het volgende beeld. Zo wordt de temporele coherentie uitgebuit. Op die momenten zal het getoonde beeld weinig fouten tonen en het perfecte beeld benaderen. Indien opeenvolgende beelden zwakkere temporele coherentie vertonen, zullen meerdere monsters van kleurwaarde veranderd zijn. Dat gebeurt wanneer er grotere veranderingen in de scene, grotere camerabewegingen of grotere illuminatieveranderingen voorgevallen zijn. Zodoende zullen meer fouten zichtbaar worden in het beeld, aangezien monsters van verschillende tijdstippen in de tijd samen zichtbaar zijn in *één* beeld.

Indien opeenvolgende beelden niet veranderen, is de temporele coherentie op zijn toppunt. Er zullen geen fouten meer zichtbaar zijn en het resulterende beeld bevat de kwaliteit van het beeld dat gemaakt is door het onderliggende



Figuur 2.31: Witte vakjes: coherentie ; groene vakjes: veranderd tov vorig beeld

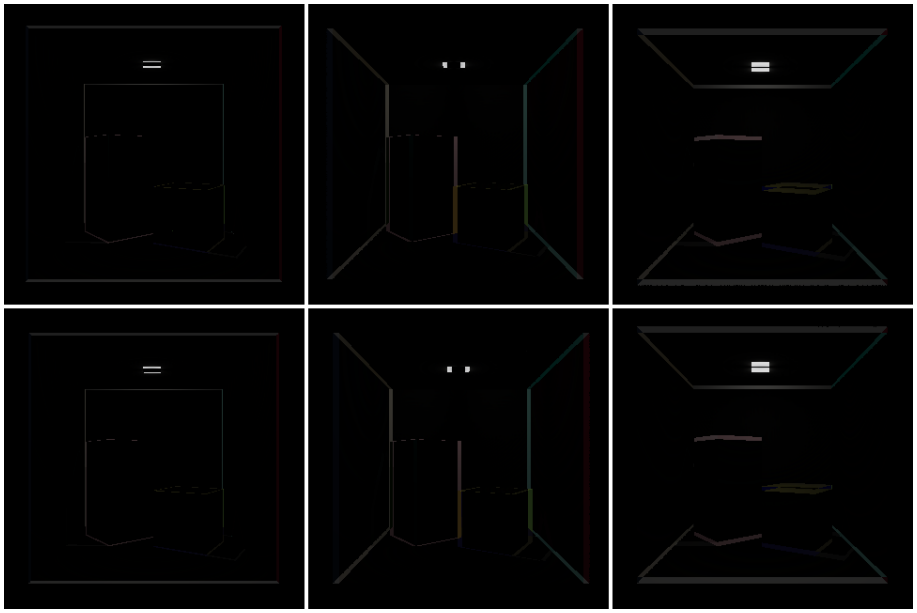
visualisatie algoritme.

Wanneer er helemaal geen temporele coherentie is, kan die dus ook niet uitgebuit worden. Frameless rendering zal dan tijdelijk leiden tot heel verwarrende beelden. Dit gebeurt wanneer men bijvoorbeeld direct (van beeld op beeld) teleporteert naar een andere plaats in de scene, zonder hiervoor interpolatie of dergelijke te gebruiken. Het vroegere beeld zal geleidelijk aan verdwijnen en gemengd worden met het nieuwe beeld. Dat nieuwe beeld wordt geleidelijk opgebouwd. Een dergelijke overgang is het lastigste voor frameless rendering en zal het langst duren. In interactieve applicaties worden dergelijke operaties gewoonlijk heel wat minder toegepast dan gewoon te bewegen doorheen de scene of lokaal dingen te bezichtigen of te veranderen.

Frameless rendering benadert op deze manier de "golden thread"[12]. Dat betekent dat bij herhaling van een aantal stappen, het beeld glad en continu zal convergeren naar het ideale beeld. Stel dat begonnen wordt met een opeenvolging van beelden van een scene, waarin geen bewegingen voorvallen. Daarna wordt verdergegaan met een opeenvolging van beelden van de scene met veel beweging en tenslotte wordt teruggekeerd naar een beeld dat niet meer verandert. Er zullen dan initieel ideale beelden getoond worden, zolang de bewegingen dus nog niet voorvallen. Wanneer de bewegingen beginnen en de vernieuwingen niet snel genoeg gebeuren, zal de temporele coherentie tijdelijk sterk verlaagd worden, waardoor de kwaliteit van het beeld achteruitgaat. Gedeeltelijke vernieuwingen worden namelijk kortstondig zichtbaar in het getoonde beeld. Daardoor verschijnen visuele artefacten, welke verspreid zijn over het gehele pad van de beweging op het beeld. Bij het afnemen van de bewegingen, zal de temporele coherentie geleidelijk aan toenemen. Dat komt omdat er telkens per beeld een deel van de pixels herberekend wordt. Het beeld zal naarmate de beweging afneemt en de tijd verdergaat minder fouten bevatten. Indien er zich minder bewegingen of langzamere bewegingen voordoen, gebeuren er namelijk minder veranderingen in het getoonde beeld. Het beeld convergeert zo geleidelijk aan terug naar het fotorealistisch beeld, dat gegarandeerd wordt door het onderliggende visualisatie algoritme. Wanneer de bewegingen totaal gestopt zijn, is een korte tijd nodig om uiteindelijk het ideale beeld terug te bereiken. Dat gebeurt op het moment dat elke pixel herberekend is sinds de laatste verandering in de scene. Het visualisatie algoritme heeft dan de kans om elke pixel de

waarde te geven die het op die moment heeft. Dit is gelijkaardig aan hoe het beeld op het netvlies van de mens uitgesmeerd wordt bij snelle bewegingen van de ogen, waarna het zich terug zal herstellen [62, 118]. Dat komt doordat het oog over een tijdsinterval van ongeveer 125 ms het inkomende licht integreert, waarbij motion blur ontstaat net als een uitsmering over het netvlies van objecten met een bepaalde retinale snelheid. Objecten hebben een retinale snelheid wanneer ze uit zichzelf bewegen, of wanneer het oog beweegt. Terwijl het beeld uitgesmeerd is, kan de mens ook niet duidelijk alle details waarnemen.

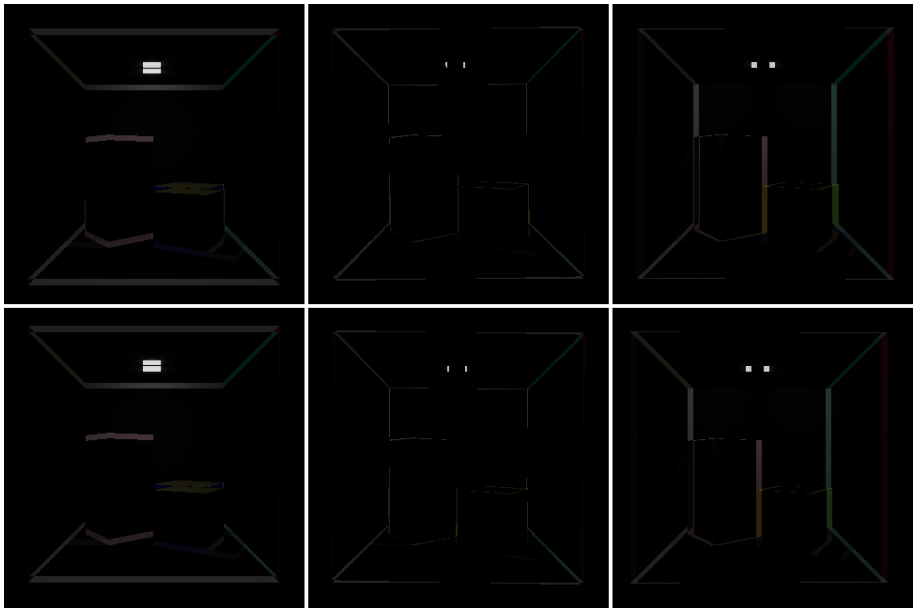
Camerabewegingen hebben de meeste invloed op de temporele coherentie, indien de bewegingen van objecten het effect niet teniet doen. Dat is een gevolg van het feit dat alle objecten op het beeldvlak van positie veranderen als de camera beweegt. Dit effect is sterker bij rotaties dan bij translaties. Translaties vertonen gewoonlijk sterkere temporele coherentie dan rotaties.



Figuur 2.32: Camera translatie: (a) Vooruit (17722 pixels), (b) Rechts (17722 pixels), (c) Opwaarts (31932 pixels), (d) Achteruit (16743 pixels), (e) Links (32170 pixels), (f) Neerwaarts (31552 pixels)

Bij een translatie volgens de optische as (vooruit en achteruit) verandert er niets op de plaats in het beeld waardoor de optische as gaat. Het object, dat op die plaats zichtbaar is, behoudt zijn positie op het beeldvlak. Hoe verder pixels afgelegen zijn van deze as, hoe groter de kans is dat de kleurwaarden zullen verschillen. De daar zichtbare objecten zullen verder verplaatst worden over het beeldvlak, naarmate ze verder van de optische as afliggen. Er zal informatie bijkomen indien voorwaarts getransleerd wordt, omdat de resolutie van de reeds zichtbare delen in de scene zal toenemen. Tevens worden objecten, die eerst bedekt waren, mogelijk zichtbaar of worden objecten, die eerst zichtbaar waren, dan bedekt. Wanneer achterwaarts getransleerd wordt neemt de resolutie af en komt er informatie bij aan de randen van het beeld. Ook is het mogelijk dat objecten dan verschijnen of bedekt worden. Bij zijwaartse, opwaartse of

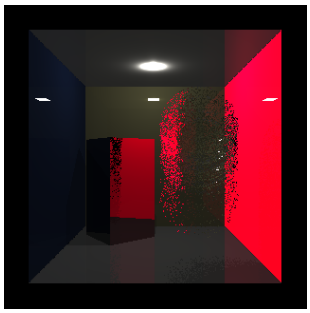
neerwaartse translaties zal in de richting van de translatie nieuwe informatie verschijnen en aan de andere kant verdwijnen. Geleidelijk zullen reeds zichtbare objecten gedeeltelijk of volledig kunnen bedekt worden of momenteel onzichtbare objecten kunnen verschijnen.
Bij rotaties van de camera zal een deel nieuwe informatie van de scene bijkomen aan de kant waarheen geroteerd wordt, en zal een deel verdwijnen aan de andere kant. Tevens zal aan die kant een tekort ontstaan aan informatie bij de nog zichtbare objecten, zodat de achtergrond er doorheen zichtbaar wordt. Rotaties over de optische as, de y-as in dit geval, veroorzaken de kleinste veranderingen. Hoe verder bepaalde punten van een object van de as afliggen, hoe sterker ze van positie zullen veranderen op het beeldvlak.



Figuur 2.33: Camera rotatie: positief over (a) X (35160 pixels), (b) Y (17392 pixels), (c) Z (35715 pixels), negatief over (d) X (34821 pixels), (e) Y (17320 pixels), (f) Z (35781 pixels)

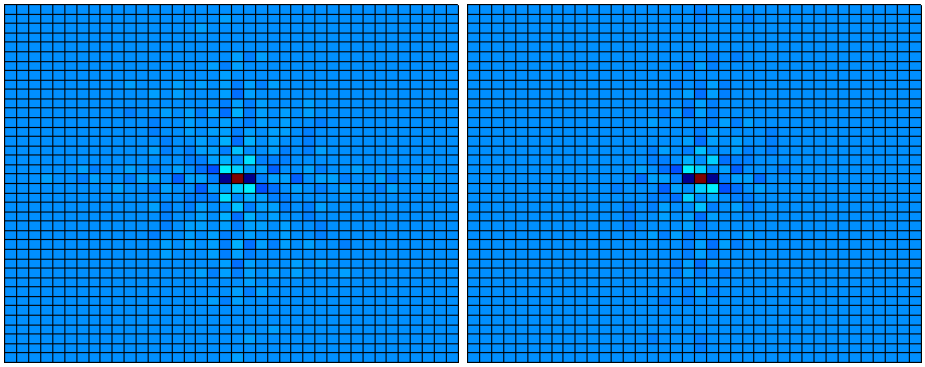
Dankzij de random sampling zal de ruis in frameless rendering bij bewegingen waargenomen worden als een benadering van motion blur [116]. Aangezien meerdere tijdstippen tegelijkertijd bemonsterd worden, zal een bewegend object tegelijkertijd op meerdere plaatsen aanwezig zijn tijdens het interval van de bemonstering. Dat interval bestaat uit de vroegst gegenereerde kleurwaarde van het object, die nog aanwezig is op het beeld, tot de laatst gegenereerde kleurwaarde van het object. Het voordeel van deze motion blur is het feit dat ze geen extra kost vormt en ook geen fouten bevat [118]. Het menselijk visueel systeem gedraagt zich trouwens op een gelijkaardige wijze, waardoor deze motion blur als natuurlijk overkomt. De mens interpreteert een individueel beeld waarop motion blur aanwezig is als beweging. Indien opeenvolgende individuele beelden motion blur bevatten, vereist het menselijk visueel systeem minder opeenvolgende beelden om gladde beweging te concluderen.

Ten tweede wordt de waargenomen kwaliteit ook anders beïnvloed door de



Figuur 2.34: Motion blur dankzij frameless rendering (bol beweegt naar boven)

frequentie-inhoud in verschillende beelden. Wanneer voornamelijk lage frequenties aanwezig zijn, zal minder ruis aanwezig zijn in het beeld. Dat komt doordat de bandbreedtevereisten voor een beeld met voornamelijk lage frequenties kleiner zijn dan voor een beeld met voornamelijk hoge frequenties. Bijgevolg moeten de monsters minder dicht bij elkaar genomen worden om aliasen te vermijden. Voor eenzelfde dichtheid aan monsters zal een beeld met voornamelijk hoge frequenties dus meer ruis vertonen. Deze bandbreedte wordt bepaald door de theorie van Shannon [42]. Die theorie beweert dat een signaal volledig hersteld kan worden van zijn monsters, indien de snelheid van de bemonstering minstens twee keer zo groot is dan de hoogste frequentie van het signaal. Er zullen dan geen aliasen aanwezig zijn. Elke bemonsterde frequentie is dan immers uniek. Een alias van een lage frequentie valt voor wanneer de bemonsterde waarden van een hoge frequentie gelijk zijn aan die van de lage frequentie. Aangezien tussenin deze waarden niet bemonsterd is, kan niet bepaald worden of het de lage frequentie of de hoge frequentie is.



Figuur 2.35: FFT op figuur met: (a) scherpe randen (b) wazige randen

Een voorbeeld van lage frequentie valt voor wanneer objecten wazige grenzen hebben. Hoge frequenties duiden scherpe veranderingen in intensiteit aan. De frequentie-inhoud wordt verder bepaald door aanwezige kleuren, patronen, het aantal objecten in de scene, hoe ver ze uit elkaar liggen, enzovoort.

Ten derde kan het menselijk visueel systeem slechts bepaalde frequenties waar-

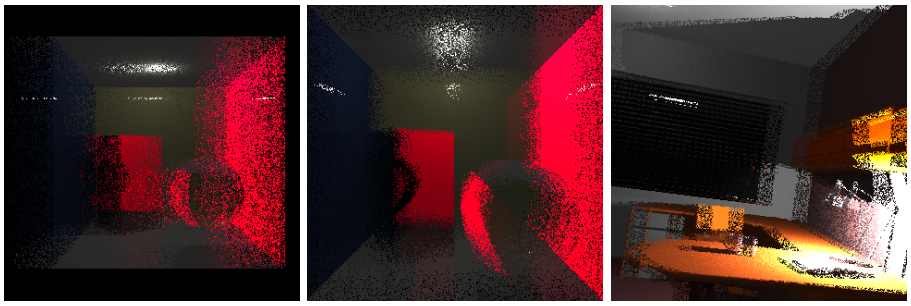
nemen [110, 71]. Vanaf een bepaalde hoge frequentie zullen spatiale frequenties niet meer waargenomen worden. Er is dus een grens op de spatiale resolutie die we kunnen waarnemen. Ook voor temporele frequenties geldt een dergelijke bovengrens, waarvan men gewoonlijk aanneemt dat ze rond 30 *à* 50 beelden per seconde ligt. Indien sneller intensiteitsverschillen voorvallen, wordt dit niet meer waargenomen. Hoe dichter deze grens benaderd wordt, hoe minder de ruis zichtbaar zal zijn. Frameless rendering heeft het meeste effect wanneer met dubbele buffering slechts een bemonsteringssnelheid behaald kan worden, waarop de temporele aliasing zichtbaar is. Volgens Ellen J. Scher Zagier geldt dat voor het bereik van vijf tot tien beelden per seconde. Wanneer met dubbele buffering geen temporele aliasing waargenomen wordt, gewoonlijk boven ongeveer vijftien beelden per seconde, zijn de waargenomen resulterende beelden gelijkwaardig aan die van frameless rendering. Dan doet frameless rendering enkel dienst als een techniek om berekeningen te vermijden.

Met behulp van supersampling in de ruimte wordt de spatiale resolutie verhoogd [35], wat resulteert in minder ruimtelijke aliasing. Door **supersampling in de tijd** wordt de temporele resolutie verhoogd en dat resulteert in gladdere bewegingen en dus minder temporele aliasing. Dergelijke supersampling wordt gewoonlijk toegepast door posities in een regelmatig rooster te nemen of random posities te bepalen.

Het toepassen van een regelmatig rooster resulteert in aliasing, aangezien er posities in de ruimte zijn die nooit bemonsterd zullen worden. Telkens worden dezelfde posities opnieuw bemonsterd, waardoor valse patronen kunnen ontstaan. Dat betekent dat frequenties, die hoger zijn dan de hoogst mogelijk te bemonsteren frequentie, bemonsterd zullen worden als een lagere frequentie. Het is daar dus een alias van. Temporele aliasing ontstaat wanneer de temporele bemonsteringssnelheid te laag ligt en wordt eveneens beïnvloed door de snelheid van bewegingen in de scene.

Indien random bemonstering toegepast wordt, zullen telkens andere posities bemonsterd worden op een niet regelmatige wijze. De posities tussenin de regelmatige posities kunnen dan ook bemonsterd worden. Het creëren van valse patronen wordt vermeden op deze manier, omdat er geen coherentie aanwezig is tussen de posities. Doch is er, door te weinig monsters te nemen, geen volledige kwaliteit beschikbaar. Er zal ruis zichtbaar zijn, maar dat is minder storend voor het menselijk visueel systeem dan aliasing. Deze ruis leidt tot een minder solide waarneming van de omgeving, aangezien bij bewegingen de grenzen en de inhoud van objecten gaten zullen bevatten en delen van objecten op voormalige plaatsen nog zichtbaar kunnen zijn.

De **uiteindelijke beeldkwaliteit** kan beoordeeld worden volgens het exacte aantal foute pixels in het gedeeltelijk vernieuwd beeld ten opzichte van een volledig berekend beeld op dat moment. Het kan ook beoordeeld worden volgens de eigenschappen van de menselijke visuele waarneming. De laatste manier is de belangrijkste, maar ook de moeilijkste. Die wordt beïnvloed door patronen, kleuren, beweging, diepte, geometrie van objecten, grootte van objecten, verspreiding in het zichtveld van de objecten, enzovoort. Volgens Ellen J. Scher Zagier worden de meeste hiervan op verschillende manieren bepaald door het menselijk visueel systeem. Herkenning van individuele objecten en hun geometrie is het moeilijkste. Daarom kan er best voor gezorgd worden dat objecten



Figuur 2.36: Voorbeelden van de resulterende minder solide waarneming bij (a) zijwaartse translatie, (b) voorwaartse translatie, en (c) rotatie in "The office"

reeds zo samenhangend mogelijk blijven.

2.3.4 Uitbreidingen

Frameless rendering kan op verschillende manieren verbeterd worden. Bishop et al. [14] en Ellen J. Scher Zagier [117, 118] bespreken verschillende mogelijkheden. Het visualiseren van de pixels is een heel dure berekening. Daarom is het wenselijk het aantal te herberekenen pixels zoveel mogelijk te minimaliseren, met behoud van een optimale kwaliteit van de waargenomen interactieve beelden. Nieuwe beelden moeten dus zo snel mogelijk convergeren richting het correcte beeld op die moment. Dat kan bereikt worden door nieuwe monsterkandidaten zo te selecteren dat toekomstige beelden sneller benaderd worden III. Dergelijke benaderingen kunnen via beeldreconstructie III van een te dun aantal monsters nog correcter gemaakt worden.

I Bemonstering

Om het beperkt aantal pixels, dat herbemonsterd kan worden, zo efficiënt mogelijk te gebruiken, kunnen betere methoden aangewend worden om monsters te kiezen in het beeldvlak. Zo wordt dan importance sampling gedaan. Gebieden, die het meest nood hebben aan vernieuwingen, worden frequenter behandeld. Door rekening te houden met de perceptuele eigenschappen van de mens kan bepaald worden waar het visualisatie algoritme nieuwe monsters moet nemen, zoals bijvoorbeeld in gebieden met veel kleurverschillen of gebieden met grenzen van objecten.

Spatiale grenzen zorgen ervoor dat objecten onderscheiden kunnen worden van de rest van de scene. Expliciete grenzen vallen sterk op door de variatie in kleur aan weerszijden van de grens. Dankzij het Mach band effect [61] worden ze versterkt waargenomen. Door meer monsters te nemen op deze grenzen, blijven ze beter aaneengesloten waardoor de integriteit van het beeld sterker behouden wordt. Vaak hebben objecten gelijkaardige kleuren binnen hun grenzen. Het menselijk visueel systeem vult daar kleine fouten zelf een beetje in. Impliciete grenzen zijn grenzen die er eigenlijk niet zijn, maar door de mens afgeleid worden. Deze bevatten geen sterke kleurvariaties aan weerszijden en zijn moeilijker detecteerbaar. Ze worden bijgevolg niet geholpen op deze manier.

In de tijd worden grenzen gecreëerd door bewegingen. Vloeiende beweging leidt tot betere interactiviteit, zodat ook bij bewegende objecten het beste meer monsters geconcentreerd kunnen worden. Indien slechts *één* enkel object beweegt, is de grootste verandering in de scene natuurlijk aanwezig op het gebied waar dit object terechtkomt in het huidige beeld en vanwaar het kwam.

Zoals eerder vermeld, is het ook nodig de rest van het beeld te bemonsteren op een goed verspreide wijze om nieuwe veranderingen te ontdekken en zo toch een gelijktijdige beeldvernieuwing te ondervinden. Het is ook nuttig de bemonstering te concentreren op speculaire objecten, omdat deze afhankelijk zijn van hun positie ten opzichte van de camera en ze zorgen voor veranderingen binnen de grenzen van een object.

Verder kunnen kleurwaarden reeds berekend worden voor een toekomstige positie van een object of van de camera. Indien een bepaald object of de camera beweegt, kan voorspeld worden waar dat object of de camera zich in een volgend tijdstip zal bevinden. Zo kan bijvoorbeeld een occlusiefout, die gaat voorvallen, reeds op voorhand bemonsterd worden, zodat ze minder sterk aanwezig zal zijn.

Tenslotte kan rekening gehouden worden met de positie op het scherm waarop de gebruiker zich concentreert via eye tracking. De pixels die zich bevinden in de periferie van het kijkvolume vereisen een lagere vernieuwingssnelheid dan de pixels die zich nabij het centrum bevinden. Er kan ook rekening gehouden worden met de verwachte snelheid van een object op het netvlies van het oog. Het oog neemt de objecten met de laagste retinale snelheid het nauwkeurigste waar. Die kunnen best sterker bemonsterd worden.

II Herprojectie

Het is mogelijk om de oude monsters niet enkel te behouden, maar ook om hun positie op het beeldvlak aan te passen aan de recentste camera- en objectparameters. De objectparameters worden gebruikt om de bijgehouden 3D-positie van de monsters aan te passen. Daarna worden de cameraparameters gebruikt voor herprojectie van de monsters. Op die manier zullen ze op de correcte positie staan, maar er kunnen dus nog steeds fouten ontstaan doordat het object, waarvan de kleur getoond wordt, geheel of gedeeltelijk speculair is of omdat een ander object het eigenlijk bedekt op die moment. Aangezien op die plaats geen monster genomen is, zal de zichtbaarheid niet correct bepaald zijn. Dat is de taak van de primaire stralen die vanaf de camera geschoten worden.

III Reconstructie

De resulterende beeldkwaliteit, na het nemen van de monsters en de toepassing van de herprojectie, kan verhoogd worden met behulp van **reconstructiemethoden**. Het beeld kan gemanipuleerd worden om zaken, die een negatieve invloed hebben, te vervangen door zaken die minder storend zijn. Filters kunnen toegepast worden in de beeldruimte om onder andere gaten binnen objecten op te vullen, gaten in grenzen aan te vullen, enzovoort. Met behulp van een filter kunnen ook storende frequenties voor het menselijk visueel systeem vervangen worden door andere minder storende frequenties.

De kleurwaarden van oudere monsters kunnen vervangen worden door bijvoorbeeld kleuren van recentere buurpixels. Indien geen herprojectie toegepast wordt zal de motion blur verdwijnen en dat leidt volgens Ellen J. Scher Zagier

tot minder gladde bewegingen. Door de oude kleur te mengen met kleuren van recentere buren kunnen accuratere beelden bekomen worden.

IV Anti-aliasing

Tenslotte kan het idee van frameless rendering tevens toegepast worden op anti-aliasing [119]. Anti-aliasing kan dan gedaan worden op momenten dat er voldoende tijd voor beschikbaar is.

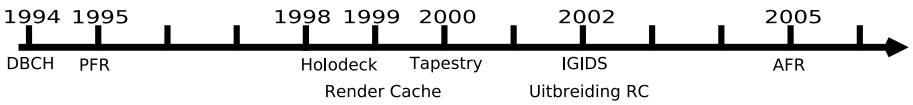
V Belangrijkste methoden

De implementatie van de belangrijkste methoden wordt in het volgende hoofdstuk toegelicht. Zij leggen de belangrijkste principes van frameless rendering voor. De originele paper [14] brengt het fundamentele idee met zich mee, welke verklaart dat niet telkens alle pixels in elk nieuw beeld berekend moet worden, indien dat niet haalbaar is. Deze werkwijze is verder ontsproten uit de gedachte, die besproken wordt door Bergman et al. [12]. Zij kwamen op het idee dat, wanneer het beeld berekend is en statisch blijft, er geen werk meer geleverd wordt door de CVE. Die tijd kan even goed besteed worden aan het verder verfijnen van de kwaliteit van dat statische beeld. Zodoende tonen zij zo snel mogelijk een ruw beeld aan de gebruiker, dat indien het statisch blijft, in opeenvolgende en op elkaar steunende stappen verder verfijnd wordt. Dat gebeurt op een zo vloeiend mogelijke manier, met als gevolg dat de gebruiker zo weinig mogelijk wordt afgeleid. De volgende paper over de Render cache [106] beweert dat het veel minder tijd kost om reeds gevisualiseerde punten te herprojecteren bij camerabewegingen of te verplaatsen bij objectbewegingen, dan de pixels waarop ze terechtkomen opnieuw te visualiseren. Verder worden verschillende concepten in *één* kader samengebracht. Adaptive frameless rendering [32] beschrijft een systeem, waarbij kleurvariaties geviseerd worden, omdat die plaatsen de grootste visuele veranderingen opleveren op het beeld voor de gebruiker. De grootste variaties worden dan in tijd en ruimte het snelst opnieuw berekend.

Er zijn ook een deel andere frameless rendering methoden bedacht. Deze worden besproken in de volgende sectie.

2.3.5 Gerelateerd werk

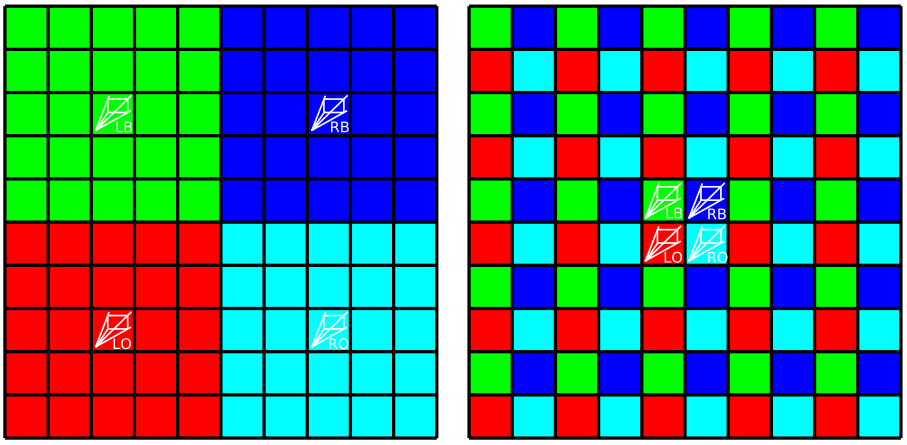
Deze sectie bespreekt verschillende technieken, die gebruikmaken van het concept frameless rendering. Hun voor- en nadelen worden vergeleken. De tijdslijn geeft een korte historische schets van het ontstaan van de belangrijkste technieken.



Figuur 2.37: Tijdslijn, DBCH: Double Buffering Considered Harmful, PFR: Practically Frameless Rendering, IGIDS: Interactive Global Illumination in Dynamic Scenes, AFR: Adaptive Frameless Rendering

I Practically Frameless Rendering

Hoewel frameless rendering geschikter is voor gebruik met pixel-gebaseerde visualisatie algoritmen zoals raytracing, wordt het toch toegepast op rasterisatie [115]. Ook hier resulteert een stijging van het aantal beelden per seconde door wederom minder pixels te bemonsteren per beeld. Tevens is er een verbetering van de interactiviteit bij applicaties die geen hoge snelheid kunnen halen. De invoer zal namelijk vaker bemonsterd worden, maar in tegenstelling tot de oorspronkelijke frameless rendering zullen de camera- en objectparameters slechts per beeld vernieuwd worden, dat dan bestaat uit een gedeelte nieuwe pixels en de overige oudere pixels. Het levert ook gratis motion blur, wat altijd voordelig is in interactieve applicaties. Deze motion blur is spijtig genoeg slechts een benadering. Andere motion blur technieken bieden een hogere kwaliteit ten koste van extra berekeningstijd. Tegelijkertijd brengt het ook het nadeel met zich mee dat er nergens op het beeld iets in focus is, aangezien de motion blur aanwezig is op het hele beeld. Bijgevolg kan een mens zich nergens echt op concentreren, aangezien het menselijk visueel systeem dergelijke visuele aanknopingspunten nodig heeft.



Figuur 2.38: (a) De framebuffer bestaat uit vier kwadranten. Elke keer wordt *één* kwadrant ingevuld met de corresponderende camera. (b) Telkens *één* kwadrant gegenereerd is, worden alle kwadranten ingevoegd in het finale beeld op de aangegeven positie. De aangegeven camera's staan hier op de posities vanwaar ze de scene visualiseren, namelijk het midden van de pixel.

Practically frameless rendering berekent telkens een kwart van de pixels van het uiteindelijke beeld en toont dit dan in combinatie met de overige drie kwart eerder berekende pixels. Aangezien rasterisatie geen pixel-gebaseerd visualisatie algoritme is, moet het op een andere manier gebeuren dan bij bijvoorbeeld raytracing. De techniek bereikt het door een beeld in de framebuffer te beheren die de resolutie van het finale beeld heeft. Het splitst dat beeld op in vier delen, zodat vier kwadranten resulteren. Telkens berekent het een een kwart van de resolutie van dat uiteindelijke beeld en slaat dat op in *één* van de kwadranten van de framebuffer. Bij elk kwadrant heeft de camera een lichtjes verschillende positie. Daardoor berekent de techniek niet telkens exact hetzelfde en wordt een

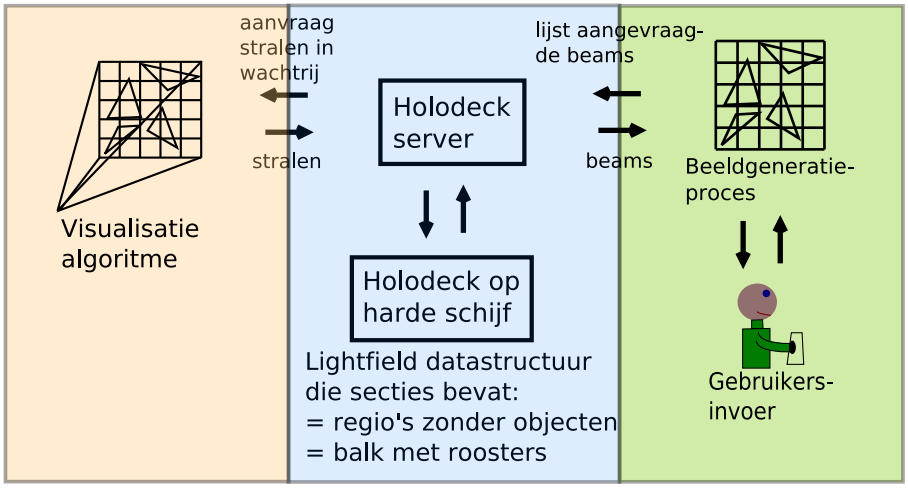
hogere resolutie verkregen, namelijk die van het finale beeld. Telkens combineert de techniek het laatst berekende kwart pixels met de drie kwart pixels die er net voor berekend werden. Het deelt het finale beeld op in vierkanten van telkens vier pixels groot. Dan bouwt het dat beeld op door elk kwadrant kleuren te laten leveren aan de overeenkomende pixels. Het kwadrant linksboven levert de pixel linksboven, het kwadrant linksonder de pixel links onder, enzovoort. Op deze manier zijn er vier beelden nodig om convergentie met het exacte beeld te verkrijgen, wanneer de scene niet meer verandert.

Ze stellen ook nog enkele variaties voor, die de bemonstering minder regelmatig maken. Daardoor stijgt onder andere de kwaliteit van de motion blur. Maar deze variaties lijden allemaal aan het probleem dat er niet automatisch terug een volledige resolutie bekomen wordt, wanneer de scene terug statisch wordt.

Hieruit blijkt nogmaals dat frameless rendering beter geschikt is voor pixelgebaseerde visualisatie algoritmen. De implementatie voor rasterisatie brengt extra kosten en problemen met zich mee. De basis implementatie biedt geen stochastische bemonstering en geen focuspunten voor het oog. De variaties leveren uiteindelijk geen volledige resolutie en hebben daarenboven elk hun eigen nadelen. Bovendien bezit rasterisatie de reeds vermelde nadelen 2.2.4.

II Holodeck

Radiantie, berekend door een raytracer, blijft gelijk langs de hele lengte van de straal [56, 57]. Daarom stelt deze paper voor informatie van berekende stralen op te slagen in een lightfield datastructuur. Een lightfield is een 4D-functie die radiantie voorstelt als een functie van een onbelemmerde lichtstraal in 3D-ruimte. Tijdens opeenvolgende beelden wordt de geschikte informatie hergebruikt om beelden sneller te kunnen genereren dan wanneer de raytracer elke pixel moet herberekenen. De informatie wordt opgeslagen in een cache en op de harde schijf.

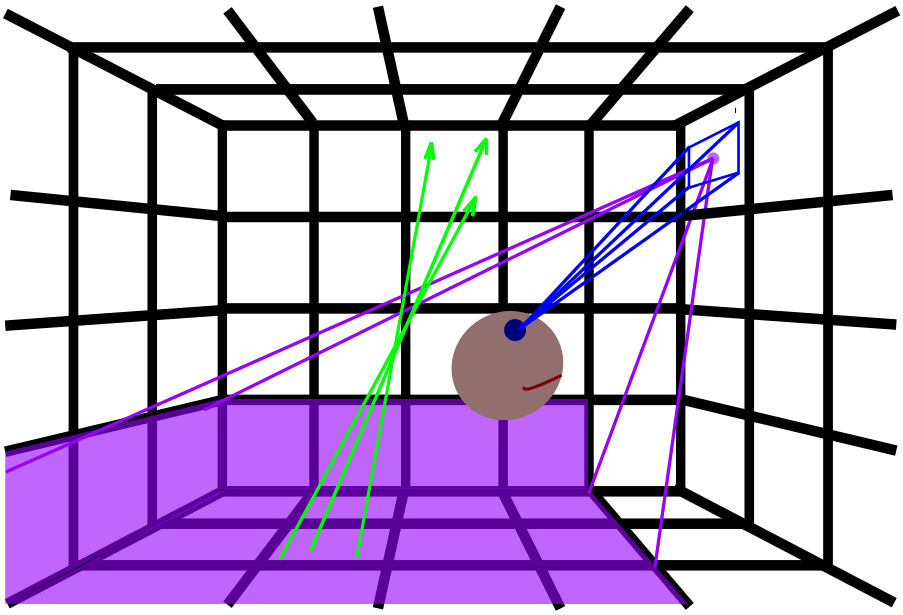


Figuur 2.39: Holodeck onderdelen

Het algoritme bevat, buiten de raytracer, een holodeck server en een beeldvormingsproces. De server bepaalt waar de raytracer monsters moet nemen en beheert de datastructuur, die uit meerdere holodecksecties bestaat. De secties stellen elk een regio in de scene voor, die vrij is van objecten zodat lichtstralen daar onbelemmerd zijn. Ze hebben de vorm van een balk, of zelfs een parallellepepidum, waarin elke wand een rooster is. Stralen, die door eenzelfde roostercel de holodecksectie binnenkomen en door eenzelfde roostercel de sectie verlaten, worden samen gebundeld in beams. Zo kunnen ze later efficiënt teruggehaald worden voor hergebruik in volgende beelden. De holodeck server beheert tevens de cache, die slechts een kleine grootte heeft. Het bevat enkel de recentst gebruikte stralen zodat ze nog sneller opgehaald kunnen worden. De datastructuur op de harde schijf kan daarentegen zeer groot worden, aangezien die alle stralen opslaat. De begeleidende informatie omvat onder andere de berekende kleur, de inkomende en uitgaande intersectiepunten van de straal met de roostercellen van de datastructuur om de exacte oorsprong en richting van de straal te kunnen berekenen en de afstand van de straal om die te kunnen herprojecteren. Nieuwe monsters worden bepaald aan de hand van de gekozen beams. Voor elk is een reeds berekend aantal monsters en een gewenst aantal monsters gespecifieerd. Hoe groter het aantal gewenste monsters is ten opzichte van het reeds berekende aantal voor een bepaalde beam, hoe eerder die beam aan bod zal komen. Wanneer een beam aan bod komt, worden de oorsprongen en richtingen van de stralen random gekozen, maar zo dat ze binnen de beam en een bepaalde afstand van het kijkpunt gelegen zijn. De te bemonsteren beams bevinden zich in een wachtrij met een bepaalde grootte. Indien die overschreden wordt, zal de wachtrij geledigd worden om interactiviteit te behouden.

Het beeldvormingsproces geeft aan de server door welke beams opgehaald moeten worden om het beeld op te bouwen. De server bezorgt dan eerst de benodigde stralen uit de cache en daarna de benodigde stralen van de harde schijf. Vervolgens worden de nieuwe door de raytracer berekende stralen gestuurd. Vooraleer het proces kan doorgeven welke stralen kunnen dienen, zal het berekenen welke stralen dichtbij de kijkpositie passeren. Die vindt het door te zoeken welke cellen zichtbaar zijn voor de camera en welke cellen aan de tegengestelde kant van het rooster mede bepalend zijn. Beams die aan beide kanten een van de gevonden cellen doorkruisen zijn de benodigde beams. Deze beams worden meestal gehaald uit *één* bepaalde holodecksectie. Zelden worden ze toch gehaald uit meerdere secties, wanneer bijvoorbeeld het kijkpunt tussenin secties ligt in plaats van in een sectie. Omdat stralen niet altijd exact doorheen het kijkpunt gaan, worden ze geherprojecteerd, zodat hun kleurwaarde op de juiste positie op het scherm belandt. Zonder herprojectie zal blurring ontstaan, maar occlusiefouten zullen blijven.

De reconstructie van het beeld gebeurt met een quadtree, een Voronoi diagramma of een mesh met Delaunay triangulatie zoals bij tapestry IV. Het maakt bij de laatste twee mogelijkheden gebruik van de grafische hardware. De holodeck deelt het beeld bij de quadtree op in vier gelijke delen. Daarna deelt het elk deel, waarvoor meer dan *één* monster beschikbaar is, recursief verder op in vier delen. Het tekent de kwadranten, waarvoor monsters voorzien zijn, met de kleur van het monster. Bij nieuwe monsters of bij het wegsmijten van monsters past het de quadtree aan. Indien bij toevoeging een monster op dezelfde plaats terechtkomt als een ander monster, neemt het degene die het dichtste bij het



Figuur 2.40: Een voorbeeld van een holodecksectie. De camera bevindt zich binnen de sectie en kijkt naar rechtsboven. De intersectie van het kijkvolume met het rooster van het holodeck bepaalt de cellen, waarmee de geschikte beams gevonden kunnen worden. De drie groene stralen behoren tot eenzelfde beam. Deze beam kan niet dienen voor het huidig zicht. De beams, die wel kunnen dienen, gaan door de paarsgekleurde cellen richting de cel waarmee het kijkvolume intersecteert.

kijkpunt passeert. Monsters, die een te grote hoek maken met het kijkpunt, zal de holodeck niet gebruiken. Occlusiefouten zijn dus mogelijk. Ook moet de holodeck per beeld alle monsters herprojecteren. In het geval van een Voronoi diagramma, tekent de holodeck monsters als Voronoi polygonen. Het tekent elk monster volgens zijn diepte als een kegel, die van bovenaf gezien wordt met de top op de positie van het monster. Deze kegel wordt benaderd met behulp van driehoeken. Door rekening te houden met de diepte kan de holodeck veel occlusiefouten vermijden. Het Voronoi diagramma levert minder hoekige resultaten dan de quadtree en kan progressief verbeterd worden. Nadelen zijn plotse illuminatieveranderingen, flat shading en de nood aan extra geometrie. De 3D-mesh met Delaunay triangulatie werkt gedeeltelijk zoals in tapestry, wat Maryann Simmons [87] uitgebreid uitlegt. Het bezit het voordeel dat het voor volgende beelden hergebruikt kan worden in het geval van kleine camerabewegingen. Nieuwe monsters kunnen dan incrementeel toegevoegd worden. Bij grote camerabewegingen zullen artefacten optreden, indien de mesh niet gereconstrueerd wordt. Die reconstructie kan de visualisatie dan weer sterk vertragen bij grote camerabewegingen. Verder levert het Gouraud interpolatie en een progressieve verfijning. Het levert daardoor gladdere en consistentere beelden dan de Voronoi representatie.

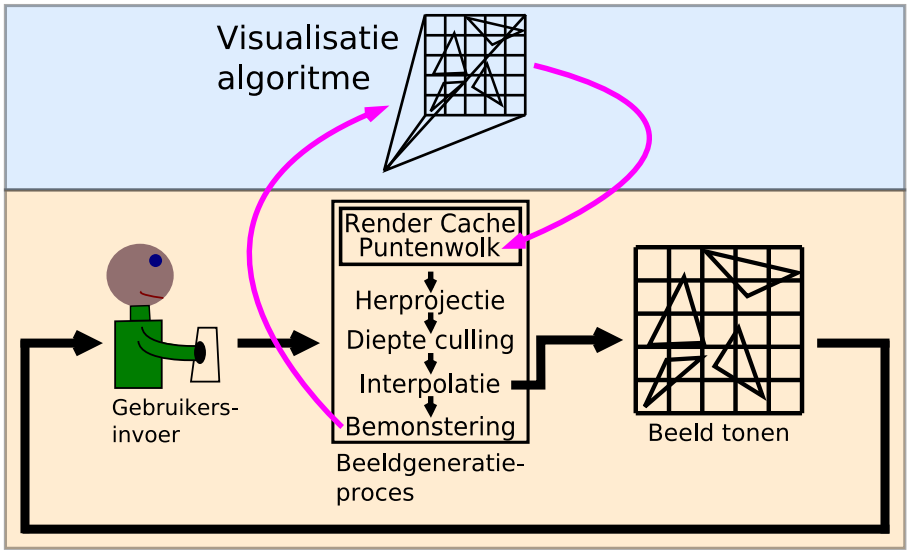
Bij kleine camerabewegingen zal veel informatie van de cache nuttig zijn en

worden nieuwe monsters progressief bijgevoegd, maar bij grote camerabewegingen kunnen de monsters in de cache waarschijnlijk weinig bijdragen en daarom wordt het grootste deel ervan genegeerd. Er zullen dan voornamelijk nieuwe monsters beschikbaar worden en gebruikt worden. De holodecksectieroosters zullen zichtbaar zijn, zodat de gebruiker zich kan oriënteren. Wanneer de camera niet beweegt worden monsters uit beide de cache en de datastructuur genomen en worden tevens nieuwe monsters gebruikt. Zo zal de resolutie en de kwaliteit van de quadtree progressief stijgen en bijgevolg ook die van het beeld.

De raytracer kan gestopt en herstart worden om nieuwe stralen te genereren na veranderingen aan de scene of de datastructuur kan leeggemaakt worden bij grote veranderingen. Foute monsters, die nog berekend moeten worden, zullen niet meer berekend worden en foute monsters, die opgeslagen waren, worden buiten beschouwing gelaten.

III Render Cache

Dit systeem [106] combineert verschillende eenvoudige technieken en heuristieken. Een pixel-gebaseerd visualisatie algoritme berekent de monsters. Die worden daarna net zoals bij het holodeck-systeem opgeslagen in een cache. Een beeldvormingsproces zorgt ervoor dat de monsters uit de cache op de juiste positie op het beeld belanden via herprojectie. Aangezien herprojectie occlusiefouten kan opleveren en de cache niet altijd de juiste monsters bevat om het hele beeld op te vullen, past de render cache filters toe om fouten zoveel mogelijk weg te werken. Adaptieve bemonstering zorgt ervoor dat de dure berekening van monsters zo efficiënt mogelijk gericht wordt op de belangrijke plaatsen. Dankzij het idee van frameless rendering buit de render cache temporele beeldcoherentie uit. De opeenvolging van beelden wordt minder sterk gehinderd, doordat het visualisatie algoritme uit de beeldvernieuwingslus gehaald is.



Figuur 2.41: Volgorde van de operaties, waarbij het visualisatie algoritme in een aparte thread uitgevoerd wordt.

Het visualisatie algoritme berekent de monsters asynchroon van het beeldvormingsproces. Dat proces staat in voor een aantal taken. Het integreert nieuwe monsters in de cache en verwijdert oude monsters uit de cache. Daarnaast richt het nieuwe berekeningen op de belangrijke gebieden en genereert het met regelmatige tussenpozen een volledig beeld dat telkens getoond zal worden op het scherm. Zo is er een onafhankelijke vernieuwing van shading waarden en het beeld. Er moet niet gewacht worden totdat het visualisatie algoritme gedaan heeft met het visualiseren van een bepaald deel monsters. Op die manier zal een vloeiende beeldvernieuwing mogelijk blijven. Wanneer het visualisatie algoritme te traag is, zal tijdens veranderingen enkel de kwaliteit van het beeld achteruitgaan in de vorm van een groter aantal zichtbare artefacten. Op deze manier wordt een progressieve verfijning gekregen. Het desbetreffende implementatiegedeelte 3.3 bespreekt dit verder.

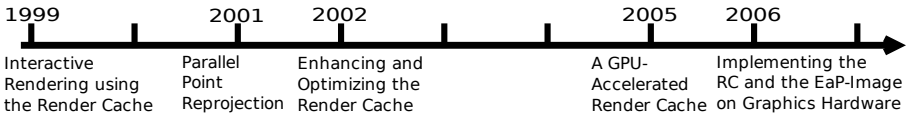
Bij het genereren van een nieuw beeld zullen de monsters in de cache eerst geherprojecteerd worden met behulp van de huidige cameraparameters en een Z-buffer. Monsters komen daardoor steeds op de juiste positie op het beeld terecht. Maar aangezien de cache niet altijd voor elke pixel een monster bevat, zullen gaten overblijven in het beeld. Ook is er dankzij herprojectie kans op occlusiefouten. Dit systeem maakt daarom gebruik van beeldreconstructie en adaptieve bemonstering.

De beeldreconstructie maakt gebruik van een diepte culling stap en een interpolatiefilter. Het zijn beide heuristieken. De diepte culling lost reeds een deel van de occlusies op, terwijl de interpolatiefilter kleine gaten opvult.

De bemonstering wordt geleid door een prioriteitsbeeld, dat telkens gecreëerd wordt bij de opbouw van een beeld. Dat prioriteitsbeeld heeft dezelfde grootte als het te tonen beeld, zodat per pixel een prioriteit moet toegekend worden. De prioriteit is afhankelijk van een aantal heuristieken. Het wordt onder andere bepaald door de leeftijd van het monster dat op die pixel geprojecteerd wordt, aangezien oudere monsters een grotere kans hebben dat ze niet meer nuttig kunnen bijdragen. De leeftijd van elke pixel verhoogt met een constante snelheid telkens een nieuw beeld af is. Een kleurveranderingsheuristiek verhoogt voortijdig de prioriteiten van pixels die buren zijn van een pixel die van kleur verandert. Wanneer een pixel van kleur verandert, is er namelijk veel kans dat zijn buren ook van kleur veranderen. Verder worden ook de leeftijden voortijdig verhoogd van monsters, die buiten het beeldvlak geprojecteerd worden. Tenslotte worden de prioriteiten van pixels verhoogd, die gelegen zijn in regio's waar weinig monsters op terechtkomen. Deze hebben een grote kans dat ze gaten vertonen. De prioriteit kan makkelijk aangepast worden met andere heuristieken. Error diffusion dither wordt toegepast op dat prioriteitsbeeld om aan de hand van de prioriteiten een pixel aan te nemen of af te wijzen als kandidaat voor herbemonstering. Deze dither zorgt ervoor dat delen van het beeld goed gerepresenteerd worden waar vernieuwing het hardste nodig is en zorgt tevens voor een goede spatiale verdeling, zodat in andere regio's nieuw ontstaande veranderingen vroeg gedetecteerd kunnen worden en zodat voorkomen wordt dat veel nieuwe monsters vlak bijeen gekozen worden. Het levert tenslotte ongeveer het vooraf bepaald aantal kandidaten op.

De cache heeft in tegenstelling tot de holodeck een beperkte grootte, zodat alle monsters na een tijdje overschreven worden. Het is iets groter dan de grootte van het doelbeeld. Op die manier wordt verzekerd dat monsters relatief recent en bijgevolg nuttig blijven, aangezien nieuwe monsters telkens een oud monster moeten overschrijven. Nieuwe monsters worden ingevoegd in deze cache op twee manieren. Indien het nieuwe monster dezelfde 3D-positie bevat dan het oude monster, wordt het rechtstreeks overschreven nadat hun kleuren vergeleken zijn. Als de kleuren ongelijk zijn, worden de leeftijden van de buren verhoogd. Wanneer het nieuwe monster een andere 3D-positie bezit zal op een round-robin manier een oud monster overschreven worden. Telkens de hele cache doorzoeken is te duur.

Dynamische scenes worden ondersteund door de kleurveranderingsheuristiek. Ze worden tevens gedeeltelijk ondersteund doordat "rigid body"-transformaties direct toegepast kunnen worden op de overeenkomende monsters in de render cache. Op die manier komen ook deze monsters direct op de correcte positie terecht. Als een object bijvoorbeeld transleert, is de beweging van elk monster van dat object gekend. Ze worden allemaal met dezelfde hoeveelheid getransleerd. Dat geldt ook voor rotaties. Spijtig genoeg werkt dit niet voor andere bewegingen, aangezien er dan meer informatie van de scene en de bewegingen nodig is. Als bijvoorbeeld skinning toegepast wordt, is er niet gekend welk punt het monster juist voorsteld en waar het terecht zal komen.



Figuur 2.42: Tijdslijn van render cache varianten

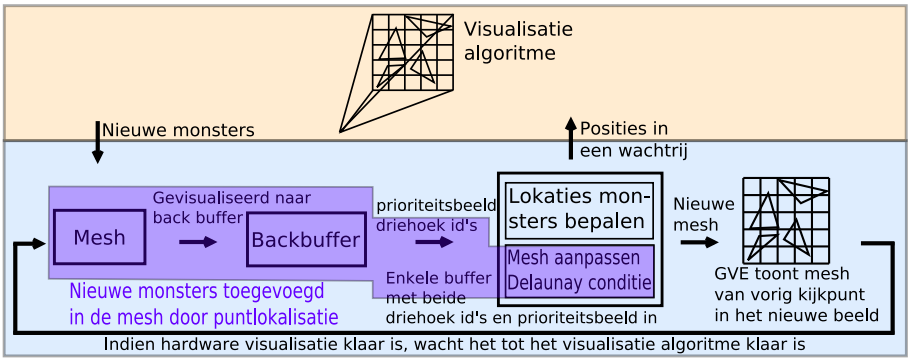
Dit basis render cache algoritme is nadien nog meerdere malen aangepast. Walter et al. hebben zo enkele verbeteringen toegevoegd in een volgende paper [105]. Zo werd een voorspellingsstap toegevoegd in het bemonsteringsproces om gaten op voorhand te ontdekken tijdens bewegingen en dan reeds te bemonsteren. In het reconstructieproces is een grotere filter toegevoegd om dunne gebieden beter op te vullen. Verder zijn de berekeningen beter geordend om geheugencoherentie uit te buiten en werd gebruik gemaakt van SSE.

Reinhard et al. [75] hebben een gedistribueerde versie uitgewerkt om het dure beeldvormingsproces te parallelliseren. Daarbij wordt het beeld opgedeeld in tegels, die elk hun eigen puntenwolk en bijhorende datastructuren bezitten. Deze tegels worden random toegewezen worden aan een van de beeldvormingsprocessen.

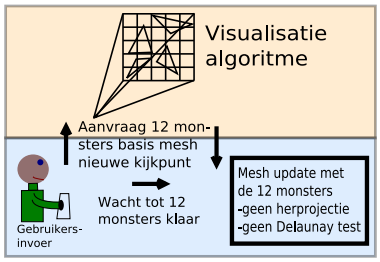
Zhu et al. [121] hebben verder een versie uitgewerkt die gebruik maakt van de grafische kaart. Tenslotte is er een combinatie uitgewerkt met het Edge-and-Point Image [95]. De laatste techniek zorgt voor een beter behoud van randen aangezien de render cache soms wazige randen oplevert tijdens veranderingen in het beeld en zorgt tevens voor een implementatie op grafische hardware. Deze laatste implementatie maakt gebruik van recentere mogelijkheden van de GVE dan de versie van Zhu et al.

IV Tapestry

Deze methode [88] bestaat net als het render cache systeem III uit een cache met een vaste grootte voor hergebruik van reeds gevisualiseerde monsters, een reconstructor om een redelijk beeld te verkrijgen van een klein aantal monsters en een prioriteitsbeeld om de bemonstering adaptief te gidsen zodat sneller convergentie optreedt. Het gebruikt tevens herprojectie om monsters aan het huidig kijkpunt aan te passen.



Figuur 2.43: Volgorde operaties wanneer er geen bewegingen gebeuren.

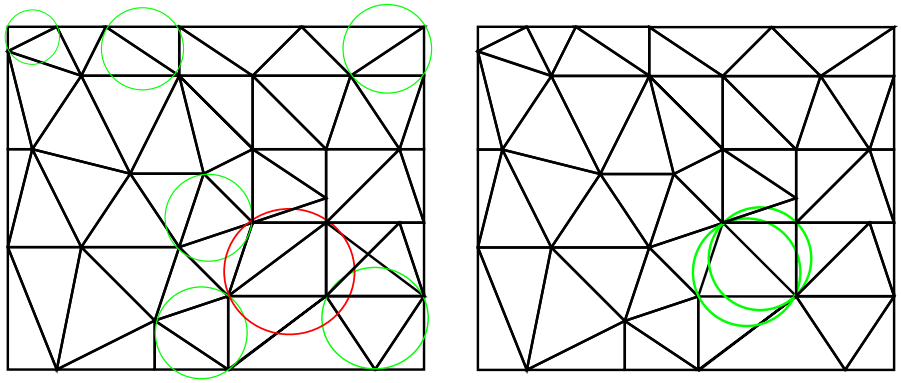


Figuur 2.44: Wanneer bewegingen gebeuren worden eerst deze operaties uitgevoerd, gevolgd door die van het statische geval.

Voor elk nieuw beeld zal tapestry een aantal operaties doen. Het bepaalt waar monsters genomen moeten worden met behulp van een **prioriteitsbeeld**, dat de reconstructor zal genereren. Prioriteiten steunen voornamelijk op kleurverschillen tussen vertices, maar worden ook lichtjes verhoogd bij diepteverschillen tussen vertices. Monsters bevatten bovendien leeftijden, die bij bewegingen verhoogd worden en ook bijdragen aan de prioriteit. De prioriteiten worden toegekend aan driehoeken en hun geprojecteerde grootte zal impliciet mee de kans op selectie bepalen. Verder wordt ervoor gezorgd dat de rest van het beeld uniform bemonsterd wordt, door een quasi-random Sobol sequentie posities te laten genereren. Dat zorgt ervoor dat niet totaal random posities genomen worden, maar dat er toch een mooie spreiding over het hele beeld bereikt wordt.

Daarna worden de monsters gegeneerd door een raytracer. Die worden dan opgeslagen in een cache, welke daarnaast dienst doet als een geometrische re-

presentatie voor weergave op het beeld. Tapestry gebruikt namelijk een dynamische mesh van driehoeken, die een uitbreiding is van de mesh die de holodeck gebruikt. Monsters worden geprojecteerd op een bol rondom het kijkpunt en stellen de vertices voor van de mesh. Aangezien de wereldcoördinaten van de monsters ook opgeslagen worden, is het een 2.5D-mesh. De diepte van de monsters ten opzichte van het kijkpunt kan zo later afgeleid worden. Er wordt een minimum afstand tussen monsters opgelegd en de driehoeken van de mesh moeten voldoen aan de Delaunay conditie. Dat betekent dat de vertices van de driehoek het dichtste bij elk intern punt van die driehoek moeten liggen. Er mogen dus geen vertices van een aangrenzende driehoek dichterbij liggen. Dat kan getest worden door een cirkel te trekken rond de vertices, waarin dan geen vertices mogen liggen van andere driehoeken. Aangezien de diepte ook gebruikt wordt, gebeurt dit met een kegel in plaats van een cirkel, waarvan de top start op de kijkpositie. De conditie garandeert een betere beeldkwaliteit, aangezien de vorm van de driehoeken vrij gelijkaardig zal blijven. Er ontstaan dan bijvoorbeeld geen uitgerekte smalle driehoeken, die resulteren in artefacten en berekeningsfouten. Wanneer geen camerabewegingen voorvallen, zullen nieuwe monsters oude monsters vervangen zodat de driehoeken niet moeten aangepast worden. De mesh zal dynamisch evolueren tijdens camerabewegingen, om zich aan te passen aan de nieuwe kijkpunten.



Figuur 2.45: (a) De driehoek met de rode cirkel voldoet niet aan de Delaunay conditie. De andere driehoeken wel, waarbij enkelen aangegeven zijn met groene cirkels. (b) Voorbeeld van een geldige Delaunay triangulatie, waarbij de ongeldige driehoek vervangen is. Deze zal op het gedeelte van de bol aanwezig zijn waardoor het kijkvolume van de camera gaat. De vertices van de driehoeken bevatten de door het visualisatie algoritme berekende kleuren en de pixels binnen de driehoeken zullen een geïnterpoleerde kleur krijgen van de GVE.

Daarom zijn bepaalde operaties noodzakelijk op de mesh. De locatie van nieuwe monsters moet bepaald worden en bij het invoegen van deze monsters worden nieuwe driehoeken aangemaakt. Dankzij de dieptewaarden kan ontdekt worden wanneer driehoeken niet meer richting de camera wijzen. Hun vertices moeten verwijderd worden. Tevens worden telkens een deel verouderde monsters verwijderd en ook op plaatsen waar sterke kleur- of diepteverschillen voorvallen tussen oudere monsters en een nieuw monster worden de oudere monsters verwijderd. Bij verwijdering van monsters worden hun ook driehoeken weg-

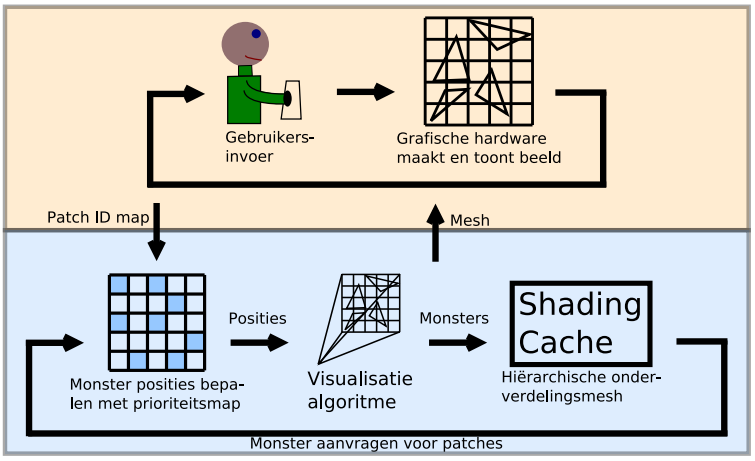
gesmeten, waarna gaten ontstaan in de mesh die dan opnieuw getrianguleerd moeten worden. In beide gevallen zal achteraf de Delaunay conditie terug getest worden. Indien een dergelijke test faalt, wordt de mesh aangepast tot ze slaagt.

Vervolgens wordt de cache gebruikt voor reconstructie en wordt er een beeld getoond. Daarvoor worden de driehoeken door de rasterisatie hardware getoond, waarbij Gouraud shading zorgt voor de opvulling van de driehoeken.

Als de camera niet beweegt wordt progressief een beter beeld verkregen doordat de mesh nieuwe monsters blijft opnemen tot een niveau dat juist groter is dan het aantal pixels. Op dat moment is de hoogste kwaliteit bereikt. Kleine camera bewegingen leiden tot kleine veranderingen in de mesh, terwijl grote camera bewegingen grote veranderingen teweegbrengen, zodat meer tijd nodig zal zijn om de mesh aan te passen aan het huidige kijkpunt. Hierbij zullen monsters, die niet meer kunnen bijdragen aan het huidige kijkpunt, verdwijnen uit de mesh.

V Interactive Global Illumination In Dynamic Scenes

Ook dit systeem [92] bestaat uit gelijkaardige onderdelen als het render cache systeem III. Er is een visualisatie algoritme, een beeldvormingsproces, een cache en een prioriteitsbeeld.

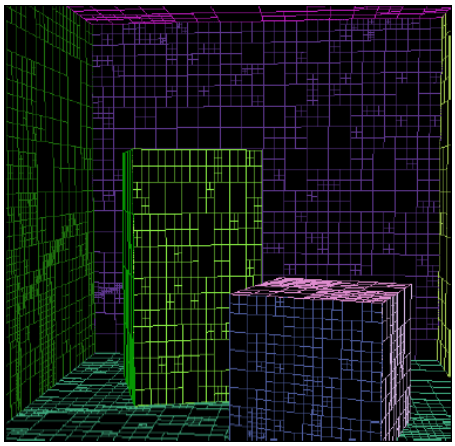


Figuur 2.46: Operatie volgorde

De belangrijkste vernieuwingen in dit systeem zijn de manier waarop de monsters opgeslagen worden in de cache en hoe die cache als een efficiënte representatie behandeld kan worden door de grafische hardware. De **shading cache** bevindt zich in wereldruimte en monsters worden lui toegevoegd. Het bestaat uit een hierarchische onderverdelingsmesh, die vastgehecht is aan elke primitieve in de scene. De recentst berekende monsters worden opgeslagen als vertices op de patches in de mesh. De hiërarchie wordt geïnitialiseerd als een enkele onopgedeelde patch per geometrisch object in de scene en zal op bepaalde regio's dieper verfijnd worden wanneer meer monsters beschikbaar worden op die regio. Als een patch aangewezen wordt door het bemonsteringsproces voor vernieuwing, worden monsters berekend voor zijn vertices. Indien

meer precisie nodig is, wordt de patch opgedeeld in vier deelpatches en worden monsters berekend voor hun vertices. Bewegingen van camera en objecten gebeuren onafhankelijk van de bemonstering van shading waarden. Er wordt verder een verschil gemaakt tussen diffuse en niet-diffuse oppervlakken. Voor niet-diffuse oppervlakken wordt de uitgaande radiantie opgeslagen voor het huidige kijkpunt, terwijl voor diffuse oppervlakken de inkomende radiantie wordt opgeslagen. Texture mapping kan dan toegepast worden.

De shading cache heeft een bepaalde grootte en wanneer die bereikt is, zullen oude patches weggesmeten worden. Leeftijden stijgen wanneer patches niet in beeld komen of minder dan een pixel innemen op het beeldvlak.



Figuur 2.47: Onderverdelingsmesh [92]

Het **beeldvormingsproces** en de daarbij behorende bemonstering van gebruikersinteracties wordt asynchroon uitgevoerd van het vernieuwen van de shading cache en het handhaven van het prioriteitsbeeld. Geometrie en texturen kunnen zo altijd accuraat gereproduceerd worden. Ze zullen zelfs na bewegingen op de juiste positie terechtkomen, waarbij randen en oppervlakken geen gaten zullen bevatten. De resolutie van de shading van de objecten kan daarentegen niet volledig gelijktijdig vernieuwd worden dankzij de onafhankelijke vernieuwing van de kleurwaarden. Zo zal de frequentie van beelden per tijdseenheid niet afhangen van de snelheid waarmee de resolutie verfijnd kan worden of de grootte van de resolutie. De kwaliteit van de beelden hangt daar wel van af. De grafische hardware kan vlakke primitieven direct weergeven, maar de weergave van gebogen oppervlakken leidt tot fouten. Daarom worden ze getesseleerd. Die tesselatie zal dan getoond worden in plaats van de shading cache, waarbij de kleur van de vertices daar bepaald wordt door interpolatie. Wanneer de scene veel verandert, wordt de resolutie van de shading cache op de bewegende objecten kleiner gemaakt, zodat herberekeningen binnen de beschikbare tijd kunnen gebeuren. Bij statische beelden zal de shading cache verfijnen totdat er voor elke pixel een patch is. De hoogste kwaliteit is dan bereikt. Deze reconstructie van beelden gebeurt met behulp van de grafische hardware. Die zorgt voor Gouraud interpolatie tussen de monsters in de shading cache, voor het oplossen van de zichtbaarheid en voor texture mapping. Belangrijkere gebieden worden sterker bemonsterd door het bemonsteringsproces en zullen dus dieper verfijnd worden, terwijl in gebieden met weinig monsters een sterkere interpolatie zal gebeuren.

Het beeldvormingsproces geeft een patch identificeermap terug aan het **bemonsteringsproces**, wanneer die dat nodig heeft. Het zal dan aan de hand van de momenteel zichtbare patches een prioriteitsmap creëren in beeldruimte. De prioriteiten worden toegekend, gebaseerd op camera-, object- of lichtbronbewegingen en de waarschijnlijkheid dat de huidig getoonde kleuren fouten vertonen.

Die fout wordt geschat in de objectruimte van de cache door berekeningen te doen op de kleuren van de vertices van een patch. Niet-diffuse objecten veranderen van kleur bij camerabewegingen en dragen daarom ook bij aan de prioriteit. Hoe meer kans er is dat ze van kleur veranderen, hoe sterker de leeftijd van hun patches (tijd van aanmaak tot de huidige moment) gescaleerd zal worden. Bewegende objecten worden gedetecteerd door de random bemonstering, doordat wanneer kleurwaarden van een patch veranderen, de buurpatches een stijgende waarschijnlijkheid op fouten zullen krijgen. Hun prioriteiten zullen dan stijgen en hun leeftijd wordt ook verhoogd. Beeldruimte monsters worden gericht naar plaatsen in de objectruimte van de cache waar meer verfijning nodig is. De patches met hoge prioriteiten moeten bemonsterd worden, maar tevens moet een goede spatiale beeldruimteverdeling bekomen worden van monsters. De eerste test bereikt een goede spatiale verdeling door telkens een random pixel en een random drempelwaarde te nemen. De prioriteit van die pixel moet boven de drempelwaarde liggen. De kans dat die pixel geselecteerd zal worden voor bemonstering hangt dus af van zijn prioriteit en het geprojecteerde gebied van de patch in pixels op het beeld. Kleine patches, die frequent aanwezig zijn bij grote kleurverschillen, hebben weinig kans op herbemonstering dankzij deze test. De tweede test (flood filling) zorgt ervoor dat gebieden met hoge frequenties benadrukt worden, door buurpatches van patches met een hoge prioriteit ook te selecteren indien ze een hogere prioriteit hebben. De posities van de aangewezen monsters komen overeen met patches, wat bepaald wordt via de identificeermap. De patches worden dan samen met het aantal te berekenen monsters voor die patch in een lijst opgeslagen en zullen vernieuwd worden. Het visualisatie algoritme is bijvoorbeeld een raytracer en zorgt ervoor dat de aangevraagde monsters berekend worden.

De ontwikkelaars van deze techniek beweren dat het cachen en interpoleren van berekende kleurwaarden in objectruimte en hun manier van bemonstering zorgen voor een beduidende verbetering in snelheid en kwaliteit ten opzichte van vorige technieken. Voornamelijk bij drastische veranderingen wordt dit duidelijk.

VI Adaptive Frameless Rendering

Adaptive frameless rendering keert terug naar een onregelmatige bemonstering van de tijd, zoals het oorspronkelijke frameless rendering [14]. Camera- en objectparameters worden nu ongeveer per nieuw gevisualiseerd monster vernieuwd, zodat sneller gereageerd kan worden op veranderingen. Deze techniek bevat twee grote asynchrone onderdelen, namelijk een bemonsteringsalgoritme en een reconstructor. Dankzij adaptieve bemonstering en adaptieve reconstructie kan het zich met zeer fijne granulariteit aanpassen aan kleurveranderingen in tijd en ruimte. Beide onderdelen hebben elk een eigen buffer om nieuwe monsters in op te slagen, waardoor de onderlinge communicatie zo klein mogelijk gehouden wordt. Verder bezitten ze elk ook hun eigen herprojectiemechanisme met eigen specifieke doeleinden. Het wordt dankzij deze techniek mogelijk een visualisatie algoritme uit te voeren op de CVE dat selectief een enkel monster herberekend en direct doorstuurt naar het beeld waarop dan een reconstructor zorgt voor kwaliteit. Maar beeldschermen verlangen momenteel nog steeds dat telkens een heel beeld doorgestuurd wordt.

De buffers van beide hoofdcomponenten hebben een vaste grootte, zodat verouderde monsters uiteindelijk zullen verdwijnen. Ze zijn vier keer zo groot als de resolutie van het te berekenen beeld en stellen een volume voor. De x- en y-as stellen de positie op het beeld voor en de t-as bepaalt de leeftijd. Per pixelindex is er namelijk een wachtrij van vier elementen. De buffer van het bemonsteringsalgoritme bezit crosshairs als elementen, terwijl de buffer van de reconstructor monsters als elementen bezit.

Het bemonsteringsalgoritme maakt gebruik van een beeldruimtetegeling. Aangezien deze op de buffer van crosshairs toegepast wordt, zullen ook deze tegels een volume voorstellen. De tegeling wordt in een korte gesloten lus zeer regelmatig aangepast aan de recentste vernieuwingen. Het is daardoor mogelijk de bemonstering te richten naar kleurveranderingen, occlusies en dissoclusies. Zo zal het frequenter monsters nemen op beeldregio's die spatiaal en temporeel belangrijk zijn zoals randen en bewegingen. Met behulp van de buffer van crosshairs worden tegelstatistieken berekend om dergelijke belangrijke plaatsen te detecteren. Verschillende gebieden zullen bijgevolg een verschillende dichtheid van monsters bezitten. Gebieden die langer niet bemonsterd zijn, moeten ook af en toe sterker vernieuwd worden. Wanneer veel bewegingen gebeuren of de gemiddelde leeftijd in veel gebieden te hard stijgt, zal het bemonsteringsalgoritme het aantal tegels verminderen. Dan is de bemonstering minder gericht en krijgen andere gebieden ook meer kans vernieuwd te worden.

Dankzij herprojectie wordt de informatie, die de buffer van crosshairs biedt, recenter gehouden dan alleen mogelijk is met nieuwe gevisualiseerde monsters. Gebieden met occlusies worden daarenboven ook rechtstreeks herbemonsterd door via herprojectie te ontdekken waar deze zich voordoen. Deze informatie wordt tevens gebruikt om de tegeling te richten op occlusies. Herprojectie zorgt er verder voor dat de bemonstering gericht wordt op speculaire oppervlakken, onderbemonsterde regio's en gebieden waar zich objectbewegingen voordoen. De herprojectie wordt tevens gericht naar de belangrijke gebieden door de tegeling. In plaats van per nieuw beeld te herprojecteren, worden hier per nieuw monster enkele crosshairs geherprojecteerd.

De communicatie tussen bemonsteringsalgoritme en reconstructor beperkt zich tot het doorzenden van nieuw berekende monsters en per nieuw beeld het doorzenden van tegelposities en gemiddelde kleurgradiënten.

De reconstructor wordt volledig op de grafische kaart uitgevoerd en toont de monsters die zich in zijn buffer bevinden op het scherm. Dat gebeurt door de monsters te herprojecteren naar de juiste posities op het beeld. Op dun bemonsterde plaatsen in het beeld worden monsters als grotere punten getoond zodat minder gaten over zullen blijven. Het gebruikt bovendien schattingen van de bemonsteringsdichtheden en gemiddelde kleurgradiënten in tijd en ruimte om een adaptieve filtering te verkrijgen. Dat is belangrijk omdat de verschillende gebieden van het beeld niet altijd even snel veranderen. De filters krijgen bijgevolg een bepaalde grootte in drie dimensies met behulp van deze informatie. Dankzij temporeel brede filters zullen oudere monsters ook een belangrijke bijdrage leveren in statische gebieden. Wanneer genoeg monsters aanwezig zijn in een gebied is zelfs anti-aliasing mogelijk. Met behulp van nauwe spatiale filters zal de filter randen scherp houden. In dynamische gebieden zullen voornamelijk nieuwe monsters gebruikt worden dankzij nauwe temporele filters, zodat die gebieden

sneller vernieuwd worden. Dat levert een lagere latentie op, maar het gaat wel ten koste van kwaliteit, aangezien de filter spatiaal breder zal interpoleren. Het resulterende gebied in het beeld zal dan waziger zijn.

Via herprojectie wordt er bij elk nieuw beeld voor gezorgd dat monsters steeds op de juiste pixel terechtkomen. Oude monsters dragen dan nuttiger bij dan wanneer ze op hun oude beeldposities blijven staan. Dankzij de filters en de bemonstering is er geen Z-buffer nodig. Door de directe herbemonstering en de tegeling wordt ervoor gezorgd dat genoeg monsters genomen worden op die plaatsen zodat de foute monsters snel verdwijnen. Occlusies vallen voor in dynamische gebieden. Filters zorgen ervoor dat de nieuwere monsters meer bijdragen in dergelijke gebieden, zodat oudere occluderende monsters weinig kans hebben zichtbaar te zijn.

In het implementatiegedeelte 3.4 wordt dieper ingegaan op deze techniek.

VII Vergelijking

Deze subsectie vergelijkt de belangrijkste technieken en verklaart waarom bepaalde technieken gekozen zijn voor implementatie. De cache, reconstructie en bemonstering zijn de hoofdonderdelen van elke techniek en worden daarom vergeleken. De cache moet verouderde monsters doen verdwijnen, nieuwe monsters goed invoegen en het bemonsteringsproces helpen om fouten te detecteren. Het bemonsteringsproces zorgt ervoor dat belangrijke gebieden nieuwe monsters krijgen. De reconstructie zorgt tenslotte voor het maskeren van fouten in de cache. Tabel VII.4 bevat een korte samenvatting van de belangrijke eigenschappen.

VII.1 Cache De holodeck is de enige techniek, die reeds berekende informatie permanent bijhoudt. Bij de opbouw van nieuwe beelden is er dan meer informatie beschikbaar. Spijtig genoeg wordt een groot gedeelte van die informatie ongeldig bij sceneveranderingen. Kleine lokale objectbewegingen kunnen een beetje opgevangen worden, maar schaduwen worden bijvoorbeeld niet vernieuwd. Daarom is deze techniek voornamelijk bruikbaar voor interactieve walkthroughs, waarvoor het dan ook bedoeld is. Verder ondersteunt de techniek enkel beperkte camerabewegingen, aangezien de beeldrepresentatie anders volledig gereconstrueerd moet worden. Indien dat niet gebeurt, zullen occlusiefouten optreden in de vorm van artefacten. Tijdens camerabewegingen worden er trouwens geen nieuwe monsters gegenereerd, wat wel het geval is bij de andere technieken. Tapestry past de mesh dynamisch aan en moet die bijgevolg niet telkens volledig herbouwen na grotere camerabewegingen. Toch is ook deze techniek voornamelijk geschikt voor interactieve walkthroughs, aangezien het eveneens slechts beperkte objectbewegingen ondersteunt. Volgens Tole et al. [92] is het niet eenvoudig om daar verandering in te brengen. De shading cache kan dynamische scenes en bewegende lichtbronnen veel efficiënter behandelen dan de render cache en tapestry. Het duidt ongeldig geworden monsters voor vernieuwing aan met een patch identificeermap. Bewegende lichtbronnen zorgen er bij de andere twee technieken voor dat veel monsters in de cache ongeldig worden en de beeldkwaliteit bijgevolg langer verlaagd zal blijven. Ook na grote camerabewegingen is deze techniek efficiënter, omdat monsters opgeslagen zijn in de 3D-scene zelf. Geometrie wordt dan altijd accuraat weergegeven. De GVE bezit minstens een ruwe representatie van alle objecten en positioneert ze bijgevolg altijd op de juiste positie. De objectranden zullen juist weergegeven worden,

net als mogelijk aanwezige texturen. Hoge frequenties worden dus sneller correct weergegeven. Het invoegen van monsters gebeurt efficiënter dan toevoeging in de mesh van tapestry. Het nadeel is dat oppervlakken lokaal parametriseerbaar moeten zijn en gebogen oppervlakken getesseleerd moeten worden. Het vereist dus kennis van de geometrie van de scene en is er daardoor ook afhankelijk van. De andere technieken kunnen potentieel alle soorten geometriëen aan. Aangezien voor elk object tevens minstens een enkele onopgedeelde patch aanwezig moet zijn, zal er voor complexe scenes een soort culling, versnellingsstructuur of dergelijke nodig zijn om het aantal primitieven te verminderen. Voor een goede beeldkwaliteit is er zelfs per zichtbare primitieve minstens *éé*n monster nodig. Deze techniek zal minder winst opleveren naarmate de complexiteit van de op het scherm zichtbare primitieven stijgt. Tapestry verwijdert monsters uit de cache als die vol zit, tijdens camerabewegingen om niet langer geldige monsters te verwijderen en bij detectie van diepte- of kleurverschillen bij het invoegen van een monster in een driehoek. De shading cache markeert patches, die zichtbaar waren in het vorig beeld, en verwijdert onzichtbare monsters. Deze techniek moet de resolutie van de mesh verkleinen bij camerabewegingen.

VII.2 Reconstructie De holodeck, tapestry en shading cache systemen voegen nieuwe monsters toe in een representatie die direct weergegeven kan worden door de GVE. Daarom kan het hergebruikt worden over meerdere opeenvolgende beelden, in plaats van telkens opnieuw een beeld te moeten opbouwen van de monsters in de cache en te moeten filteren, zoals wel het geval is bij de render cache en adaptive frameless rendering. De drie technieken zijn bijgevolg minder sterk afhankelijk van de monsterdichtheid. De beeldkwaliteit en de snelheid, waarmee beide laatste technieken convergeren naar een optimaal beeld, is ervan afhankelijk. Er kunnen daar wel gaten aanwezig zijn als de monsterdichtheid in de puntenwolk op die plaats te laag is om opgelost te worden door interpolatie. De grootte van de cache moet lineair stijgen met de beeldresolutie om dergelijke gaten te vermijden tijdens stilstand [75]. Aangezien de interpolatie van adaptive frameless rendering adaptief is, lost het meer fouten op dan de render cache. De drie andere technieken interpoleren daarentegen alle drie beter in dergelijke beeldregio's, zodat er geen gaten op het beeld aanwezig zullen zijn. Bij de eerste twee technieken bezet de beeldrepresentatie de hele bol rondom de camera met driehoeken. Bij de shading cache worden monsters opgeslagen in een hiërarchische onderverdelingsmesh in de 3D-scene zelf. Ze voeren allemaal Gouraud interpolatie uit met de GVE, zodat spatiale coherentie uitgebuit wordt. Dat levert het voordeel dat beelden weergegeven kunnen worden tegen hogere snelheden en dat de resolutie van het beeld dan ook een stuk groter gemaakt kan worden binnen de capaciteit van de GVE. Het nadeel van deze GVE-representaties is het feit dat het tijdrovender is om er nieuwe monsters in in te voegen en uit te verwijderen. Daarom zijn deze methoden efficiënter in gevallen waarin slechts weinig monsters in de cache aanwezig zijn. Hoe voller de cache wordt, hoe kleiner de winst van de technieken zal zijn. De holodeck en tapestry hebben het nadeel dat scherpe randen afwezig zijn in de geproduceerde beelden. Het maakt daarbij enkel gebruik van de diepte om objectranden juist voor te stellen. Wanneer de camera vrij lang stilstaat en de resolutie van de mesh reeds hoog genoeg is, worden de randen scherper. Bij camerabewegingen worden ze snel terug vervormd. Ze geven scenegeometrie

dus niet accuraat weer. Tenslotte kan de snelheid, waarmee beelden getoond worden tijdens camerabewegingen, sterk variëren als de mesh groter wordt. Dat kan opgelost worden indien geruild wordt voor beeldkwaliteit. De shading cache heeft tenslotte de nadelen dat aliasingartefacten en schaduwlekken zichtbaar zijn wanneer de cache nog niet genoeg onderverdeeld is. Tenslotte zijn enkel de filters van adaptive frameless rendering spatiaal en temporeel adaptief. Ze zorgen op die manier als enige voor een snel temporeel antwoord, zodat het beeld bij temporele veranderingen recent zal zijn en anti-aliasing mogelijk is bij kleine temporele veranderingen. De techniek vindt impliciet spatiale randen en zorgt ervoor dat ze niet wazig zijn. Gaten worden snel gevuld.

VII.3 Bemonstering De holodeck gebruikt een uniforme bemonstering en is dus niet adaptief. Tapestry past een adaptieve bemonstering toe met behulp van een prioriteitsbeeld. Per driehoek wordt een prioriteit berekend, waarbij het grootste gewicht gegeven wordt aan kleurverschillen tussen de vertices en het kleinste gewicht aan diepteverschillen tussen vertices. De prioriteit wordt gescaleerd met de leeftijd van de drie vertices. De driehoek wordt dan geprojecteerd op het prioriteitsbeeld en met een quasi-randomgenerator worden monsters beeldruimte-afhankelijk gekozen. De shading cache leidt de bemonstering door onderverdeling van de mesh naar belangrijke gebieden volgens camera-, object- en lichtbronbewegingen. Het berekent ook een prioriteit op basis van kleurverschillen, maar hier tussen de vertices in een patch. Bij projectie op een prioriteitsbeeld kiest het in een eerste stap random monsters. Daarna concentreert het op hoge frequenties met een tweede stap. Niet-diffuse oppervlakken komen sneller aan bod door hun leeftijd bij camerabewegingen te scaleren met een heuristiek, gebaseerd op de materiaaleigenschappen. Objectbewegingen worden eerst gedetecteerd met de randomstap en dan verouderd. Diepteverschillen zijn overbodig, aangezien objecten steeds accuraat weergegeven worden. De render cache bepaalt prioriteiten aan de hand van de leeftijd van monsters en de lokale monsterdichtheid. Het houdt onrechtstreeks rekening met kleurverschillen door buurmonsters in de cache sneller te verouderen bij een kleurverschil. Adaptive frameless rendering richt de bemonstering op temporele en spatiale kleurverschillen, waarvan de prioriteit berekend wordt uit een ruimte-tijdsvolume in plaats van een enkel statisch beeld. Het houdt tevens rechtstreeks rekening met occlusies en disocclusies.

VII.4 Andere eigenschappen De gebruiker moet zich bij de holodeck het best binnen op voorhand gespecifieerde roosters begeven, want daarbuiten zullen de resultaten minder goed zijn. De holodeck ondersteunt rechtstreeks kijkpuntafhankelijke effecten, zoals reflecties. Tapestry en de shading cache buiten de GVE uit voor het prioriteitsbeeld, de reconstructie, het bepalen van de zichtbaarheid en het invoegen van monsters in de beeldrepresentatie. Tapestry moet ook nog de herprojectie op de GVE doen, wat bij de shading cache overbodig is. De shading cache heeft weinig extra kosten buiten de tesselatie. Het kan daardoor het grootste deel van de beschikbare verwerkingskracht gebruiken voor de bemonstering. Adaptive frameless rendering kan monsters goed verdeeld over de tijd doorsturen naar de reconstructor.

	Holodeck	Tapestry	SC	RC	AFR
Dynamische scenes	Nee	Nee	Ja	Ja	Ja
Accurate geometrie	Nee	Nee	Ja	Nee	Nee
Accurate texturen	Nee	Nee	Ja	Nee	Nee
Beperkingen op geometrie	Nee	Nee	LP	Nee	Nee
GVE Representatie	DT	DT	HM	Nee	Nee
Gaten in het beeld	Nee	Nee	Nee	Ja	Ja
Permanente opslag	Ja	Nee	Nee	Nee	Nee
Adaptieve bemonstering	Nee	PB, DT	PB, HM	PB	KT
Afhankelijk #primitieven	Nee	Nee	Ja	Nee	Nee
Afhankelijk #primitieven zichtbaar op het beeld	Ja	Ja	Ja	Nee	Nee
Afhankelijk beeldresolutie	Minder	Minder	Minder	Ja	Ja

Tabel 2.1: Samengevatte vergelijking van de verschillende technieken. LP: Lokaal parametriseerbaar, DT: Delaunay Triangulatie, HM: Hiërarchische onderverdelingsmesh, PB: prioriteitsbeeld, KT: kd-boom

VII.5 Conclusie De render cache en adaptive frameless rendering zijn gekozen voor implementatie. De render cache maakt gebruik van eenvoudige en snelle technieken, waardoor het eenvoudig uitgebreid kan worden en vertaald kan worden naar andere omgevingen. Zo zijn er reeds meerdere aanpassingen geïmplementeerd, zoals vermeld is in III. In tegenstelling tot tapestry en de holodeck, ondersteunen de render cache en adaptive frameless rendering wel dynamische scenes. Verder heeft de complexiteit van de scene er geen rechtstreekse invloed op [11]. Onrechtstreeks wel, omdat het visualisatie algoritme dan trager nieuwe monsters zal berekenen. De onderdelen van de technieken zullen dus niet trager uitgevoerd worden naarmate een complexere scene gevisualiseerd wordt, terwijl de shading cache het dan steeds lastiger zal krijgen. De performantie zal ongeveer lineair stijgen met de grootte van de cache. Alle monsters in die cache moeten namelijk geherprojecteerd worden en de datastructuren hebben ongeveer de grootte van de beeldresolutie of een klein veelvoud daarvan. Dit bevordert de scaleerbaarheid voor grote scenes en kan efficiënt geparallelliseerd worden. Beide technieken kunnen bovendien alle soorten geometriëen aan, terwijl de shading cache dat niet kan. Tenslotte bemonsteren alle technieken, behalve adaptive frameless rendering, de tijd op regelmatige intervallen [32] en dat leidt tot een grotere latentie. Ook verouderen ze monsters met een constante snelheid.

Hoofdstuk 3

Implementatie

Dit hoofdstuk behandelt de geïmplementeerde algoritmen. De eerste sectie 3.1 bespreekt het visualisatie algoritme, dat gebruikt wordt voor versnelling met behulp van frameless rendering. Daarna volgt in de tweede sectie 3.2 een vergelijking tussen dubbele buffering en frameless rendering. Vervolgens zal de derde sectie 3.3 de implementatie van het render cache systeem uiteenzetten en tenslotte zal de laatste sectie 3.4 de implementatie van adaptive frameless rendering beschrijven.

3.1 Visualisatie algoritme

Wald et al. [97] hebben in 2002 OpenRT ontwikkeld. Dat is een een realtime raytracer, die kan uitgevoerd worden op standaard massaproductiecomputers. Het is sterk geoptimaliseerd, waarbij het algoritme zelf sneller gemaakt is. Wald et al. beweren dat het een grootte-orde sneller is dan de tot dan toe bereikte snelheden voor raytracers. Er is tevens een API voor ontwikkeld, die de communicatie verzorgt tussen applicaties en de raytracer. Deze API komt grotendeels overeen met OpenGL, zodat programma's eenvoudig overgezet kunnen worden van het gebruik van rasterisatie naar raytracing of andersom. Het maakt het daarnaast ook eenvoudig om de verschijning van objecten aan te passen door programmeerbare shaders gewoon te kunnen invoegen of vervangen. De licht- en cameratypes kunnen ook aangepast worden via programmeerbare licht- of camerashaders.

De achterliggende raytracer ondersteunt verdeling over netwerken en dynamische scenes. De objecten in de scene hebben elk een eigen versnellingsstructuur, zodat ze onafhankelijk van de andere objecten aangepast kunnen worden. Het is mogelijk enkel veranderingen door te sturen, in tegenstelling tot OpenGL die telkens de hele scene moet doorsturen. Het maakt geen gebruik van benaderingen en is geoptimaliseerd door het uitbuiten van parallellisme en coherentie op verschillende gebieden. De belangrijkste effecten worden ondersteund, namelijk directe illuminatie, zachte schaduwen, indirecte illuminatie, reflecties, refracties en caustics. Tenslotte is het mogelijk allerlei soorten primitieven te gebruiken.

OpenRT wordt in deze thesis gebruikt als visualisatie algoritme, waarop frameless rendering toegepast wordt. Als primitieven worden enkel driehoeken gebruikt. Die zijn onderverdeeld in een aantal objecten, zodat OpenRT er effi-

ciënter mee kan omgaan. De scene wordt ingelezen met een MGF parser [107] of een BART parser [59]. Een Phong-shader zal de kleurwaarden bepalen van punten op de objecten.

3.2 Vergelijking van frameless rendering met dubbele buffering

Deze sectie levert net zoals in de oorspronkelijke paper [14] een vergelijking tussen dubbele buffering en frameless rendering. Daarvoor is er een applicatie geïmplementeerd die beide technieken kan uitvoeren. De eerste sectie 3.2.1 bespreekt de implementatie van dubbele buffering, terwijl de tweede sectie 3.2.2 de implemenatie van frameless rendering behandelt. Beide implementaties worden getest op een aantal voorbeeldanimaties, welke besproken wordt in de eerste sectie van het volgende hoofdstuk 4.4.1.

3.2.1 Dubbele buffering

Dubbele buffering genereert met behulp van een timer periodiek nieuwe beelden. Elke pixel moet daarbij voor elk nieuw beeld herberekend worden. Er is dus weinig extra code nodig buiten het visualisatie algoritme. Telkens een gelijke tijdsperiode verstreken is, onderneemt de implementatie de volgende stappen. Eerst past het de camera- en objectparameters aan om het huidig moment voor te stellen. Daaropvolgend voert het een lus uit om alle pixels van het beeld af te gaan en te berekenen. De resulterende monsters worden opgeslagen op de overeenkomende positie in de beeldbuffer. Wanneer alle pixels berekend zijn, wordt de beeldbuffer doorgestuurd naar de GVE. Die wisselt de oude beeldbuffer dan om met de nieuwe beeldbuffer. Tenslotte toont het de nieuwe beeldbuffer op het scherm.

3.2.2 Frameless Rendering

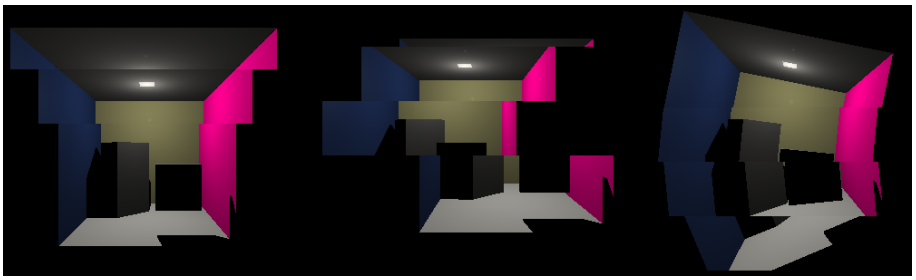
In het ultieme geval toont frameless rendering elk berekend monster direct op de overeenkomende pixel in het beeld. Maar dit is tijdrovend met de huidig beschikbare middelen, aangezien telkens een hele beeldbuffer doorgestuurd moet worden naar de GVE. Daarom wordt hier met behulp van een timer periodiek een gedeelte van de pixels herberekend. Dat zal telkens eenzelfde aantal pixels zijn.

I Selectie van nieuwe monsters

De implementatie van frameless rendering moet de posities voor het gedeelte nieuwe monsters selecteren. Dit is op verschillende manieren mogelijk, maar zoals uiteindelijk zal blijken, is een random bemonstering een goede keuze wanneer geen rekening gehouden wordt met de inhoud van de pixels. De technieken in de volgende secties 3.3 3.4 houden daar wel rekening mee, zodat ze met behulp van importance sampling betere resultaten kunnen bereiken.

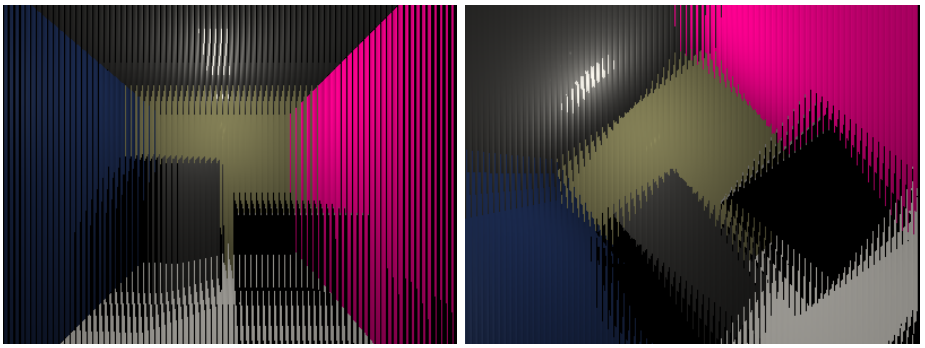
In een eerste implementatie worden monsters sequentieel geselecteerd. Dit

vraagt net als bij dubbele buffering weinig extra kosten, maar leidt tot onaangename resultaten, zoals later besproken zal worden.



Figuur 3.1: Beeldverscheuring bij sequentiële bemonstering

Een tweede implementatie deelt het totaal aantal pixels door het aantal te herberekenen pixels. Dat resultaat is de spatie die zich tussen elk geselecteerd monster zal bevinden. Per nieuw beeld wordt *één* pixel opgeschoven, zodat elk nieuw monster vlak naast een juist berekend monster uit het vorig beeld zal liggen.



Figuur 3.2: (a) Voorwaartse translatie (b) Rotatie

De derde implementatie is de belangrijkste en enkel deze zal hier verder besproken worden. Het selecteert de nieuwe pixels random. Het maakt daarbij gebruik van de SIMD-georiënteerde Mersenne Twister [80] voor het snel genereren van pseudo-random getallen. Dergelijke pseudo-random generatoren hebben een aantal nuttige eigenschappen. Wanneer een bepaald getal aan bod gekomen is, kan niet afgeleid worden welk het volgend getal zal zijn. Elk getal zal ongeveer even vaak gegenereerd worden over een lange termijn en tenslotte mag de volgorde van opeenvolgende getallen zich niet herhalen. SFMT bezit deze eigenschappen, maar het heeft wel een periode van $2^{19937-1}$. Dit betekent dat er toch herhaling zal optreden. Dat gebeurt pas na een zeer lange tijd en er zijn weinig andere pseudo-random generatoren, waarbij de periode zo groot is.

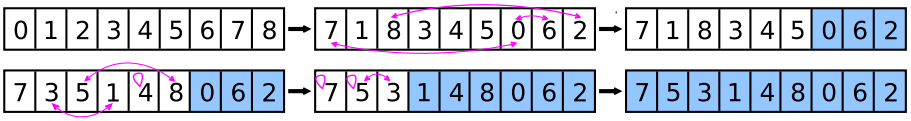
De applicatie duidt per beeld *één* vijfde van het aantal pixels random aan voor herberekening. OpenRT zal deze pixels dan van kleurwaarden voorzien. Telkens de camera of objecten aangepast worden, zal dat direct invloed heb-

ben op de monsters die dan nog berekend moeten worden. De camera- en scenetoestandsveranderingen worden bemonsterd met een hogere snelheid dan de vernieuwing van een heel beeld. Zodoende stellen de monsters verschillende momenten voor in de tijd. Ook deze bemonstering kan random gedaan worden. Dan kan er best wel een interval vastgelegd worden waarbinnen een nieuw monster genomen zal worden, anders kan het zijn dat er weer af en toe te lang gewacht moet worden, waardoor de interactiviteit daalt. Als voorbeeld zal deze applicatie de gebruikersinvoer om de 50ms bemonsteren en zal het *één* volledig beeld per seconde kunnen berekenen.

Figuur 3.3: De groene streepjes duiden de bemonstering van de gebruikersinvoer aan. De zwarte streepjes duiden de weergave van een nieuw beeld aan, waarbij $\frac{4}{5}$de van de monsters reeds in het vorige beeld aanwezig waren en $\frac{1}{5}$de van de monsters nieuw berekend zijn.

II Random selectie met behulp van een tabel

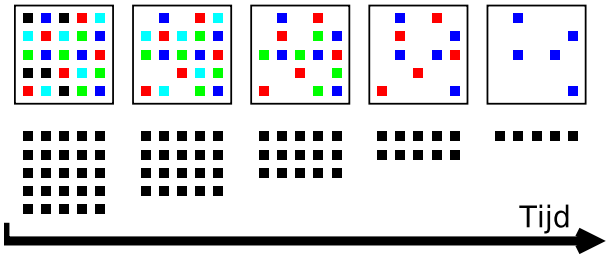
Indien monsters random gekozen worden, is het mogelijk dat het lang duurt vooraleer bepaalde pixels herbemonsterd worden en kunnen bepaalde andere pixels zeer vaak herbemonsterd worden. Daarom is er een gewone versie van deze random bemonstering geïmplementeerd, maar ook een versie die dit probleem verhelpt. Dat is de methode die hieronder verder besproken wordt.



Figuur 3.4: Voorbeeldtabel voor een beeld met een resolutie van 3×3 pixels dat telkens $\frac{1}{3}$de van het aantal pixels vernieuwt. De paarse pijlen geven aan welke pixelposities omgewisseld worden, doordat de index van de ene random gekozen is en de andere op dat moment laatst staat in het deel van de tabel dat aangepast mag worden. Telkens een pixelpositie gekozen wordt, verkleint dat deel met *één* index. Het blauwe gebied is telkens het onaanpasbaar gedeelte. Nadat alle pixelposities aan bod geweest zijn, zal dit zich herhalen. Er wordt dan gestart vanaf de nieuwe tabel.

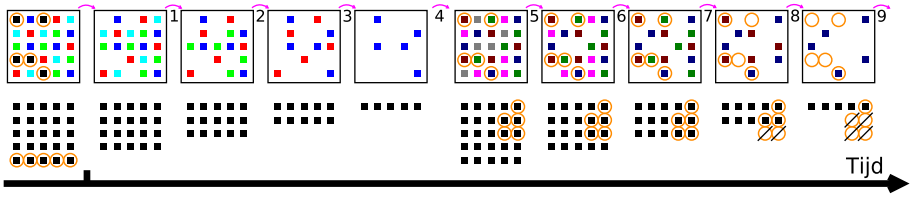
Om te verzekeren dat uiteindelijk alle pixels herberekend zullen worden, is er een tabel aanwezig die bijhoudt welke pixels reeds bemonsterd zijn. De tabel bevat initieel alle pixelposities in stijgende volgorde, alhoewel dat niet verplicht is. De tabel is dus even groot als het aantal pixels. Bij het aanduiden van monsters, zal random een index van de tabel genomen worden. In de tabel zit op die plaats een bepaalde pixelpositie, welke dan genomen wordt om door de raytracer te laten bemonsteren. Die pixelpositie wordt nu van plaats verwisseld met de pixelpositie die achteraan in de tabel zit. Stel dat N het aantal pixels voorstelt. De index van de volgende pixelpositie zal dan random gekozen worden

uit het bereik van nul tot $N-1$. Daarna zal het omgewisseld worden met de index op positie $N-1$. Zo wordt gegarandeerd dat er geen pixelposities meerdere keren genomen kunnen worden vooraleer alle pixelposities herberekend zijn. Dit proces zal zich herhalen totdat alle pixels aan de beurt geweest zijn. Daarna wordt opnieuw gestart met een bereik van N indices, waaruit random gekozen zal worden. Het proces herhaalt zich dan op dezelfde manier. Zodoende zullen alle pixels herberekend zijn nadat vijf "frameless"beelden getoond zijn.



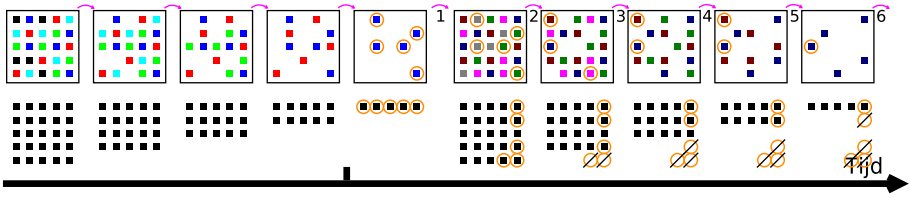
Figuur 3.5: Voorbeeld van een beeld met een resolutie van 5×5 pixels, waarbij telkens $\frac{1}{5}$de van het aantal pixels vernieuwd wordt. Na vijf beelden zullen alle pixels vernieuwd zijn. De frameless renderer convergeert dan naar het correcte huidige beeld, indien er geen sceneveranderingen of camerabewegingen gebeurd zijn sinds het eerste van deze vijf beelden.

Indien vijf beelden per seconde getoond worden, betekent dit dat alle pixels herberekend zullen zijn na *één* seconde. Wanneer nu toch camerabewegingen of sceneveranderingen voorvallen tijdens het verloop van deze tabel en stoppen na de vernieuwing van het eerste gedeelte, zal de tabel een tweede keer doorlopen moeten worden voor convergentie met de huidige moment.



Figuur 3.6: Wanneer de veranderingen vlak na het eerste gedeelte stoppen zijn er maximum negen beelden nodig voor convergentie. Het kan natuurlijk zijn dat toevallig alle veranderde pixels vernieuwd worden na het eerste gedeelte van de tweede iteratie. Dan zijn slechts vijf beelden nodig voor convergentie, wat tevens het minimum is als op deze manier gewerkt wordt. De oranje cirkels geven aan welke pixels nog waarden bevatten van het vorige beeld. De tweede rij vol zwarte pixels is een aanduiding van de hoeveelheid pixels, die nog vernieuwd moeten worden eer de tabel volledig afgewerkt is.

Het is mogelijk de afwerking van de tabel te onderbreken wanneer veranderingen ophouden en dan een nieuwe tabel te starten. Het beeld zal dan convergeren in maximum vijf beelden, indien ondertussen geen nieuwe verandering start.



Figuur 3.7: Wanneer de veranderingen vlak voor het laatste gedeelte van de eerste iteratie stoppen zijn maximum zes beelden nodig voor convergentie en minimum vijf. De oranje cirkels duiden in dit geval de pixels aan die tijdens de afwerking van de eerste tabel nog vernieuwd zijn vlak na de veranderingen.

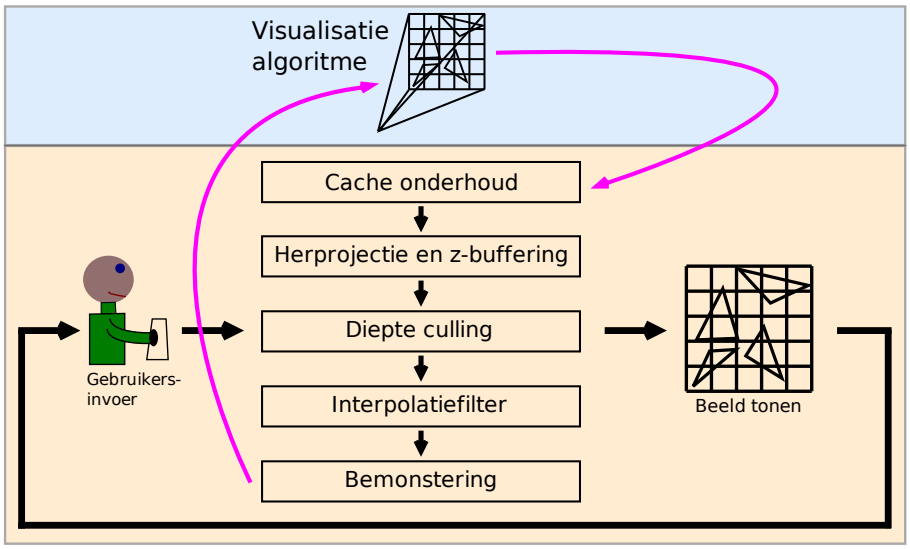
3.3 Render Cache

Walter et al. [106] maken gebruik van verschillende concepten om de gebruiker snel van visuele terugkoppeling te voorzien. Deze concepten omvatten progressieve verfijning, uitbuiten van spatiotemporele beeldcoherentie, herprojectie, adaptieve bemonstering en asynchrone vernieuwing van het beeld en de shadingwaarden. In de eerste 3.3.1 sectie wordt de implementatie van de asynchrone beeldvernieuwingslus kort besproken. De tweede sectie 3.3.2 zal de implementatie van de onderdelen verder uitleggen. De derde sectie 3.3.3 bespreekt de geïmplementeerde onderdelen uit de tweede paper van Walter et al. [105]. De laatste sectie 3.3.4 bespreekt de implementatie van de beeldreconstructie op de grafische kaart.

3.3.1 Asynchrone componenten

De methode bevat twee grote componenten. Er is een beeldgeneratieproces en een visualisatie algoritme. Het beeldgeneratieproces wordt uiteengezet in de volgende sectie 3.3.2. Het visualisatie algoritme staat in voor het berekenen van de monsters. Beide componenten worden asynchroon uitgevoerd in plaats van sequentieel, zodat de vernieuwing van het beeld onafhankelijk is van de vernieuwing van de shadingwaarden. Nieuwe beelden kunnen bijgevolg sneller gecreëerd worden dan de snelheid van het visualisatie algoritme zou toelaten in het sequentiële geval. Het beeldgeneratieproces zal namelijk niet telkens moeten wachten totdat een bepaald aantal monsters klaar zijn om een beeld op te bouwen. Na een vast interval toont het een beeld, onafhankelijk van het feit dat het visualisatie algoritme al dan niet klaar is met de berekening van een bepaalde verzameling monsters. Aangezien het beeldgeneratieproces toch een minimaal aantal nieuwe monsters nodig heeft om nuttige beelden te genereren, is de beeldweergavesnelheid toch nog een beetje afhankelijk van de snelheid van het visualisatie algoritme.

Er wordt gebruik gemaakt van twee threads. Indien slechts *één* processor gebruikt wordt, zal continu afgewisseld worden tussen de uitvoering van een deel van de eerste component en een deel van de tweede component. Op deze wijze wordt parallelle verwerking nagebootst. Indien meerdere processoren aanwezig zijn, zal de ene component gewoonweg op de ene processor uitgevoerd worden, terwijl de andere component parallel op de andere processor uitgevoerd wordt.



Figuur 3.8: Twee asynchrone componenten.

De communicatie tussen beide threads gebeurt door extra buffers bij te houden in beide threads. Wanneer het beeldgeneratieproces klaar is met het prioriteitsbeeld en weet welke monsters het wil aanvragen, worden deze buffers doorgestuurd naar het visualisatie algoritme. Ze worden gekopieerd in een extra lokale buffer. Wanneer het visualisatie algoritme klaar is met de berekeningen van de vorige aanvragen, zal het de buffers kopiëren naar de lokale buffers waarmee het zijn operaties uitvoert. Deze twee operaties worden in een mutex gezet, want ze moeten sequentieel gebeuren zodat er geen overschrijving mogelijk is terwijl het visualisatie algoritme zijn berekeningen doet. De monsters die berekend zijn door het visualisatie algoritme worden op dezelfde manier doorgegeven aan het beeldgeneratieproces.

Indien het visualisatie algoritme sneller de monsters berekend heeft dan de beschikbare tijd, kunnen eventueel extra monsters berekend worden. Wanneer het algoritme trager monsters berekend heeft dan de beschikbare tijd, kan het beeldgeneratieproces ondertussen gewoon een beeld genereren met de reeds aanwezige monsters in de render cache.

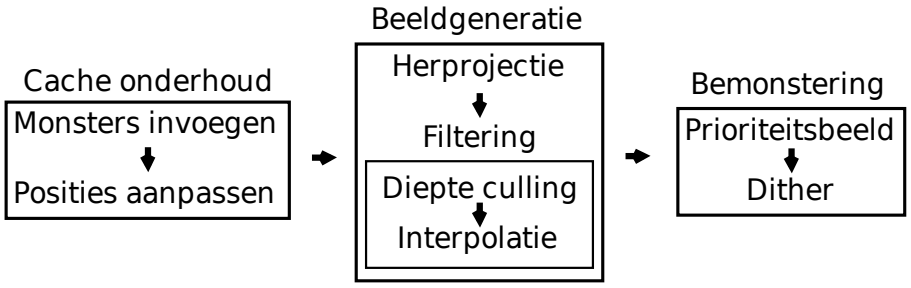
Als een wachtrij gebruikt wordt voor bemonsteringsaanvragen en de render cache steeds te traag nieuwe monsters berekent, kunnen oude bemonsteringsaanvragen best weggeworpen worden zodat de wachtrij niet blijft aangroeien. Dan zullen waarschijnlijk monsters berekend worden die niet meer van toepassing zijn op het huidig beeld. Daarbij moet ervoor gezorgd worden dat monsters niet sequentieel berekend worden, want anders zal het laatste deel van het beeld weinig aan bod komen. Het is bovendien sneller om ineens buffers van waarden te kopiëren dan individuele monsters. Daarbij moet anders telkens gebruik gemaakt worden van mutexen. Tenslotte is er in deze techniek geen baat bij om nieuwe monsters in de cache te integreren terwijl geen nieuw beeld gegenereerd wordt. Het moet er gewoon zijn wanneer de berekening van het beeld start.

Vanwege deze redenen worden buffers met bemonsteringsaanvragen gekopieerd en wordt ervoor gezorgd dat de beschikbare tijd om te bemonsteren optimaal benut wordt.

Aangezien niet elk monster dezelfde berekeningstijd vereist, kan het aantal berekende monsters variëren. Bij korte opeenvolgende bewegingen zal dit waarschijnlijk minder sterk verschillen. Daarom is het bijvoorbeeld mogelijk ruwweg het aantal aan te vragen monsters, dat berekend kan worden, te voorspellen. De berekeningstijd hangt onder andere af van het materiaal dat het object bezit.

3.3.2 Werking van het beeldgeneratieproces

Het beeldgeneratieproces is verantwoordelijk voor drie grote taken. Het zorgt voor het onderhoud van de cache, voor het periodiek tonen van een beeld en voor het aanduiden van de door het visualisatie algoritme te bemonsteren pixels. Subsectie I behandelt het cache onderhoud. Subectie II beschrijft de beeldgeneratie en tenslotte zet subsectie III het bemonsteringsproces uiteen.



Figuur 3.9: Drie taken van het beeldgeneratieproces.

I Cache onderhoud

Nieuw berekende monsters worden opgeslagen in de cache, zodat ze in volgende beelden hergebruikt kunnen worden. Deze cache heeft een vaste grootte, die ongeveer gelijk is aan het aantal pixels in het beeld. Er is geëxperimenteerd met een cache die exact even groot is als het beeld en met caches die groter zijn. Het gevolg van deze vaste grootte is, dat eens deze cache gevuld is, elk nieuw berekend monster een monster in de cache zal moeten overschrijven. Op die manier zal de cache altijd redelijk recente data bevatten, want oude monsters zullen uiteindelijk verdwijnen doordat een leeftijd bijgehouden wordt die mede bepalend is voor zijn kans op herbemonstering. Het overschrijven van oudere monsters in de cache leidt ook tot snellere operaties op de cache, aangezien het aantal operaties lineair stijgt met de grootte van de cache. Daardoor stijgt het ook lineair met de beeldresolutie.

Het overschrijven van vroegere monsters gebeurt op twee manieren. Een nieuw monster is ofwel een herbemonstering van een oude 3D-wereldlocatie ofwel is het een bemonstering van een nieuwe locatie. Bij het toevoegen van een monster wordt daarom eerst gekeken of het monster op de overeenkomende positie door dezelfde 3D-locatie gaat. Als dat het geval is, overschrijft het nieuwe monster

dat oude monster op die positie in de cache. In het andere geval wordt een klein gedeelte van de cache onderzocht in round-robin volgorde. Dat betekent hier dat de hele cache onderverdeeld is in groepen van bijvoorbeeld acht pixels. Afwisselend wordt een andere groep genomen en doorzocht. Het oudst gevonden monster van die groep wordt dan overschreven.

Telkens monsters berekend zijn door het visualisatie algoritme worden die in deze cache opgeslagen, samen met een aantal eigenschappen. Zo wordt de door het visualisatie algoritme berekende kleur, 3D-locatie en object-ID opgeslagen. Verder krijgt het een leeftijd toegekend, die initieel nul is en gelijkmatig veroudert bij elk nieuw gegenereerd beeld. Tenslotte wordt nog een beeld-ID opgeslagen, dat telkens zal verwijzen naar zijn geherprojecteerde positie in het puntbeeld.

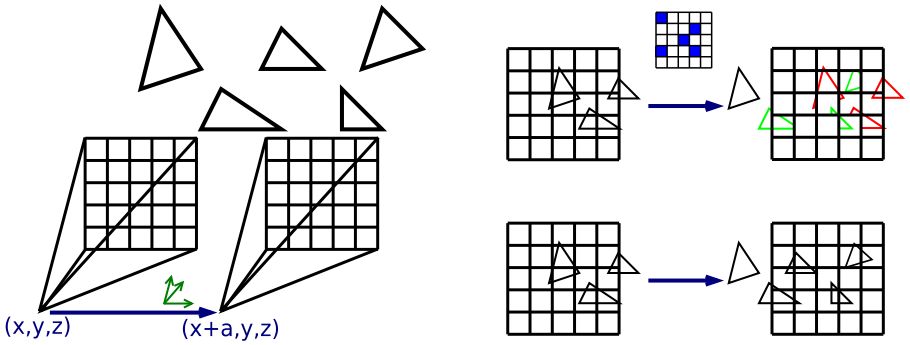
II Beeldgeneratie

Het beeldgeneratieproces zal telkens zo snel mogelijk een benadering moeten afleveren van het eigenlijke beeld op een bepaald moment. Vaak zal het dat moeten verwezenlijken met behulp van slechts een beperkt aantal monsters. De kwaliteit zal afhangen van hoe sterk de monsters in de cache het huidig tijdstip voorstellen en hun aantal. Om regelmatig een nieuw beeld te tonen, wordt een timer gebruikt, die telkens na eenzelfde tussenpoze het beeldgeneratieproces op de hoogte brengt. Het proces zal de monsters in de cache gebruiken om het beeld op te bouwen in een aantal stappen:

1. Herprojectie
2. Diepte culling filter
3. Interpolatie filter

II.1 Herprojectie Deze stap herprojecteert elk monster uit de cache naar zijn nieuwe positie op het beeldvlak, daarbij gebruikmakende van de huidige cameraparameters. Dankzij deze stap wordt het beeld dus sneller aangepast aan camerabewegingen, want het geeft de aanwezige monsters van de statische objecten in de scene direct op de juiste nieuwe positie weer. Dat is mogelijk omdat de 3D-positie van elk monster ook opgeslagen is in de cache. Die 3D-positie vermenigvuldigt het met de inverse cameramatrix om cameracoördinaten te verkrijgen. Daarna deelt de stap het resultaat door de w-coördinaat om het resultaat te normaliseren. Tenslotte vermenigvuldigt het de x-coördinaat met de helft van de breedte van het beeld en de y-coördinaat met de helft van de hoogte van het beeld. Het resultaat is de x en y positie op het beeld.

Het render cache systeem ondersteunt gedeeltelijk dynamische objecten. Objecten, die getransleerd of geroteerd worden, kunnen sneller op hun juiste positie op het scherm weergegeven worden. Aangezien de ID's van de objecten opgeslagen zijn bij de cache-elementen, kunnen de monsters gevonden worden die behoren tot de bewegende objecten. Hun 3D-posities kunnen dan ook getransleerd of geroteerd worden. Bij de beeldgeneratie komen ze door de herprojectie dan terecht op hun nieuwe positie. Dit werkt enkel voor translatie en rotatie, aangezien dat rigid body transformaties zijn. Die behouden de onderlinge afstand tussen twee individuele punten van het object. Scalering komt bijvoorbeeld niet in aanmerking. Het render cache systeem is namelijk niet op de



Figuur 3.10: (a) Cameratranslatie naar rechts volgens x-as (b) Herberekening van 5 posities en geen herprojectie. De groene lijnen geven de posities aan die wel juist staan, de rode geven de posities aan die verouderd zijn. De delen van de driehoeken buiten de beelden zijn enkel ter verduidelijking van de posities. Deze zijn niet zichtbaar voor de respectievelijke camera's. (c) Herberekening van 5 posities en wel herprojectie. De verouderde monsters worden ook op de nieuwe posities geplaatst door herprojectie.

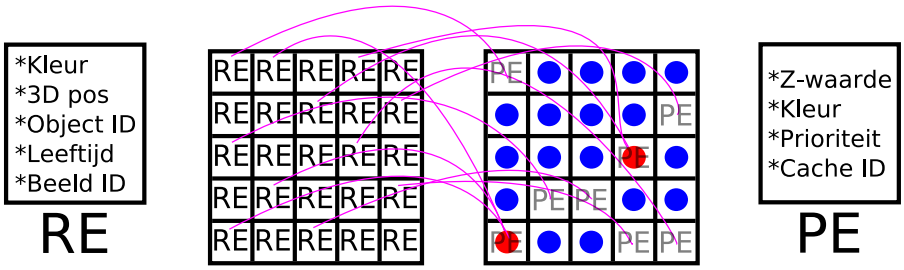
hoogte van welke punten op het object voorgesteld worden door de monsters. Twee individuele punten van een object, die getransleerd of geroteerd worden, kunnen verplaatst worden met dezelfde constante functie. Bij een scalering kunnen bepaalde punten op dezelfde positie blijven, terwijl andere punten sterk verplaatst worden.

De geherprojecteerde monsters, die op een pixel terechtkomen, worden voorlopig opgeslagen in een datastructuur, die het beeld voorstelt. Dat is het puntbeeld en bevat even veel elementen dan dat er pixels zijn in het beeld. De eigenschappen van de individuele elementen in dit puntbeeld worden aangepast aan de eigenschappen van de monsters. Deze bevatten de kleur, de dieptewaarde, de prioriteit en het cache-ID. De dieptewaarde is de z-waarde die berekend werd bij de herprojectie. De prioriteit komt aan bod in subsectie III. Het cache-ID geeft aan welk element in de cache geprojecteerd werd op deze plaats. Figuur 3.11 toont het een mogelijke projectie op het puntbeeld.

Herprojectie kan leiden tot monsters die het beeldvlak verlaten, foute kijkpuntafhankelijke kleurwaarden en occlusiefouten. De monsters die buiten het beeld vallen, worden niet gebruikt en hun leeftijd wordt bovendien telkens verhoogd met een extra increment. Dit is een heuristiek. Monsters die niet meer op het scherm aanwezig zijn, dragen niets bij tot het huidige beeld en zullen dat waarschijnlijk ook niet doen in daaropvolgende beelden. Bijvoorbeeld indien men een tijdje naar rechts beweegt met de camera, zullen de monsters die links naast het beeld vallen niet meer van pas komen voor de nieuwe beelden.

De kijkpuntafhankelijke fouten, zoals bijvoorbeeld speculaire oppervlakken, worden niet behandeld door deze techniek. Herprojectie projecteert een bepaald monster op de juiste positie, maar zijn kleurwaarde kan fout zijn, aangezien de kleur vanuit een andere hoek vrij verschillend kan zijn. De techniek zal pas herstellen van deze fouten wanneer er genoeg nieuwe monster berekend zijn vanaf de huidige camera positie.

Occlusiefouten vallen voor in een aantal gevallen. Wanneer meerdere mon-



Figuur 3.11: RE: rendercache-element, PE: puntbeeldelement. Deze afbeelding is een mogelijk voorbeeld van een herprojectie van de elementen van de rendercache. De paarse lijnen geven aan welke elementen op welke pixels terechtkomen. Rode bollen geven aan waar meerdere elementen op dezelfde pixel terechtkomen. Blauwe bollen geven pixels aan die geen element ontvangen. De rendercache-elementen waar geen paarse lijnen vertrekken vallen buiten het beeld.

sters op eenzelfde pixel geprojecteerd worden zal een keuze gemaakt moeten worden welk monster getoond zal worden. Een andere occlusiefout valt voor wanneer een bepaald oppervlak een achterliggend oppervlak gaat bedekken. Als er dan geen monster beschikbaar is in de rendercache die dat bedekkend oppervlak voorstelt en ook op die pixel geherprojecteerd wordt, zal een onderliggend oppervlak zichtbaar zijn, indien daar wel een monster van geherprojecteerd wordt op die pixel. Tenslotte kan het gebeuren dat er gaten in het beeld ontstaan, aangezien monsters naast het beeld terecht kunnen komen of dat er meerdere monsters op dezelfde positie terecht kunnen komen. Aangezien er ongeveer evenveel monsters in de cache aanwezig zijn als er pixels zijn, zullen er mogelijk pixels overschieten die geen monster ontvangen.

De problemen, die voorkomen wanneer meerdere monsters op dezelfde pixel terechtkomen, worden opgelost door Z-buffering. Bij de herprojectie wordt een z-waarde verkregen die aangeeft hoe ver het monster van de camera afligt. Daardoor kan telkens, wanneer een monster op een pixel geprojecteerd wordt dat reeds een monster bevat, getest worden of het nieuwe monster dieper ligt. Indien het verder van de camera ligt wordt het oude monster behouden, anders wordt het nieuwe monster op die plaats gezet. Deze soort occlusiefouten worden bijgevolg correct behandeld.

II.2 Beeldreconstructie Voor de reconstructie van het beeld uit een incomplete verzameling monsters, maakt het beeldgeneratieproces gebruik van een aantal heuristieken. Dat zijn methoden die snel tot benaderde resultaten leiden, maar niet altijd de optimale oplossing vinden. Deze worden in de volgende twee stappen uitgevoerd.

De diepte culling stap verhelpt gedeeltelijk de occlusiefout, waarbij onderliggende oppervlakken zichtbaar zijn door bovenliggende oppervlakken. Dat gebeurt met behulp van een filter. Aangezien dergelijke monsters gewoonlijk heel wat dieper liggen dan hun buren, neemt het een 3×3 buurt voor elke pixel op het beeldvlak. Daarvan berekent het een gemiddelde diepte, die enkel

rekening houdt met pixels waarop zich monsters bevinden. Indien de diepte van de pixel sterk verschilt van dit gemiddelde, werpt de filter het monster op deze pixel weg. Deze drempelwaarde is bijvoorbeeld 110% van de gemiddelde diepte. Spijtig genoeg leidt deze heuristiek ook tot het onterecht verwijderen van monsters, bijvoorbeeld aan de randen van objecten waar grote diepteverschillen aanwezig kunnen zijn. Maar dat verbergt de render cache grotendeels in de interpolatiestap, die het daarna toepast. De gewichten van de diepte culling filter worden zo bepaald:

- 1 voor de centrumpixel, de directe buurpixels en de diagonale buurpixels
- 0 indien geen monster aanwezig is op een pixel

1	1	1
1	1	1
1	1	1

De interpolatiestap zorgt voor **interpolatie** tussen buurpixels, waardoor gaten in de oppervlakken van objecten gedeeltelijk verborgen zullen worden. Het gaat voor elke pixel weer zijn 3×3 buurt beschouwen om een gewogen gemiddelde te nemen van de kleurwaarden. De gewichten worden als volgt bepaald:

- 4 voor de centrumpixel
- 2 voor de directe buurpixels
- 1 voor de diagonale buurpixels
- 0 indien geen monster aanwezig is op een pixel

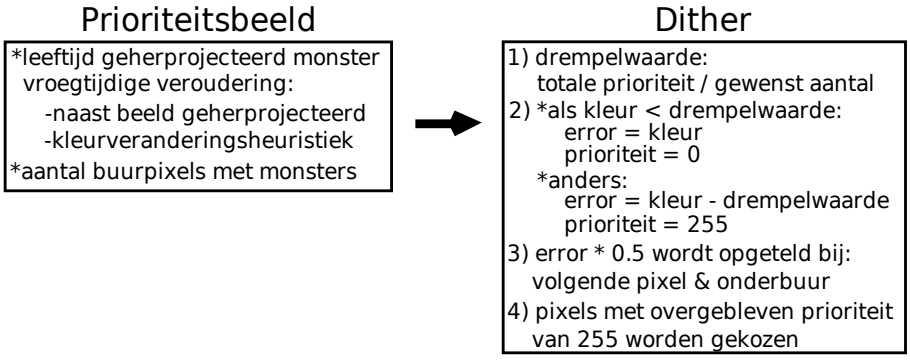
1	2	1
2	4	2
1	2	1

Indien geen van de negen pixels een monster bevat, blijft de kleur van het vorige beeld behouden of wordt de pixel zwart getoond. Anders wordt het gewogen gemiddelde genomen als kleur.

III Bemonstering

De bemonstering zorgt ervoor dat punten in de rendercache relevant blijven voor huidige temporele evoluties. Aangezien het visualisatie algoritme minder monsters neemt dan noodzakelijk is om telkens een heel beeld voor te stellen, moet dit onderdeel de posities voor herbemonstering goed kiezen. Dit onderdeel gebruikt, net zoals de beeldreconstructie, heuristieken. Zoals eerder vermeld zijn objectgrenzen en bewegingen goede posities voor nieuwe monsters. Ook moet een goede spatiale verdeling gehandhaafd worden om nieuw ontspringende bewegingen te detecteren.

Voor het verwezenlijken van deze doelen maakt de render cache gebruik van een prioriteitsbeeld. Prioriteiten van beeldpunten worden berekend aan de hand



Figuur 3.12: Onderdelen van de bemonstering

van de leeftijd van het monster dat erop geprojecteerd wordt en aan de hand van de interpolatiefilter. Oudere monsters zullen meer kans hebben niet langer correct te zijn. Daarom krijgt de pixel, waarop ze geherprojecteerd zijn, een prioriteit gelijk aan deze leeftijd. Zo wordt gegarandeerd dat uiteindelijk alle monsters vervangen zullen worden. Elk monster in de rendercache krijgt initieel een leeftijd van 0 toegekend. Bij elk nieuw beeld verhoogt de techniek deze leeftijd met een constant increment.

Door monsters sneller te verouderen zullen ze sneller overschreven worden in de rendercache en zullen ze meer bijdragen aan de prioriteit van de pixel waarop ze herprojecteren. Op die manier kan de techniek herbemonstering concentreren op bepaalde zaken. Het zal zo enkele heuristieken toepassen. Monsters die door herprojectie naast het beeld vallen, zal het bijvoorbeeld sneller verouderen, aangezien ze waarschijnlijk niet meer van direct nut zullen zijn. Verder concentreert het bijvoorbeeld met een kleurveranderingsheuristiek op veranderingen in de scene, zoals occlusie- en illuminatieverschillen. De 3D-positie van een nieuw berekend monster vergelijkt het met de 3D-positie van het monster dat op de pixel, waarvoor het nieuw monster genomen is, aanwezig is. Als dat dezelfde positie is, vergelijkt het de kleurwaarden van beide monsters. Bij grote kleurveranderingen veroudert het de omliggende monsters. Een bewegend object bestaat meestal uit meerdere pixels op het beeld, welke gewoonlijk aaneengesloten zijn. Daarom is de kans groot dat daar ook veranderingen gebeurd zijn. Bij illuminatieverschillen geldt dezelfde motivatie. Nieuwe bemonsteringsheuristieken kunnen ingevoegd worden, door de leeftijd van monsters te verhogen.

Tijdens de interpolatie stap past de render cache de prioriteit verder aan. Gebieden, waar weinig monsters aanwezig zijn, krijgen een hogere prioriteit. Zodoende baseert de techniek de prioriteit op het aantal buren monsters toegekend kregen tijdens de herprojectie. Indien noch de pixel zelf, noch de buren monsters bevatten, dient het een maximale prioriteit van 255 toe. Indien buren wel monsters bevatten, past het de prioriteit naargelang aan. De minimale prioriteit is 0. Een pixel heeft 8 buren. Daarom krijgt de pixel per buur een toegevoegde prioriteit van 255/9.

Het prioriteitsbeeld bevat nu voor elke pixel een prioriteit. Hieruit moet slechts een fractie van de pixels aangeduid worden voor herbemonstering. **Error diffusion dither** wordt tot dit doel toegepast op het prioriteitsbeeld. Op die

manier wordt een goede afweging bekomen tussen gerichte bemonstering en goed spatiaal verdeelde bemonstering. Normaal wordt deze techniek gebruikt om grijswaardebeelden om te zetten naar een binair beeld (zwarte punten of witte punten). Daarbij worden donkere gebieden voorgesteld door veel zwarte punten, en lichte gebieden door weinig goed verdeelde zwarte punten. Gebieden van hoge prioriteit komen overeen met de donkere gebieden, terwijl de andere gebieden minder goed verspreide monsters krijgen. Eenvoudige dithering doet enkel voor elke pixel een vergelijking met een drempelwaarde, maar dat leidt tot een groter verlies aan kwaliteit. Error diffusion dither houdt een soort geschiedenis bij. Telkens een waarde groter is dan de vaste drempelwaarde, verspreidt het dit extra gedeelte onder buurpixels. Pixels met hoge waarden zullen hun buren een grotere kans geven ook de drempelwaarde te overtreffen, terwijl pixels, die slechts een beetje groter zijn dan de drempelwaarde, hun buren amper zullen helpen. Pixels met lage waarden worden niet gekozen en helpen hun buren ook meer dan pixels die juist kleiner zijn dan de drempelwaarde.

255	255	255	255
255	255	255	255
255	255	255	255

0	383	255	255
383	255	255	255
255	255	255	255

0	255	255	255
383	255	255	255
255	255	255	255

0	255	0	383
383	255	383	255
255	255	255	255

0	255	0	255
383	255	383	255
255	255	255	255

0	255	0	255
383	255	510	0
255	255	255	383

0	255	0	255
383	319	255	0
255	255	319	383

0	255	0	255
542	0	255	0
255	414	319	383

0	255	0	255
255	0	255	0
335	414	319	383

0	255	0	255
255	0	255	0
0	582	319	383

0	255	0	255
255	0	255	0
0	255	419	383

0	255	0	255
255	0	255	0
0	255	255	255

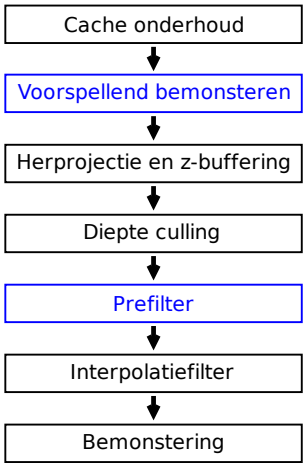
Figuur 3.13: Voorbeeld van error diffusion dither. Alle pixels hebben hier een prioriteit van 255 en er zijn ongeveer 8 pixels gevraagd. Volgens de uitgelegde regels in de tekst hierboven en hieronder worden 7 pixels gekozen. De totale prioriteit is namelijk 3060 en de drempelwaarde is bijgevolg $3060/8 = 382.5$. De waarden in de tabel zijn afgerond en zijn opgesteld van links naar rechts en van boven naar onder.

De dithering volgt het prioriteitsbeeld volgens de scanlijnen. Per lijn wisselt het van richting. Bij elk punt vergelijkt het de prioriteit met de volgende drempelwaarde $\frac{totale\ prioriteit}{aantal\ aan\ te\ vragen\ monsters}$. Indien de prioriteit hoger is, zal het de pixel aanduiden voor herbemonstering. Het trekt de drempelwaarde dan af van de prioriteit. De helft van deze resterende prioriteit telt het dan op bij de volgende pixel op de scanlijn en bij de prioriteit van de pixel die zich op dezelfde positie bevindt in de volgende scanlijn. Indien de prioriteit lager is, zal het de pixel niet aanduiden voor herbemonstering en zal het de helft van de prioriteit optellen bij dezelfde pixels als in het vorige geval.

3.3.3 Uitbreidingen

Walter et al. [105] hebben nog enkele uitbreidingen toegevoegd aan het oorspronkelijke render cache systeem. De berekeningen zijn geherorganiseerd om geheugencoherentie uit te buiten, SSE is toegepast om parallellisme uit te buiten, voorspellend bemonsteren zorgt ervoor dat gaten sneller opgevuld worden en tenslotte vult een 7×7 filter gaten op in gebieden waar de monsterdichtheid

te laag is om dat te verwezenlijken met de standaard 3×3 interpolatiefilter. De laatste twee uitbreidingen zijn geïmplementeerd en worden hier besproken.



Figuur 3.14: Operaties per nieuw beeld. De blauwe operaties zijn nieuw.

I Voorspellend bemonsteren

De bemonsteringstechniek van de originele render cache bezit latentie, die gereduceerd kan worden door voorspellend te bemonsteren. Wanneer de camera beweegt of wanneer objecten bewegen kunnen delen van de scene zichtbaar worden, die grote gebieden innemen op het beeld. Daarvoor zijn geen monsters beschikbaar. De originele bemonsteringstechniek zal dergelijke gebieden telkens pas ontdekken nadat de bewegingen reeds gebeurd zijn, omdat het bemonsteringsposities selecteert aan de hand van het huidig beeld. Pas daarna zal het trachten de gaten op te vullen. Zo is er minstens *één* beeld vertraging tussen de aanvraag van een monster en de integratie in de rendercache. Het is nu mogelijk om niet te wachten totdat dit gebeurt en reeds op voorhand dergelijke gebieden te ontdekken. Wanneer de camera of objecten bewegen, zullen ze in korte opeenvolgende tijdsstappen veel kans hebben dezelfde snelheid en richting aan te houden. Daarom is het mogelijk te voorspellen hoe het beeld er zoveel tijd later ongeveer zal uitzien. Het verhoogt de beeldkwaliteit tijdens bewegingen, ten koste van een lage toegevoegde berekeningstijd.

In deze implementatie is voorspellend bemonsteren enkel geëxperimenteerd voor camerabewegingen. Telkens de camera bewogen heeft, wordt voorspellend bemonsteren geactiveerd voor het volgend beeld. Een deel van het totaal aantal monsters dat berekend kan worden, zal ter beschikking gesteld worden van de voorspellende bemonstering. Wanneer de camera stilstaat zal voorspellend bemonsteren gedeactiveerd worden, zodat dat aantal bemonsteringsaanvragen terug volledig gebruikt kan worden voor het standaard bemonsteringsproces.

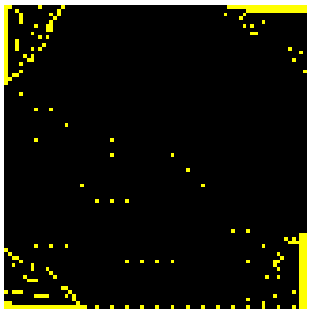
De snelheid en richting van de camera worden telkens gebruikt om de verwachte cameratoestand van acht beelden later te berekenen. De huidige cameraparameters worden daarvoor eerst gekopieerd. Op die kopie worden de bewegingen toegepast. Met behulp van deze nieuwe cameratoestand worden

alle monsters in de rendercache geherprojecteerd op een beeld met een lagere resolutie. Herprojectie zal hier minder berekeningstijd kosten, aangezien er geen Z-buffer bijgehouden moet worden. Het is enkel nodig te ontdekken welke pixels een monster bevatten en welke niet. Bovendien moeten slechts grote lege gebieden ontdekt worden en moet slechts een deel van de pixels bemonsterd worden in die lege gebieden. Daarom is de breedte en de hoogte van dit voorspeld beeld 25% van respectievelijk de breedte en de hoogte van het echte beeld. Op die manier worden monsters juist dicht genoeg opeen genomen, zodat de interpolatiefilter de gebieden kan opvullen. Bijgevolg wederom besparingen gedaan op het aantal te bemonsteren pixels, aangezien niet meer monsters genomen worden dan nodig is. Elke pixel, die geen monster bevat in dat lagere resolutie beeld, zal bemonsterd worden tenzij dat aantal lege pixels nog te hoog is. Indien het aantal lege pixels boven een bepaalde drempelwaarde ligt, wordt het gedeeld door de drempelwaarde. Die waarde wordt opgeslagen in een float en wordt telkens opgeteld bij zichzelf. De integere waarde die er telkens het dichtste boven ligt, bepaalt de overeenkomende index van de pixel waarvoor een monster genomen wordt. De overblijvende pixels worden niet bemonsterd. Indien voorspellend bemonsteren geactiveerd is, wordt maximum 25% van het aantal bemonsteringsaanvragen erdoor in beslag genomen. De overige 75% worden gebruikt door het originele bemonsteringsproces. In figuur 3.1 is een voorspelling getoond, die gebruikt wordt om figuur 4.15 te voorspellen.

II Prefilter

De originele interpolatiefilter is een kleine gewogen 3×3 filter. Dat is zo gekozen omdat kleine filters randen beter behouden. Als de gebieden dicht bemonsterd zijn, zal een hogere kwaliteit behouden worden. Kleine gaten zullen dankzij deze filter opgevuld worden, maar grotere gaten zullen leeg blijven. Dergelijke gaten ontstaan bij grote bewegingen of wanneer nieuwe monsters te traag berekend worden. Wanneer geen monster ligt binnen een 3×3 omgeving van een pixel, zal het niet ingevuld worden. Met grote filters is het eenvoudiger om grote lege gebieden op te vullen. Aangezien dat bij de originele render cache ontbreekt, wordt een dergelijke filter toegevoegd. Die filter zal uniform zijn om berekeningskosten te drukken, want vermenigvuldigingen zijn bijvoorbeeld niet nodig. Tabel 3.1 toont de gebruikte 7×7 filter.

1	1	1	1	1	1	1
1	1	1	1	1	1	1
1	1	1	1	1	1	1
1	1	1	1	1	1	1
1	1	1	1	1	1	1
1	1	1	1	1	1	1
1	1	1	1	1	1	1

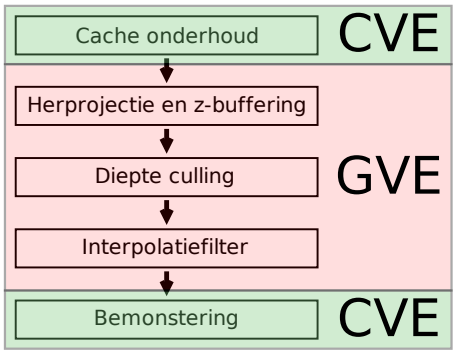


Tabel 3.1: (a) 7×7 prefilter (b) Voorspelling

De prefilter zal eerst werken op het resulterende beeld van de diepte culling en het resultaat wegschrijven naar het finale beeld. Daarna zal de kleinere interpolatiefilter van de originele render cache werken op het resulterende beeld van de diepte culling. Deze resultaten zullen die van de prefilter overschrijven. Enkel gebieden, waar de kleine interpolatiefilter niets genereert en dus gaten overblijven, blijven dan ingevuld met de resultaten van de prefilter. De dicht bemonsterde gebieden behouden zo een hogere kwaliteit en de prefilter werkt enkel waar de interpolatiefilter tekortschiet.

3.3.4 Beeldgeneratie op de GVE

De huidige grafische kaarten zijn goed geschikt om het beeldgeneratiegedeelte van de render cache uit te voeren. Z-buffering en filtering zijn namelijk belangrijke operaties op de grafische kaart. Deze operaties kunnen bijgevolg sterk versneld worden, waardoor er meer tijd beschikbaar zal zijn voor het visualiseren van nieuwe monsters. De andere delen van de render cache, zoals het onderhouden van de puntenwolk en de dithering, zijn te lastig om uit te voeren op de GVE en bevinden zich nog steeds op de CVE [95].



Figuur 3.15: Onderdelen die op de GVE gedaan worden staan in het roos gebied.

De rendercache-elementen worden met behulp van vertex buffer objecten (VBO) op de GVE opgeslagen in geheugen met hoge performantie [3]. Daarna projecteren vertex- en fragmentprogramma's de VBO-elementen op een textuur van een framebuffer object (FBO) [4]. Ondertussen schrijven ze ook dieptes weg naar een dieptetextuur van het FBO. De beeld-ID's schrijven ze eveneens tegelijkertijd weg naar een textuur van het FBO. Dat is mogelijk omdat de huidige grafische kaarten multiple render targets (MRT) ondersteunen. Htet fragmentprogramma schrijft dan tegelijkertijd naar de twee doeltexturen van de FBO, die op dat moment gebonden zijn. De dieptetextuur wordt automatisch gevuld, aangezien er een dieptecomponent gegenereerd is in het FBO. Na het vullen van deze FBO, passen fragmentprogramma's de diepte culling filter en de interpolatiefilter toe. Het uiteindelijk bekomen beeld wordt tenslotte getoond. Dankzij het gebruik van FBO's is er een minimale tijd vereist voor het doorgeven van informatie tussen verschillende stappen.

De volgende subsecties leggen de verschillende onderdelen verder uit. Subsectie I bespreekt de manier waarop monsters uit de rendercache terechtkomen op

de GVE. Subsectie II handelt over de herprojectie van deze monsters op het beeldvlak. Uiteindelijk zet subsectie III uiteen hoe dat beeld verder gefilterd wordt en uiteindelijk terechtkomt op het scherm.

I Puntenwolk

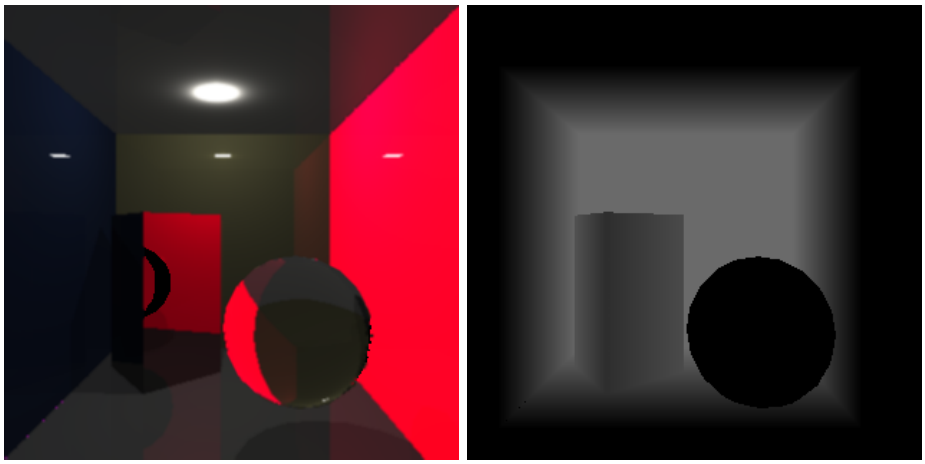
In de rendercache op de CVE bevinden zich elementen, die elk een eigen kleur, 3D-locatie en beeld-ID bezitten. De kleur en 3D-locatie zullen dienen om de positie en kleur van het element op het beeld te bepalen. Het beeld-ID is de beeldpositie waarop het element de laatste keer herprojecteerde. Dat is nodig om na de herprojectie te bepalen op welke pixel elk element terechtgekomen is. Aan de hand van de leeftijd van dat element bepaalt de render cache de overeenkomende prioriteit. Ook kan het zo het cache-ID van die pixel juist toekennen. De puntenwolk op de CVE blijft behouden voor het beheren van de cache.

Voor de drie elementeigenschappen maakt de applicatie nu vertex buffer objecten aan, zodat een tweede kopie van de puntenwolk verkregen wordt op de GVE voor snelle herprojectie. Dat gebeurt met de OpenGL-functies glGenBuffers, glBindBuffer en glBufferData. De drie gegenereerde buffers krijgen dezelfde grootte dan die van de rendercache. Vooraleer elk nieuw beeld gecreëerd wordt, zal de applicatie deze vertex buffer objecten vernieuwen met de nieuw gevisualiseerde monsters. Het zal deze elementen eerst toevoegen aan de puntenwolk op de CVE. Daarna zal het de VBO's met behulp van de OpenGL-functie glMapBuffer write-only mappen in het geheugen van de client. De drie benodigde eigenschappen van alle elementen van de rendercache op de CVE schrijft het dan in de overeenkomende VBO's. Aangezien deze elementen zich eender waar en sterk verspreid kunnen bevinden in de puntenwolk, is het niet efficiënt om bijvoorbeeld met glBufferSubData een groot verspreid gedeelte van de VBO te vernieuwen. Tenslotte haalt de applicatie de VBO's terug uit het geheugen van de client met de OpenGL-functie glUnmapBuffer.

II Herprojectie

Vooraleer de herprojectie start, bindt de applicatie een FBO met twee kleurenbuffers en *één* dieptebuffer. Dan laadt het de huidige inverse cameramatrix met de OpenGL-functie glLoadMatrixf. Vervolgens herprojecteert het de elementen van de VBO's. Dat gebeurt door hen eerst te binden met de OpenGL-functie glBindBuffer. Voor de VBO, die de ID's bevat, moet het tevens een vertex attribuut array definieren met de functie glVertexAttribPointer.

Daarna maakt de applicatie de arrays van de VBO's functioneel via glEnableClientState en de ID's-VBO via glEnableVertexAttribArray. Wanneer deze operaties gebeurd zijn, bindt het de herprojectie-shader. Deze bevat enkel extra code om de ID's weg te schrijven naar de tweede textuur van de FBO dankzij MRT. De functie glDrawArrays zorgt dan voor de visualisatie van de VBO's met behulp van de gebonden shader. De applicatie ontbindt achteraf de shader. Ook maakt het de VBO's onbruikbaar via glDisableClientState en glDisableVertexAttribArray. Door de functie glReadPixels, leest het de bekomen textuur van ID's terug naar een array op de CVE. Tenslotte ontbindt het ook de FBO en kan de filtering beginnen.

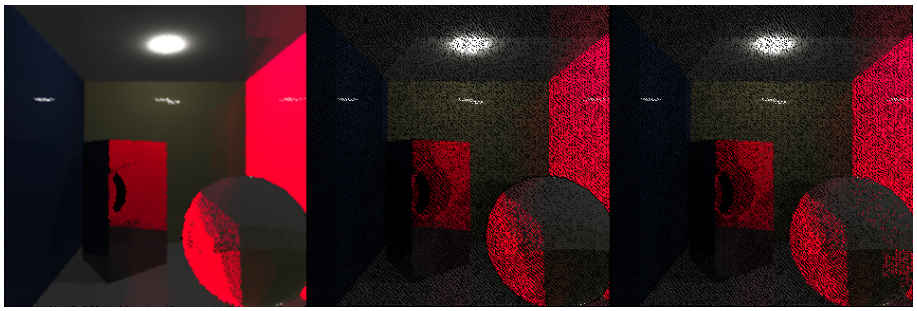


Figuur 3.16: Een voorbeeld diepte textuur (rechts) voor de scene (links) vanuit een bepaald camerastandpunt.

III Filtering

In deze stap zal de applicatie twee filters uitvoeren, namelijk de diepte culling filter en de interpolatiefilter. Dat gebeurt in twee stappen. Bijgevolg moet de applicatie het resultaat van de eerste filter schrijven naar een tweede FBO.

De eerste stap begint met het binden van de tweede FBO. Bij deze FBO bindt de applicatie slechts *één* kleurenbuffer, aangezien er enkel kleuren weggeschreven moeten worden. Tevens bindt het de shader, die zal zorgen voor de diepte culling. Deze bevat gelijkaardige code als de code van de overeenkomende functie in de software versie van de render cache. Aan deze shader geeft het het bekomen resultaat van de herprojectie mee in de vorm van twee texturen. De eerste bevat de kleurwaarden en de tweede de dieptewaarden. Vervolgens creëert de applicatie een vierhoek, waarop het de textuur toont. Daarna ontbindt het de shader en de FBO.



Figuur 3.17: Tijdens een snelle beweging: (a) Finaal beeld met ook interpolatiefilter toegepast, (b) Diepte culling filter op herprojectie toegepast (c) Enkel herprojectie

De tweede stap wordt rechtstreeks uitgevoerd op het uiteindelijke beeld. De

applicatie moet dus geen FBO binden, maar wel een shader. Die shader bevat gelijkaardige code als de code van de interpolatiefilter in de software versie van de render cache. Aan deze shader geeft de applicatie het resultaat van de vorige filter mee in de vorm van een textuur. Die textuur bevindt zich in de tweede FBO en wordt hier gebonden. Daarna creëert de applicatie een vierhoek, waarop deze textuur getoond wordt. Tenslotte ontbindt het de shader en is de generatie van het huidige beeld ten einde.

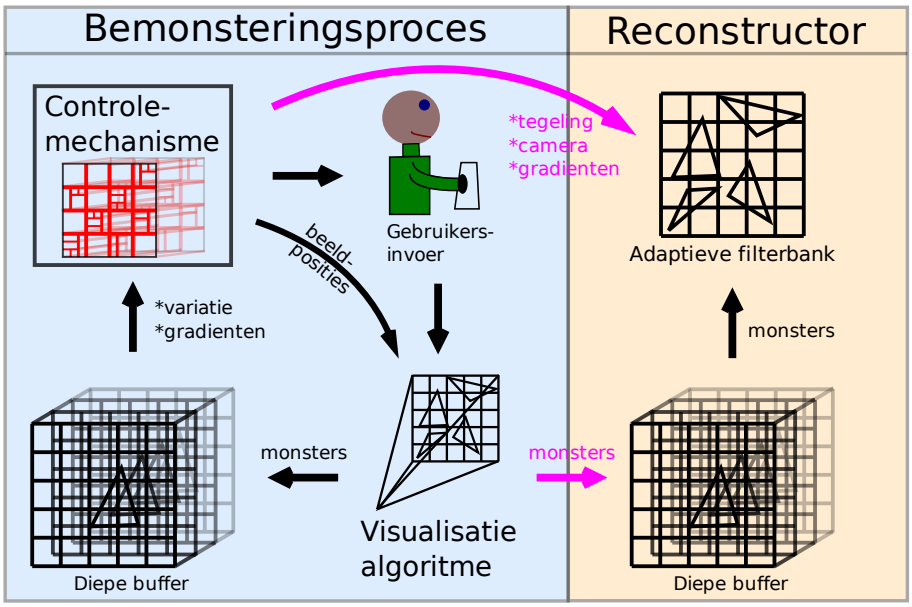
3.4 Adaptive Frameless Rendering

Deze techniek is ontworpen door Dayal et al. [32]. Het maakt gebruik van gelijkaardige concepten als de render cache techniek, zoals adaptieve bemonstering, herprojectie, reconstructie en frameless rendering. Maar het verschilt op een aantal gebieden en introduceert nieuwe ideeën, zoals toegelicht is in VII. Belangrijk is het feit dat het zich met een zeer fijne granulariteit aanpast aan kleurveranderingen in ruimte en tijd.

De eerste sectie 3.4.1 schetst in het kort de verschillende onderdelen van het systeem. De tweede sectie 3.4.2 zal de implementatie van het bemonsteringsproces beschrijven en de derde sectie 3.4.3 beschrijft de implementatie van de reconstructor.

3.4.1 Systeem overzicht

De techniek bevat twee hoofdcomponenten, namelijk het bemonsteringsproces en de reconstructor. Die worden asynchroon van elkaar uitgevoerd, elk in hun eigen thread.



Figuur 3.18: Onderdelen van adaptive frameless rendering.

Het bemonsteringsproces bevat een controlesysteem, een visualisatie algoritme en een diepe buffer. Het controlesysteem zal telkens aan de hand van de reeds opgeslagen monsters in de diepe buffer bepalen waar het visualisatie algoritme vervolgens zal moeten bemonsteren. Daarbij gebruikt het een beeldruimtetegeling die zich continu aanpast aan kleurvariaties. Die monsters worden vervolgens opgeslagen in de diepe buffer van het bemonsteringsproces en in die van de reconstructor.

De reconstructor bestaat uit een adaptieve filterbank en een aparte diepe buffer. De applicatie voert het volledig uit op de GVE, behalve het onderhoud van de diepe buffer. De reconstructor zal periodiek door een timer verwittigd worden dat het een nieuw beeld moet genereren. Het zal daarvoor monsters uit zijn diepe buffer halen en die op de juiste posities op het scherm herprojecteren. De adaptieve filterbank zal tijdens dat proces filters uitvoeren om de kwaliteit van het beeld te verhogen. Daarbij maakt het in verschillende beeldregio's gebruik van de corresponderende lokale kleurgradiënten en de lokale dichtheid van monsters om lokaal adaptieve reconstructie te bieden.

Beide hoofdcomponenten bezitten elk hun eigen diepe buffer om gegevensoverdracht en synchronisatievereisten tussen beide threads te beperken. De diepe buffer van de reconstructor wordt gebruikt om reeds gevisualiseerde monsters te hergebruiken in volgende beelden, terwijl de diepe buffer van het bemonsteringsproces informatie levert om nieuwe monsters naar belangrijke gebieden te gidsen. Deze levert ook informatie aan de reconstructor.

Herprojectie zorgt ervoor dat de monsters telkens aangepast worden aan de huidige cameraparameters. Dat levert nieuwe informatie aan het bemonsteringsproces, zodat het zich nog beter kan concentreren op belangrijke posities. Bij de beeldreconstructie plaatst het de monsters op de juiste beeldposities om de huidige cameratoestand te reflecteren.

Het algoritme kan zich in beide hoofdcomponenten aanpassen aan spatiale en temporele kleurvariaties. In het bemonsteringsproces wordt de selectie van nieuwe monsters in de tijd en in de ruimte bepaald via het opleggen van een kansverdeling aan de pixels in het beeld. In de reconstructor wordt er aangepast aan de variaties door filters gebruik te laten maken van kleurgradiënten in ruimte en tijd. Het bemonsteringsproces zal daarom, telkens een nieuw beeld gegenereerd moet worden, de huidige cameraparameters en tegelinginformatie doorsturen naar de reconstructor. De tegelinginformatie bestaat uit de volgende elementen:

- beeldcoördinaten van elke tegel
- gemiddelde spatiale gradiënten in de balk van de tegel
- gemiddelde temporele gradiënten in de balk van de tegel

3.4.2 Bemonsteringsproces

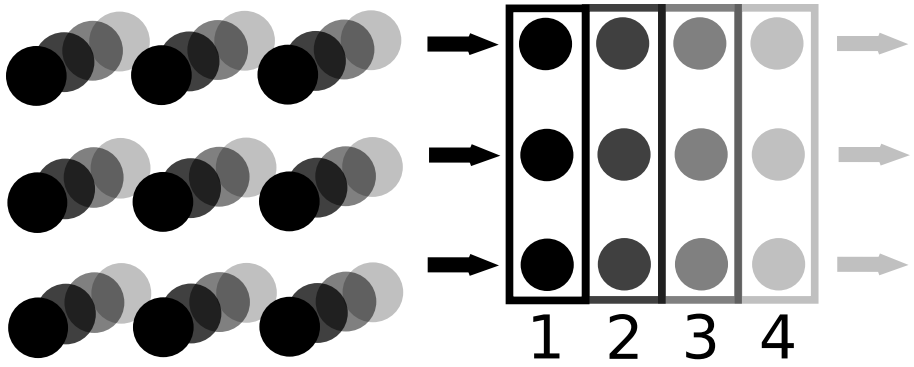
Het bemonsteringsproces moet het visualisatie algoritme continu leiden naar de gebieden in het beeld die het meeste nood hebben aan herbemonstering. Adaptive frameless rendering concentreert op gebieden waar veel kleurveranderingen

gebeuren in de tijd en/of in de ruimte. Kleurveranderingen in de ruimte duiden onder andere op randen van objecten, terwijl kleurveranderingen in de tijd duiden op bewegingen van objecten. Dat zijn visueel belangrijke aanknopingspunten, die bijgevolg frequenter nieuwe monsters zullen ontvangen.

Deze techniek bevat een aangepaste buffer, die besproken is in subsectie I. Om belangrijke gebieden met kleurvariatie te ontdekken, wordt continu een beeldruimtetegeling onderhouden. Subsectie II bespreekt dat. Aangezien opeenvolgende beelden en verschillende gebieden in dat beeld niet altijd even snel veranderen in de tijd, zal de techniek het aantal tegels geregeld aanpassen om nuttig te blijven. Wanneer snelle, grote veranderingen optreden, moet er sneller en ruw in de tijd herbemonsterd worden. Het crosshairsysteem, dat aan bod komt in subsectie III, zal daarbij gebruikt worden om het aantal tegels te bepalen. Herprojectie wordt toegepast in dezelfde gebieden waar nieuwe monsters genomen worden om de inhoud van de diepe buffer gericht recenter te houden en occlusies te ontdekken. Subsectie IV spit dit dieper uit. Tenslotte zet subsectie V de juiste volgorde van operaties uiteen.

I Diepe buffer

Er is gekozen voor een 3D-buffervolume omdat monsters verzameld worden over tijd en ruimte en omdat verschillende gebieden van het beeld verschillende dichtheden van monsters kunnen bevatten. Gebieden, waar veel veranderingen voorvallen, zullen frequenter nieuwe monsters ontvangen, terwijl de oudere monsters op die plaatsen in de buffer moeten verdwijnen. Op die manier worden de juiste, overbodige oude monsters sneller verwijderd. Ze zullen niet de monsters van andere gebieden, die misschien veel trager vernieuwd worden, doen verdwijnen. Zodoende kunnen monsters in dergelijke gebieden langer nuttig bijdragen aan het beeld.



Figuur 3.19: Diepe buffer: (a) Vooraanzicht (b) Zij-aanzicht

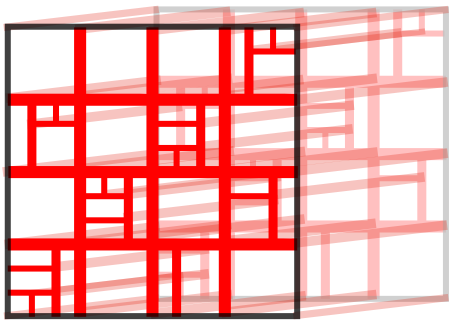
De buffer bezit voor elke pixel in het beeld een wachtrij. Dat is een FIFO (first in first out) datastructuur. Die heeft een bepaalde lengte en in deze implementatie wordt, net zoals in de paper, vier als lengte genomen. Bijgevolg bevat de buffer $breedte \times hoogte \times diepte$ posities, die opgevuld kunnen worden met crosshairs. Subsectie III verklaart de reden waarom crosshairs opgeslagen

worden door het bemonsteringsproces in plaats van individuele monsters. Wanneer een bepaalde wachtrij een nieuwe crosshair ontvangt, wordt deze achteraan toegevoegd (vooraan dus in de diepe buffer).

In deze implementatie is de diepe buffer geïmplementeerd als vier arrays van crosshairobjecten, die elk de grootte van het beeld hebben. De applicatie voegt een nieuwe crosshair toe in de eerste array op de corresponderende (x, y)-positie, nadat de reeds aanwezige monsters op die pixelpositie naar een volgende array op de corresponderende (x, y)-positie verplaatst zijn. Indien een crosshair aanwezig was in de vierde array, zal die uit de diepe buffer verdwijnen. Telkens nieuwe monsters berekend zijn door het visualisatie algoritme, creëert de applicatie crosshairs die direct terechtkomen in de diepe buffer. Het stuurt de individuele monsters tevens direct naar de diepe buffer van de reconstructor.

II Tegeling in beeldruimte

Een beeldruimtetegeling zal continu kleurvariaties opvolgen om belangrijke gebieden te ontdekken en de bemonstering daarop te concentreren. Het algoritme zal daarbij het heel beeld opdelen in tegels, die van grootte kunnen verschillen om een bepaalde kansverdeling toe te kennen aan de pixels binnen die tegel. Het is dus een 2D-tegeling en toch volgt het kleurvariaties in drie dimensies, namelijk de x- en y-as van het beeld en de tijd. Samen met de temporele diepte van de buffer, stelt elke tegel bijgevolg een volumeblok (balk) voor. Het monster in de tegel met de grootste temporele diepte bepaalt de diepte van de volumeblok. Kleurvariaties en andere statistieken van tegels zullen berekend worden op alle monsters in die volumeblokken om rekening te houden met de inhoud van een dynamisch ruimte-tijdsvolume in plaats van met een enkel statisch beeld.

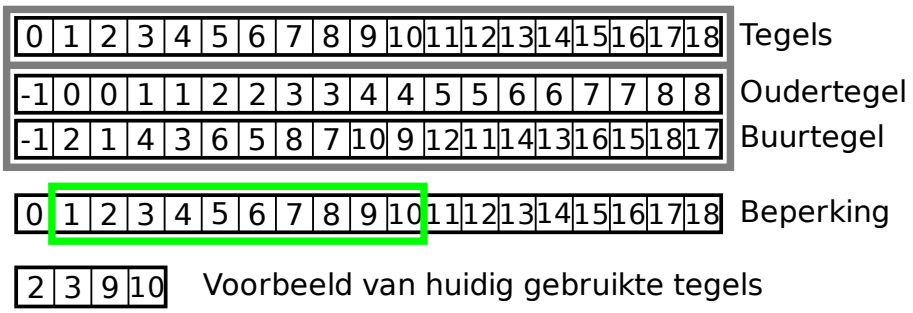


Figuur 3.20: Tegeling met volumeblokken.

Hieronder zal eerst de implementatie van de tegeling met behulp van een kd-boom uitgelegd worden II.1. Daarna volgt een verklaring van de door de tegeling opgelegde kansverdeling aan de pixels II.2. Tenslotte wordt besproken hoe de tegeling zich aanpast aan de hoeveelheid verandering in de scene II.3.

II.1 Kd-boom De tegeling is geïmplementeerd als een kd-boom. De applicatie houdt daarvoor twee arrays bij. De ene array bezit alle mogelijke tegels en de andere array bezit de indices van alle tegels die momenteel actief zijn. In de onderstaande afbeelding 3.21 duiden de rijen binnen de grijze omkadering de eerste array aan en de onderste rij duidt de tweede array aan. Beide arrays

zijn even groot, maar voor de tweede wordt een integere waarde bijgehouden die aanduidt waar de array stopt. De eerste array zal altijd hetzelfde blijven. Die wordt *één* keer op voorhand gevuld met alle mogelijke tegelobjecten. Er is namelijk een klasse tegel, die enkele eigenschappen zal opslaan zoals de index naar de oudertegel en de index naar de buurtegel. De tegeling moet zich lokaal kunnen aanpassen als antwoord op gebruikersinvoer en animaties. Dat gebeurt aan de hand van berekende statistieken zoals kleurvariaties. Daarom zal de tweede array vaak van inhoud veranderen. Als tegels gesplitst of samengevoegd worden, verdwijnen de indices van die tegels uit de tweede array. De nieuw gecreëerde tegels worden ingevoegd. Oude indices worden gewisseld met de achterste indices in de array, terwijl nieuwe indices achteraan toegevoegd worden. Daarna wordt de nieuwe grootte van de array bepaald door de bijgehouden integere waarde te veranderen.



Figuur 3.21: Klein voorbeeld van arrays voor de tegeling.

Aanpassingen gebeuren door tegels met veel kleurvariatie op te splitsen en tegels met weinig kleurvariatie samen te voegen. Dat is zo geïmplementeerd dat elke tegel slechts *één* geldige buur heeft, waarmee hij samengevoegd kan worden. Dat is de buur, waarvan hij de index bezit. Verschillende tegels kunnen dus een andere grootte bezitten. Omdat het aantal tegels vastgelegd is, zal het bemonsteringsproces het splitsen en samenvoegen van tegels kunnen uitvoeren tot aan een bepaald criterium voldaan is. Telkens de tegeling aangepast wordt, zal de tegel met de grootste variatie opgesplitst worden en de twee tegels met de kleinst opgetelde variatie samengevoegd worden totdat er ongeveer een gelijke hoeveelheid kleurvariatie overschiet in alle tegels. Het aantal tegels verandert op die manier niet. De kleurvariatie van een tegel wordt berekend met formule 3.1.

$$\nu_{tile} = \frac{\sum_{i=1}^{n}(L_i - L_m)^2}{n} \tag{3.1}$$

- L_i = luminantie van een sample,
- L_m = gemiddelde luminantie in de volumeblok = $\frac{\sum_{i=1}^{n} L_i}{n}$
- n = aantal aanwezige monsters n in het volumeblok

Eerst moet dus de gemiddelde luminantie L_m in het volumeblok berekend worden. Daarna wordt de variatie ν_{tile} van de tegel berekend. Oudere monsters krijgen in deze berekening een lager gewicht, zodat een sneller antwoord mogelijk is op veranderingen in de scene. De functie $e^{-3.47\alpha}$ bepaalt de gewichten,

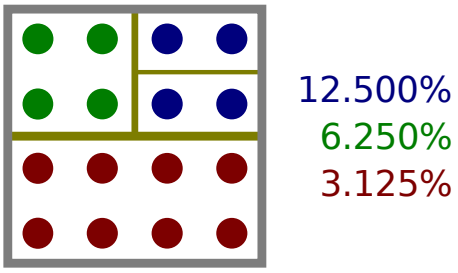
waarbij α de leeftijd van het monster is. Voor deze exponentiële functie is op voorhand een tabel berekend met kleine discrete stappen, zodat deze berekening niet telkens gedaan moet worden tijdens de hele uitvoering. De leeftijd wordt meegegeven als integere index en geeft dus in constante tijd de overeenkomende waarde terug van de functie.

Voor de aanpassing van de kd-boom moet telkens efficiënt de tegel met de grootste kleurvariatie bepaald kunnen worden en de twee tegels met de kleinste kleurvariatie. Er is dus een sortering nodig op de verzameling van tegels volgens de grootte van de kleurvariatie. Aangezien er tussen opeenvolgende aanpassingen van de tegeling slechts een fractie van de tegels van kleurvariatie kunnen veranderen, zou het niet efficiënt zijn om telkens alle tegels te sorteren. Verder kunnen de kleurvariaties van tegels veranderen tussenin opeenvolgende aanpassingen van de kd-boom. Aangezien de sortering gebeurt volgens de kleurvariatie, is het vereist dat elementen efficiënt gevonden, verwijderd en ingevoegd kunnen worden in de datastructuur. Daarom worden twee sets bijgehouden. Sets hebben een gemiddelde logaritmische complexiteit voor het vinden (en dan verwijderen) en invoegen van elementen. Tegels, waarbij de kleurvariatie veranderd is, worden bijgevolg verwijderd uit de set en daarna met hun nieuwe variatie terug gesorteerd toegevoegd. Dit vereist slechts een fractie van de berekeningen ten opzichte van het telkens opnieuw sorteren van alle tegels. De eerste set bevat een lijst van alle tegels samen met hun berekende kleurvariaties en de tweede set bevat een lijst van alle tegels die samengevoegd kunnen worden en hun opgetelde kleurvariatie. Tijdens het splitsen en samenvoegen kan zo efficiënt de volgende te splitsen en samen te voegen tegels gevonden worden.

Wanneer de scene en camera statisch blijven, zal de kleurvariatie vrij gelijkaardig verdeeld blijven doorheen de tijd. Wanneer veranderingen gebeuren, zal het controlemechanisme met behulp van nieuw berekende monsters de corresponderende gebieden ontdekken en zal de tegeling aangepast worden. Om ervoor te zorgen dat de tegeling toch redelijk verdeeld blijft over het beeld, wordt de kd-boom in de diepte vooraan en achteraan beperkt, zoals getoond wordt in afbeelding 3.21. Op die manier ontstaat een uniform rooster, waarbinnen kleine kd-bomen zich kunnen opsplitsen tot een bepaalde diepte. Die diepte is zo ingesteld dat de kleinste tegel een negentigtal pixels omvat. De maximum grootte van een tegel is zo ingesteld dat er zes verschillende niveau's van tegelgroottes mogelijk zijn. De array van huidige tegels zal nooit tegelijkertijd kindertegels bevatten, waarvan een ouder reeds aanwezig is in die array. Elke pixel op het beeld zal slechts tot *één* tegel tegelijkertijd kunnen behoren van de huidig actieve tegels in die array.

II.2 Kansverdeling De tegeling onderhoudt nu de controle over de kansen op het bemonsteren van elke pixel. Elke tegel heeft een gelijke kans op bemonstering, omdat een uniforme kansverdeling toegepast wordt bij het random kiezen van een tegel. Ook voor het kiezen van een pixel binnen een dergelijke tegel wordt een uniforme kansverdeling toegepast. Aangezien tegels van grootte kunnen verschillen, hebben pixels in kleinere tegels een grotere kans gekozen te worden dan pixels in grotere tegels. Het is dan mogelijk om grotere kansen toe te kennen aan belangrijkere gebieden, door hen met meerdere kleine tegels te

bedekken. De minder belangrijke gebieden krijgen een kleinere kans om behandeld te worden, door er grotere tegels over te plaatsen. Daarom zullen kleinere tegels geplaatst worden over belangrijke gebieden zoals bijvoorbeeld objectranden en objectbewegingen. Hier zal doorheen de tijd vaker bemonsterd worden, zodat de gemiddelde leeftijd van de monsters hier kleiner wordt. De statische gebieden zullen veel minder vaak bemonsterd worden, zodat daar de gemiddelde leeftijd van de monsters zal toenemen. De oude monsters zullen er langer recent en dus nuttig blijven. Dat nut kan nu langer uitgebuit worden.



Figuur 3.22: Voorbeeld van de kansverdeling bij een mogelijke tegeling.

Aangezien alle gebieden op deze manier een kans hebben op herbemonstering, ook al is die op sommige plaatsen klein, kunnen nieuw opduikende bewegingen ook tijdig ontdekt worden terwijl er toch een sterkere concentratie is op spatiaal en temporeel detail. Bij het sequentieel afwerken van de belangrijkste gebieden, worden dergelijke nieuwe bewegingen pas later ontdekt en wordt spatiaal detail overgeslagen.

II.3 Aantal tegels aanpassen De aanpassing aan temporele kleurvariaties gebeurt ook door het aantal tegels te veranderen. Als er in de tijd veel veranderingen voorvallen, is het moeilijker om de spatiale kleurverdeling op te volgen. Daarom kunnen er beter minder tegels gebruikt worden. Zo wordt er minder geconcentreerd op spatiale verschillen en zal er in de tijd sneller opnieuw een ruwer gebied bemonsterd worden, waardoor de tijd beter voorgesteld wordt.

Indien meer tegels gebruikt worden, zal er in de tijd minder snel opnieuw bemonsterd worden in eenzelfde ruw gebied en kan het spatiaal detail sterker bemonsterd worden. Kleinere tegels zullen ook geplaatst worden op gebieden met beweging. Er zal telkens verzekerd worden dat de kleurverandering over ruimte en tijd ongeveer gelijk is in alle blokken via formule 3.2.

$$\frac{dC}{ds}S = i\frac{dC}{dt}T \tag{3.2}$$

- $\frac{dC}{ds}$ = gemiddelde van de spatiale kleurgradiënten over het hele beeld
- $\frac{dC}{dt}$ = gemiddelde van de temporele kleurgradiënten over het hele beeld
- S = gemiddelde breedte van de tegels
- T = gemiddelde leeftijd van de monsters in elke tegel
- i = een constante = relatieve belangrijkheid van temporele en spatiale kleurverandering

Het gewenst aantal tegels S wordt berekend door vergelijking 3.2 te herformuleren 3.3.

$$S = \frac{i\frac{dC}{dt}T}{\frac{dC}{ds}} \tag{3.3}$$

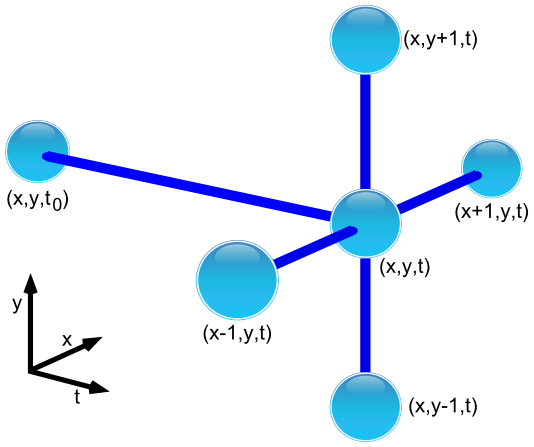
In het kort samengevat is concentratie op spatiale en temporele randen belangrijk in statische scenes en in dynamische scenes met weinig veranderingen. De lokale dichtheid van monsters zal dan over ruimte en tijd variëren, waarbij zich in de nabijheid van randen een grotere dichtheid van monsters zal bevinden. In dynamische scenes met veel veranderingen is het nuttiger overal op het beeld nieuwe monsters te berekenen om een gelijkmatige vernieuwing op het beeld waar te nemen. Dankzij het feit dat elk nieuw monster recenter is dan een vorig genomen monster, kan de bemonstering beter aangepast worden aan de tijd dan wanneer regelmatige intervallen genomen worden.

III Crosshairs

De diepe buffer van het bemonsteringsproces slaat crosshairs op in plaats van individuele monsters. Deze crosshairs dienen om kleurgradiënten te schatten over tijd en ruimte. Telkens wanneer monsters berekend worden voor een bepaalde pixel, zullen er ineens meerdere monsters berekend worden om een crosshair op te vullen. Een dergelijke crosshair bestaat uit zes monsters. De positie waar het monster genomen wordt, is het centrummonster van de crosshair (x, y, t). Op deze plaats is vroeger in de tijd reeds een monster (x, y, t_0) genomen. Die heeft de crosshair toen gestart. De overige vier monsters worden op hetzelfde moment berekend dan dat het huidige centrummonster berekend wordt. Dat zijn de vier directe buren van het centrummonster.

- $(x+1, y, t)$
- $(x-1, y, t)$
- $(x, y+1, t)$
- $(x, y-1, t)$

Tabel 3.2: 4 directe buren



Tabel 3.3: Crosshair mechanisme

Die directe buren bevinden zich op een afstand van *één* pixel breedte ten opzichte van het centrummonster en zijn op hetzelfde ogenblik met dezelfde camera- en sceneparameters gevisualiseerd als het centrummonster. De horizontale spatiale kleurgradiënten worden berekend met behulp van formule 3.4.

De verticale spatiale kleurgradiënten worden berekend met formule 3.5.

$$\frac{|(L_{x,y,t} - L_{x+1,y,t}) - (L_{x,y,t} - L_{x-1,y,t})|}{2} \tag{3.4}$$

$$\frac{|(L_{x,y,t} - L_{x,y+1,t}) - (L_{x,y,t} - L_{x,y-1,t})|)}{2} \tag{3.5}$$

De temporele kleurgradiënt wordt berekend aan de hand van het centrummonster (x, y, t) en het vroeger gemaakte centrummonster (x, y, t_0) met behulp van formule 3.6.

$$\frac{|L_{x,y,t} - L_{x,y,t_0}|}{t - t_0} \tag{3.6}$$

Bij het berekenen van de gemiddelde kleurgradiënten in een tegel, worden gewichten toegekend aan elke crosshair, zodat oudere crosshairs minder bijdragen aan het gemiddelde. Ook hier wordt de functie $e^{-3.47\alpha}$ gebruikt om het gewicht te leveren.

IV Herprojectie

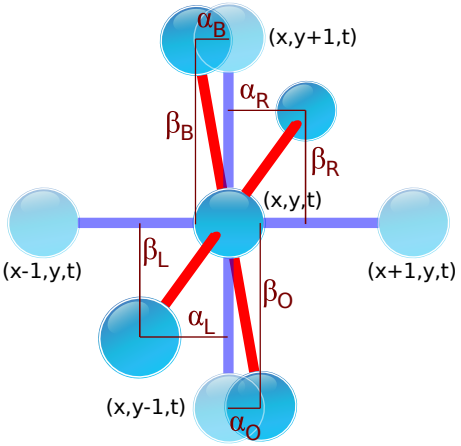
Telkens nadat de vijf monsters van een nieuwe crosshair berekend zijn en de corresponderende tegelstatistieken herberekend zijn, herprojecteert het bemonsteringsproces random vijf crosshairs, die zich bevinden in dezelfde tegel, naar posities die het huidig tijdstip representeren. Zodoende wordt herprojectie op dezelfde manier dan het bemonsteringsproces gericht naar de belangrijke regio's. Herprojectie helpt bij het vinden van dergelijke belangrijke gebieden. De geherprojecteerde crosshairs worden achteraan in de diepe buffer (vooraan in de wachtrij dus) weggehaald op de vroegere positie en vooraan ingevoegd in de diepe buffer op de nieuwe positie. De nieuwe positie wordt bepaald door herprojectie van het centrummonster volgens de nieuwe cameraparameters.

De diepe buffer zal op deze manier recenter gehouden worden, dan wanneer enkel herbemonstering gebruikt wordt om de diepe buffer te vernieuwen, aangezien herprojectie sneller berekend kan worden dan herbemonstering. Na herprojectie kan de crosshair in een andere tegel terechtkomen. Dan is het dus nodig om de statistieken van bron- en doeltegel aan te passen. De spatiale gradiënten van de crosshair worden herberekend met de nieuwe spatiale locaties van de monsters in de crosshair. Het gemiddeld horizontale luminantieverschil wordt berekend met formule 3.7 en het gemiddeld verticale luminantieverschil met formule 3.8. De temporele gradiënten van de crosshair worden herberekend door het absolute verschil te nemen van het geherprojecteerde centrummonster en het nieuwste monster op die pixel in de wachtrij. Het resultaat wordt gedeeld door de leeftijd van het nieuwste monster.

$$\frac{\frac{\alpha_L|lum_C - lum_L|}{\alpha_L+\beta_L} + \frac{\alpha_R|lum_C - lum_R|}{\alpha_R+\beta_R} + \frac{\alpha_B|lum_C - lum_B|}{\alpha_B+\beta_B} + \frac{\alpha_O|lum_C - lum_O|}{\alpha_O+\beta_O}}{\frac{\alpha_L}{\alpha_L+\beta_L} + \frac{\alpha_R}{\alpha_R+\beta_R} + \frac{\alpha_B}{\alpha_B+\beta_B} + \frac{\alpha_O}{\alpha_O+\beta_O}} \tag{3.7}$$

$$\frac{\frac{\beta_L|lum_C - lum_L|}{\alpha_L+\beta_L} + \frac{\beta_R|lum_C - lum_R|}{\alpha_R+\beta_R} + \frac{\beta_B|lum_C - lum_B|}{\alpha_B+\beta_B} + \frac{\beta_O|lum_C - lum_O|}{\alpha_O+\beta_O}}{\frac{\beta_L}{\alpha_L+\beta_L} + \frac{\beta_R}{\alpha_R+\beta_R} + \frac{\beta_B}{\alpha_B+\beta_B} + \frac{\beta_O}{\alpha_O+\beta_O}} \tag{3.8}$$

$$
\begin{aligned}
\alpha_L &= |scrX_L - scrX_C| \\
\alpha_R &= |scrX_R - scrX_C| \\
\alpha_B &= |scrX_B - scrX_C| \\
\alpha_O &= |scrX_O - scrX_C| \\
\beta_L &= |scrY_L - scrY_C| \\
\beta_R &= |scrY_R - scrY_C| \\
\beta_B &= |scrY_B - scrY_C| \\
\beta_O &= |scrY_O - scrY_C|
\end{aligned}
$$



Tabel 3.4: Geherprojecteerde crosshair

- lum_C, lum_L, lum_R, lum_B, lum_O: luminantie van respectievelijk het centrum-, linker-, rechter-, boven-, ondermonster van de crosshair
- $scrX_C$, $scrX_L$, $scrX_R$, $scrX_B$, $scrX_O$: geherprojecteerde x-waarde van respectievelijk het centrum-, linker-, rechter-, boven-, ondermonster van de crosshair
- $scrY_C$, $scrY_L$, $scrY_R$, $scrY_B$, $scrY_O$: geherprojecteerde y-waarde van respectievelijk het centrum-, linker-, rechter-, boven-, ondermonster van de crosshair

Herprojectie helpt het bemonsteringsproces bij zijn detectie van disocclusies, occlusies, camera-afhankelijke illuminatieveranderingen en camera-onafhankelijke beweging. Adaptive frameless rendering berekent aan de hand van de herprojectie nog een aantal statistieken die gecombineerd met de kleurvariatie de uiteindelijke foutwaarde zullen bepalen van de tegel (hier tegel x).

Door herprojectie zullen in gebieden, die disocclusies bevatten, monsters verplaatst worden naar andere gebieden. Zodoende kunnen deze gebieden sneller ontdekt worden en krijgen ze een hogere kans op bemonstering door formule 3.9.

$$
v_{tile} = 1 - \min\left(1, \frac{m\frac{\sum_{j=1}^{nrTiles}(sb_j - n_j)}{nrTiles}}{sb - n}\right) \tag{3.9}
$$

- $nrTiles$ = aantal tegels
- sb = aantal monsters, die tegel x kan bevatten
- n = aantal monsters, dat momenteel aanwezig zijn in tegel x
- sb_j = aantal monsters, die tegel j kan bevatten (s = tegelresolutie, $b = 4$)
- n_j = aantal monsters, dat momenteel aanwezig zijn in tegel j

- m = aantal keer dat het aantal lege monsterposities in tegel x groter moet zijn dan het gemiddeld aantal lege monsterposities in alle tegels om de tegeling te beïnvloeden

Occlusies worden bij herprojectie ontdekt door telkens na herprojectie een straal naar de camera te schieten. Dat gebeurt vanaf de 3D-lokatie van het geherprojecteerde monster en dient enkel om te testen of er een intersectie voorvalt tussen het monster en de camera. Indien dat het geval is, wordt het monster bedekt. Er zal direct een nieuwe crosshair gegenereerd worden die deze crosshair overschrijft in de diepe buffer. De gegenereerde monsters worden tevens gezonden naar de reconstructor. Bovendien zal in de tegels, waarin occlusie voorvalt (hier tegel x), een foutwaarde bepaald worden met formule 3.10.

$$O_{tile} = \frac{|O|}{sb} \tag{3.10}$$

- $|O|$ = aantal bedekte monsters in de blok van een tegel
- sb = aantal monsters, die tegel x kan bevatten

Uiteindelijk bepaalt formule 3.11 de fout van een tegel.

$$E_{tile} = s\left(\kappa\frac{\nu_{tile}}{\sum_j^{|tiles|}\nu_j} + \lambda\frac{\upsilon_{tile}}{\sum_j^{|tiles|}\upsilon_j} + (1-\kappa-\lambda)\frac{o_{tile}}{\sum_j^{|tiles|}o_j}\right) \tag{3.11}$$

- $|tiles|$ = aantal tegels
- ν_{tile}, ν_j = kleurvariatie van respectievelijk tegel x en j
- υ_{tile}, υ_j = disocclusiefout van respectievelijk tegel x en j
- o_{tile}, o_j = occlusiefout van respectievelijk tegel x en j
- s = tegel grootte
- $\kappa, \lambda, (\kappa+\lambda)$ liggen allen in het bereik $[0, 1]$
- $\nu_{tile}, \upsilon_{tile}, o_{tile}$ worden genormaliseerd door hen te delen door respectievelijk de totaal opgetelde ν, υ en o van alle tegels.

V Volgorde operaties

Het bemonsteringsproces initialiseert zijn eigen diepe buffers en die van de reconstructor door elke pixel in het beeld te visualiseren met dezelfde camera- en sceneparameters. Tevens initialiseert het de tegeling met tegels van gelijke grootte. Het bovengrensniveau wordt daarvoor genomen, dus het kleinst mogelijk aantal tegels. Voor alle mogelijke tegels neemt het proces *één* monster op de pixelpositie in het midden van de tegel. Dat slaat het op in de tegel. Zo is er voor elke tegel reeds een temporeel monster beschikbaar. Vanaf dan zal het bemonsteringsproces steeds opnieuw de operaties doen die hieronder beschreven worden.

Eerst vernieuwt het de camera- en sceneparameters naar het huidig moment t. Vervolgens kiest het random een tegel. Daarin zoekt het naar het laatst gevisualiseerde monster (x, y, t_0) dat een crosshair gestart heeft. Daarna worden

de overige vijf monsters ((x, y, t), $(x+1, y, t)$, $(x-1, y, t)$, $(x, y+1, t)$ en $(x, y-1, t)$) gegenereerd door het visualisatie algoritme. Die krijgen de huidige tijd t toegekend.

Als die vijf monsters gevisualiseerd zijn, berekent het proces de spatiale gradiënten en de temporele gradiënt. Vervolgens voegt het de crosshair als enkel element toe in de eerste array van zijn diepe buffer op positie (x, y). De vijf monsters voegt het elk apart toe in de diepe buffer van de reconstructor op hun overeenkomstige posities. De pixelpositie van het nieuwe monster zal behoren tot een bepaalde tegel. De kleurvariatie van die tegel past het aan met de formule 3.1. Samen met de andere formules 3.9 en 3.10 past het dan de algemene foutwaarde van de tegel 3.11 aan.

Per nieuw berekende crosshair herprojecteert het bemonsteringsproces een aantal crosshairs. Het zal dus binnen de huidige tegel random vijf crosshairs herprojecteren. Het herberekent de gradiënten van deze crosshairs. Ook schiet het proces voor alle vijf crosshairs een straal van het centrummonster naar de camera om te kijken of er geen object tussen ligt, dat het monster bedekt. Indien het bedekt wordt, zal het een nieuwe crosshair genereren op dezelde beeldpositie. Als temporeel monster neemt het dan het geherprojecteerde centrummonster. De andere vijf monsters visualiseert het direct en het berekent de gradiënten. Het stuurt de nieuwe monsters naar de diepe buffer van de reconstructor. De crosshair voegt het proces toe in zijn diepe buffer. Indien het centrummonster niet bedekt wordt, zal het de huidige crosshair behouden. De vijf crosshairs kunnen ofwel terug binnen de huidige tegel terechtkomen ofwel in andere tegels. Het past daaropvolgend de kleurvariaties van bron- en doeltegels aan.

Na de herprojectie kiest het bemonsteringsproces random een nieuwe pixelpositie (x', y') in de huidige tegel. Daar neemt het proces *één* monster (x', y', t) en slaat het op in de huidige tegel. Er is zo een crosshair gestart voor deze pixelpositie. Het stuurt het monster ineens door naar de diepe buffer van de reconstructor.

Wanneer dit proces zich honderd keer herhaald heeft, zal het controlemechanisme tegels splitsen en samenvoegen tot de kleurvariaties in de verschillende tegels weer ruwweg overeenkomen.

Indien de reconstructor ondertussen een beeld gaat tonen, stuurt het bemonsteringsproces de informatie van de tegeling en de nieuwe cameraparameters door naar de reconstructor.

3.4.3 Reconstructor

De reconstructor bestaat uit twee delen, namelijk de adaptieve filterbank en de diepe buffer. De diepe buffer en zijn voorstelling op de GVE staan beschreven in subsectie I. De GVE voert de adaptieve filterbank uit met behulp van FBO's en vertex- en fragmentprogramma's. Zodoende produceert het periodiek een gefilterd beeld op het scherm. Daarbij herprojecteert het telkens de monsters in zijn diepe buffer om de huidige cameratoestand voor te stellen. Subsectie II bespreekt dat. De reconstructie past zich ook aan de monsterdichtheid aan. Dat is uitgelegd in subsectie III. Tevens maakt het gebruik van de tegelinginformatie, dat het bemonsteringsproces herhaaldelijk zendt wanneer een nieuw beeld geconstrueerd wordt. Dankzij die informatie kunnen de filters zich lokaal aanpassen aan temporele en spatiale kleurvariaties. Subsectie IV bespreekt

dat verder. Tenslotte bespreekt subsectie V de volgorde van operaties en de implementatie op de GVE.

I Diepe buffer en GVE buffer

De diepe buffer van de reconstructor is op dezelfde wijze opgebouwd als die van het bemonsteringsproces. Het enige verschil is dat deze individuele monsters opslaat in plaats van crosshairs, aangezien crosshairs hier geen toegevoegde waarde hebben. Ze zouden enkel meer geheugen in beslag nemen. Een monster in de diepe buffer bevat een kleur, een 3D-positie en een leeftijd.

Uiteindelijk is de buffer vervangen door *één* wachtrij, zoals aangegeven is in de paper. Die houdt enkel de recentste monsters bij, namelijk vier keer zoveel monsters dan de grootte van de beeldresolutie. Dankzij deze wachtrij verdwijnen oudere monsters sneller en zo ontstaat een recenter beeld.

De applicatie maakt bij de start van het algoritme VBO's aan voor de kleuren, 3D-posities en leeftijden van de monsters. Elke keer de reconstructor een beeld gaat tonen, stuurt het de monsters in zijn diepe buffer door naar de VBO.

II Herprojectie

Alle monsters worden geherprojecteerd, zodat ze op de juiste nieuwe positie van het scherm terechtkomen volgens de huidige cameraparameters. Dankzij herprojectie dragen oudere monsters meer bij aan de kwaliteit van het beeld, dan wanneer het niet toegepast wordt. Meerdere kleurwaarden zullen dankzij herprojectie op de juiste positie weergegeven worden, en dat zelfs bij sterke camerabewegingen.

III Splats

Op bepaalde gebieden in het beeld zullen weinig monsters aanwezig zijn, zoals bijvoorbeeld bij disocclusies en aan de randen van het scherm bij camerarotaties. Daarom worden monsters op het beeldvlak geplaatst als een gevulde vierhoek met een bepaalde adaptieve grootte. Die grootte kan bepaald worden met gl_PointSize en wordt aangepast, gebaseerd op de bedekkingsmap van het vorige beeld. Monsters worden namelijk, tijdens het berekenen en wegschrijven van de kleuren van dat beeld naar een FBO-textuur, verzameld in een dergelijke bedekkingsmap. Dat is mogelijk dankzij multiple render target en het gebruik van een FBO. Het aantal monsters, dat op een bepaalde pixel terechtkomt, en hun gemiddelde splatgroottes worden opgeteld en opgeslagen in een tweede FBO-textuur.

Tijdens het splatten van de monsters op het nieuwe beeld, bindt de applicatie de bedekkingsmaptextuur van het vorige beeld. Op de juiste positie in die textuur, gebruikt het de aanwezige info om de gemiddelde splatgrootte op die pixelpositie te berekenen. Dat gemiddelde wordt vermenigvuldigd met vier wanneer de dichtheid van monsters te klein is op die pixelpositie. Dat gebeurt hier bij pixels, die door minder dan vier monsters beïnvloed worden. Zo kan de reconstructor snel een groter gebied vullen. Bij pixels, die door meer dan 32 monsters beïnvloed worden, wordt er vermenigvuldigd met 0.7, zodat de splatgrootte gradueel zal afnemen.

IV Adaptieve filters

De reconstructor voert lokaal adaptieve filters uit om de beeldkwaliteit te verhogen door onder andere de resulterende temporele incoherentie zoveel mogelijk weg te werken. Deze filters worden lokaal aangepast omdat verschillende gebieden van het beeld kunnen veranderen met verschillende snelheden. Die aanpassing gebeurt volgens lokale schattingen van de dichtheid van monsters en van spatiale en temporele kleurgradiënten. De lokale schatting van de dichtheid is het aantal monsters uit de bedekkingsmaptextuur. Het bemonsteringsproces zal, telkens de reconstructor een nieuw beeld genereert, de gemiddelde spatiale en temporele gradiënten berekenen en doorsturen. De adaptieve filterbank zal de grootte en vorm van de filters daarna corresponderend aanpassen. De eerste paragraaf IV.1 bespreekt de invloed van de kleurgradiënten en de tweede paragraaf IV.2 bespreekt de invloed van de dichtheid van monsters. De formules voor de filteromvang komen in de derde paragraaf IV.3 aan bod, terwijl de vierde paragraaf IV.4 de filterkernel bespreekt. Tenslotte worden enkele voorbeeldfilters getoond in de laatste paragraaf IV.5.

IV.1 Gradiënten Waar het beeld statisch is in de tijd, zijn de temporele kleurgradiënten klein. De temporele coherentie is groot, wat betekent dat oude monsters minder kans hebben om verouderde kleurwaarden te bezitten. De filter moet hiervoor temporeel breed zijn. Hierdoor worden aan oude monsters ook hoge gewichten gegeven in de filter. Zo worden temporeel gedetailleerde en temporeel geanti-aliaste beelden verkregen, wat voornamelijk bij kleine bewegingen veel nut heeft.

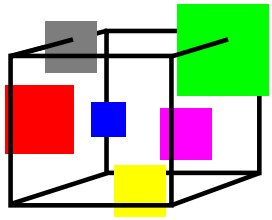
Waar de scene dynamisch is in de tijd, zijn de temporele kleurgradiënten groot. De temporele coherentie is dan laag en oude monsters bezitten een grotere kans om foute kleurwaarden te bevatten. De filter zal bijgevolg oudere monsters sterker mijden door hen een laag gewicht te geven. Er zullen op die manier sneller vernieuwingen van bewegingen getoond worden. Temporeel nauwe filters kunnen dit verwezenlijken. Ze bieden een lage latentie.

Waar de scene dynamisch is in de ruimte, zijn de spatiale kleurgradiënten groot, zoals bij een rand van een object. Daar wordt een spatiaal nauwe filter gebruikt, zodat buurmonsters geen grote invloed hebben. De randen zullen dan niet wazig zijn.

Waar de scene statisch is in de ruimte, zijn de spatiale kleurgradiënten klein. Op deze plaats zal een spatiaal brede filter werken, zodat gaten, die eventueel aanwezig zijn, gevuld worden.

IV.2 Dichtheid van monsters De dichtheid van monsters bepaalt het volume dat de filter zal innemen. Dus zal een spatiaal en temporeel nauwere filter werken in gebieden waar veel monsters aanwezig zijn. Het is dan niet nodig om extra monsters te gebruiken, aangezien er reeds een voldoende aantal monsters voorhanden is. Zo zal het resultaat ook minder wazig zijn. Als er weinig monsters aanwezig zijn in een gebied, zal het volume van de filter groot zijn om meer monsters te kunnen laten bijdragen aan dat gebied. Zo kunnen gaten opgevuld worden. Op een pixel moet namelijk geen monster aanwezig zijn met deze splattechniek. Er zullen grotere splats zijn in deze gebieden, zodat verder afgelegen monsters toch terecht kunnen komen op lege pixels en door de filter geïncorporeerd kunnen worden.

IV.3 Omvang van de filter De omvang van de filters volgt de x-, y- en t-dimensie en omsluit een ruimte-tijdsvolume dat een gelijke kleurvariatie tracht voor te stellen in elke dimensie met behulp van de zojuist besproken schattingen van de kleurgradiënten (G_x, G_y, G_t) en de totale volumebeperking V_s. De filter bestaat dus uit een lengte volgens de x-as of e_x, een lengte volgens de y-as of e_y en een lengte volgens de t-as of e_t. De formules hieronder bepalen de omvang volgens deze assen. Formule 3.12 impliceert dat een lage gradiënt in een bepaalde dimensie overeenkomt met een grotere omvang in die dimensie. Een grote gradiënt komt overeen met een kleinere omvang in die dimensie. Formule 3.13 impliceert dat de totale omvang gebonden is door de dichtheid van monsters. R_l is de lokale snelheid waarmee nieuwe monsters genomen worden en is uitgedrukt in monsters per pixel per seconde. De afbeelding IV.3 toont een pixelvolume, waarbij de verschillende vierhoeken splats zijn van verschillende monsters, die een bijdrage leveren aan deze pixel.



Tabel 3.5: Pixelvolume

$$e_x G_x = e_y G_y = e_t G_t \tag{3.12}$$

$$V_s = e_x e_y e_t = \frac{1}{R_l} \tag{3.13}$$

$$e_x = \sqrt[3]{\frac{V_s G_y G_t}{G_x^2}}, e_y = \sqrt[3]{\frac{V_s G_x G_t}{G_y^2}}, e_t = \sqrt[3]{\frac{V_s G_x G_y}{G_t^2}} \tag{3.14}$$

IV.4 Filterkernel Als kernel wordt een Gaussiaanse filter gebruikt. Formule 3.15 stelt dit voor. De constante c in formule 3.15 dient om het volume van de Gaussiaan *één* te maken, wat in dit geval overbodig is.

$$G(x, y, t) = c\, e^{-\frac{x^2+y^2+t^2}{2\sigma^2}} \tag{3.15}$$

Sigma is de standaarddeviatie en wordt hier voorgesteld door de inverse filteromvangen. Die bepaalt het gebied waarbinnen 95% van het gewicht zich bevindt. Bij een scherpe spatiale rand is deze standaarddeviatie kleiner dan *één*. Sigma is hier een vector, in plaats van een enkel getal. Zodoende zal er geen uniforme heuvel ontstaan, maar een heuvel die in elke dimensie verder, even ver of minder ver gaat.

$$machtE = \frac{x^2 + y^2 + t^2}{\sigma^2} \tag{3.16}$$

$$= (diff_x^2, diff_y^2, leeftijd^2) \cdot \left(\frac{1}{e_x^2}, \frac{1}{e_y^2}, \frac{1}{e_t^2}\right) \tag{3.17}$$

$$= \left(diff_x^2 + \frac{1}{e_x^2}\right)\left(diff_y^2 + \frac{1}{e_y^2}\right)\left(leeftijd^2 + \frac{1}{e_t^2}\right) \tag{3.18}$$

$$diff_x = m_x - p_x \tag{3.19}$$

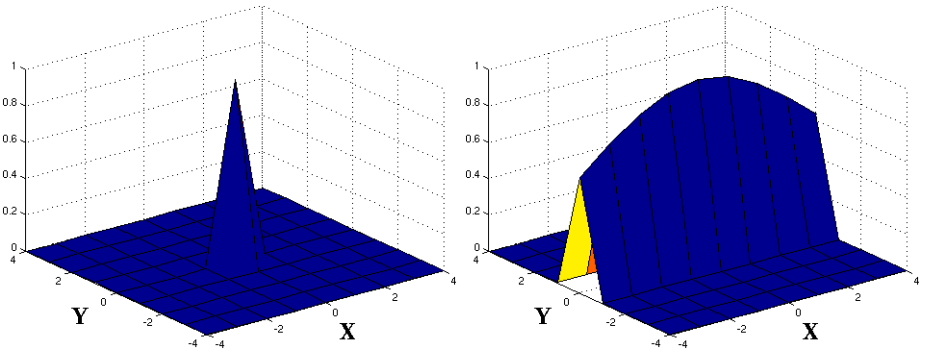
$$diff_y = m_y - p_y \tag{3.20}$$

De bijdrage van een monster, waarvan een fragment van de splat terechtkomt op de huidige pixel, wordt berekend door eerst de afstand ($diff_x$, $diff_y$) van het monster (m_x, m_y) tot het centrum van de pixel (p_x, p_y) te berekenen. De juiste positie binnen het pixelvolume is dan bepaald. Dat verschil vormt samen met de leeftijd van het monster een vector. Door het dotproduct te nemen van deze vector en sigma, wordt de macht bekomen. Die zal gedeeld worden door twee en maal -1 gedaan worden. De exponentiële functie wordt dan toegepast op het resultaat en zo wordt het gewicht voor het huidige fragment bekomen. De kleur van het monster wordt dan vermenigvuldigd met dat gewicht. De gewogen kleuren en de gewichten van alle fragmenten, die op deze pixel terechtkomen, worden opgeteld. Normalisatie op de pixel gebeurt dan achteraf door de totaal opgetelde kleuren te delen door het totaal opgeteld gewicht. Dat dient om de juiste helderheid te bereiken.

$$gewicht = exp(\frac{-machtE}{2}) \tag{3.21}$$

$$FinaleKleur = vec4(kleur * gewicht, gewicht) \tag{3.22}$$

IV.5 Enkele voorbeeldfilters Omdat het onmogelijk is om een 3D-volume filter te tonen, zullen de gewichten van de filter telkens getoond worden volgens twee assen.



Figuur 3.23: (a) x-,y-gradiënt groot (b) x-gradiënt groot, y-gradiënt klein

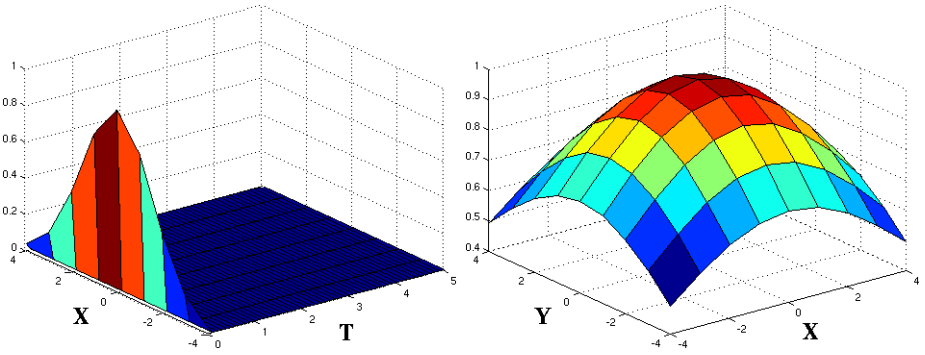
Figuur 3.23 toont filters, die de techniek gebruikt wanneer de dichtheid van

monsters groot is en de beeldregio statisch is. Een statische beeldregio bevat een lage temporele gradiënt.

Links toont het de filter, waarbij de x- en y-gradiënten groot zijn. Rechts toont het de filter, waarbij de x-gradiënt groot is en de y-gradiënt klein. Dit duidt aan hoe de filter verschilt bij gelijke spatiale gradiënten ten opzichte van verschillende spatiale gradiënten. Bij grote spatiale gradiënten zal de filter snel een laag gewicht geven.

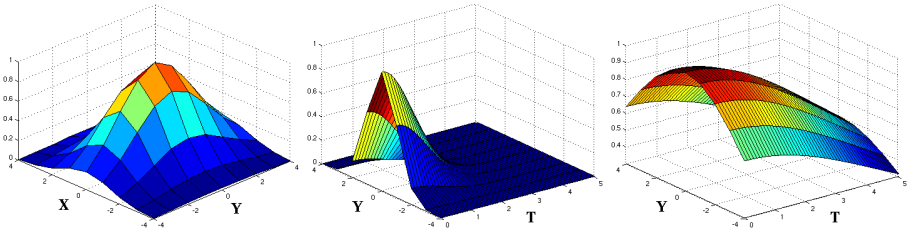
De linker afbeelding in figuur 3.24 toont een filter, die de techniek toepast wanneer de dichtheid van monsters groot is en de beeldregio dynamisch. Een dynamische regio betekent een grote temporele gradiënt. Het heeft een grote x-,y- en t-gradiënt. Oudere monsters zullen snel weggefilterd worden.

De rechter afbeelding in figuur 3.24 toont een filter, die gebruikt zal worden wanneer de dichtheid van monsters laag is en alle gradiënten groot. Dit duidt aan dat meer monsters genomen worden, om bijvoorbeeld gaten op te vullen.



Figuur 3.24: (a) Dynamische beeldregio (b) Lage monsterdichtheid

De laatste figuur hieronder toont nog enkele mogelijke situaties. De meest linkse filter valt voor in dynamische gebieden met een gemiddelde monsterdichtheid en grote spatiale gradiënten, waarbij de x-gradiënt de grootste is. De tweede filter wordt gebruikt bij een lage monsterdichtheid, een statische beeldregio en lage gradiënten. De derde filter wordt gebruikt in een statisch gebied met een groot aantal monsters, een grote x-gradiënt en lage y- en t-gradiënten.



V Volgorde operaties

Bij de initialisatie van de reconstructor maakt de applicatie VBO's aan om de eigenschappen van de monsters in op te slagen. Tevens maakt het een FBO

aan. In totaal zal die FBO vier texturen bevatten. De eerste textuur is degene waarop de gefilterde kleuren en gewichten zullen terechtkomen. De tweede en de derde textuur worden gebruikt voor de bedekkingsmap. Die zal de reconstructor per beeld moeten wisselen, omdat de resultaten van het tegelijkertijd schrijven en lezen van een FBO-textuur ongedefinieerd zijn. Tegelijkertijd lezen en schrijven is noodzakelijk, aangezien binnen dezelfde shader de kleuren via MRT weggeschreven worden naar de eerste gebonden textuur en de splatgrootte en aantal splats naar de tweede gebonden textuur. Tenslotte is er nog een vierde FBO-textuur nodig, waarop de reconstructor de gradiënten en de tegeling zal visualiseren.

Telkens de reconstructor een nieuw beeld moet creëren, slaat het de eigenschappen van de monsters op in de overeenkomende VBO's. Het bemonsteringsproces zal de reconstructor dan de huidige tegeling en gemiddelde gradiënten van elke tegel doorsturen. Daarna bindt de reconstructor een FBO en de vierde textuur van die FBO. De tegeling en gradiënten worden op die textuur gevisualiseerd. Elke tegel zal daarbij als kleur de overeenkomende gemiddelde gradiënten bevatten. Een kleur bevat dan in het R-kanaal de gemiddelde horizontale spatiale gradiënt, in het G-kanaal de gemiddelde verticale spatiale gradiënt en in het B-kanaal de gemiddelde temporele gradiënt.

Na deze operaties ontbindt de reconstructor de vierde textuur. De eerste textuur wordt gebonden, leeggemaakt en terug ontbonden. Daarna bindt de reconstructor de eerste textuur en de bedekkingsmaptextuur, die op die moment leeg is. Vervolgens maakt het de VBO's actief en bindt het een shader, die een vertex- en een fragmentprogramma bevat. Blending mode staat daarbij op additieve blending. De reconstructor geeft aan de shader de andere bedekkingsmaptextuur mee, die de bedekkingsgegevens bevat van het vorig beeld. Het geeft bovendien de vierde textuur met de gradiënten mee. Via glDrawArrays zal de visualisatie met behulp van de adaptieve filters starten. Het vertexprogramma projecteert de monsters op het scherm als gevulde vierhoeken. Dat kan met behulp van GL_POINTS. Het programma bepaalt daarbij de grootte van de splat, zoals uitgelegd is in subsectie III. Tevens berekent het hier de filteromvangen met behulp van de vierde FBO-textuur. Dankzij vertex texture fetch kan het vertexprogramma namelijk de lokale gradiënten voor een bepaalde pixel uit de textuur halen. De monsterdichtheid wordt gehaald uit de bedekkingsmaptextuur. De berekening gebeurt zoals vermeld is in subsectie IV.

Na het vertexprogramma zal het fragmentprogramma zijn werk doen. Het vertexprogramma geeft daarbij informatie door aan het fragmentprogramma, zoals de filteromvangen, de huidige splatgrootte en de kleur, het centrum en de leeftijd van het monster. Met behulp van die informatie zal het fragmentprogramma dan over alle pixels gaan, waarop zich punten bevinden. Het berekent eerst de afstand tussen het monster en het centrum van de pixel. Daarna berekent het de macht van de exponentiële functie, het gewicht en tenslotte de finale kleur van het monster, zoals uitgelegd is in subsectie IV.4. Dankzij MRT schrijft het fragmentprogramma de kleur weg naar de eerste FBO-textuur en de splatgrootte naar de tweede of derde FBO-textuur. De eerste FBO-textuur zal in de overeenkomende R-,G- en B-kanalen de kleur RGB-kleur bevatten en in het A-kanaal het gewicht. Dankzij additieve blending worden alle kleuren bij elkaar opgeteld, alle gewichten bij elkaar opgeteld en alle splatgroottes bij elkaar opgeteld van de monsters die op de corresponderende pixels terechtkomen.

Per monster wordt in het R-kanaal van de bedekkingsmap het aantal monsters opgeteld, dat terechtkomt op de corresponderende pixel. Na deze operaties ontbindt de applicatie de shader en worden de VBO's inactief gemaakt. Ook de FBO wordt ontbonden en de bedekkingsmap, waaruit juist gelezen is, wordt leeggemaakt.

Omdat alle kleuren en gewichten opgeteld zijn, is de juiste kleur nog niet bereikt. Deze laatste stap bindt een laatste fragmentshader en geeft daaraan de eerste FBO-textuur mee. Alle opgetelde kleuren worden gedeeld door de opgetelde gewichten en de juiste kleur zal dan getoond worden op het beeld.

Hoofdstuk 4

Resultaten

Dit hoofdstuk beschrijft de resultaten, die bekomen zijn met de verschillende implementaties. De eerste sectie 4.1 behandelt het systeem, de programmeertalen en andere parameters die invloed hebben op de implementaties. De tweede sectie 4.2 beschrijft de voornaamste problemen, die opdoken tijdens de programmatie. Daaropvolgend zal de derde sectie 4.3 de performantie en flessenhalzen tonen van de verschillende systemen en de laatste sectie bespreekt de bekomen visuele resultaten 4.4.

4.1 Systeemspecificatie

De implementaties zijn uitgevoerd op dezelfde computer. Tabel 4.1 geeft een overzicht van de gebruikte software en hardware. OpenRT legt een beperking op de mogelijke gebruiksmiddelen. Het vereist het gebruik van het besturingssyteem linux, van de compilers gcc of icc en het kan enkel op een 32-bitsysteem gebruikt worden.

Hardware	
Hoofdgeheugen	1GB
CVE	AMD Athlon 64 X2 Dual Core Processor 4000+
GVE	NVidia 8600GT, 540MHz kloksnelheid, 256MB geheugen
Software	
Besturingssyteem	Gentoo
C++	Gcc-4.1.1
Qt4	versie 4.3.1
OpenRT	NONCOMMERCIAL-1-4-R6-IA32-GCC34X-2006-04-24 SDK-NONCOMMERCIAL-Date-2005-08-17
OpenGL	versie 2.1.2, driver 173.14.09
Glew	versie 1.3.5
GLSL	versie 1.20

Tabel 4.1: Systeemspecificatie

4.2 Problemen

Tijdens de implementatie doken een aantal lastige problemen op. De geschreven shaders voor OpenRT konden op bepaalde posities geen gebruik maken van de eenvoudige operator + =, aangezien dat totaal andere resultaten opleverde dan gewoon + te gebruiken.

Verder leverde OpenRT een ongeïnitialiseerde variabele met RTState. Na een korte tijd dook dit probleem reeds op, maar het was niet direct duidelijk dat OpenRT de reden was. Telkens deze variabele problemen gaf, waren alle eerder geproduceerde versies van de implementaties ineens onbruikbaar. Via het programma Valgrind (versie 3.2.3) is dit probleem ontdekt. Met een eenvoudige memset was het probleem definitief opgelost.

De implementaties werden oorspronkelijk uitgevoerd op een andere GVE, een 6600GT. Maar deze haalde slechts een snelheid van 180 tot en met 400ms bij het teruglezen van een beeld van 256×256 pixels. Ook had de kaart last met vertex texture fetch. Met de nieuwere GVE waren er geen problemen. Die haalt een snelheid van 6ms bij het teruglezen van dat beeld.

Tenslotte staan er redelijk wat fouten en dubbelzinnigheden in de paper over adaptive frameless rendering [32]. Het is bijvoorbeeld niet helemaal duidelijk waar geherprojecteerde monsters terug ingevoegd moeten worden, aangezien ze op twee plaatsen zichzelf tegenspreken over hoe moet toegevoegd worden in de wachtrijen. Ook in de pseudocode van het bemonsteringsproces staan fouten in de variabelen en worden operaties herhaald, terwijl dat geen nut lijkt te hebben. De formule v_{tile} in de paper doet een sommatie over alle tegels van exact dezelfde operatie, die niets met die tegel te maken heeft. Die formule is waarschijnlijk fout.

4.3 Performantie

De volgende secties wijden uit over de timing- en profilingresultaten van de verschillende technieken. Voor profiling is Oprofile (versie 0.9.3) gebruikt. Dankzij performantie-analyse kunnen de flessenhalzen in het systeem bepaald worden. Deze analyse wordt gedaan met behulp van een referentie-animatie.

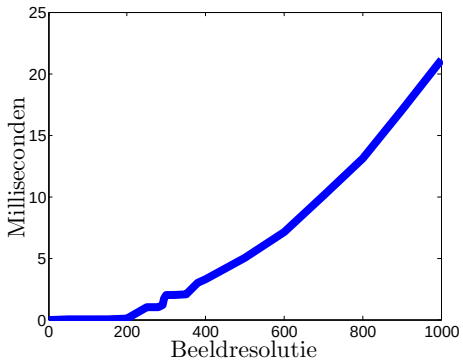
4.3.1 Frameless Rendering versus Dubbele Buffering

Subsectie I toont de snelheid, die gehaald wordt bij het doorsturen van een beeld naar de GVE. Dat is getest voor verschillende beeldresoluties. Daarna wordt de snelheid van OpenRT getest door dubbele buffering te gebruiken in gevallen met camerabeweging, objectbewegingen en verschillende reflectiedieptes. Daarbij wordt de invloed van enkele mogelijke samenstellingen van objecten vergeleken. Dit is beschreven in sectie II. Vervolgens is een applicatie getest die dubbele buffering uitvoert. Resultaten daarvan zijn te vinden in subsectie III. Tenslotte bevat subsectie IV de resultaten van frameless rendering.

I GVE

Figuur 4.1 toont de behaalde snelheid bij het gebruik van glDrawPixels. Dat is belangrijk omdat dubbele buffering deze kost kan aflossen over een heel beeld, terwijl frameless rendering veel meer beelden zal moeten doorsturen naar de

GVE. Zo is het duidelijk dat de applicatie best geen volledige frameless rendering toepast, waarbij voor elke nieuwe pixel een nieuw beeld doorgestuurd wordt. Dan is de kost van het sturen van het beeld al groter dan de kost van het raytracen van *één* pixel.



Figuur 4.1: Scalering met beeldresolutie. De beeldresolutie-as geeft $n \times n$ aan.

II OpenRT

Met behulp van de BART-scene "kitchen" wordt OpenRT getest. "Kitchen" is een beschrijven van een scene en een animatie van 22 seconden. De scene bevat 110.595 driehoeken, die onderverdeeld zijn in 393 meshes. "Kitchen" is speciaal gemaakt [60] om verschillende eigenschappen van een interactief visualisatie algoritme te testen. Zo test het een klein gedetailleerd object in een grote niet gedetailleerde omgeving (auto in keuken), cache performantie (past niet binnen L2-cache), dynamische scenes en benaderingsalgoritmen. Dynamische scenes worden getest door translaties, rotaties en scaleringen van de auto. Benaderingsalgoritmen worden getest door beelden met hoge en beelden met lage temporele coherentie te tonen in de animatie.

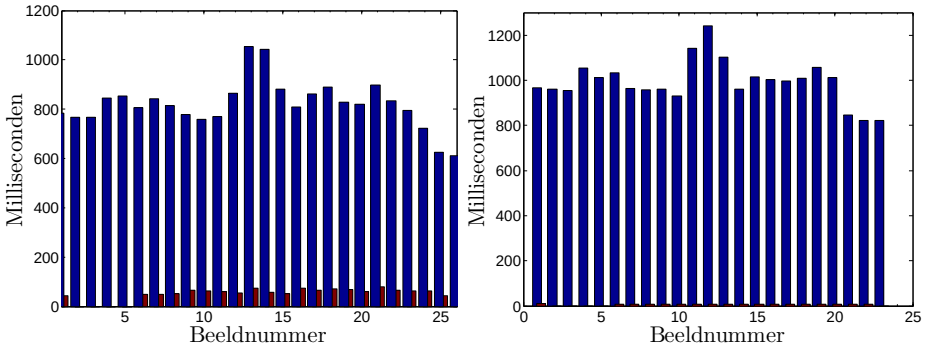
II.1 Objectsamenstelling De beeldresolutie is ingesteld op 320×320 en de reflectiediepte op nul. OpenRT verlangt dat verschillende driehoeken samengepakt worden in *één* object. Deze objectsamenstelling heeft ook effect op frameless rendering, aangezien dubbele buffering de kost van het herbouwen van de versnellingsstructuur weer zoals in I kan aflossen over een heel beeld, terwijl frameless rendering per beeld weer minder tijd heeft.

In de eerste test wordt een object gecreëerd voor elke afzonderlijke driehoek. Enkel voor het genereren van deze objecten vraagt de applicatie al enorm veel tijd. Dit is duidelijk niet de bedoeling. Daarom wordt overgeschakeld op test twee.

In de tweede test wordt voor elke mesh, zoals aangeduid door de bartparser zelf, een object gemaakt. Dat levert dus 393 objecten. Met behulp van deze indeling kan via dubbele buffering de snelheid bepaald worden, die getoond wordt in figuur 4.2 (a).

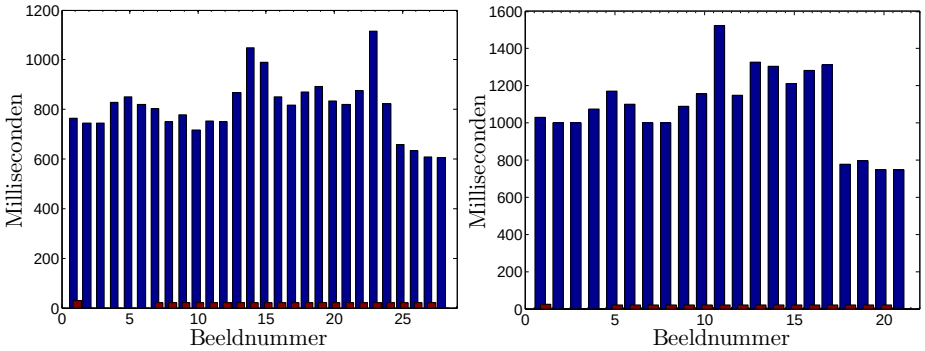
Aangezien bijvoorbeeld de auto en andere voorwerpen in de scene uit meerdere meshes bestaan, zullen ze ook uit meerdere objecten bestaan. Daarom wordt in de derde test voor elk voorwerp een object aangemaakt, waarvan de

snelheid te zien is in figuur 4.2 (b). Dat levert 58 objecten op. Dit geeft dus een lagere snelheid voor de visualisatie en een hogere snelheid voor het aanpassen van de versnellingsstructuur. Die aanpassing gebeurt met het OpenRT commando rtInitNewFrame. Het is duidelijk dat de snelheid van de visualisatie overheerst.



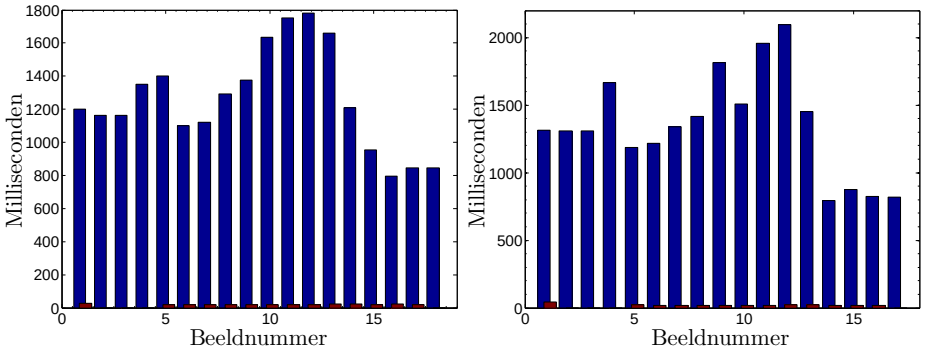
Figuur 4.2: "Kitchen" in OpenRT met objecten: (a) volgens aangegeven meshes (b) volgens voorwerpen in scene. Blauwe balken tonen de benodigde tijd voor visualisatie. Rode balken tonen de benodigde tijd voor het aanpassen van de versnellingsstructuur in OpenRT.

Daarom is tenslotte een laatste test gedaan. Voor elke mesh wordt weer een object aangemaakt, behalve voor het voorwerp (auto) dat beweegt. De documentatie bij OpenRT raadt aan driehoeken, die dezelfde animatie ondergaan, te combineren in *één* object. Nu zijn er 308 objecten aanwezig. Dat geeft de resultaten in figuur 4.3 (a).



Figuur 4.3: (a) "Kitchen" in OpenRT met objecten volgens aangegeven meshes. De geanimeerde auto bevindt zich apart in *één* object. (b) Reflectiediepte *één*

II.2 Reflectiediepte De reflectiediepte kan ingesteld worden bij OpenRT. Het verder recursief schieten van stralen wordt dan beperkt tot deze maximale diepte. Dat heeft een invloed op de snelheid waarmee gevisualiseerd wordt. Dat is duidelijk in figuren 4.3 (b), 4.4 (a) en (b).



Figuur 4.4: (a) Reflectiediepte twee (b) Reflectiediepte drie

III Dubbele buffering

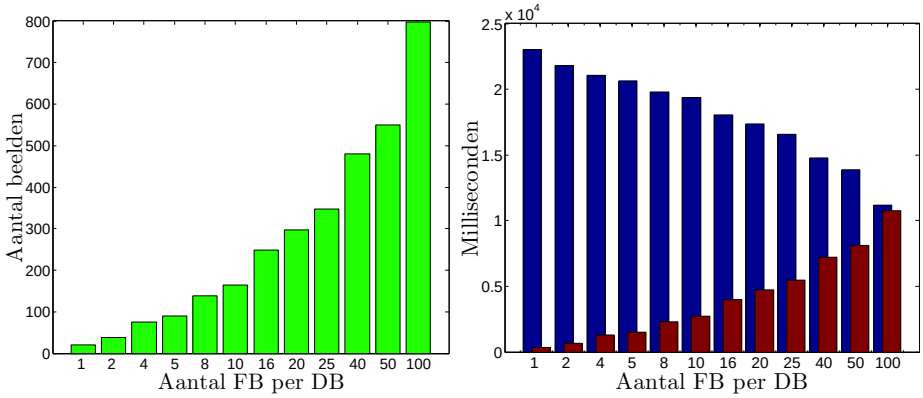
Het is mogelijk om ofwel per bepaald interval een nieuw beeld te genereren, ofwel direct een nieuw beeld te tonen als het af is en direct aan het volgende beeld beginnen. De laatste mogelijkheid levert een variabel aantal beelden, aangezien niet elk beeld even lang duurt om te berekenen. Bij reflectiediepte nul, kan dubbele buffering bij een vast bepaald interval ongeveer slechts *één* beeld per seconde bereiken voor een 320×320 beeld. Dat is niet voldoende om een vloeiende waarneming te bereiken. Het beeld bezit de volle kwaliteit, maar de animatie schokt duidelijk. Bij een variabel aantal beelden per seconde, kan het 28 beelden tonen voor de animatie van 22 seconden, zoals getoond wordt in figuur 4.3 (a).

IV Frameless rendering

Standaard frameless rendering zou per nieuw gevisualiseerde pixel de camera- en sceneparameters moeten aanpassen en het resultaat direct op het beeld moeten tonen. De beelden bij dubbele buffering duurden gemiddeld een 800 milliseconden om getoond te worden voor een 320×320 resolutie. Dat betekent een gemiddelde kost per pixel van ongeveer $(\frac{800}{320 \times 320})$ 0.0078125 milliseconden. Maar het aanpassen van de versnellingsstructuur kost per scene-aanpassing gemiddeld twintig milliseconden. Als er 1100 beelden getoond worden met frameless rendering, zal er geen tijd zijn om echt een pixel te visualiseren, aangezien 20×1100 de 22 seconden animatie oplevert. Daarom wordt geëxperimenteerd met een lager aantal beelden. Als er 40 $(\frac{800}{20})$ beelden getoond worden, zal elk beeld even veel tijd kosten om te visualiseren dan als er nodig is om de versnellingsstructuur aan te passen.

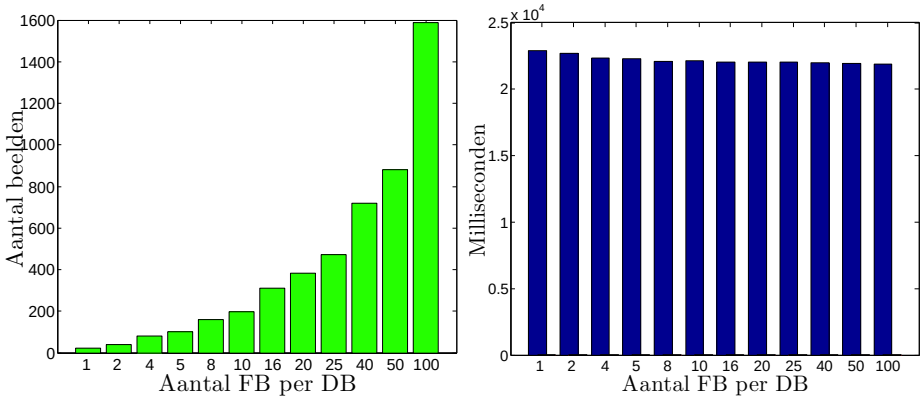
Er is opeenvolgend geëxperimenteerd met een gewenst aantal van 1, 2, 4, 5, 8, 10, 16, 20, 25, 40, 50 en 100 beelden per dubbel gebufferd beeld. De resultaten bevinden zich in figuur 4.5. Het resultaat, waarbij *één* beeld gewenst is per dubbel gebufferd beeld, toont dat de structuur voor het random kiezen van monsters en het gebruiken van de tabel een vertraging oplevert. Pure dubbelbuffering laat 28 beelden van volle kwaliteit per seconde toe. Nu zijn dat er 21. Bij het scaleren naar een hoger aantal beelden valt het op dat de aanpassing van de versnellingsstructuur telkens een belangrijkere rol gaat spelen en

het aantal beelden niet langer laat verdubbelen vanaf een tiental beelden per dubbel gebufferd beeld.



Figuur 4.5: Met sceneveranderingen: (a) Aantal bekomen beelden voor de animatie van 22 seconden (b) Benodigde tijd voor visualisatie in blauw en voor het aanpassen van de versnellingsstructuur in rood.

Wanneer geen sceneveranderingen gedaan worden en de versnellingsstructuur van OpenRT bijgevolg niet aangepast moet worden, is er een veel betere scalering, zoals te zien is in figuur 4.6. Bij camerabewegingen en illuminatieveranderingen is er dus geen probleem met de scalering, aangezien ze geen aanpassing vragen van de versnellingsstructuur. Deze scalering zal ook bij de andere frameless rendering technieken invloed hebben op de snelheid, waarmee nieuwe monsters berekend kunnen worden.



Figuur 4.6: Zonder sceneveranderingen: (a) Aantal bekomen beelden voor de animatie van 22 seconden (b) Benodigde tijd voor visualisatie in blauw en versnellingsstructuur, die niet aangepast wordt, in rood (amper zichtbaar).

4.3.2 Render Cache

Elke implementatie van de render cache staat in een aparte sectie. De originele render cache wordt besproken in subsectie I. De render cache, die uitgebreid is met een prefilter en voorspellend bemonsteren, staat in subsectie II. Tenslotte komt de render cache, waarvan een gedeelte uitgevoerd wordt op de GVE, aan bod in subsectie III.

I Originele Render Cache

Bij de implementatie van de render cache worden twee threads gebruikt. Aangezien de gebruikte computer twee processoren bezit, moet de performantie normaal zowiezo reeds verdubbelen, wat in het dubbele buffering geval twee beelden per seconde betekent. Het beeldgeneratieproces van de render cache vereist in totaal, zoals getoond in tabel I, gemiddeld tussen de 40 en 50ms. Daarom zouden er per dubbelgebufferd beeld, zonder de raytracingtijd mee te tellen, op een enkele processor tussen de 20 en 25 beelden gemaakt moeten kunnen worden. Als verzekerd wordt dat de raytracer telkens evenveel tijd gebruikt dan het beeldgeneratieproces, is er tussen de 80 en 100ms nodig om een enkel beeld te genereren. Dan kunnen een tiental beelden per dubbel gebufferd beeld getoond worden. Omdat dit een dual core is, betekent dat dat deze tijd gehalveerd zou moeten worden en er terug tussen de 20 en 25 beelden getoond zouden moeten worden.

De timer in Qt genereert bij frameless rendering en dubbele buffering op de juiste tijdsstippen een tijdsevent, tenzij de vorige bewerkingen nog niet afgerond zijn. Dit lijkt niet het geval te zijn bij de render cache. In een initiële implementatie lukt het slechts om vijf beelden te tonen per dubbel gebufferd beeld. Daarom is er tijdens de uitvoering van het programma getest hoeveel tijd de verschillende onderdelen van de applicatie in beslag namen. De timer roept enkel de functie paintGL aan en die duurt 40ms. Ook de mutexen bij het kopiëren zijn niet de oorzaak van deze vertraging. Bij het inspecteren van de systeemmonitor valt op dat *één* processor 100% van de tijd in beslag neemt en de andere processor slechts een 30%. Het leek dus alsof de raytracer en het beeldgeneratieproces niet gelijkwaardig verdeeld zijn. Doch bij het timen van de raytracer was het duidelijk dat ze gelijkwaardig verdeeld zijn. Daarom is uiteindelijk de Qt-timer vervangen door een forlus. Dankzij die vervanging lukte het ineens wel om beide processoren voor 100% in beslag te nemen. De 22-seconden durende "kitchen"-animatie kan uitgevoerd worden, terwijl er 541 beelden getoond worden. Dat betekent ongeveer 25 beelden per seconde. De animatie is bijgevolg vloeiend. De raytracer verbruikt een gelijkwaardige tijd als het beeldgeneratieproces wanneer het $\frac{1}{32}$ste van het beeld vernieuwt per door het beeldgeneratieproces gegenereerd beeld. Dankzij de herprojectie en de filtering zijn de beelden zelfs vrij goed. Bij sterke bewegingen zijn er delen van het beeld missende aan de zijkanten van het beeld of bij dissoclusies.

Tabel I toont de tijd in milliseconden, die de onderdelen van de render cache nodig hebben voor een 320×320 beeld. Deze resultaten zijn gemiddeld over de hele "kitchen"-animatie, maar verschillen gewoonlijk weinig per beeld. Het is duidelijk dat de herprojectie het meeste tijd in beslag neemt, buiten het raytracen van monsters.

Cache vernieuwen	2, 340070ms
Objectposities aanpassen	0, 841912ms
Herprojectie	19, 549600ms
Diepte Culling	3, 172794ms
Interpolatiefilter	6, 990808ms
Prioriteitsbeeld	1, 976100ms
Dithering	1, 687500ms
Leeftijden verhogen	1, 917280ms
Beeld tonen	1, 472430ms
Totale tijd	39, 948494ms

Tabel 4.2: Benodigde tijd per onderdeel van de render cache voor een beeld van resolutie 320×320.

Via Oprofile is het ook duidelijk dat raytracing en het beeldgeneratieproces elk ongeveer de helft van de tijd in beslag nemen. De samengetelde duurste OpenRT-functies verbruiken zo rond de $46,7815\%$ van de hele uitvoeringstijd. Bij het beeldgeneratieproces wordt herprojectie het langst uitgevoerd. De tijd, die raytracing verbruikt, moet niet even lang zijn dan die van het beeldgeneratieproces. Hier is dat gedaan, omdat de implementatie twee threads bezit en de tweede processor anders niet volledig in gebruik genomen zou worden. Op die manier worden de processoren met twee threads maximaal benut. Met behulp van drie threads kan op de processor, waarop het beeldgeneratieproces uitgevoerd wordt, geruild worden tussen het aantal beelden en de kwaliteit. De raytracer kan daar een deel van de tijd afnemen om ook monsters te nemen.

II Uitgebreide Render Cache

Door het toevoegen van de prefilter en het voorspellend bemonsteren, stijgt de tijd, die het beeldgeneratieproces vereist. Tabel II geeft de tijden weer voor beide onderdelen bij een beeld met resolutie 320×320. De prefilter is daarbij onafhankelijk van de animatie, aangezien die op elke pixel uitgevoerd wordt en is daarmee een grotere flessenhals geworden dan de herprojectie. Het voorspellend bemonsteren is ook onafhankelijk van de animatie, aangezien alle monsters geherprojecteerd worden. Het is goedkoper dan de echte herprojectie, aangezien er hier geen Z-buffer nodig is. Er moeten enkel gaten in het beeld gevonden worden en er moet dus niet bepaald worden welke kleur bepaalde pixels hebben. Het aantal nieuwe monsters dat berekend zal worden, is ook onafhankelijk van het voorspellend bemonsteren. Wanneer er teveel monsters aangevraagd worden met voorspellend bemonsteren, zal $\frac{nrAanvragen}{nrMogelijkeAanvragen}$ als interval genomen worden om in de array van aanvragen monsters te kiezen. Zo zullen er niet meer monsters gekozen worden dan het aantal, dat vooropgesteld is. Wanneer er minder aanvragen zijn dan vooropgesteld is, zal de drempelwaarde bij de dithering aangepast worden, zodat er daar meer genomen worden. Het uiteindelijk aantal aangevraagde monsters blijft op die manier ongeveer gelijk. In de paper [105] houdt de prefilter slechts 10% in van het hele beeldgeneratieproces, maar daar is tevens gebruik gemaakt van SIMD-optimalisaties en hier niet.

Als het aantal monsters, dat aangevraagd mag worden, even groot blijft als bij de originele render cache in de vorige sectie I, dan haalt deze implementatie

Prefilter	26, 891447ms
Voorspellen	6, 819080ms

Tabel 4.3: Benodigde tijd van de nieuwe onderdelen voor een beeld van resolutie 320×320.

304 beelden voor de animatie van 22 seconden, ofwel een veertiental beelden per seconde. Indien het aantal aan te vragen monsters aangepast wordt, zodat de raytracer evenveel tijd nodig heeft dan het beeldgeneratieproces, kunnen 289 beelden bekomen worden.

III GVE Render Cache

In deze implementatie bevindt zich geen prefilter en geen voorspellend bemonsteren. Het doet enkel een aantal delen van de originele render cache op de GVE. Daarbij wordt bijvoorbeeld de grootste flessenhals, namelijk herprojectie, sneller gemaakt. De aanpassing van de scene met behulp van rtInitNewFrame bevindt zich in de thread van de raytracer. Het beeldgeneratieproces nam 40ms tijd in beslag in de originele render cache. Door een deel ervan op de GVE uit te voeren, wordt een gemiddelde van 20ms bereikt. Omdat de aanpassing van de versnellingsstructuur reeds een gemiddelde van 20ms in beslag neemt, is het niet mogelijk beide processoren evenveel werk te geven met behulp van de twee threads. De berekeningen van de thread met de raytracer kunnen niet zo laag gekregen worden dan die van het beeldgeneratieproces. Er moeten buiten de animaties ook nog monsters gevisualiseerd worden. Daarom wordt in een eerste test de thread van de raytracer behouden op 40ms zoals bij de originele render cache. In een tweede test wordt de animatie van de auto weggelaten. Dan moet enkel de camera aangepast worden en is er geen aanpassing nodig van de versnellingsstructuur. Bij die tweede test kunnen beide processoren wel even zwaar bezet worden en zal het beeldgeneratieproces evenveel keer uitgevoerd worden als de raytracer.

De eerste test levert 969 beelden op, waarbij de raytracer ongeveer 484 keer wordt uitgevoerd. Er worden dus ongeveer 44 beelden per seconde gegenereerd. In de helft van de nieuwe beelden zijn dus geen nieuwe monsters aanwezig, maar de herprojectie en filtering zorgen er wel voor dat het een ander, recenter beeld oplevert. De raytracer levert telkens $\frac{1}{32}$ste van de beeldresolutie aan nieuwe monsters (per twee beelden ongeveer).

In de tweede test worden 1.015 beelden gegenereerd. Dat zijn ongeveer 46 beelden per seconde. Elk beeld incorporeert $\frac{1}{50}ste$ van de beeldresolutie aan nieuwe monsters, gevisualiseerd door de raytracer. Aangezien de thread van de raytracer enkel monsters moet nemen en de thread van het beeldgeneratieproces enkel een beeld moet tonen, is dit de maximale opstelling met behulp van twee threads. In totaal zal de raytracer in dit geval altijd evenveel monsters nemen voor een bepaalde seconde. Wanneer het deze doorstuurt naar het beeldgeneratieproces heeft weinig belang. Het beeldgeneratieproces zal ook altijd evenveel beelden genereren voor een bepaalde seconde, dus kan het er best zoveel mogelijk genereren.

De verschillende tijden van de nieuwe onderdelen staan vermeld in tabel III.

Samen met de onderdelen, die zich nog steeds op de CVE bevinden (cache aanpassen, objectposities aanpassen, dithering en leeftijden verhogen), wordt iets minder dan 20ms bereikt. Het vullen van het prioriteitsbeeld kan normaalgezien ook op de GVE gebeuren, maar dat is hier niet het geval. Dit is vrij duur, vergeleken met de andere onderdelen. De herprojectie is beduidend sneller. Ook de filters zijn veel sneller. Het aanpassen van de VBO zou eventueel kunnen door niet telkens alle monsters door te zenden. Maar zoals uit deze resultaten blijkt, kan de techniek eerst beter op andere plaatsen geoptimaliseerd worden.

Aanpassen van de VBO	$2,617733$ms
Herprojectie	$0,491625$ms
Enkel teruglezen	$1,871921$ms
Filter 1 en op FBO textuur tekenen	$0,369458$ms
Filter 2 en op beeld tekenen	$0,051231$ms
Prioriteitsbeeld vullen (Niet op GVE)	$4,107389$ms
Totale tijd GVE gedeelte	$5,401968$ms
Totale tijd nieuwe onderdelen	$9,509357$ms

Tabel 4.4: Benodigde tijd van de GVE-onderdelen voor een beeld van resolutie 320×320.

4.3.3 Adaptive Frameless Rendering

Ook hier is de Qt-timer vervangen door een forlus om de CVE maximaal te bezetten. Voor dezelfde animatie als bij de vorige technieken, kan adaptive frameless rendering voor een resolutie van 320×320 ongeveer zestien beelden per seconde behalen. De reconstructor vereist daarbij een gemiddelde van 61ms per beeld. Er werden namelijk 360 beelden gegenereerd voor de animatie van 22 seconden. Tabel 4.3.3 toont de vereiste tijd per onderdeel. De tegelinginfo-overdracht omvat ook het berekenen van de gemiddelde spatiale en temporele kleurgradiënten. Bij het tekenen van de tegeling in de textuur moeten rechthoeken gemaakt worden, die als kleur de gemiddelde gradiënten krijgt in het R,G en B kanaal.

Vernieuwen van de VBO's	$17,761111$ms
Overdracht tegelinginfo	$10,791666$ms
Tegeling tekenen in FBO-textuur	$18,330555$ms
Herprojectie en filters	$13,355555$ms
Totale reconstructietijd (per beeld)	$60,238887$ms

Tabel 4.5: Benodigde tijd van de onderdelen van de reconstructor.

Het bemonsteringsproces heeft tijdens deze animatie 35.001 keer geïtereerd. Daarbij zijn er in totaal 363.236 monsters gevisualiseerd. Dat aantal houdt alle monsters in, die gevisualiseerd zijn, ook degene die zich bij in de crosshairs bevinden. In tabel 4.3.3 bevinden zich de uitvoeringstijden van de verschillende onderdelen van het bemonsteringsproces. De tegeling wordt om de honderd keer aangepast. Daarom wordt zijn uitvoeringstijd gedeeld door honderd en opgeteld

bij de totale uitvoeringstijd van het bemonsteringsproces. $0,60929247 \times 35.001$ geeft ongeveer 22ms.

Camera aanpassen	$0,049170$ms
Random tegel kiezen	$0,001685$ms
Nieuwste monster erin vinden	$0,002542$ms
Crosshair vervolledigen	$0,068083$ms
Diepe buffers en statistieken aanpassen	$0,005942$ms
Herprojectie (de vijf keer samen)	$0,118596$ms
Randompositie kiezen in tegel	$0,002228$ms
Nieuw temporeel monster visualiseren	$0,017542$ms
Tegeling aanpassen (de 100 keer samen)	$34,350447$ms
Totale bemonsteringtijd (per iteratie)	$0,60929247$ms

Tabel 4.6: Benodigde tijd van de onderdelen van het bemonsteringsproces.

In deze opstelling verbruikt het bemonsteringsproces per lus $0,085625$ms in visualisatie (crosshair vervolledigen plus nieuw temporeel monster visualiseren) en $0,52366747$ms in de andere berekeningen, zoals het aanpassen van de kd-boom. De aanpassing van de kd-boom verbruikt meer dan de helft van de beschikbare tijd, namelijk $0,34350447$ms per lus. De overschot van het bemonsteringsproces verbruikt $0,180163$ms. Daarin bevindt zich ook nog een gedeelte visualisatie, aangezien de herprojectie $0,118596$ms voorstelt en bij occlusies nieuwe monsters zal genereren. Verder stellen de geherprojecteerde monsters ook recente informatie voor. Voor occlusies zal die kloppen, aangezien er bij occlusies een straal geschoten wordt om te testen of het monster het dichtste bij de camera ligt. Voor speculaire effecten, illuminatieveranderingen en veranderingen van de objectoppervlakken kan deze informatie fout zijn. Het is mogelijk de tegeling minder adaptief te maken door meer monsters te visualiseren vooraleer de tegeling aangepast wordt. Op die manier zullen er meer nieuwe correcte monsters per seconde bijkomen.

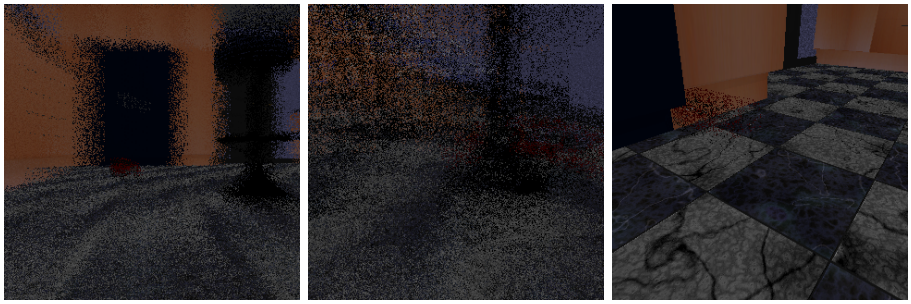
Wanneer de tegeling na 400 iteraties aangepast wordt, neemt deze evenveel tijd in beslag dan het visualiseren van nieuwe monsters. Er zijn dan 536.580 monsters genomen. Dat betekent 24.390 monsters per seconde, ofwel $\frac{1}{4}$de van het aantal pixels per seconde. De tegeling is nu $6,673295$ keer aangepast per seconde, ofwel *één* keer per 3.655 monsters, aangezien er 58.725 iteraties van het bemonsteringsproces uitgevoerd zijn.

Omdat het animeren van de auto telkens veel tijd kost, is er hier enkel gebruik gemaakt van animatie met de camera. Adaptive frameless rendering gaat namelijk ongeveer per zes nieuwe monsters de camera- en sceneparameters aanpassen, tenzij er via herprojectie occlusies ontdekt worden. De animatie met de auto vraagt dan zeer veel tijd. Het is mogelijk de animatie van de auto pas na een groter tijdsinterval te vernieuwen. Dan zullen monsters binnen een klein interval (generatie van zes monsters plus de tijd voor de andere onderdelen van het bemonsteringsproces) geen nieuwe cameraparameters gebruiken en binnen een gekozen, groter interval, geen nieuwe sceneparameters gebruiken. Als dat interval klein genoeg is, zodat de mens het niet waarneemt, is dat visueel geen probleem.

4.4 Visuele resultaten

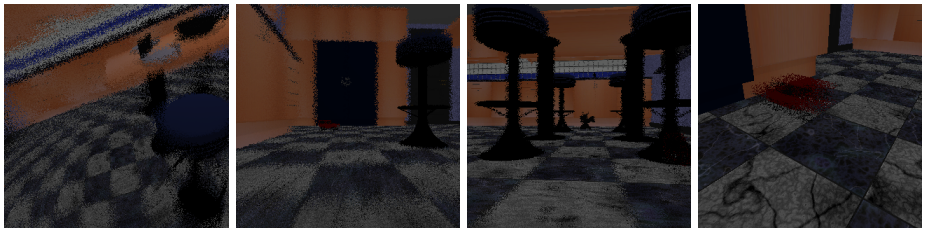
4.4.1 Frameless Rendering

Hoe meer beelden per seconde getoond worden, hoe vloeiender de bewegingen overkomen. Maar hoe dichter de dubbele buffering benaderd wordt in aantal beelden, hoe duidelijker de beelden zullen zijn. Wanneer het aantal frameless beelden dus niet te hoog is, kan van beide voordelen genoten worden, maar het blijft dan natuurlijk een benaderd beeld, dat ruis bevat. Een dergelijk beeld is weliswaar slechts zeer kort zichtbaar.



Figuur 4.7: Acht beelden per dubbel gebufferd beeld: (a) Redelijk trage beweging (b) Snelle beweging (c) Camera stil, auto beweegt

Als de camerabeweging te snel gaat, zal er door deze basis frameless rendering geen nuttig beeld getoond worden 4.7 (b). Het beeld is dan een mengelmoes van allerlei tijdsstippen, waarin de beeldinhoud sterk verandert. Als de camerabeweging stopt, zal het beeld zeer snel terug de volledige kwaliteit tonen. In 4.7 (c) staat de camera stil en is enkel het gebied, waarin de auto beweegt nog fout. Tijdens kleine camerabewegingen is er veel coherentie tussen opeenvolgende beelden, zodat een vloeiende animatie zichtbaar is 4.7 (a).



Figuur 4.8: Acht beelden per dubbel gebufferd beeld, tijd is vijf keer vertraagd: (a) Redelijk snelle beweging (b) Trage beweging (c) Trage beweging (d) Camera stil, auto beweegt

Figuur 4.8 toont dezelfde animatie, maar de tijd is hier vijf keer trager gemaakt aangezien deze voornamelijk snelle bewegingen bevat. Frameless rendering is bovendien nuttiger om visualisaties te versnellen, die dichtbij de grens van vloeiende animatie liggen. Dubbele buffering levert daar juist te traag nieu-

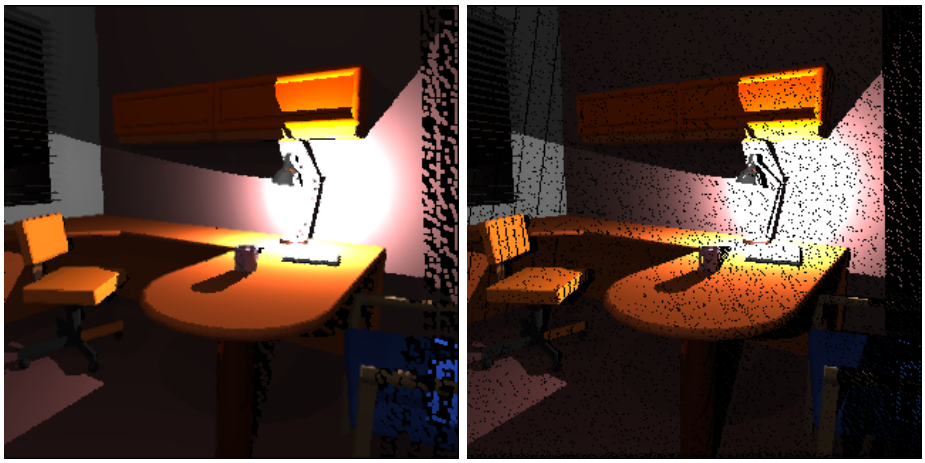
we beelden, terwijl frameless rendering daar het aantal beelden sterk verhoogt. Omdat de opeenvolgende beelden daar sowieso al een betere temporele coherentie bezitten, zijn er bovendien nog minder fouten aanwezig. Vijf keer trager betekent dus dat dubbele buffering ongeveer vijf beelden per seconde kan tonen, wat nog steeds vrij schokkerige beelden geeft. Frameless rendering kan er nu ongeveer 25 per beeld tonen. De animatie van 22 is nu 110 seconden lang. Frameless rendering heeft 572 beelden geproduceerd, terwijl dubbele buffering er 130 getoond heeft. Dat is een iets groter aantal, omdat de 800ms waar we vanuit gaan een gemiddelde is en sommige beelden dus minder tijd vereisen, waardoor sneller een nieuw beeld gemaakt wordt. Dankzij de rustigere bewegingen produceert frameless rendering ook nuttigere beelden en een zachte animatie. In figuur 4.8 (b), (c) en (d) zijn zo enkele situaties getoond. Zelfs tijdens een iets snellere beweging, zoals in figuur 4.8 (a), wordt nog steeds een nuttig beeld getoond. Er is ook nuttige motion blur aanwezig, zoals in figuur 4.8 (d).

Wanneer geen tabel gebruikt wordt, zullen er pixels zijn, die pas na een geruime tijd herberekend worden of potentieel nooit herberekend worden. Vaak worden dan trouwens dezelfde pixels herberekend en bijgevolg wordt er geen progressieve verfijning bekomen. Dit zorgt ervoor dat de meeste beelden een mengelmoes zijn van heel oude kleurwaarden, heel nieuwe kleurwaarden en een hoop kleurwaarden met een leeftijd daartussen.

De implementatie, die niet random het beeld vernieuwt, maar volgens een vast patroon, levert minder aangename resultaten op. De artefacten zijn steeds op dezelfde positie aanwezig en het lijkt niet alsof overal een vernieuwing gebeurt. De sequentiele vernieuwing is erger dan degene die met een gelijk interval. Het vernieuwt het beeld op een heel onaangename wijze, aangezien delen van het beeld constant wegscheuren en onafhankelijk lijken.

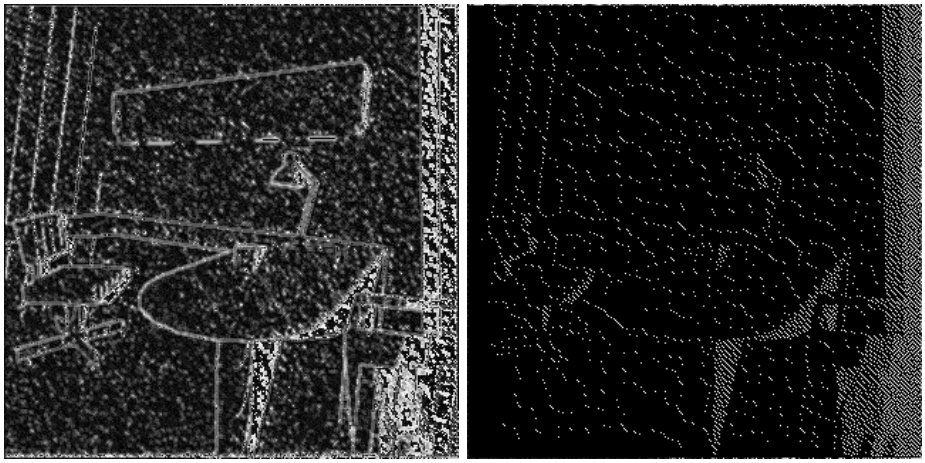
4.4.2 Render Cache

In de eerste figuur 4.9 beweegt de camera van links naar rechts. Als gevolg



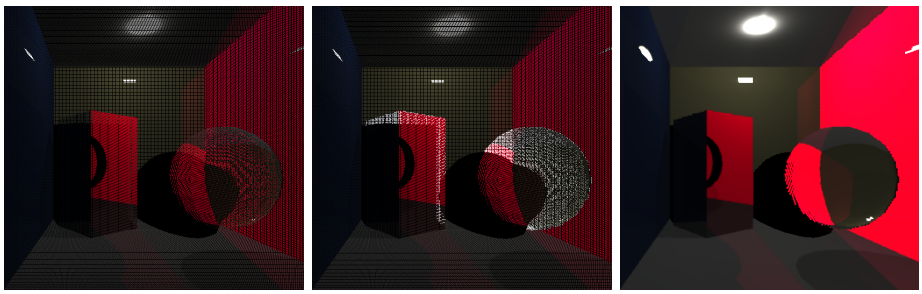
Figuur 4.9: (a) Finaal beeld (b) Enkel herprojectie

daarvan komt er rechts een deel van de scene in beeld, waar nog geen monsters voor aanwezig zijn in de cache. Direct nadat de render cache dat gebied ontdekt heeft, genereert het monsters voor dat gebied. Figuur 4.10 (a) toont het gegenereerde prioriteitsbeeld. De witte pixels duiden de belangrijke posities aan. Het gebied dat rechts in beeld komt heeft zo een hoge prioriteit, net als de poot van de bureau en nog een aantal gebieden, waar eerst occlusies voorvielen en nu het achterliggende in beeld komt.



Figuur 4.10: (a) Prioriteitsbeeld (b) Dithering

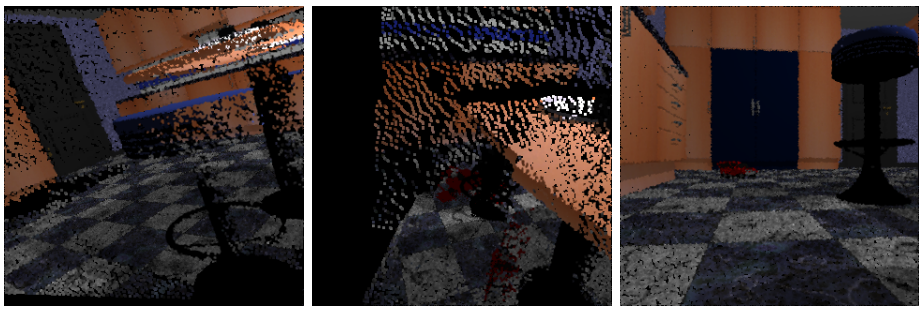
Figuur 4.10 (b) toont de uiteindelijk aangevraagde monsterposities met behulp van error diffusion dither. Dat genereert een beperkt aantal aanvragen van het prioriteitsbeeld. Daarbij zorgt het ervoor dat die liggen in de belangrijke gebieden en tevens goed verspreid zijn over het beeld, zoals hier te zien is. Figuur 4.9 toont het geherprojecteerd beeld (b) en toont het finaal beeld (a). De herprojectie bevat duidelijk meer fouten dan het finale beeld, dat gefilterd wordt met de diepte culling en de interpolatiefilter om gaten op te vullen.



Figuur 4.11: (a) Herprojectie (b) Diepte culling (c) Interpolatie

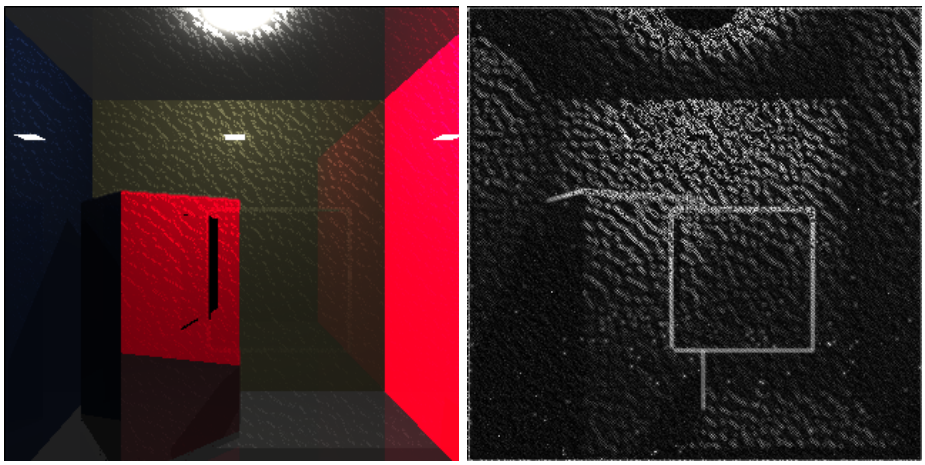
De beeldreconstructiestappen worden getoond in figuur 4.11. Hier is gestart vanaf een enkel beeld en er werden geen monsters toegevoegd in de cache. Daarna is met de camera naar een bepaalde positie bewogen. Er is hier te zien hoe eerst de monsters op het beeld geprojecteerd worden met behulp van Z-buffering

(a). Daarna wordt in (b) de diepte culling filter toegepast. De witte pixels in de bol en op de balk geven de posities aan, waarvoor geen occluderende monsters beschikbaar zijn. De bol en balk zouden dus doorzichtig zijn, dankzij een tekort aan monsters. Maar met behulp van deze filter worden de onderliggende monsters verwijderd (op die witte pixels dus). De derde afbeelding (c) toont het uiteindelijk beeld, waarop ook de interpolatiefilter toegepast is.



Figuur 4.12: (a en b) Snelle camerabeweging (c) Trage camerabeweging

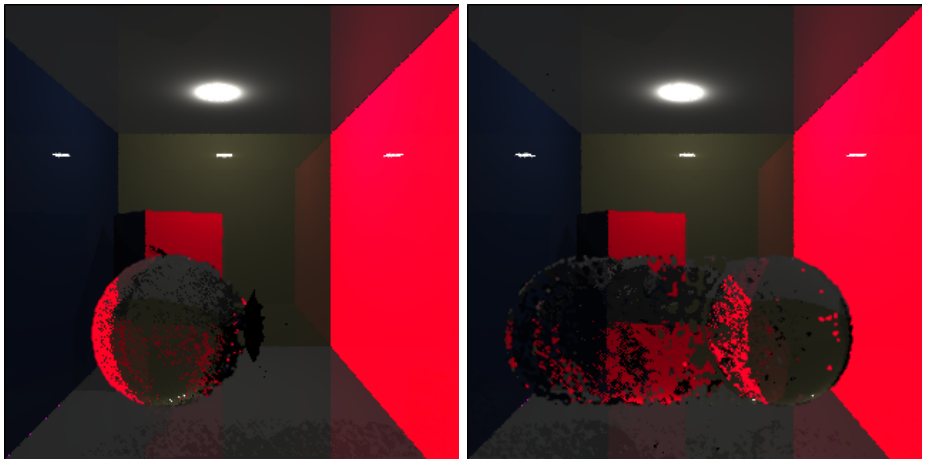
De kleurveranderingsheuristiek is eenvoudig te tonen door de lichtintensiteit in de scene aan te passen. In figuur 4.13 staat rechts het prioriteitsbeeld. Daarin wordt weergegeven hoe de gebieden, die het snelst en hardst beïnvloed worden door de illuminatieverandering, reeds een hogere prioriteit hebben. Als er geen camerabeweging gebeurt, zullen alle nieuwe monsters op dezelfde 3D-positie genomen worden. Dan wordt hun kleurwaarde vergeleken met de kleurwaarde van het monster dat eerder op die positie op het beeld aanwezig was. Indien de kleur verschilt, wat hier dus het geval is, zullen de prioriteiten van de buurpixels verhoogd worden. In de animatie zetten die buren hun buren weer aan tot snellere bemonstering, zodat deze prioriteiten uitbreiden totdat de hele scene de nieuwe lichtintensiteit ontvangen heeft.



Figuur 4.13: Kleurveranderingsheuristiek

Figuur 4.12 vergelijkt enkele snelle camerabewegingen (a en b) met een tra-

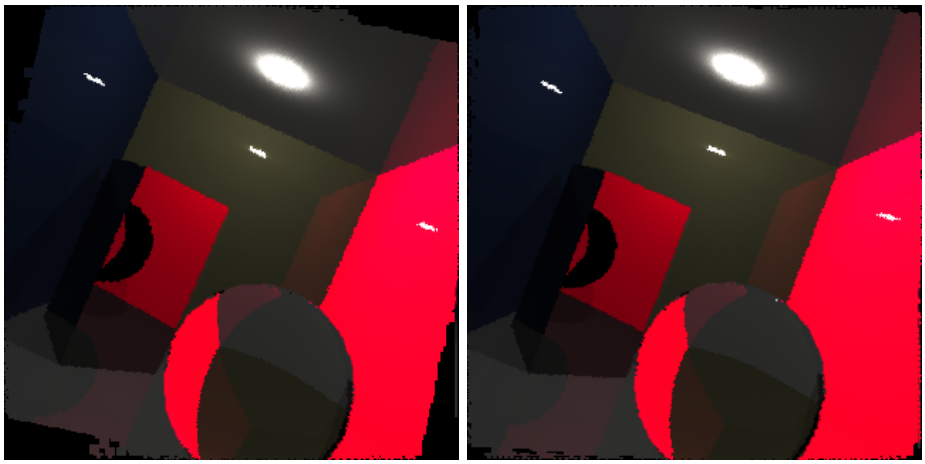
ge camerabeweging (c). Omdat de beweging in (a) even in ongeveer dezelfde richting gebeurt, toont de render cache een redelijk nuttig beeld. Het bezit dan nog informatie van een deel van het beeld en dat deel zal zich gelijkmatig verplaatsen over het beeld. Wanneer nog harder bewogen wordt en snel verwisseld wordt van richting, ontstaat eerder een beeld zoals in (b). Voor een groot deel van het beeld is nog geen informatie aanwezig, zodat er niks voor getoond kan worden. Bovendien zal een ander groot deel van het beeld door elkaar gemengd worden, zodat er veel op elkaar herprojecteren en er gaten ontstaan. Figuur 4.12 (c) toont een trage camerabeweging, waarbij het grootste deel van de aanwezig informatie in de cache maar een kleine positiewijziging zal doormaken, zodat het beeld goed gevuld blijft. Mogelijke gaten worden weggefilterd.



Figuur 4.14: Object wordt: (a) aangepast in cache (b) niet aangepast in cache

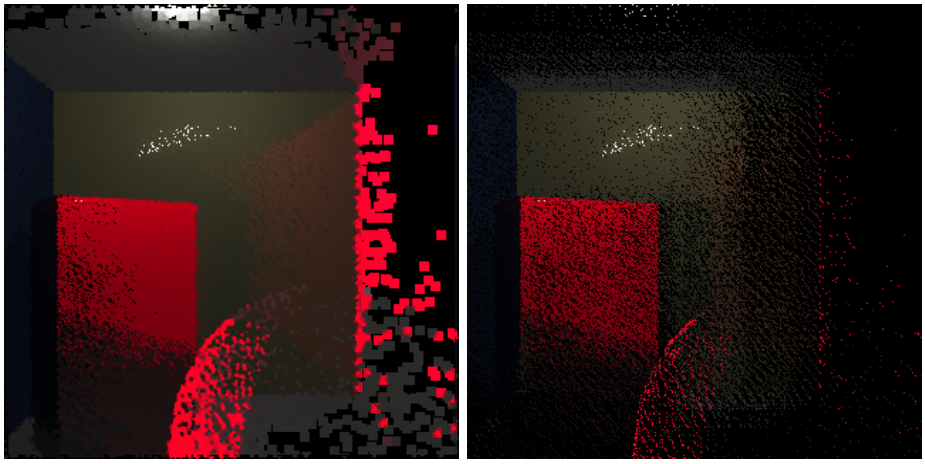
In figuur 4.14 wordt een object snel getransleerd van rechts naar links. Bij dit voorbeeld wordt een vijfde van het beeld vernieuwd per 200ms. Figuur (a) toont het geval wanneer er rekening gehouden wordt met "rigid-body"-transformaties. Via het object-ID worden alle monsters van dat object in de cache aangepast aan de huidige objectparameters. Figuur (b) toont het geval wanneer er geen rekening mee gehouden wordt. Het object is op een viertal plaatsen tegelijkertijd te zien. Indien meerdere beelden gegenereerd worden in de tijd en bijgevolg meer de sceneparameters bemonsterd worden, zal dit sterker lijken op motion blur en er aangenamer uitzien. Bij de linkse figuur heeft het prioriteitsbeeld reeds het dissoclusiegat ontdekt en gedeeltelijk opgevuld. Dat is mogelijk omdat de monsters in de cache van dat object verplaatst zijn, zodat het gat vroeger ontstaat en direct groot is. De kleurwaarden op de bol zijn nog niet volledig aangepast, omdat het object doorzichtig is en de kleurwaarden dus trager vernieuwd worden.

De geïmplementeerde uitbreidingen van de render cache komen nu aan bod. Figuur 4.15 (a) toont een snelle camerarotatie, waarbij niet voorspeld wordt. Het resultaat is dat er geen monsters beschikbaar zijn voor de zwarte plaatsen in de hoeken. Figuur 4.15 (b) toont een voorbeeld van voorspellend bemonsteren. Door herprojectie van de monsters op een beeldvlak, gebruikmakende van voorspelde cameraparameters van enkele beelden later, worden de zwarte



Figuur 4.15: (a) Niet voorspeld (b) Voorspeld

hoeken ontdekt en op voorhand bemonsterd.



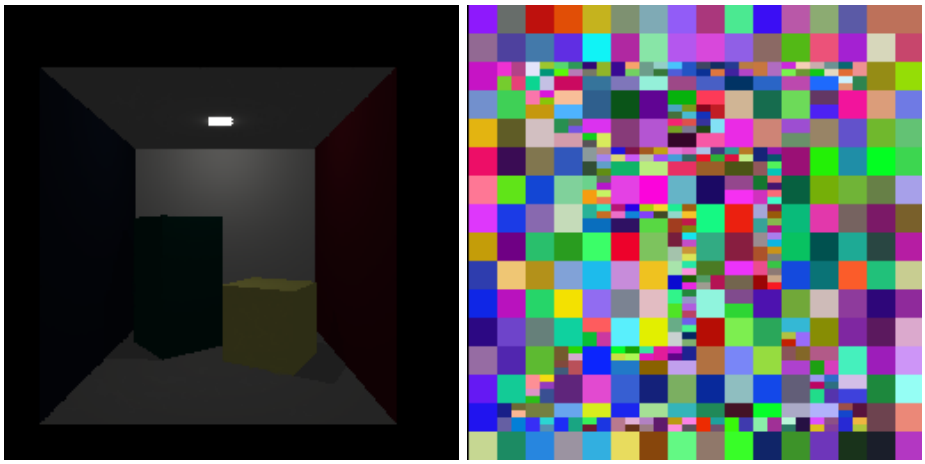
Figuur 4.16: (a) Prefilter (b) Zonder filter

De prefilter wordt getoond in figuur 4.16. Er is een snelle, grote beweging toegepast, zodat er weinig informatie beschikbaar is voor een deel van het beeld. Rechts is het beeld weergeven, waarop de monsters alleen maar geherprojecteerd zijn. Links is het beeld zichtbaar, waarop de prefilter toegepast is. Het is duidelijk dat de monsters uitgebreid zijn tot grotere vierhoeken in heel dunne gebieden, en mooi opvullen in iets dichter bemonsterde gebieden.

4.4.3 Adaptive Frameless Rendering

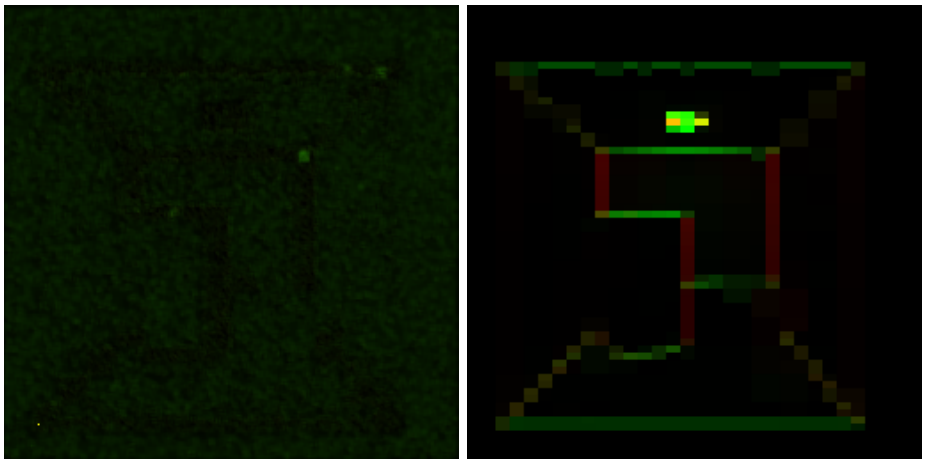
Figuur 4.17 toont een afbeelding van een scene, waarbij zowel de camera als de scene statisch zijn. In (a) staat het gereconstrueerd beeld en in (b) is de huidige tegeling te zien. Omdat de tegeling zich rond de randen van de scene

bevindt, zijn de randen van de objecten scherp. Er wordt geen interpolatie over de randen gedaan. Aangezien de scene statisch is, zal de splatgrootte overal klein zijn, zodat ook daardoor geen wazige randen aanwezig zijn.



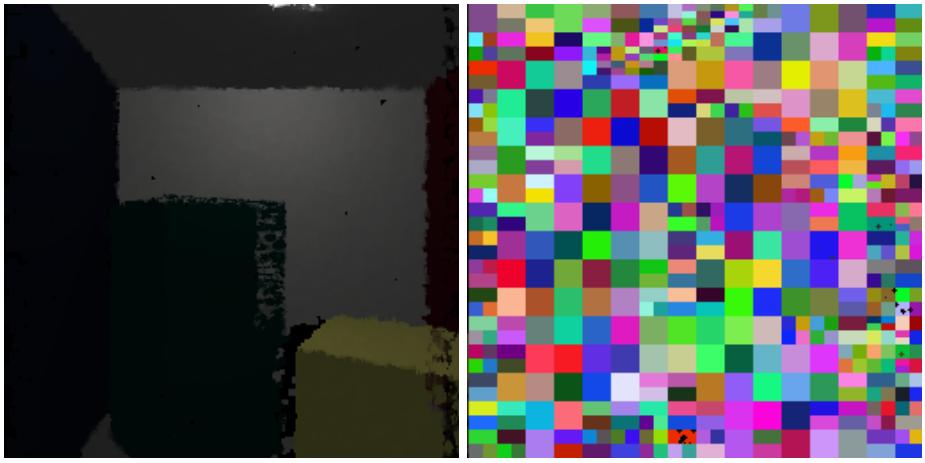
Figuur 4.17: (a) Finaal beeld (b) Kd-boom (tegeling)

In figuur 4.18 staat links (a) de omgekeerde bemonsteringsdichtheid van dezelfde scene. Dat is de bedekkingsmap, waarin het aantal monsters, die op die pixel terechtkomen, en hun totaal opgetelde splatgrootte in opgeslagen zijn. Het groen kanaal bevat die totale splatgrootte. Randen van objecten zijn daar vrij zwart, wat betekent dat de splatgroottes van de monsters, die daarop terechtkomen, klein zijn. Dat komt doordat de tegeling ervoor zorgt dat veel monsters genomen worden op de randen. Als er veel monsters aanwezig zijn in een gebied, gaat de gemiddelde splatgrootte geleidelijk aan achteruit. De figuur is zelfs in het geheel vrij zwart, omdat er overal veel monsters aanwezig zijn bij een statische scene en camera.



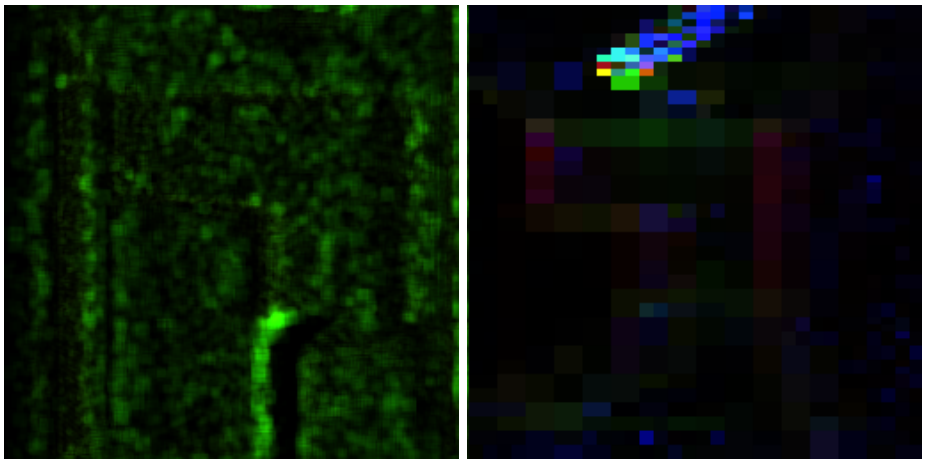
Figuur 4.18: (a) Gemiddelde splatgrootte (b) Gemiddelde gradiënten

Rechts in de figuur (b) zijn de gemiddelde gradiënten weergegeven. Per tegel wordt namelijk de gemiddelde gradiënt genomen van alle monsters binnen dat tegelvolume. Aangezien er zich kleine tegels rond grote kleurvariaties bevinden en daar bijgevolg grote kleurgradiënten aanwezig zijn, zullen deze gradiënten sterk aan de randen geconcentreerd zijn. Hier is de tegeling enkel geconcentreerd op spatiale randen aangezien de scene en camera statisch zijn.



Figuur 4.19: (a) Finaal beeld (b) Kd-boom

Wanneer de camera beweegt, worden bijvoorbeeld figuren 4.19 en 4.20 bekomen. De camera is zich hier snel linksvooruit aan het begeven. Dat is ook te zien aan de temporele gradiënt in 4.20 (b).

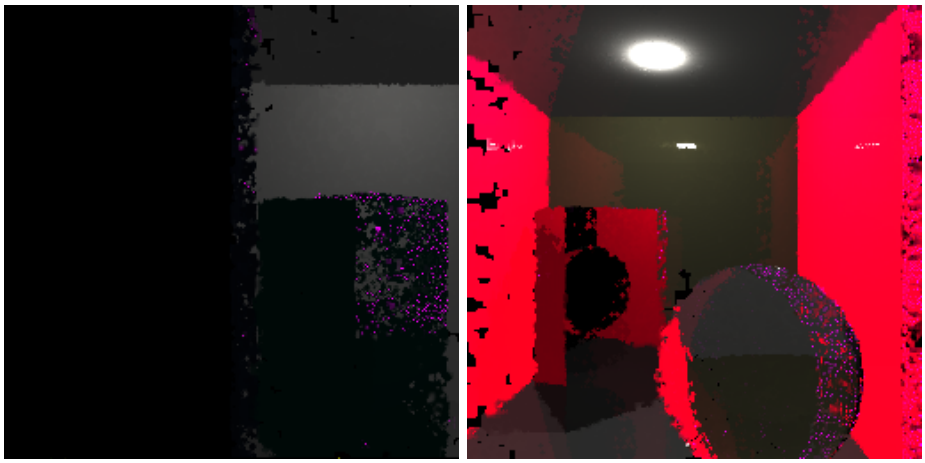


Figuur 4.20: (a) Gemiddelde splatgrootte (b) Gemiddelde gradiënten

In figuur 4.19 is de kd-boom zich reeds aan het concentreren op de occlusie aan de rechterkant van de groene balk. Ook rechtsonder bij de occlusie door de gele balk, is de kd-boom reeds onderverdeeld. In de daaropvolgende beelden

heeft het bemonsteringsproces genoeg monsters genomen om de gaten in de balken op te vullen. In de vroege momenten bij een beweging zijn er dus soms objecten zichtbaar door andere objecten. Dat komt omdat er nog niet genoeg monsters genomen zijn op die plaats om de balk op te vullen.

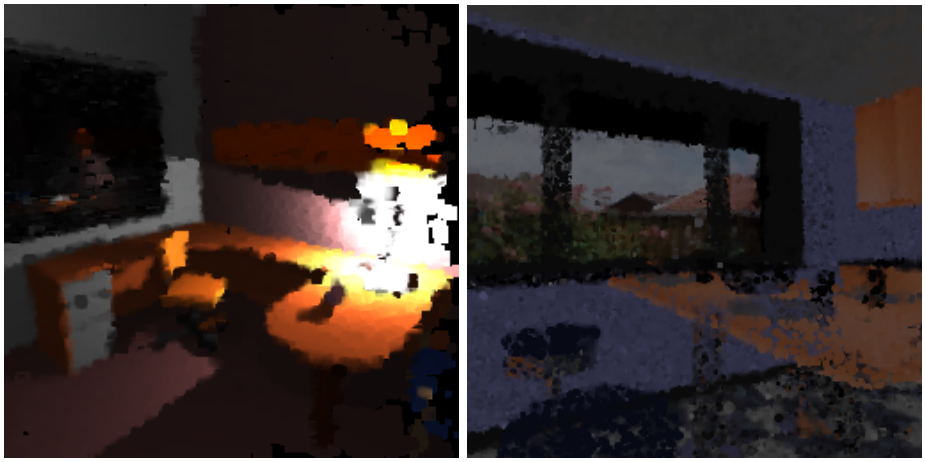
In figuur 4.20 (a) is er een lichtgroene streep te zien op de plaats waar eerst de gele balk weggegaan is en daarna de groene balk is overgeschoven. De tegeling heeft zich hier reeds vroeg geconcentreerd, waardoor de groene balk op die plaats reeds gedeeltelijk opgevuld is en de splatgroottes tijdelijk groot zijn om ervoor te zorgen dat het hele gebied volgezet is. Door de beweging zijn de gemiddelde spatiale gradiënten minder sterk uitgesproken in figuur 4.19 (b). De tegeling is er groter, zodat de gradiënten meer uitgespreid worden. Dat komt omdat de kd-boom zich bij bewegingen minder sterk moet aanpassen aan spatiale details. Het moet de tijd beter bemonsteren en bij de filters voorkeur geven aan recentere monsters, zodat het huidig tijdsstip beter voorgesteld wordt.



Figuur 4.21: Directe bemonstering bij occlusiedetectie

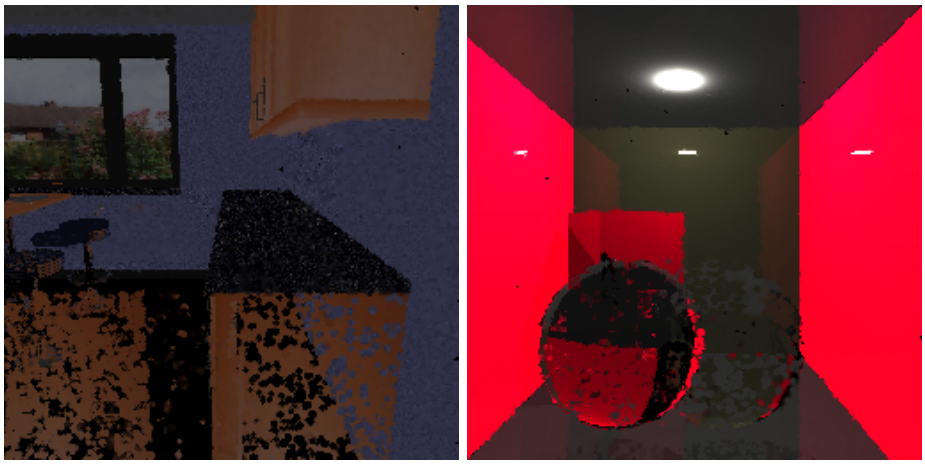
Occlusies worden ontdekt door de herprojectie in het bemonsteringsproces. Een oud monster van achteraan in de buffer (vooraan in de wachtrij dus) komt vooraan terecht in de buffer. Indien dat monster bedekt is, heeft de herprojectie een dergelijke occlusie ontdekt. Er zal dan direct een nieuwe crosshair gegenereerd worden op die positie in de buffer. Figuur 4.21 toont met paarse pixels dergelijke posities aan tijdens een grote camerabeweging. Bij beide figuren is de camera bewogen van rechtsvoor naar linksachter, zodat aan de rechterkant van de objecten occlusies voorvallen.

Figuur 4.23 toont een snelle rotatie (a) en een snelle zijwaartse translatie (b) van de camera. Adaptive frameless rendering produceert snel ruwe resultaten, zoals de figuren tonen. Het gebruikt weinig monsters en grote splatgroottes tijdens dergelijke snelle bewegingen. Daarenboven interpoleert het sterk en neemt het weinig oude monsters op in zijn resultaten aangezien de temporele gradiënt hoog is. Bij de "Kitchen"-scene wordt hier zelfs reflectiediepte drie gebruikt. Het nadeel van deze snelle ruwe resultaten is het feit dat sommige objecten tijdelijk sterk doorzichtig kunnen zijn, wegens een tekort aan monsters.



Figuur 4.22: (a) Camerarotatie (b) Zijwaartse cameratranslatie

Ook bij het verplaatsen van objecten worden snel ruwe resultaten getoond. Wanneer het bemonsteren van de scene duur is en er bijgevolg weinig monsters genomen worden, lijkt het toch even te duren eer het object volledig weg is van zijn originele positie. Figuur 4.23 (a) toont dit. Wanneer de bemonstering van de scene goedkoper is, gaat het vrij snel en gelijkaardiger aan de verplaatsing in de render cache, waarbij de objecten in de cache zelf aangepast moeten worden 4.23 (b).



Figuur 4.23: (a) Objectbeweging in dure scene (b) Objectbeweging in goedkopere scene

Tenslotte is de kwaliteit en de performantie van dit algoritme sterk afhankelijk van verschillende parameters. Zo is er de constante i, die de relatieve belangrijkheid bepaalt van de temporele gradiënten ten opzichte van de spatiale gradiënten. Verder moet bepaald worden hoeveel keer de tegeling aangepast

zal worden, hoeveel keer er herbemonsterd moet worden per nieuwe crosshair, welke grootte de constante m heeft bij het berekenen van de disocclusies en welk gewicht de verschillende onderdelen van de tegelfout krijgen. De constante i is momenteel ingesteld op $0,001$ om het te scaleren naar seconden. De constante m is ingesteld op drie om niet te snel tegels op te delen. Het aantal herprojecties is vijf gebleven en het aantal keren dat de tegeling vernieuwd wordt is ook gelijk gebleven. De onderdelen van de tegelfout krijgen elk 0.33 als gewicht.

Hoofdstuk 5

Conclusie

De thesis is gestart met een korte inleiding in computergrafieken en visualisatie. Daarna is er gesteld dat raytracing steeds aantrekkelijker wordt voor interactieve visualisatie, maar dat het nog steeds vrij zware berekeningen vereist.

Een idee om raytracing te versnellen is het uibuiten van temporele coherentie met behulp van frameless rendering. Er zal dan niet telkens een heel beeld pixel per pixel berekend worden, maar per beeld zal een gedeelte van het aantal pixels herberekend worden met de nieuwste invoerparameters. Op die manier wordt een gladdere animatie bekomen.

Standaard frameless rendering kan bovendien uitgebreid worden of geïntegreerd worden in een visualisatiekader. Zo zijn er reeds meerdere technieken bedacht. In deze thesis is een vergelijking opgesteld en werden er enkele uitgekozen voor implementatie.

De implementaties tonen aan dat het inderdaad mogelijk is om raytracing interactief te visualiseren met behulp van dergelijke benaderingsalgoritmen. Er is daarbij gebruik gemaakt van een voorbeeldanimatie en een vaste beeldresolutie. Dubbele buffering kan voor die animatie ongeveer iets meer dan twee beelden per seconde halen, indien het op beide processoren zou uitgevoerd worden. Twee beelden per seconde levert geen interactieve resultaten. Met behulp van de render cache is het mogelijk ongeveer 25 beelden per seconde te tonen. Dat ligt boven de ondergrens voor animatie van ongeveer 18 beelden per seconde. Deze beelden zijn van een lagere kwaliteit dan die van dubbele buffering. Dat is hier grotendeels te danken aan het feit dat deze animatie sterk varieert. Maar de beelden leveren wel een vloeiende animatie op. Bij trage camerabewegingen is de kwaliteit veel beter, zoals verwacht was. Bij snelle camerabewegingen is de inhoud van sommige beelden minder goed onderscheidbaar.

De GVE-implementatie van de render cache levert zelfs 44 beelden per seconde. De rest van het algoritme blijft gelijk. Bij het herhaaldelijk bemonsteren van de invoer moet er wel opgepast worden dat deze niet teveel tijd in beslag gaat nemen. De camera-invoer is daarbij goedkoop, in tegenstelling tot de animatie van de auto.

De uitbreiding van de render cache voegt een prefilter toe en voorspellend bemonsteren. Deze combinatie werkt goed wanneer de camera een bepaalde richting aanhoudt. Maar bij deze "Kitchen"-animatie zijn er redelijk wat variaties in de richting van de camera, zodat een deel van die bemonsterde waarden

verspilde tijd is. Aan de hand van de geproduceerde beelden wordt dat duidelijk omdat er zwarte gaten verschijnen.

Adaptive frameless rendering is effectiever in het bemonsteren van nieuwe gebieden dan de render cache. Het creëert sneller een ruw beeld. Daarom is deze techniek handiger bij het snel doorwandelen van een scene. Het nadeel is dat bij een lagere bemonsteringssnelheid veel objecten een tijdje doorzichtig zijn. Bij de "Kitchen"-animatie is dit meermaals het geval. De camera beweegt dan ook nogal snel. Deze techniek behaalt 16 beelden per seconde voor dezelfde animatie. Objectbewegingen lukken redelijk goed in scenes, waarin voldoende monsters berekend kunnen worden. In de "Kitchen"-scene heeft de techniek meer last met het verplaatsen van objecten. Een deel van het object komt direct op de juiste positie terecht, maar er blijven artefacten achter. De "rigid body"-transformaties werken goed bij de render cache. Het levert een directe aanpassing van de positie van het object. Enkel de shading kleuren passen zichzelf pas later aan.

Er was vooropgesteld dat deze technieken benaderde beelden genereren tijdens dynamische situaties en volle kwaliteitsbeelden bij statische situaties. De render cache zal steeds zijn interpolatiefilter toepassen en zal daarom een lichte benadering afleveren. Adaptive frameless rendering moet normaal de volle kwaliteit afleveren, aangezien zijn adaptieve filters zich aanpassen aan een dicht bemonsterd gebied. Bij stilstand zal elke pixel dicht bemonsterd zijn en dan worden geen buurmonsters opgenomen.

Deze technieken leveren zoals verwacht de hoogste kwaliteit wanneer kleine bewegingen ondernomen worden. Bij een teleportatie of wanneer men door een muur wandelt, zal er even geen beeld beschikbaar zijn. De tijd, die dan gewacht moet worden, valt hier nog mee.

5.1 Toekomstig Werk

Het menselijk visueel systeem moet zich concentreren op bepaalde posities op het beeld. De posities, waarop het zich concentreert, zullen scherp moeten zijn. De andere gebieden van het beeld, die zich in de periferie bevinden zijn minder belangrijk. Door integratie van eye tracking is het mogelijk te bepalen waarheen een persoon kijkt en bijgevolg die posities beter te bemonsteren. Bij gecompliceerde bewegingen zal het menselijk visueel systeem bovendien meer moeten werken en bijgevolg zullen meerdere andere zaken het ontgaan.

Volgens Dayal et al. [32] kunnen betere filters ontwikkeld worden dan degene die zij momenteel gebruiken in hun adaptieve filterbank. Verder kunnen ook perceptuele kwaliteitsmodellen geïntegreerd worden om de bemonstering voornamelijk te concentreren op de gebieden die voor de mens belangrijk zijn volgens een fijnere onderverdeling dan nu het geval is.

Dankzij frameless rendering is het mogelijk monsters goed verdeeld over de tijd door te sturen naar het beeldscherm, in plaats van telkens een heel beeld samen te stellen. Daarom zou het handig zijn als dergelijke beeldschermen op de markt kwamen.

Bibliografie

[1] http://www.cs.unc.edu/~ibr/pubs.html.

[2] http://100fps.com/how_many_frames_can_humans_see.htm.

[3] Song Ho Ahn. http://www.songho.ca/opengl/gl_vbo.html.

[4] Song Ho Ahn. http://www.songho.ca/opengl/gl_fbo.html.

[5] Kurt Akeley, David Kirk, Larry Seiler, Philipp Slusallek, and Brad Grantham. When will ray-tracing replace rasterization? 2002.

[6] Tomas Akenine-Möller. Fast 3d triangle-box overlap testing. 2001.

[7] Takaaki Akimoto, Kenji Mase, and Yaushito Suenaga. Pixel-selected ray tracing. 1991.

[8] John Amanatides and Andrew Woo. A fast voxel traversal algorithm for ray tracing. 1987.

[9] Joseph Anderson and Barbara Anderson. http://www.uca.edu/org/ccsmi/ccsmi/classicwork/Myth%20Revisited.htm, 1993.

[10] Arthur Appel. Some techniques for shading machine renderings of solids. 1968.

[11] Nils Beck and Andrè*Hinkenjann*. Direc - distributing the render cache to pc-clusters for interactive environments. 2005.

[12] Larry Bergman, Henry Fuchs, Eric Grant, and Susan Spach. Image rendering by adaptive refinement. 1986.

[13] Oliver Bimber. The ultimate display - what will it be? 2005.

[14] Gary Bishop, Henry Fuchs, Leonard McMillan, and Ellen J. Scher Zagier. Frameless rendering: Double buffering considered harmful. 1994.

[15] J. F. Blinn and M. E. Newell. Texture and reflection in computer generated images. *Communications of the ACM*, 19:542–546, 1976.

[16] James F. Blinn. Simulation of wrinkled surfaces. In *Computer Graphics (SIGGRAPH '78 Proceedings)*, volume 12, pages 286–292, August 1978.

[17] Solomon Boulos, Dave Edwards, J. Dylan Lacewell, Joe Kniss, Jan Kautz, Peter Shirley, and Ingo Wald. Interactive distribution ray tracing. 2006.

[18] Solomon Boulos, Dave Edwards, J. Dylan Lacewell, Joe Kniss, Jan Kautz, Peter Shirley, and Ingo Wald. Packet-based whitted and distribution ray tracing. 2007.

[19] J.E. Bresenham. Algorithm for computer control of a digital plotter. 1965.

[20] J.E. Bresenham. Run length slice algorithm for incremental lines. 1985.

[21] Wayne E. Carlson. http://design.osu.edu/carlson/history/timeline.html, 2004.

[22] Nathan A. Carr, Jesse D. Hall, and John C. Hart. The ray engine. 2002.

[23] Edwin E. Catmull. *A Subdivision Algorithm for Computer Display of Curved Surfaces*. PhD thesis, Dept. of CS, U. of Utah, December 1974.

[24] Chih-Chang Chen and Damon Shing-Min Liu. Use of hardware z-buffered rasterization to accelerate ray tracing. 2007.

[25] Shenchang Eric Chen, Holly E. Rushmeier, Gavin Miller, and Douglass Turner. A progressive multi-pass method for global illumination. 25:165–174, July 1991.

[26] James H. Clark. Hierarchical geometric models for visible surface algorithms. 1976.

[27] Robert L. Cook. Stochastic sampling in computer graphics. *ACM Transactions on Graphics*, 5(1):51–72, January 1986.

[28] Robert L. Cook, Thomas Porter, and Loren Carpenter. Distributed ray tracing. 18(3):137–45, July 1984.

[29] Satyan Coorg and Seth Teller. A spatially and temporally coherent object space visibility algorithm. 1996.

[30] Robert A. Cross. Interactive realism for visualization using ray tracing. 1995.

[31] Lucia Darsa, Bruno Costa, and Amitabh Varshney. Walkthroughs of complex environments using image-based simplication. 1998.

[32] Abhinav Dayal, Cliff Woolley, Benjamin Watson, and David Luebke. Adaptive frameless rendering. 2005.

[33] Philippe Decaudin. Cartoon looking rendering of 3D scenes. Research Report 2919, INRIA, June 1996.

[34] Andreas Dietrich, Ingo Wald, Carsten Benthin, and Philipp Slusallek. The openrt application programming interface - towards a common api for interactive ray tracing. 2003.

[35] Mark A.Z. Dippé and Erling Henry Wold. Antialiasing through stochastic sampling. 1985.

[36] Philip Dutre, Kavita Bala, and Philippe Bekaert. *Advanced Global Illumination*. A K Peters, Ltd., 2006.

[37] Randy Fernando, Mark Harris, Matthias Wloka, and Cyril Zeller. Programming graphics hardware. 2004.

[38] Tim Foley and Jeremy Sugerman. Kd-tree acceleration structures for a gpu raytracer. 2005.

[39] Heiko Friedrich, Johannes Günther, Andreas Dietrich, Michael Scherbaum, Hans-Peter Seidel, and Philipp Slusallek. Exploring the use of ray tracing for future games. 2006.

[40] Henry Fuchs, Z. M. Kedem, and Bruce F. Naylor. On visible surface generation by a priori tree structures. 14(3):124–133, July 1980.

[41] Thomas A. Funkhouser and Carlo H. Séquin. Adaptive display algorithm for interactive frame rates during visualization of complex virtual environments. 1993.

[42] Rafael C. Gonzalez and Richard E. Woods. *Digital Image Processing*. Addison-Wesley, second edition, 2002.

[43] Amy Gooch. http://www.cs.utah.edu/npr/.

[44] Cindy M. Goral, Kenneth E. Torrance, Donald P. Greenberg, and Bennett Battaile. Modelling the interaction of light between diffuse surfaces. 18(3):212–22, July 1984.

[45] Henri Gouraud. Continuous shading of curved surfaces. 1971.

[46] Ned Greene, Michael Kassy, and Gavin Miller. Hierarchical z-buffer visibility. 1993.

[47] Roy A. Hall and Donald P. Greenberg. A testbed for realistic image synthesis. 1983.

[48] Peter Houska. Directional lightmaps. 2006.

[49] Henrik Wann Jensen. Photon maps in bidirectional monte carlo ray tracing of complex objects. *Computers & Graphics*, 19(2):215–224, March 1995.

[50] Henrik Wann Jensen and Niels Jørgen Christensen. Global illumination using photon maps. 1996.

[51] Gregory S. Johnson, Juhyun Le, Christopher A. Burns, and William R. Mark. The irregular z-buffer and its application to shadow mapping. 2004.

[52] James T. Kajiya. The rendering equation. 20:143–150, August 1986.

[53] Filip Karlsson and Carl Johan Ljungstedt. Ray tracing fully implemented on programmable graphics hardware. 2004.

[54] Eric P. Lafortune and Yves D. Willems. Bidirectional path tracing. 1993.

[55] Hayden Landis. *Chapter 5: Production-Ready Global Illumination*. 2002.

[56] Gregory Ward Larson. The holodeck: A parallel ray-caching rendering system. 1998.

[57] Gregory Ward Larson and Maryann Simmons. The holodeck ray cache: An interactive rendering system for global illumination in nondiffuse environments. *ACM Transactions on Graphics*, 18(4):361–398, October 1999.

[58] Jonas Lext and Tomas Akenine-Möller. Towards rapid reconstruction for animated ray tracing. 2001.

[59] Jonas Lext, Ulf Assarsson, and Tomas Möller. http://www.ce.chalmers.se/research/group/graphics/BART/.

[60] Jonas Lext, Ulf Assarsson, and Tomas Möller. Bart: A benchmark for animated ray tracing. 2000.

[61] R. Beau Lotto, S.Mark Williams, and Dale Purves. Mach bands as empirically derived associations. 96:5245–5250, April 1999.

[62] Karen A. Manning. Eye-movement-dependent loss in vision and its time course during vergence. 1986.

[63] William R. Mark, Leonard McMillan, and Gary Bishop. Post-rendering 3d warping. 1997.

[64] Tomas Möller and Ben Trumbore. Fast, minimum storage ray triangle intersection. 1997.

[65] Karol Myszkowski, Takehiro Tawara, Hiroyuki Akamine, and Hans-Peter Seidel. Perception-guided global illumination solution for animation rendering. 2001.

[66] M.E. Newell, R.G. Newell, and T.L. Sancha. A solution to the hidden surface problem. 1972.

[67] Bill Nichols and Susan J. Lederman. http://www.grand-illusions.com/articles/persistence_of_vision/.

[68] Juri A. Oudshoorn. Ray tracing as the future of computer games. 1999.

[69] Rick Parent. *Computer Animation: Algorithms and Techniques*. Elsevier Science (USA), 2002.

[70] Steven Parker, William Martin, Peter-Pike J. Sloan, Peter Shirley, Brian Smits, and Charles Hansen. Interactive ray tracing. 1999.

[71] Kadir A. Peker and Ajay Divakaran. Adaptive fast playback-based video skimming using a compressed-domain visual complexity measure. 2004.

[72] Bui Tuong Phong. Illumination for computer generated pictures. *Communications of the ACM*, 18(6):311–317, June 1975.

[73] Lynn Pocock and Judson Rosebush. *The Computer Animator's Technical Handbook*. Morgan Kaufmann, 2001.

[74] Timothy J. Purcell, Ian Buck, William R. Mark, and Pat Hanrahan. Ray tracing on programmable graphics hardware. 2002.

[75] Erik Reinhard, Peter Shirley, and Charles Hansen. Parallel point reprojection. 2001.

[76] Alexander Reshetov, Alexei Soupikov, and Jim Hurley. Multi-level ray tracing algorithm. 2005.

[77] Philippe C.D. Robert, Severin Schoepke, and Hanspeter Bieri. Ray tracing using gpu-accelerated image-space methods. 2007.

[78] J.G. Rokne, B. Wyvill, and X. Wu. Fast line scan-conversion. 1990.

[79] S. M. Rubin and T. Whitted. A 3-dimensional representation for fast rendering of complex scenes. *Computer Graphics*, 14(3):110–116, July 1980.

[80] Mutsuo Saito. An application of finite field: Design and implementation of 128-bit instruction-based fast pseudorandom number generator. 2007.

[81] William J. Schroeder, Jonathan A. Zarge, and William E. Lorensen. Decimation of triangle meshes. 1992.

[82] Daniel E. Sevo. http://hem.passagen.se/des/hocg/hocg_1960.htm, 2008.

[83] Jonathan Shade, Dani Lischinski, David H. Salesin, Tony DeRose, and John Snyder. Hierarchical image caching for accelerated walkthroughs of complex environments. 1996.

[84] William Shoaff. http://cs.fit.edu/~wds/classes/graphics/History/history/history.html, 2000.

[85] François Sillion, George Drettakis, and Benoit Bodelet. Efficient impostor manipulation for real-time visualization of urban scenery. 16:207–218, Sep 1997.

[86] Francois X. Sillion and Claude Puech. A general two-pass method integrating specular and diffuse reflection. 23:335–344, July 1989.

[87] Maryann Simmons. A dynamic mesh display representation for the holodeck ray cache system. 2000.

[88] Maryann Simmons and Carlo H. Séquin. Tapestry: A dynamic mesh-based display representation for interactive rendering. 2000.

[89] Brian Smits. Efficiency issues for ray tracing. 1998.

[90] Marc Stamminger, Jörg Haber, Hartmut Schirmacher, and Hans-Peter Seidel. Walkthroughs with corrective texturing. 2000.

[91] Seth J. Teller and Carlo H. Séquin. Visibility preprocessing for interactive walkthroughs. 1991.

[92] Parag Tole, Fabio Pellacini, Bruce Walter, and Donald P. Greenberg. Interactive global illumination in dynamic scenes. 2002.

[93] A.J. van der Ploeg. Interactive ray tracing: The replacement of rasterization? 2006.

[94] Eric Veach and Leonidas J. Guibas. Metropolis light transport. 1997.

[95] Edgar Velázquez-Armendáriz, Eugene Lee, Kavita Bala, and Bruce Walter. Implementing the render cache and the edge-and-point image on graphics hardware. 2006.

[96] Ingo Wald. *Realtime Ray Tracing and Interactive Global Illumination*. PhD thesis, 2004.

[97] Ingo Wald, Carsten Benthin, and Philipp Slusallek. A flexible and scalable rendering engine for interactive 3d graphics. 2002.

[98] Ingo Wald, Carsten Benthin, and Philipp Slusallek. A simple and practical method for interactive ray tracing of dynamic scenes. 2002.

[99] Ingo Wald, Thomas Kollig, Carsten Benthin, Alexander Keller, and Philipp Slusallek. Interactive global illumination. 2002.

[100] Ingo Wald, Thomas Kollig, Carsten Benthin, Alexander Keller, and Philipp Slusallek. Interactive global illumination using fast ray tracing. 2002.

[101] Ingo Wald, Timothy J. Purcell, Jörg Schmittler, Carsten Benthin, and Philipp Slusallek. Realtime ray tracing and its use for interactive global illumination. 2003.

[102] Ingo Wald and Philipp Slusallek. State of the art in interactive ray tracing. 2001.

[103] Ingo Wald, Philipp Slusallek, Carsten Benthin, and Markus Wagner. Interactive rendering with coherent ray tracing. 2001.

[104] John R. Wallace, Michael F. Cohen, and Donald P. Greenberg. A two-pass solution to the rendering equation: A synthesis of ray tracing and radiosity methods. 21:311–320, July 1987.

[105] Bruce Walter, George Drettakis, and Donald P. Greenberg. Enhancing and optimizing the render cache. 2002.

[106] Bruce Walter, George Drettakis, and Steven Parker. Interactive rendering using the render cache. 1999.

[107] Greg Ward. http://radsite.lbl.gov/mgf/HOME.html.

[108] Matthew Ward. http://web.cs.wpi.edu/~matt/courses/cs563/talks/history.html.

[109] J. Warnock. A hidden-surface algorithm for computer generated half-tone pictures. Technical Report TR 4–15, NTIS AD-733 671, University of Utah, Computer Science Department, 1969.

[110] Andrew B. Watson, Jr. Albert J. Ahumada, and Joyce E. Farrell. Window of visibility: A psychophysical theory of fidelity in time-sampled visual motion displays. 1986.

[111] Benjamin Watson and David Luebke. The ultimate display - where will all the pixels come from. 2005.

[112] Turner Whitted. An improved illumination model for shaded display. *Communications of the ACM*, 23(6):343–349, June 1980.

[113] Amy Williams, Steve Barrus, R. Keith Morley, and Peter Shirley. An efficient and robust ray-box intersection algorithm. 2005.

[114] Lance Williams. Casting curved shadows on curved surfaces. 12(3):270–274, August 1978.

[115] Mathias M. Wloka, Robert C. Zeleznik, and Timothy Miller. Practically frameless rendering. 1995.

[116] Matthias M. Wloka and Robert C. Zeleznik. Interactive real-time motion blur. 1996.

[117] Ellen J. Scher Zagier. Defining and refining frameless rendering. 1997.

[118] Ellen J. Scher Zagier. A human's eye view: Motion blur and frameless rendering. 1997.

[119] Ellen J. Scher Zagier. Frameless antialiasing. Technical report, 1998.

[120] Hansong Zhang. Effective occlusion culling for the interactive display of arbitrary models. 1998.

[121] Tenghui Zhu, Rui Wang, and David Luebke. A gpu-accelerated render cache. 2005.