Hoge kwaliteits punt-gebaseerde weergave op grafische hardware

Benny Giesbers

Promotor:Philippe Bekaert
Begeleider:Erik Hubo

6 juni 2005

Abstract

De laatste jaren is de interesse in punt-gebaseerde weergave sterk toegenomen. De hoofdredenen hiervoor zijn de groeiende complexiteit van de geometrische modellen en de verbetering van 3D scanapparaten. Het grote voordeel van een punt-gebaseerde representatie is de simpliciteit van de punten. Er hoeft geen connectiviteitsinformatie bijgehouden te worden en er moet geen rekening gehouden worden met speciale topologische gevallen, zoals bij tradionele weergave met polygonen. Er is de laatste jaren al veel onderzoek geweest naar het flexibel en kwalitatief weergeven van punt-gebaseerde modellen. Een groot nadeel hiervan is dat er veel intensieve berekeningen gedaan moeten worden door de processor, omdat grafische hardware geöptimaliseerd is voor weergave met polygonen. De recente grafische hardware is flexibel en programmeerbaar en daardoor geschikt om deze taak voor een groot deel over te nemen. In deze thesis zullen enkele belangrijke methoden van software -en hardware-matig punt-gebaseerde weergave besproken en vergeleken worden. Verder zal een implementatie die gebruikt maakt van de recente grafische hardware besproken worden, met een resultaatsbespreking en een conclusie.

Voorwoord

Deze thesis is geschreven ter voltooiing van mijn studie Informatica - optie Multimedia aan de transnationale Universiteit Limburg.

Ik zou graag enkele mensen bedanken die me geholpen helpen bij het tot stand brengen van de thesis. In de eerste plaats mijn promotor prof. dr. Phillipe Bekaert zonder wie deze thesis nooit tot stand zou zijn gekomen. Mijn begeleider Erik Hubo, voor zijn enorme steun, zijn talloze tips en het nalezen van mijn tekst. Ook wil ik mijn medestudenten bedanken voor de steun en vele tips tijdens het implementeren. Tenslotte wil ik ook mijn familie, mijn vrienden en vooral mijn vriendin Jennifer bedanken voor de grote steun en het nalezen van mijn thesistekst.

Lijst van figuren

2.1 Van triangles naar punten 4
2.2 Een puntmodel 5
2.3 Surfel met basiseigenschappen 5
2.4 Surfel met bijkomende gegevens 6

3.1 Q-Splat rendermethoden 15
3.2 Mapping van objectspace naar screenspace 17
3.3 EWA filtering proces 18
3.4 Gaten in weergegeven oppervlak door quantisatie 21

4.1 Loodrechte foutwaarde bij Sequential Point Trees 27
4.2 Divergerende foutwaarde bij Sequential Point Trees 28
4.3 Overzicht van de methode van Botsch 33
4.4 Overzicht van de methode van Guennebaud 35

5.1 Structuur van de tijdelijke octree 44

6.1 Stanford Bunny gerenderd op kleine en grote afstand van camera 57
6.2 Draakmodel gerenderd op kleine en grote afstand van camera . 58
6.3 Boeddha-model gerenderd op kleine en grote afstand van camera 61
6.4 Gaten in dieptebuffer door perspectieve benadering 62

Lijst van tabellen

6.1 Performantie bij weergave in lage kwaliteit 56
6.2 Performantie bij weergave in hoge kwaliteit 56
6.3 Performantie van de voorbereidingsfase 60

Inhoudsopgave

1 Inleiding **1**
- 1.1 Doel . . . 1
- 1.2 Overzicht . . . 1
- 1.3 Indeling . . . 2

2 Basisbegrippen **4**
- 2.1 Surfels . . . 4
- 2.2 Shaders . . . 6
  - 2.2.1 Vertex shaders . . . 6
  - 2.2.2 Fragment shaders . . . 7

3 Software **8**
- 3.1 Inleiding . . . 8
- 3.2 Datastructuur . . . 9
  - 3.2.1 Surfel Rendering . . . 9
  - 3.2.2 QSplat . . . 10
  - 3.2.3 Surface Splatting . . . 11
  - 3.2.4 Compacte representatie . . . 11
- 3.3 Rendering/weergave . . . 13
  - 3.3.1 Surfel Rendering . . . 13
  - 3.3.2 QSplat . . . 13
  - 3.3.3 Surface Splatting . . . 16
  - 3.3.4 Compacte representatie . . . 21
- 3.4 Algemene vergelijking . . . 22

4 Hardware **24**
- 4.1 Inleiding . . . 24
- 4.2 Datastructuur . . . 26
  - 4.2.1 Methode van Botsch . . . 26
  - 4.2.2 Methode van Guennebaud . . . 26
  - 4.2.3 Sequential Point Trees . . . 26

4.2.4 Deferred Splatting 29
4.2.5 Phong Splatting 29
4.3 Rendering/weergave 30
4.3.1 Methode van Botsch 30
4.3.2 Methode van Guennebaud 32
4.3.3 Sequential Point Trees 34
4.3.4 Deferred Splatting 37
4.3.5 Phong Splatting 38
4.4 Algemene vergelijking 40

5 Implementatie **42**
5.1 Inleiding 42
5.2 Datastructuur 43
5.2.1 Point-struct 43
5.2.2 Inlezen punten 43
5.2.3 Omzetting naar tijdelijk octree 44
5.2.4 Omzetting naar Sequential Point Tree 46
5.3 Rendering 50
5.3.1 Eerste weergavestap 50
5.3.2 Tweede weergavestap 51
5.3.3 Normalisatiestap 52

6 Resultaat **54**
6.1 Modellen 54
6.2 Performantie 55
6.3 Weergave in lage kwaliteit 55
6.4 Weergave in hoge kwaliteit 55
6.4.1 Splatgrootte 56
6.4.2 Schermresolutie en modelgrootte 57
6.4.3 Perspectiefbenadering 59
6.4.4 Sequential Point Trees 59
6.4.5 Preprocessing bij SPT 60

7 Conclusie **63**

Hoofdstuk 1

Inleiding

1.1 Doel

Het doel van deze thesis is om punt-gebaseerd renderen in het algemeen te bestuderen en om aan te tonen dat punt-gebaseerd renderen met een hoge kwaliteit mogelijk is op moderne grafische hardware. De thesis bestaat uit twee delen : enerzijds een literatuurstudie en anderzijds een implementatie. In de literatuurstudie worden de verschillende methoden alsook de voor- en nadelen van punt-gebaseerd renderen besproken, zowel voor software-als hardwarematig renderen. Er wordt begonnen met de software-matige methoden die dan gedeeltelijk als basis gebruikt worden bij de hardware-matige methoden.

In het tweede deel wordt getracht een implementatie te maken die punt-gebaseerde modellen met een hoge kwaliteit kan weergeven, op de moderne grafische hardware.

1.2 Overzicht

Al in 1974 stelde Catmull in [Cat74] dat geometrische subdivisies uiteindelijk zouden leiden tot het gebruik van punten als weergaveprimitieven. In 1985 stelde Levoy in [LW85] voor om punten te gebruiken in plaats van polygonen. De laatste jaren is de interesse in punt-gebaseerd renderen sterk gegroeid. Er zijn hiervoor 2 hoofdredenen : Ten eerste is de groeiende geometrische complexiteit van de modellen sterk gestegen ([ZPvBG01]). Deze tendens is bijvoorbeeld sterk te merken in de spelletjes -en filmindustrie. Zo bevat bijvoorbeeld een model in de Unreal 3 Engine meer polygonen dan een volledige level uit de originele Unreal.

Alhoewel driehoeken als weergaveprimitief de perfecte balans bieden tussen omschrijvende kracht en computationele last ([Dee93]), zijn er nog veel meer driehoeken nodig om realistische, organische en hoog-complexe modellen weer te geven dan dat er momenteel gebruikt worden in de spelletjes -en filmindustrie. Of zoals Alvy Ray Smith het zei in [Smi99] :

> Reality is 80 million polygons.

De grote overhead bij het beheren, verwerken en manipuleren van deze massa's aan gegevens heeft bij vele onderzoekers tot de vraag geleid of polygonen wel ideaal zijn als basis voor grafische representaties. De tweede reden is de ontwikkeling van 3D-fotografie -en scansystemen, die tegenwoordig in staat zijn om gigantische, gedetailleerde volumes van punten te genereren uit complexe voorwerpen ([LPC+00] en [MPN+02]).

Het grote voordeel aan punt-gebaseerd renderen is de simpliciteit van de punten. Er moet geen connectiviteitsinformatie bijgehouden worden en geen rekening gehouden geworden met speciale topologische gevallen. Er is ondertussen al heel wat onderzoek verricht naar software-matig punt-gebaseerd renderen. Het is mogelijk om beelden met hoog detail te renderen, met technieken als geavanceerde shading, anti-aliasing en transparantie.

Het nadeel aan software-matig renderen is echter dat deze methode een groot deel van de CPU-kracht vereist. Bij het renderen met de GPU is de CPU vrij voor andere taken, wat het in de praktijk een pak bruikbaarder maakt. Aanvankelijk was punt-gebaseerd renderen met de GPU niet mogelijk, omdat de grafische hardware volledig geöptimaliseerd is voor het renderen van polygonen. Tegenwoordig is de grafische hardware flexibeler en programmeerbaarder geworden, waardoor het mogelijk wordt om punt-gebaseerd te renderen op de GPU.

1.3 Indeling

Enkele belangrijke basisbegrippen die doorheen de thesis gebruikt zullen worden, worden toegelicht in hoofdstuk 2. De methoden om punt-gebaseerde weergaven te realiseren worden onderverdeeld in twee groepen : software-matige en hardware-matige methoden. De groep van software-matige methoden bestaat al enkele jaren en er is al veel onderzoek naar verricht. Deze algoritmen vormen de basis voor de hardware-matige methoden en worden besproken in hoofdstuk 3. De groep van hardware-matige methoden maken gebruik van de flexibiliteit en programmeerbaarheid van recente grafische

hardware. De *hardware-accelerated* algoritmen worden besproken in hoofdstuk 4. Deze drie hoofdstukken samen vormen de literatuurstudie van deze thesis. De implementatie van deze thesis wordt besproken in hoofdstuk 5 en het resultaat hiervan in hoofdstuk 6. Tenslotte wordt er een conclusie gegeven in hoofdstuk 7.

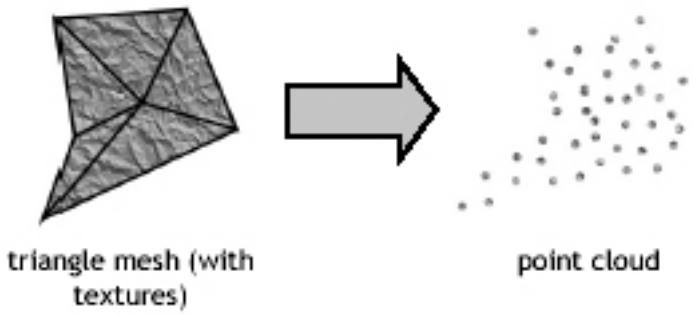
Hoofdstuk 2

Basisbegrippen

In dit onderdeel zullen enkele basisbegrippen besproken worden. Deze begrippen zullen in de rest van de tekst gebruikt worden en kunnen misschien nieuw zijn voor de lezer.

2.1 Surfels

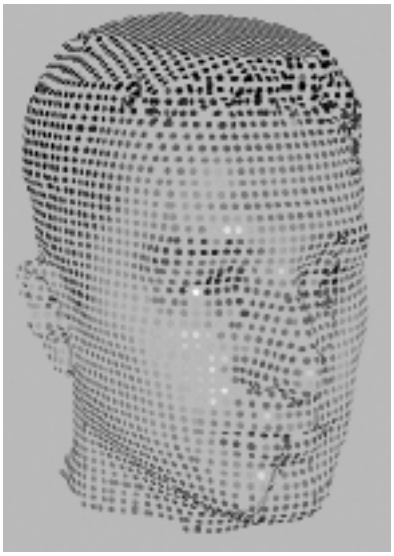
Catmull constateerde al in 1974 in [Cat74] dat geometrische subdivisies uiteindelijk zouden leiden tot het gebruik van punten als weergaveprimitieven. In [LW85] wordt voor het eerst voorgesteld om punten te gebruiken in plaats van polygonen. Het verschil tussen punten en driehoeken in 3D is analoog met het verschil tussen pixels en vector graphics in 2D.



Figuur 2.1: Van triangles naar punten

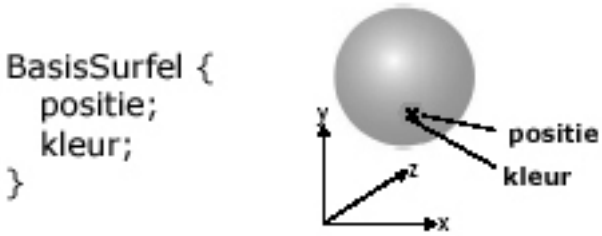
Een oppervlak of verzameling van driehoeken wordt gediscretiseerd tot een puntenwolk (of verzameling van punten) zoals in figuur 2.1. Een punt uit een puntenwolk beschrijft de geometrie en de materiaaleigenschappen van

een plaats op het oppervlak. Een groot voordeel tegenover polygoonweergave is dat informatie zoals connectiviteitsinformatie of textuurmaps niet hoeft bijgehouden te worden. De punten staan dus volledig los van elkaar, zoals in figuur 2.2.



Figuur 2.2: Een puntmodel

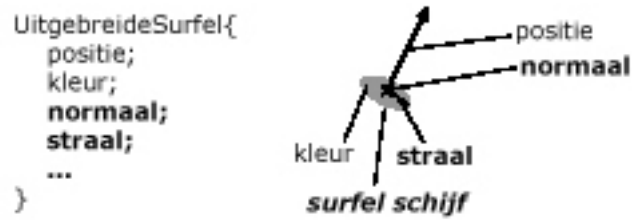
Elk punt komt overeen met een surfel of surface element. Het principe van surfels wordt voorgesteld in [PZvBG00]. Een surfel omschrijft een klein deel van het oppervlak. Een simpele basissurfel (figuur 2.3) houdt dus de positie en de kleur bij.



Figuur 2.3: Surfel met basiseigenschappen

Om een hogere kwaliteitsweergave te bekomen moeten er bijkomende gegevens in de surfels opgeslagen worden, zoals in figuur 2.4. De normaal

kan bijvoorbeeld gebruikt worden om shading te berekenen, die dan gecombineerd wordt met de kleur.



Figuur 2.4: Surfel met bijkomende gegevens

Surfels zijn de primitieven van het punt-gebaseerd weergeven : ze bevatten al de nodige informatie. Het renderen met surfels bevindt zich tussen het geometrische renderen met polygonen en het image-based renderen (IBR) in.

Surfels worden ook splats genoemd, alhoewel deze term meer wordt gebruikt voor weergegeven surfels. Het weergeven van surfels wordt ook splatting genoemd.

2.2 Shaders

De recentere grafische kaarten zijn flexibeler en programmeerbaarder dan hun voorgangers. Dit programmeren van de grafische kaart kan gedaan worden met zogenaamde shaders, kleine programma's die toelaten om hetgeen gerenderd wordt aan te passen. Er zijn twee soorten shaders : vertex shaders en fragment shaders.

2.2.1 Vertex shaders

Vertex shaders werken op de vertices, hetgeen waaruit de 3D wereld is opgebouwd. Een vertex shader bevat een aantal instructies die op elke vertex van de wereld wordt toegepast. Eigenschappen als positie, normaal, kleur, puntgrootte, ... kunnen hierbij aangepast worden. Deze instructies kunnen geschreven worden in assembly code of een hogere-niveau programmeertaal als CG of GLSL. Wanneer een vertex shader gebruikt wordt, dan wordt een deel van de weergave pijplijn vervangen door deze vertex shader. Een vertex shader kan enkel vertices manipuleren, niet aanmaken of verwijderen.

2.2.2 Fragment shaders

Fragment shaders (of ook wel pixel shaders genoemd) zijn vergelijkbaar met vertex shaders. Het grote verschil is dat de fragment shader een stap verder in de weergave pijplijn zitten dan de vertex shaders. Waar een vertex shader op vertices werkt, werkt een fragment shader op pixels. Een fragment shader kan dus gebruikt worden om de kleur van een pixel aan te passen of om bijvoorbeeld effecten als schaduw of belichting per-pixel te berekenen.

Hoofdstuk 3

Software

In dit hoofdstuk wordt een overzicht gegeven van enkele belangrijke methoden om software-matig punt-gebaseerde modellen weer te geven. Eerst wordt er nu een kort overzicht gegeven van deze methoden, daarna worden de datastructuren besproken. Vervolgens worden de weergave-methoden besproken en tenslotte wordt er een korte samenvattende vergelijking gegeven.

3.1 Inleiding

De eerste methode die besproken wordt is de *surfel rendering*-methode van Pfister. Deze werd voorgesteld in [PZvBG00] en het is het eerste algoritme dat punt-gebaseerde modellen met hoge kwaliteit weergeeft. Tot dan toe was er al onderzoek geweest naar afzonderlijke aspecten zoals efficiënte datastructuren en textuurfiltering. In [PZvBG00] worden deze technieken gecombineerd en uitgebreid tot een algoritme dat surfels gebruikt om op een efficiënte en snelle manier complexe geometrische objecten weer te geven. De term „surfels" werd voor het eerst gebruikt in deze paper ([PZvBG00]). Deze methode wordt besproken in 3.2.1 en 3.3.1.

De tweede methode die besproken wordt is QSplat. QSplat werd voorgesteld door Rusinkiewicz in [RL00] en is een belangrijk werk op het gebied van punt-gebaseerd renderen. QSplat gebruikt een snelle en compacte representatie, wat het geschikt maakt voor grote datasets en gebruik op desktop pc's. Deze methode wordt besproken in 3.2.2 en 3.3.2.

De derde methode is de *surfel splatting*-methode van Zwicker. Deze meth-

ode werd voorgesteld in [ZPvBG01] en is in staat om met een weergave met hoge kwaliteit te tonen. De methode is gebaseerd op de Elliptical Weighted Average filter of EWA filter (van Heckbert [Hec86]), een heel belangrijke anisotropisch textuurfilterings-algoritme. De EWA-filtering geeft een gerenderd beeld met een minimum aan aliasing-artefacten. Deze methode bouwt ook verder op [PZvBG00], een paper van dezelfde auteurs en besproken in 3.2.1 en 3.3.1. *Surfel splatting* wordt besproken in 3.2.3 en 3.3.3.

De vierde (en laatste software-matige) methode werd door Botsch voorgesteld in [BWK02]. Het accent hiervan ligt vooral op de datastructuur en het gebruikt een hierarchische representatie die tegelijkertijd heel efficient is op het gebied van opslag, weergavesnelheid en kwaliteit. De representatie is octree-gebaseerd en heeft een optimale balans tussen quantisatieprecisie en bemonsteringsdichtheid/resolutie. Deze methode is sneller en compacter dan QSplat ([RL00]) en de *surfel splatting*-methode van Zwicker ([ZPvBG01]) en wordt besproken in 3.2.4 en 3.3.4.

3.2 Datastructuur

3.2.1 Surfel Rendering

In deze subsectie wordt de datastructuur besproken die Pfister gebruikt in [PZvBG00]. Een bemonsteringsproces converteert geometrische objecten en diens texturen naar surfels. Hiervoor wordt een octree-gebaseerde representatie van het object gemaakt. Raycasting wordt gebruikt om 3 orthogonale LDI's (layered depth images) te maken, die samen een LDC (layered depth cube) vormen. Een LDI is een beeld van het object vanuit een bepaald camera-standpunt, met meerdere surfels langsheen elke gezichtslijn (1 surfel voor elke intersectie). Vanuit deze LDC wordt een LDC-boom gemaakt. De LDC-boom is gebaseerd op de LDI-boom, een octree met een LDI aan elke knoop. [PZvBG00] past deze methode aan door aan elke knoop of blok een LDC in plaats van een LDI te hangen, vandaar de benaming LDC-boom. De volgende stap, genaamd de 3-naar-1 reductie, herleidt de LDC's aan elke knoop tot LDI's door middel van een dichtste buur interpolatie. Deze optimalisatie heeft een positief effect op opslagruimte en renderingstijd. Deze LDI's worden ook wel surfel mipmaps genoemd, door hun gelijkenis met textuurmipmaps.

De LDI, LDC en LDI-tree komen oorspronkelijk uit het Image-Based renderen, waar deze hierarchische structuur als representatie van de scene

gebruikt wordt en een hoge kwaliteit zonder gaten garandeert.

3.2.2 QSplat

QSplat ([RL00]) gebruikt een enkele datastructuur voor culling, level-of-detail selectie en het weergeven. De progressieve representatie is compact en kan snel berekend worden, wat het geschikt maakt voor grote datasets. De representatie bestaat uit een hierarchische structuur van *bounding spheres*.

Elke knoop van de boom bevat:

- de positie van het midden van de sphere
- de straal van de sphere
- de normaal
- de breedte van de normaalkegel (enkel voor interne knopen)
- optioneel de kleur

Deze structuur is gelijkaardig met de surfel rendering van [PZvBG00]. QSplat bestaat essentiaal uit 2 fases: een voorbereidingsfase (*preprocessing*) en een weergavefase.

De hierarchie van bounding spheres kan gegenereerd worden uit puntenwolken. Uit deze puntenwolken worden de posities voor elk punt overgenomen en de straal zo berekend dat er geen gaten zichtbaar zijn tijdens het renderen. De normaal wordt bepaald met behulp van een vlak dat de vertices omvat in een kleine omgeving van het betreffende punt. Na deze stap zijn alle bladknopen van de hierarchische structuur berekend.

De volgende stap bestaat erin om de andere knopen te bepalen en zo de boom te vervolledigen. Dit algoritme ziet er als volgt uit:

```
BuildTree(vert[begin..end])//vert contains all vertices
{
 if (begin==end)
     return Sphere(vert[begin])
 else
     midpoint = PartitionAlongLongestAxis(
                                    vert[begin..end])
     leftsubtree = BuildTree(vert[begin..midpoint])
     rightsubtree = BuildTree(vert[midpoint+1..end])
```

```
    return BoundingSphere(leftsubtree, rightsubtree)
}
```

Het verdeelt de set van vertices/punten volgens de langste as van de omvattende box van deze set op een recursieve manier, totdat het aantal vertices klein genoeg is. Knoopeigenschappen als kleur en normaal worden ingesteld op het gemiddelde van de kinderen. Ook nog worden er nodes samengevoegd, om zo het aantal vertakkingen per knoop op te krikken tot ongeveer 4, waardoor de uiteindelijke boom kleiner zal zijn.

De laatste stap van de preprocessing fase is het quantiseren van knoopeigenschappen. Een knoop zal hierdoor nog maar 48 bits geheugen nodig hebben. Positie en straal (samen 13 van de 48 bits) worden relatief ten opzichte van de parent opgeslagen. Voor de normaal (14 van de 48 bits) wordt er een opzoekingstabel gebruikt met daarin meer dan 16000 normalen. De quantisatiefout bij de normalen zal nooit groter zijn dan 0.01 radialen. Het grote nut van deze quantisatie is dat de hoeveelheid data veel verminderd wordt, waardoor de Q-Splat datastructuur zeer geschikt is voor grote datasets. Het nadeel is dat er meer verwerking moet gebeuren: het encoderen tijdens het voorbereiden en het *on-the-fly* decoderen tijdens het weergeven.

3.2.3 Surface Splatting

Omdat de *surface splatting*-methode vooral op de weergave is toegespitst, is deze weinig vernieuwend op het gebied van datastructuur. Er wordt in [ZPvBG01] dezelfde hiërarchische datastructuur gebruikt als in [PZvBG00] (besproken in 3.2.1 en 3.3.1). Deze biedt een progressief weergavealgoritme dat meerdere resoluties (*multiresolutie*) ondersteunt.

3.2.4 Compacte representatie

Het algoritme voorgesteld door Botsch in [BWK02] biedt een zeer efficiënte en compacte manier om de data op te slaan. De gebruikte quantisatiemethode en datastructuur lijkt veel op die van QSplat ([RL00]), maar gaat agressiever te werk in de zin dat de benadering die gebruikt wordt een optimale balans geeft tussen quantisatiefout en bemonsteringsdichtheid. Door *entropy encoding* gecombineerd met het enkel opslaan van de meest belangrijke bits in de hierarchische structuur kan de opslagruimte per surfelpositie gereduceerd van 3 bytes tot minder dan 2 bits.

De datastructuur is octree-gebaseerd en maakt gebruik van de hiërarchie

om de data op te slaan (data wordt relatief ten opzichte van de parent opgeslagen). Er worden geen omvattende boxen gebruikt. De posities van de punten worden expliciet geëncodeerd als het midden van de cel in de knopen van de octree. Om tijdens het weergeven (back-face) culling mogelijk te maken, wordt er in de (niet-blad-)knopen van de octree een normaalkegel bijgehouden (2 bits per normaalkegel, zoals bij QSplat ([RL00]). Ook de normaal en kleur-informatie worden gequaantiseerd tot minder dan respectievelijk 2 en 1 bytes per punt.

3.3 Rendering/weergave

3.3.1 Surfel Rendering

Bij de weergave van het algoritme van Pfister ([PZvBG00]) wordt de LDC-boom op een recursieve manier doorlopen, beginnende bij de top. De top komt overeen met de blocks met de laagste resolutie en einde van de boom (=de bladknopen) bevat de blocks met de hoogste resolutie. Tijdens de recursiestappen wordt er *view frustum* en *backface culling* toegepast. Deze twee technieken worden in meer detail besproken in het hoofdstuk over QSplat (zie 3.3.2). De dichtheid van de surfels wordt geschat om zo controle te krijgen over de snelheid en kwaliteit van het weergeven.

Om het zichtbaarheidsprobleem op te lossen wordt een z-buffer en zichtbaarheidssplatten of *visibility splatting* gebruikt. Bij deze techniek worden de schijven van de surfels enkel naar de z-buffer weergegeven in een eerste stap. Deze informatie wordt daarna dan gebruikt bij het echte weergeven in een tweede stap.

De kleuren van de zichtbare surfels worden met lineare interpolatie berekend uit de verschillende surfel mipmaps. Voor belichting wordt er ook nog shading toegepast. Normaal wordt deze stap al veel eerder uitgevoerd, maar door het zichtbaarheidstesten is er nu minder werk (omdat er minder punten zijn). Tenslotte wordt het probleem van visuele artefacten als aliasing en gaten aangepakt. Tijdens het zichtbaarheidssplatten werden gaten gemarkeerd. Deze gaten worden dan gevuld met dichtste buur interpolatie met een Gaussiaanse filter.

3.3.2 QSplat

In deze sectie zal het renderen van de Q-Splat ([RL00]) datastructuur besproken worden. Dit recursieve algoritme zorgt voor *frustum* en *backface culling*, level-of-detail selectie. Kort samengevat ziet het algoritme er als volgt uit :

```
TraverseHierarchy(node)
{
 if (node is visible) //frustum & backface culling
     Skip this branch
 else if (node is leafnode)
     Draw a splat
 else if (size on screen < threshold) //level of detail
     Draw a splat
 else
     for each child in children(node) //recursive step
```

```
            TraverseHierarchy(child)
}
```

Tijdens het recursieve weergeven wordt er bij elke knoop aan *frustum* en *backface culling* gedaan :

- **Frustum culling**: Elke bounding sphere wordt vergeleken met de vlakken van het view frustum. Er zijn dan 3 mogelijkheden:
  - Ofwel ligt de bounding sphere volledig buiten het view frustum. In dit geval wordt de huidige knoop en diens kinderen niet weergegeven.
  - Als de bounding sphere volledig binnen het view frustum valt, dan gaat het recursief weergeven gewoon verder en wordt er geen frustum culling meer gedaan op de kinderen.
  - Het laatste geval is dat de bounding sphere gedeeltelijk binnen het frustum valt. In dit geval wordt er ook verder gegaan en zal er frustum culling uitgevoerd worden op de kinderen, aangezien het nog niet zeker is of deze binnen of buiten het frustum vallen.
- **Backface culling**: Ook hier zijn er weer 3 mogelijkheden:
  - Als de normaalkegel volledig weg van de viewer wijst, dan wordt de huidige knoop en diens kinderen niet verder doorlopen.
  - Wanneer een normaalkegel van een knoop daarentegen volledig naar de viewer wijst, dan wordt er geen backface culling meer gedaan op diens kinderen.
  - In het laatste geval, wanneer de normaalkegel noch volledig van de viewer weg noch volledig naar de viewer wijst, dan wordt er verder gegaan, met backface culling op de kinderen.

Om **Level-Of-Detail selectie** te bekomen wordt er tijdens het weergeven gekeken wat de geprojecteerde grootte op het scherm is. Er wordt verder gegaan met de kindknopen als deze grootte een bepaalde drempelwaarde overschrijdt. Die drempelwaarde hangt af van de gewenste framerate en de effectieve renderingstijd van het vorige frame. Wanneer de grootte klein genoeg is (kleiner dan de drempelwaarde), dan stopt de recursie en zal de huidige knoop gerenderd worden. Deze methode voorspelt dus de framerate en past de *Level Of Detail* naargelang aan.

Het renderen zelf bij QSplat gebeurt software-matig: Er wordt een splat getekend waarvan de grootte gebaseerd is op de grootte van de geprojecteerde

Figuur 3.1: Q-Splat rendermethoden

sphere. Omdat er met belichting gewerkt wordt, hangt de kleur van de splat af van de normaal en de kleur van de sphere. Bij het renderen wordt er met de Z-buffer gewerkt, opdat er geen occlusie optreedt. Voor de splatvorm bij QSplat zijn er drie mogelijkheden, zoals te zien is in figuur 3.1.

- Vierkant : Dit is de simpelste en snelste mogelijkheid. De punten worden dan weergegeven als non-antialiased vierkanten, met de `GL_POINT`-primitief van OpenGL. Het voordeel is dus de snelheid en de simpelheid, het nadeel is de lagere kwaliteit door de artefacten (anti-aliasing en hoekige randen).

- Volle cirkel : Hierbij worden een aantal driehoeken gerenderd, die dan een volle cirkel vormen. Het voordeel hiervan is dat er minder artefacten zijn, het nadeel is dat er veel meer vertices doorgestuurd moeten worden. Waar bij de vierkanten maar 1 vertex moest doorgestuurd worden, moeten er hier (3 x het aantal driehoeken) vertices doorgestuurd worden. Een andere mogelijkheid om een volle cirkel weer te geven is het gebruik van een simpele polygoon (een quad) met daarop een texture. Hierbij moeten er maar 4 vertices per punt doorgestuurd worden, maar zijn er nog altijd artefacten (vooral anti-aliasing).

- Fuzzy spot : Een fuzzy spot is een vlek met een alpha-waarde die volgens een Gaussiaanse functie verminderd. Om het blending -en occlusie-gedrag van fuzzy spots correct te implementeren werkt QSplat met 2 weergavestappen. In de eerste pass wordt er enkel naar de depth-buffer gerenderd en wordt er een kleine offset aan elke z-waarde toegevoegd. In de tweede pass worden depth-updates afgezet en wordt er gewoon weergegeven, waarbij enkel splats die het dichtst bij de viewer staan weergegeven en geblend worden en dus blending en occlusie correct worden afgehandeld. Door deze blending hebben fuzzy spots veel minder last van artefacten en geven ze dus een veel hogere renderingskwaliteit. Het grote nadeel is de performantie, die

een pak lager ligt door de twee rendering passes en het gebruik van de Gaussiaanse functie. Deze fuzzy spots komen nog vaker terug in de rest van de thesis, maar dan onder een andere naam (surfels of splats met Gaussiaanse filtering).

3.3.3 Surface Splatting

Deze methode, voorgesteld door Zwicker in [ZPvBG01], is gebaseerd op de Elliptical Weighted Average filter of EWA filter (van Heckbert [Hec86]), een belangrijke anisotropisch textuurfilterings-algoritme. De EWA-filtering geeft een beeld van hoge kwaliteit met een minimum aan aliasing-artefacten. Er zal eerst een wiskundig raamwerk voorgesteld worden, waarop deze methode steunt. Daarna zal de weergave uitgelegd worden. Een meer gedetailleerde uitwerking hiervan is te vinden in [ZPvBG01].

Wiskundig raamwerk

Bij traditionele weergave met polygonen kunnen de textuuropzoekingen op een simpele manier gedaan worden, omdat er een 2D-naar-2D-mapping bestaat tijdens het renderen. Bij splat-rendering is er geen dergelijke functie beschikbaar, dus moet er een meer expliciete texturerepresentatie zijn.

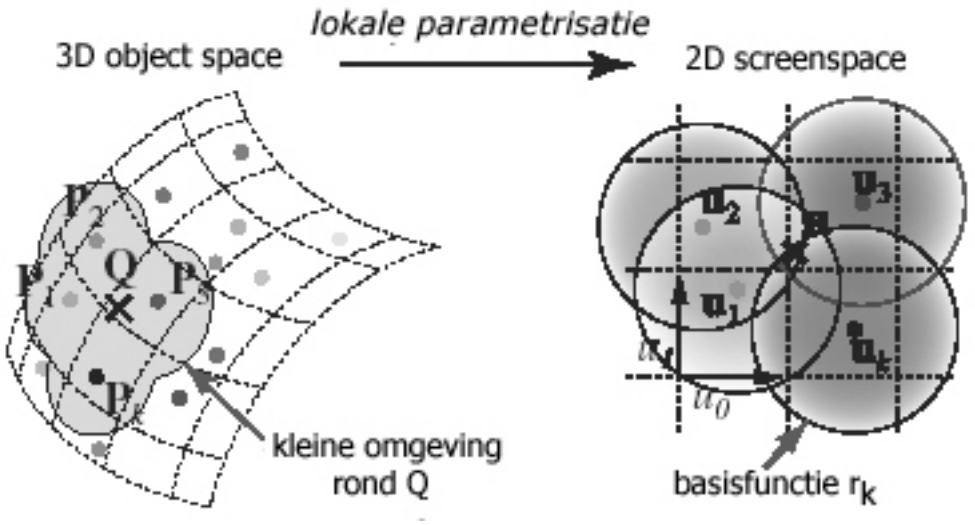
Gegeven is een verzameling van punten $\{\mathbf{P}_k\}$, waarvan elk punt een positie en een normaal heeft en waarbij er geen connectiviteitsinformatie over de punten beschikbaar is. Elk punt is geassocieerd met enkele waarden die tijdens het voorbereiden bepaald zijn, namelijk een symmetrische basisfunctie r_k en coefficiënten w_k^r,w_k^g en w_k^b die continue functies voorstellen voor de rode, groene en blauwe kleurcomponenten. Voor een gegeven punt $\mathbf{Q}$ op het oppervlak zoeken we nu een lokale textuurparametrisatie in een kleine omgeving rond $\mathbf{Q}$. $\mathbf{u}$ is de coordinaat van $\mathbf{Q}$, $\mathbf{u}_k$ is de coordinaat van $\mathbf{P}_k$. We bepalen nu de continue oppervlakte-functie $f_c(\mathbf{u})$ als een gewogen som :

$$f_c(u) = \sum_{k \in I\!N} w_k r_k(u - u_k) \tag{3.1}$$

Tijdens het weergeven wordt deze functie $f_c(\mathbf{u})$ dan naar schermruimte (*screenspace*) gewarpt, gefilterd en bemonsterd. Het warpen in [ZPvBG01] is een simpele concatenatie met een mapping $x = m(u) : I\!R^2 \rightarrow I\!R^2$:

$$g_c(x) = (f_c \circ m^{-1})(x) = f_c(m^{-1}(x)) \tag{3.2}$$

Vervolgens moet er gefilterd worden, wat ervoor zal zorgen dat er geen aliasing meer is. Het filteren gebeurt door een convolutie met een low-pass



Figuur 3.2: Mapping van objectspace naar screenspace

filter h en het resultaat is een band-gelimiteerde functie in schermruimte :

$$g'_c(x) = g_c(x) \otimes h(x) = \int_{\mathbb{R}^2} g_c(\xi) h(x - \xi) d\xi \tag{3.3}$$

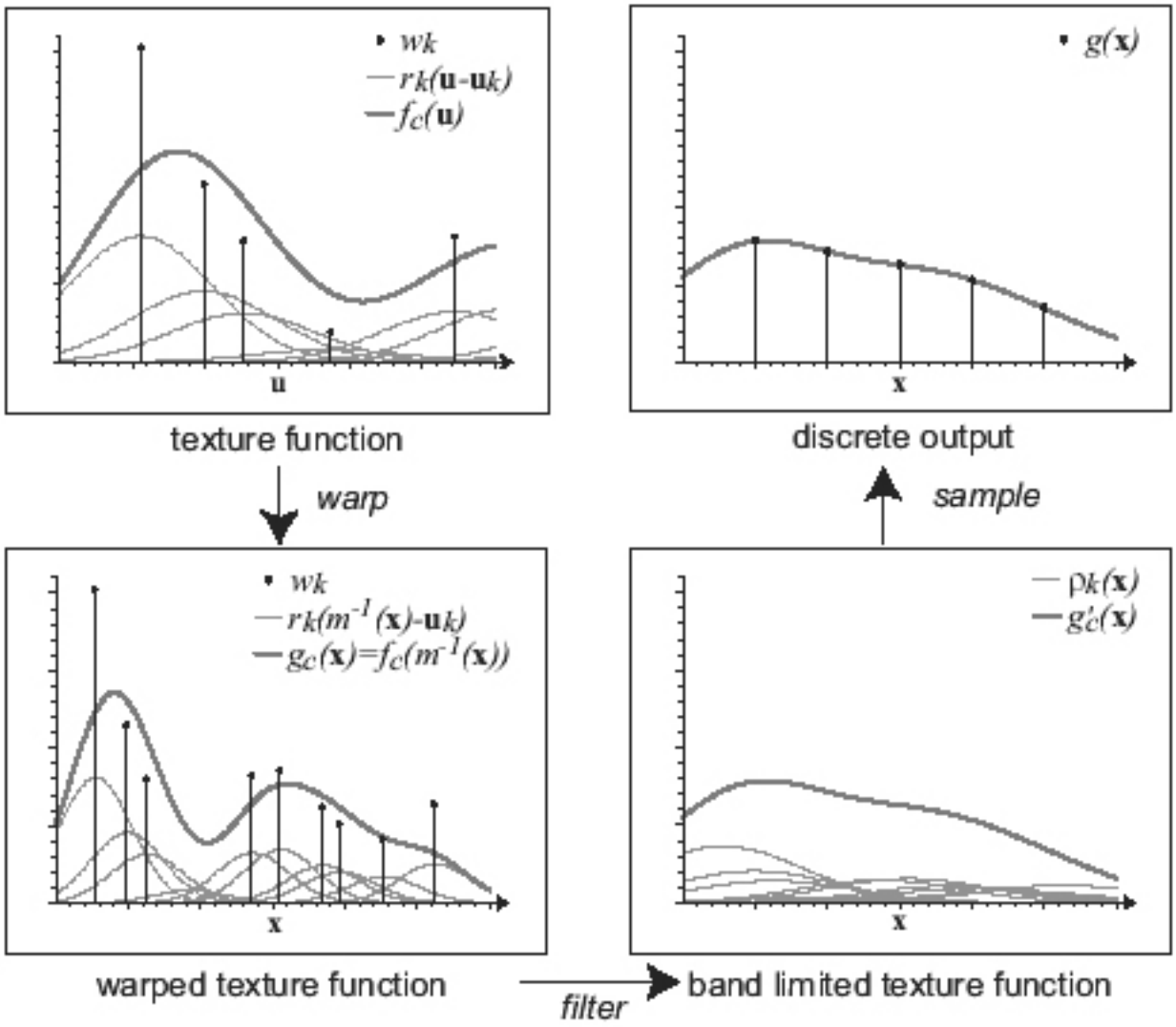
Tenslotte wordt in [ZPvBG01] deze functie $g'_c(x)$ dan bemonsterd door vermenigvuldiging met impulse i om een discrete output te bekomen :

$$g(x) = g'_c(x) i(x) \tag{3.4}$$

Figuur 3.3 geeft een overzicht van de verschillende stappen (warping, filtering, bemonstering). Samengevat kan de formule van $g'_c(x)$ als volgt geschreven worden :

$$\begin{aligned} g'_c(x) &= \int_{\mathbb{R}^2} h(x - \xi) \sum_{k \in \mathbb{N}} w_k r_k(m^{-1}(\xi) - u_k) d\xi \\ &= \sum_{k \in \mathbb{N}} w_k \rho_k(x) \qquad (3.5) \\ \text{waarbij } \rho_k(x) &= \int_{\mathbb{R}^2} h(x - \xi) r_k(m^{-1}(\xi) - u_k) d\xi \qquad (3.6) \end{aligned}$$

We noemen $\rho_k(x)$ de *resampling kernel*. Vergelijking 3.5 zegt ons dat we eerst de basisfuncties r_k kunnen warpen en filteren en daarna de *resampling kernels* berekenen en deze optellen in schermruimte. Om de integraal in 3.6



Figuur 3.3: EWA filtering proces

nog te vereenvoudigen wordt in [ZPvBG01] de mapping $m(u)$ vervangen door de lokale affiene benadering ervan, m_{u_k}, in een punt u_k :

$$\begin{aligned} m_{u_k}(u) &= x_k + J_{u_k}.(u - u_k) \\ \text{met } x_k &= m(u_k) \\ \text{en de Jacobiaan } J_{u_k} &= \frac{\vartheta m}{\vartheta u}(u_k) \end{aligned} \tag{3.7}$$

Hiermee kunnen we formule 3.6 herschrijven tot :

$$\begin{aligned} \rho_k(x) &= \int_{I\!R^2} h(x - m_{u_k}(u_k) - \xi) r'_k(\xi) d\xi \\ &= (r'_k \otimes h)(x - m_{u_k}(u_k)) \\ \text{waarbij } r'_k(x) &= r_k(J_{u_k}^{-1} x) \ \ (=\text{gewarpte basisfunctie}) \end{aligned} \tag{3.8}$$

Formule 3.8 zegt ons dat we de schermruimte *resampling kernel* ($\rho_k(x)$) kunnen schrijven als een convolutie van een gewarpte basisfunctie r'_k en een low-pass filter h. Hiermee is het wiskundig raamwerk compleet. De volgende stap is het toepassen van dit resultaat bij het renderen.

Schermruimte EWA

Zwicker gebruikt in [ZPvBG01] elliptische Gaussianen voor de basisfuncties en de low-pass filter, omdat deze gesloten zijn onder affiene mappingen en convoluties. Hiermee kunnen we de *resampling kernel* uitdrukken als een enkele Gaussiaan, die dan efficiënt gebruikt kan worden tijdens het weergeven.

Een elliptische Gaussiaan $G_V(x)$ met variantiematrix V is gedefinieerd als :

$$G_V(x) = \frac{1}{2\pi \mid V \mid^{\frac{1}{2}}} e^{\frac{-1}{2} x^T V^{-1} x}, \tag{3.9}$$

waarbij $\mid V \mid$ de determinant van de variantiematrix V is. De Gaussiaanse vorm van de basisfuncties r_k en de low-pass filter h schrijven we dan als :

$$\begin{aligned} r'_k(x) = r(J^{-1}x) = G_{V_k^r}(J^{-1}x) = \frac{1}{\mid J^{-1} \mid} G_{JV_k^r J^T}(x) \text{ en} \\ h(x) = G_{V^h}(x), \end{aligned}$$

waarbij V_k^r en V^h de variantiematrices van, respectievelijk, de basisfuncties en de low-pass filter voorstellen. De *resampling kernel* uit 3.8 kan dan

geschreven worden als een enkele Gaussian met een variantiematrix die de gewarpte basisfunctie en de low-pass filter combineert. Als $V^h = I$, dan krijgen we :

$$\begin{aligned}\rho_k(x) &= (r'_k \otimes h)(x - m(u_k)) \\ &= \frac{1}{\mid J^{-1} \mid}(G_{JV_k^r J^T} \otimes G_I)(x - m(u_k)) \\ &= \frac{1}{\mid J^{-1} \mid} G_{JV_k^r J^T + I}(x - m(u_k)) \end{aligned} \tag{3.10}$$

V_k^r moet voor elk punt zo gekozen worden dat het de puntendichtheid rond het huidig punt benadert. Dit kan vast gekozen worden of tijdens de voorbereiding voor elk punt berekend worden in de omgeving van het punt. Tenslotte kunnen we formule 3.10 in formule 3.5 voegen en dan krijgen we :

$$g'_c(x) = \sum_{k \in I\!N} w_k \frac{1}{\mid J^{-1} \mid} G_{JV_k^r J^T + I}(x - m(u_k)) \tag{3.11}$$

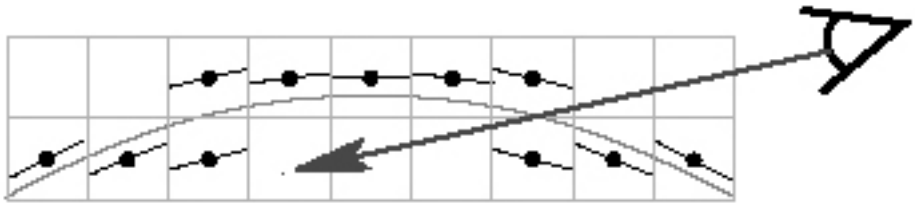
De implementatie van deze formule wordt in de volgende sectie besproken.

Renderingsalgoritme

Voor elk punt moet tijdens het weergeven de *resampling kernel* uit formule 3.10 gekend zijn. Hiervoor moet in de eerste plaats de Jacobiaan berekend worden. Deze Jacobiaan is een mapping van 2D naar 2D die de transformatie van de coördinaten van de lokale oppervlakteparametrisatie naar het viewport voorstelt. Dit wordt berekend door een concatenatie te maken van de affiene transformatie die van objectruimte naar cameraruimte mapt, een perspectieve projectie naar schermruimte mapt en de viewport mapping doet.

Naast deze berekening wordt het huidige punt gemapt naar screenspace, wat een punt $m(u_k)$ oplevert. Daarna wordt de zojuist berekende resampling kernel gecentreerd op $m(u_k)$. Elke pixel van de *resampling kernel* opgeteld in een accumulatiebuffer (onderdeel van de framebuffer). Deze accumulatiebuffer wordt op het einde gebruikt om een normalisatie uit te voeren. Verder wordt er een z-waarde berekend voor elke pixel die dan gebruikt wordt om aan zichtbaarheidssplatten te doen : punten/objecten dichtbij de viewer worden gerenderd, punten/objecten hierachter niet. Het shaden wordt op de pixels van de framebuffer gedaan, zodat er enkel shading wordt berekend op de zichtbare punten.

Kort samengevat ziet het renderingsalgoritme er als volgt uit:



Figuur 3.4: Gaten in weergegeven oppervlak door quantisatie

```
for each point P {
  project P onto screenspace;
  calculate resampling kernel G;
  splat G;
}
for each pixel px of the framebuffer {
  shade px;
}
```

3.3.4 Compacte representatie

Botsch neemt in [BWK02] het *surface splatting*-algoritme over zoals besproken in 3.2.3 en 3.3.3. Hierdoor is de weergave van hoge kwaliteit, met amper aliasing en gaten ([ZPvBG01]).

Door de efficiëntere dataopslag kan het soms gebeuren dat er nog gaten in de weergegeven oppervlakken zitten, zoals getoond wordt in figuur 3.4. Om dit op te lossen worden er kleine correctie-vectoren (genaamd *afstand attributen*) aan elke surfelpositie toegevoegd.

3.4 Algemene vergelijking

De tot nu toe besproken methoden om punt-gebaseerde modellen weer te geven. In deze sectie worden ze kort vergeleken met elkaar.

In 3.2.1 en 3.3.1 worden voor het eerst surfels als weergaveprimitieven gebruikt. Er wordt een efficiënte hiërarchische datastructuur gebruikt en de weergavekwaliteit is goed, met lichte aliasingartefacten. Deze methode is goed, maar er is nog veel plaats voor verbetering op gebied van snelheid en kwaliteit.

In 3.2.2 en 3.3.2 wordt Q-Splat voorgesteld. Dit is een heel belangrijk werk dat zich vooral toespitst op de datastructuur. Het is uiterst geschikt voor grote datasets, door zijn snelle voorbereiding en compacte representatie. Dit in combinatie met de level-of-detail-controle zorgen ervoor dat het geschikt is voor gebruik op desktop-pc's. De weergavekwaliteit is ook goed, maar bevat artefacten (aliasing en gaten).

In 3.2.3 en 3.3.3 wordt de bekende EWA filtering (gebruikt voor textuurfiltering) toegepast op het weergeven met surfels. Dit geeft een uitstekende weergavekwaliteit met vrijwel geen artefacten. Door de dure filtering en de meerdere weergavestappen is dit een computationeel zwaar algoritme dat veel van de CPU vraagt. De nadelen zijn de zware renderingskost en het niet gebruik maken van hardware-acceleratie door vertex en fragment shaders. In 4.2.2 en 4.3.2 zal een uitbreiding op deze methode getoond worden die wel gebruik maakt van hardware-acceleratie.

In 3.2.4 en 3.3.4 wordt een zeer compacte representatie getoond die een sterke „compressie" toepast op de data. Deze methode is sneller en compacter dan de vorige methoden, maar heeft zoals de eerste twee een minder goede weergavekwaliteit, door artefacten.

De methoden die tot nu toe besproken zijn, zijn in staat om op een snelle en efficiënte manier en in hoge kwaliteit punt-gebaseerd te renderen. Er worden vaak geävanceerde datastructuren gebruikt die technieken als *level-of-detail*-selectie en *culling* toelaten. Door filtering-technieken (zoals *EWA Surface Splatting*, zie 3.3.3) kan er een hoge visuele kwaliteit bereikt worden.

Het grote nadeel is nog dat er een zware computationele last op de CPU ligt. Met de programmeerbaarheid en de flexibiliteit van de recente grafische hardware is het heel interessant om een groot deel van de computationele last van de CPU naar de GPU te verschuiven. Dit wordt besproken in het volgende hoofdstuk.

Hoofdstuk 4

Hardware

In dit hoofdstuk worden enkele van de belangrijkste algoritmen besproken om hardware-matig punt-gebaseerde modellen weer te geven. Deze methoden maken gebruik van de programmeerbaarheid en flexibiliteit van de recentere grafische hardware. Eerst wordt er nu een kort overzicht gegeven van voorafgaand werk en de methoden die besproken gaan worden, daarna worden de datastructuren besproken. Vervolgens worden de weergave-methoden besproken en tenslotte wordt er een korte samenvattende vergelijking gegeven.

4.1 Inleiding

In [RPZ02] wordt de EWA filtering, die besproken werd in 3.3.3, gebruikt op de grafische hardware. Ze renderen elke splat als een vierkant met een textuur in objectruimte (*objectspace*). Hierdoor moet er voor elke splat vier punten verwerkt worden, waardoor er „slechts" 2 tot 3 miljoen splats per seconde weergegeven worden.

In [CH02] wordt een octree-gebaseerde spatiale datastructuur met daarin punten en driehoeken voorgesteld. Op deze datastructuur worden zichtbaarheidsberekeningen gedaan, zonder gebruik te maken van de z-buffer. Het correcte weergeven van de scene wordt opgelost door de cellen van de octree te sorteren en van achter naar voor te renderen. Hierdoor moet er geen tweede renderingstap gedaan worden, zoals bij de meeste andere algoritmen. Het nadeel is dat er artefacten optreden, omdat de cellen van achter naar voor met elkaar geblend worden zonder enige dieptecontrole.

In 4.2.1 en 4.3.1 wordt de methode van Botsch besproken, die in [BK03]

voorgesteld werd. Deze methode maakt gebruik van de grafische hardware, geeft een hoge kwaliteit en is efficiënt in bandbreedte. Er worden snelheden tot 10 miljoen splats per seconde gehaald in [BK03]. Deze methode zal samen met de *Sequential Point Trees* van Dachsbacher, die zodadelijk besproken worden, gebruikt worden in de implementatie.

De tweede methode die besproken wordt, werd door Guennebaud voorgesteld in [GP03]. Deze methode is vergelijkbaar met die van Botsch, alleen is deze gebaseerd op de „EWA filtering"-methoden (zie 3.3.3 en 3.2.3). Het wordt besproken in 4.2.2 en 4.3.2.

Vervolgens worden de *Sequential Point Trees* (SPT) van Dachsbacher besproken. Deze methode werd voorgesteld in [DVS03]. Deze datastructuur is gebaseerd op de hierarchische representaties die eerder besproken zijn, maar aangepast zodat het geschikt is om op een sequentiële manier op de grafische hardware te gebruiken. Er wordt eerst een hierarchische datastructuur opgebouwd die dan sequentieel geherstructureerd wordt. Zonder filtering en met simpele belichting kunnen er tot 60 miljoen punten per seconde weergegeven worden of 85 miljoen punten wanneer er culling gebruikt wordt. Deze wordt besproken in 4.2.3 en 4.3.3. Deze methode zal samen met de renderingsmethode van Botsch gebruikt worden in de implementatie.

De vierde methode is de zogenaamde *Deferred splatting*-methode, voorgesteld door Guennebaud in [GBP04]. Deze methode spitst zich vrijwel volledig toe op het renderen (niet op de datastructuur) en zorgt ervoor dat de zware filterberekeningen van de fragment shaders enkel worden uitgevoerd op de punten die ook echt zichtbaar zijn. Deze wordt kort besproken in 4.2.4 en 4.3.4.

De vijfde en laatste methode heet *Phong splatting* en werd door Botsch in [BK03] voorgesteld. Deze bouwt verder op de eerdere methode van Botsch (4.2.1 en 4.3.1) en zorgt onder andere voor een betere shading door geen constante normaal per splat bij te houden, maar een lineair variërend normaalveld. Deze wordt kort besproken in 4.2.5 en 4.3.5.

4.2 Datastructuur

4.2.1 Methode van Botsch

Er wordt geen dynamische datastructuur gebruikt in [BK03]. De data (die als statisch wordt gezien) wordt opgeslagen in het AGP-geheugen van de videokaart, zodat het heel snel gebruikt kan worden door de GPU, in DMA mode. Er wordt hierbij gebruik gemaakt van de `NV_vertex_array_range`-extensie.

4.2.2 Methode van Guennebaud

In [GP03] gebruikt Guennebaud een hierarchische datastructuur, die culling en progressieve weergave toelaat. De datastructuur is een simpele octree die doorlopen wordt. Hoe dieper er in de octree gegaan wordt, hoe hoger de resolutie van het beeld.

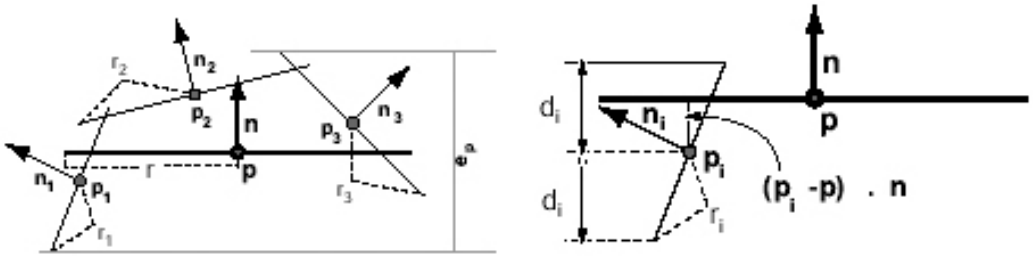
Frustum en *backface culling* gebeuren door middel van normaalkegels. Elke block van de octree houdt zo'n normaalkegel bij en bij het doorlopen van de octree wordt deze kegel getest met het frustum van de kijker, vergelijkbaar met de werkwijze van QSplat ([RL00]). Per resolutie wordt er een grote buffer bijgehouden. Elke knoop van de octree wijst naar een van deze buffers en houdt een pointer bij naar het begin en het einde van de reeks punten die bij deze buffer hoort, zoals in [RPZ02]. Tijdens het weergeven wordt er dan een grote buffer met de punten die weergegeven moet worden gereconstrueerd.

4.2.3 Sequential Point Trees

De datastructuur van de Sequential Point Tree ([DVS03]) is octree-gebaseerd. Gegeven als input een puntenwolk, wordt er een octree opgebouwd. Deze octree bevat de punten van de puntenwolk als bladknopen. Elke knoop van de octree stelt een punt voor, met een positie p, een normaal n en een diameter d. De knopen met kinderen omvatten al deze kinderen. De positie p en de normaal n worden ingesteld op het gemiddelde van de kinderen en de diameter d wordt zo gekozen dat de knoop al de kinderen omvat. De berekening hiervan gebeurt door een bounding disk te maken die al de kinderen omvat. Een knoop kan dus gezien worden als een bounding sphere. Het is ook mogelijk om attributen als kleur, textuur -of materiaalinformatie in een knoop bij te houden.

Naast de noodzakelijke attributen (positie, normaal en diameter), worden er ook nog twee waarden bijgehouden bij elke knoop van de octree: r_{min} en r_{max}. Deze twee waarden zullen we later gebruiken om te bepalen welke punten er weergegeven zullen worden. Voor de berekening van deze waarden moeten we twee foutwaarden berekenen : de *loodrechte foutwaarde* en de *divergerende foutwaarde*. Hoe groter deze waarden, hoe slechter deze knoop zijn kinderen benadert. Het gebruik van deze waarden in het algoritme zal ervoor zorgen dat de representatie zich zowel aan puntdichtheden als aan de vorm van het oppervlak zal aanpassen (bv. veel splats voor randen, weinig splats voor vlakke stukken).

De *loodrechte foutwaarde* e_p is de minimum afstand tussen de twee schijven die alle kinderen omvatten (zie figuur 4.1). Deze waarde is vooral gevoelig voor randen en kan berekend worden volgens vergelijking 4.1.



Figuur 4.1: Loodrechte foutwaarde bij Sequential Point Trees

$$e_p = max\{((p_i - p) \circ n) + d_i\} - min\{((p_i - p) \circ n) - d_i\} \qquad (4.1)$$
$$\text{met } d_i = r_i\sqrt{1 - (n_i \circ n)^2}$$

In formule 4.1 staan p en n respectievelijk voor de positie en de normaal van de knoop, p_i en n_i stellen de posities en normalen van de kinderen van deze knoop voor. De bewerking $\circ$ staat voor het crossproduct.

Door het weergeven moet de foutwaarde nog wat aangepast worden, omdat deze geprojecteerd wordt op het scherm. De aangepaste versie van e_p noemen we $\tilde{e_p}$ en kan berekend worden volgens 4.2.

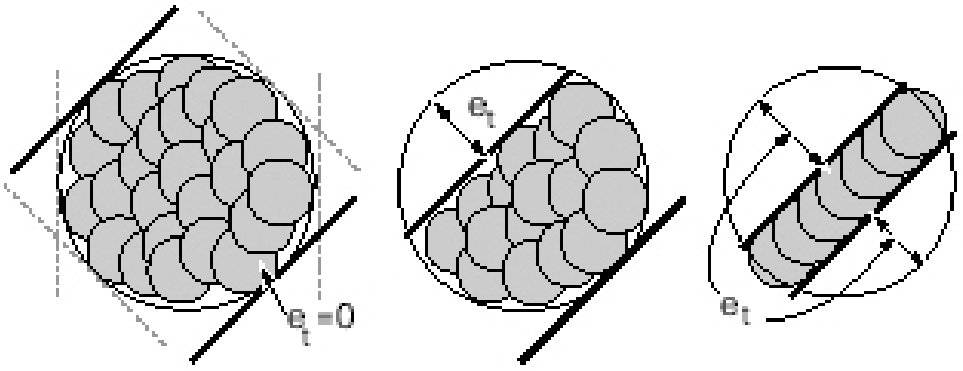
$$\tilde{e_p} = e_p sin(\alpha)/r \qquad (4.2)$$
$$\text{met } \alpha = \angle(v, n)$$
$$\text{en } r = \mid v \mid$$

In formule 4.2 staat het symbool $\angle$ voor de hoek tussen twee vectoren. α is dus de hoek tussen v (de view vector) en n (de normaal van de huidige schijf).

De *divergende foutwaarde* e_t zegt hoeveel oppervlak er teveel wordt bedekt door de schijf of, met andere woorden, hoe goed of hoe slecht de schijf door de kinderen benaderd wordt. In [DVS03] wordt deze e_t berekend door een zogenaamde *slab* rond de kinderen te vormen. Hierbij wordt een variërende oriëntatie gebruikt en uiteindelijk wordt de *slab* met de kleinste diameter gekozen, zoals getoond in figuur 4.2. De *divergende foutwaarde* wordt dan berekend met : $e_t = d_{disk} - d_{slab}$. Negatieve e_t's wordt ingesteld op 0 en net als bij de *loodrechte foutwaarde* moet van ook deze versie naar een aangepaste image-versie gemaakt worden. Deze aangepaste versie van e_t noemen we $\tilde{e_t}$ en kan berekend worden volgens 4.3.

$$\begin{aligned} \tilde{e_t} &= e_t cos(\alpha)/r \\ \text{met } \alpha &= \angle(v, n) \\ \text{en } r &= \mid v \mid \end{aligned} \tag{4.3}$$

In formule 4.3 staat het symbool $\angle$ voor de hoek tussen twee vectoren. α is dus de hoek tussen v (de view vector) en n (de normaal van de huidige schijf).



Figuur 4.2: Divergerende foutwaarde bij Sequential Point Trees

De twee foutwaarden (e_p en e_t) worden samengevoegd tot een enkele fout-

waarde (e_g) (en de image-space versie $\tilde{e_g}$), volgens 4.4. e_g is maximaal de straal van de huidige knoop. Indien voor alle knopen van de octree e_g gelijk is aan de straal, dan hebben we de Q-Splat-representatie ([RL00]).

$$e_g = max_\alpha\{e_p sin\alpha + e_t cos\alpha\} = (e_p^2 + e_t^2)^{1/2} \tag{4.4}$$

en

$$\tilde{e_g} = e_g/r$$

Voor elke knoop wordt $\tilde{e}$ bepaald, dit is $\tilde{e_g}$ of $\tilde{e_p} + \tilde{e_t}$. De octree wordt dan recursief doorlopen en voor elke node wordt e bekeken. Als deze e kleiner is dan een drempelwaarde ε, dan wordt er een splat met grootte $\tilde{d} = d/r$ getekend. In het andere geval, wanneer e groter is, dan wordt er recursief verdergegaan met de kinderen. Na het berekenen en combineren van de foutwaarden is de datastructuur klaar voor weergave.

4.2.4 Deferred Splatting

De *Deferred splatting*-methode die Guennebaud in [GBP04] voorstelt concentreert zich vooral op de weergave en gebruikt een hiërarchische datastructuur die culling en *level-of-detail*-selectie toelaat. Voor de implementatie werd er gekozen voor de *Sequential Point Trees* (SPT) van Dachsbacher (zie 4.2.3 en 4.3.3).

[GBP04] vermeldt dat de *Sequential Point Trees* niet efficiënt zijn voor puur punt-gebaseerd renderen met hoge kwaliteit, omdat er veel onnodige data door de vertex shaders verwerkt moet worden. Een combinatie van de Sequential Point Trees en Deferred Splatting is veel efficiënt er, omdat Deferred Splatting net deze zwakte van SPT oplost (SPT wordt enkel gebruikt in de tweede stap).

4.2.5 Phong Splatting

Botsch gebruikt in [BK03] geen speciale datastructuur. Er wordt vooral geconcentreerd op het renderen zelf. De GPU zorgt voor het renderen, terwijl de CPU zorgt voor het beheer van de datastructuur. Hierover wordt verder niets vermeld.

4.3 Rendering/weergave

4.3.1 Methode van Botsch

Inleiding

In [BK03] wordt een methode voorgesteld om met grafische hardware splats weer te geven, gebruik makende van vertex en fragment shaders. Splats worden met elkaar geblend om een hoge kwaliteit te bekomen. De geometrische data wordt opgeslagen in het AGP-geheugen om de data-overdracht zo snel mogelijk te laten verlopen.

Het renderingsalgoritme kan onderverdeeld worden in een aantal taken :

- Zichtbaarheidssplatting (renderen naar z-buffer)
- Renderen (met gaussiaanse filtering)
- Normalisatie

Om te zorgen dat splats correct met elkaar blenden moeten we in een eerste stap enkel naar de z-buffer renderen. Aan elke waarde in de z-buffer wordt dan een kleine waarde ϵ toegevoegd, zodat tijdens het renderen zelf enkel splats geblend worden waarvan de diepte-afstand kleiner is dan deze ϵ. Na het zichtbaarheidssplatting komt de „echte renderingstap", die nu besproken zal worden.

Bepaling splatgrootte

Allereerst wordt de splatgrootte bepaald. Hierbij moet er rekening gehouden worden met de positie en de straal van de splat en met de viewport parameters. De gezochte grootte van de splat is de grootte van de splat geprojecteerd naar schermruimte (*screenspace*). Omdat het exacte resultaat zwaar is om te berekenen, wordt er geen rekening gehouden met de normaal en wordt de omvattende bol (*bounding sphere*) van de splat geprojecteerd. Deze sphere wordt op het *near plane* van het frustum geprojecteerd en dan naar schermruimte (*screenspace*) getransformeerd. Dit kan berekend worden met formule 4.5, waarbij z_{eye} de afstand is van de splat tot de camera, r de straal van de splat, n, t, b, h respectievelijk het *near clipping plane*, het bovenste punt (*de top*), het onderste punt (*de bottom*) en de hoogte van het viewport voorstellen. Het berekenen van de splatgrootte kan efficiënt door de vertex shader gedaan worden.

$$size_{win} = r \cdot \frac{n}{z_{eye}} \cdot \frac{h}{t - b} \tag{4.5}$$

Bepaling splatvorm

De volgende stap is het bepalen van de vorm van de splat. Hierbij moet er rekening gehouden worden met de normaal van de splat, nadat deze naar cameraruimte (*cameraspace*) getransformeerd is. In een fragment shader kan er de `NV_point_sprite`-extensie gebruikt worden om te bepalen hoe ver een pixel zich van het midden van de splat bevindt. De parametrisatie geeft voor elke pixel een waarde terug, $(x, y) \in [-1, 1]^2$, waarmee we dan de diepte afstand δz berekenen. δz kan berekend worden met formule 4.6, waarbij (n_x, n_y, n_z) de cameraruimte normaalvector voorstelt.

$$\delta z = -\frac{n_x}{n_z}x - \frac{n_y}{n_z}y \tag{4.6}$$

De afstand tot het midden van de splat kan dan berekend worden met $\|(x, y, \delta z)^T\|_2$. Wanneer deze waarde kleiner is dan 1, wordt er een pixel uitgetekend, anders niet. Deze δz is slecht een benadering omdat we geen rekening houden met de hoek tussen de normaal en de kijkvector. Dit kan voor gaten zorgen als de ellipsen te smal worden.

Splatfiltering

Tot nu toe kunnen we al splats als ellipsen met een juiste grootte renderen. Om de visuele kwaliteit te verbeteren moeten we ook nog aan splat filtering doen. Dit kunnen we doen door splats te mixen volgens de alpha-waarde (*blending*). We gebruiken hierbij een 1D texture met daarin waarden die stijgen volgens een Gaussiaanse functie. De alpha-waarde van een pixel wordt dan bepaald door een textuur-opzoeking te doen met de afstand tot het midden van de splat die we al hebben berekend bij het bepalen van de splatvorm, zoals in formule 4.7.

$$\alpha(x, y) = Gaussian1DTextureLookup(\|(x, y, \delta z)^T\|_2) \tag{4.7}$$

Omdat we de splatvorm aangepast hebben en niet meer met gewone vierkanten werken, moeten we er ook voor zorgen dat de diepte van elke pixel (z_{win}) aangepast wordt (ook in de eerste stap!). Dit kan berekend worden met formule 4.8. n en f zijn de afstanden tot respectievelijk het *near* -en het *far plane*, zoals in formule 4.5.

$$z_{win} = \frac{a(z_{eye} + \delta z) + b}{z_{eye} + \delta z} \tag{4.8}$$

$$\begin{aligned} \text{waarbij } a &= f/(f-n) \\ \text{en } b &= -2fn/(f-n) \end{aligned} \tag{4.9}$$

Normalisatie

De laatste stap is de normalisatie. Dit houdt in dat met behulp van *additive blending* alle α-waarden worden opgeteld en uiteindelijk gebruikt om de pixelkleur te normaliseren. Dit kan efficiënt gedaan worden door de eerste twee weergavestappen naar een offscreen buffer te doen en deze buffer dan als textuur op het volledige scherm weer te geven, met behulp van de `NV_texture_rectangle`-extensie. Met een simpele fragment shader kan deze deling door α per pixel heel simpel en snel uitgevoerd worden.

Figuur 4.3 geeft een overzichtsfiguur van het volledige algoritme.

4.3.2 Methode van Guennebaud

Inleiding

In [GP03] wordt een gelijkaardig algoritme voorgesteld, dat steunt op de *surface splatting* methode die eerder besproken is (3.2.3 en 3.3.3).

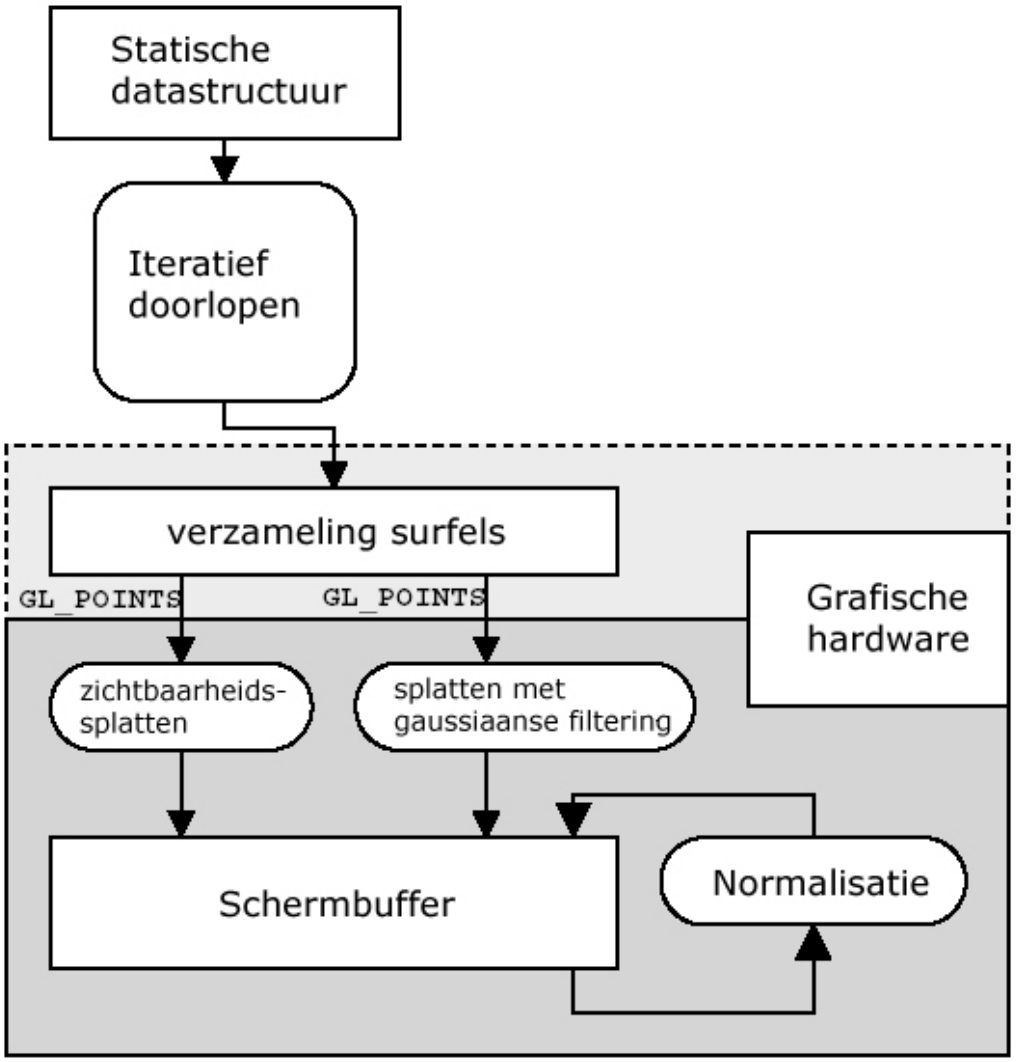
Ook hier is het algoritme onderverdeeld in 3 taken :

- Zichtbaarheidssplatting (renderen naar z-buffer)
- Renderen (EWA splatting)
- Normalisatie

Zichtbaarheidssplatting

De eerste stap (zichtbaarheidssplatting of *visibility splatting*) is ongeveer hetzelfde als die van het vorige algoritme. Er wordt enkel naar de z-buffer gerenderd. De buffer wordt met een kleine waarde getranslateerd weg van het gezichtspunt, volgens de kijk-richting. Deze z-buffer wordt dan tijdens het renderen gelezen zodat enkel splats op de voorgrond of dicht genoeg bij de voorgrond met elkaar opgeteld worden. Om een correct gevulde z-buffer te hebben, moet ook hier een per pixel dieptecorrectie gebeuren.

Gegeven een surfel met positie P^c en normaal N^c en een punt Q^c op het vlak van deze surfel, het punt Q^p als projectie van Q^c op het *near plane* en Q^v als schermruimte coördinaten van Q^c. Het is nu de bedoeling om de diepte of Q^c_z te berekenen, gegeven Q^v. Dit kan op een snelle manier gedaan worden met formule 4.10, waarbij a en B met een vertex shader worden berekend en diepte met een fragment shader. De fragment shader wordt veel aangeroepen en de berekening van de diepte vereist maar 2 instructies (1 DP3 en 1 ADD).



Figuur 4.3: Overzicht van de methode van Botsch

$$Q_z^c = depth = a - Q^v.B \tag{4.10}$$
$$\text{waarbij } a = g1 + \frac{g2}{P^c.N^c} N^c. \begin{bmatrix} r/t \\ t/n \\ 1 \end{bmatrix}$$
$$\text{en } B = \frac{g2}{P^c.N^c} \begin{bmatrix} N_x^c \frac{2r}{v_w} \\ N_y^c \frac{2t}{v_h} \end{bmatrix}$$
$$\text{en } g1 = \frac{f+n}{f-n}$$
$$\text{en } g2 = \frac{2fn}{f-n}$$

In formule guennebauddepthcorrection staan parameters r, t, n en f respectievelijk voor de rechterkant, de top, de afstand tot het *near* en de afstand tot het *far plane* van het *view frustum*. v_w en v_h staan voor de breedte en de hoogte van de *viewport*. De rest van de variabelen werden al eerder toegelicht, in de bovenstaande paragraaf.

Renderen

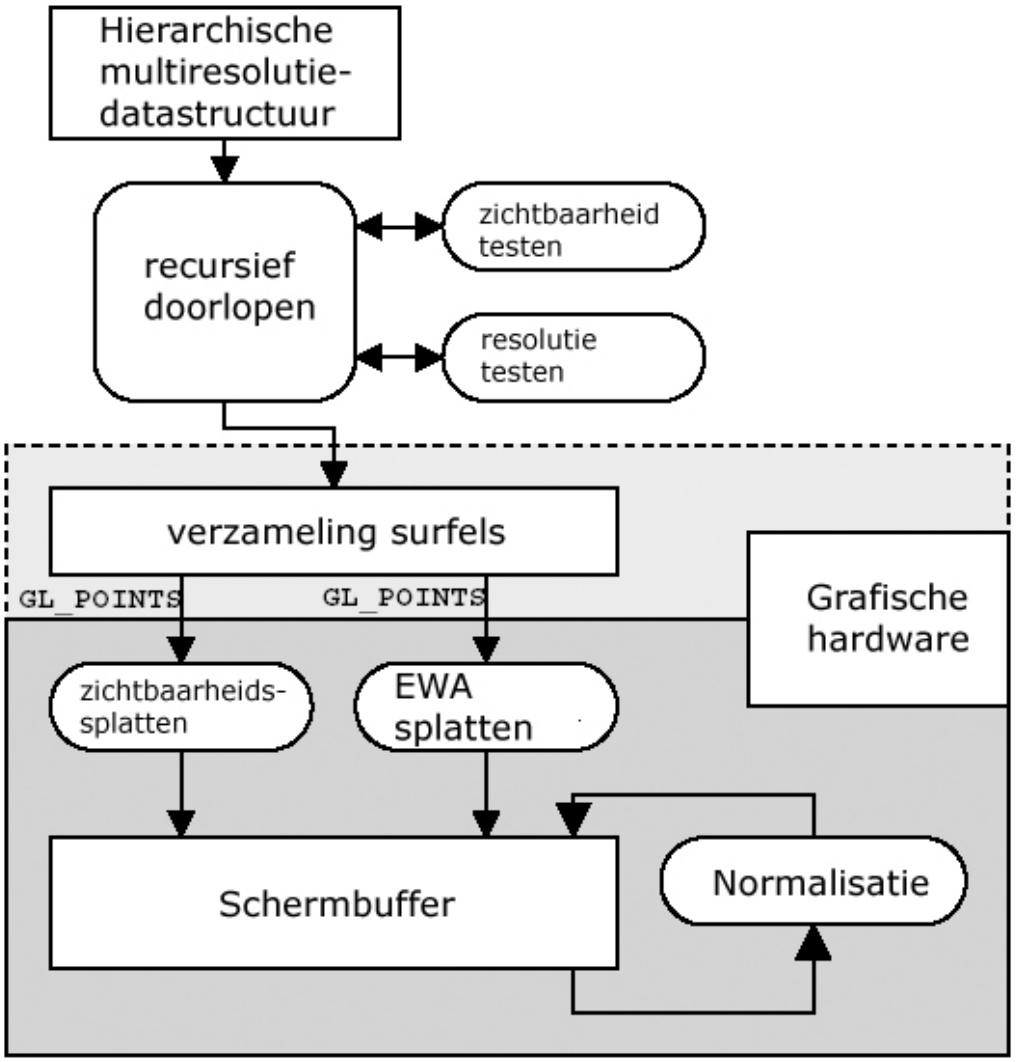
De volgende stap is de weergave. Hier worden in een vertex shader de parameters van de Gaussiaanse convolutiefilter berekend, gewarpt naar schermruimte, puntgrootte bepaald en belichting berekend. Een fragment shader berekent dan de afstand tot het midden van de splat. Indien de pixel binnen de splat valt, wordt de pixelkleur vermenigvuldigd met de juiste waarde van de Gaussiaanse filter. In het andere geval, wordt er geen pixel weergegeven.

Normalisatie

De laatste stap (normalisatie) is hetzelfde als de normalisatiestap van het vorige algoritme. Er wordt met additieve blending gerenderd naar een buffer buiten het scherm die dan als een vierkant met een textuur op het scherm wordt weergegeven. Een fragment shader doet ondertussen de deling door de α-component. Figuur 4.4 geeft een overzichtsfiguur van deze methode.

4.3.3 Sequential Point Trees

De representatie die tot nu toe is voorgesteld kan gebruikt worden om (software-matig) recursief weer te geven, vergelijkbaar met eerder besproken



Figuur 4.4: Overzicht van de methode van Guennebaud

methoden zoals Q-Splat. Om het efficiënt gebruik op grafische hardware mogelijk te maken, zal deze representatie omgevormd moeten worden tot een array. Zoals eerder vermeld, houdt Dachsbacher in [DVS03] twee extra attributen per knoop bij : r_{min} en r_{max}. r_{min} wordt ingesteld op e_g/ε. Standaard is deze ε gelijk aan 1, maar door een hogere ε te gebruiken, kan de framerate stijgen, ten koste van de weergavekwaliteit. r_{min} stelt dan de minimumafstand voor, waardoor de recursieve test simpeler en intuïtiever wordt : $r > r_{min}$. De r_{max}-attribuut van elke knoop wordt ingesteld op de r_{max}-waarde van de parent van de knoop (of ∞ in het geval van de root). Als r niet constant is, dan kan het soms gebeuren dat r kleiner is dan de r_{min} van een knoop, maar groter dan de r_{max} van alle kinderen, wat voor gaten zal zorgen. Om dit te voorkomen wordt er aan elke r_{min} en r_{max} een kleine waarde toegevoegd.

De recursieve test kan nu vervangen worden door een niet-recursieve test : $r \in [r_{min}, r_{max}]$. Vervolgens worden de punten uit de octree in een simpele array gestopt en deze array wordt gesorteerd volgens r_{max}. Het weergave-algoritme gebeurt nu als volgt :

- Bepaal een ondergrens voor r (door gebruik te maken van een omvattende box rond het object)
- Zoek de eerste entry in de lijst met $r_{max} \leq min\{r\}$
- Stuur alle entries van het begin tot aan de net uitgekozen entry naar de GPU
- Voor elk punt berekent de GPU (met een vertex shader) nu r en doet de test : $r \in [r_{min}, r_{max}]$. Punten waarbij de test faalt worden naar oneindig verplaatst.
- Punten waarbij de test slaagt worden weergegeven, met als splatgrootte $\tilde{d}$.

De CPU zorgt dus voor een ruwe selectie van „weer te geven punten" (de grote blok die naar de GPU gestuurd wordt) waarna de GPU deze selectie verfijnt (de niet-recursieve test). Gemiddeld blijft er 60 tot 90% van de punten over na de twee selectiestappen. Het is ook mogelijk om de CPU een linkergrens van de lijst te laten bepalen (in plaats van entries vanaf het begin door te sturen). De winst hierdoor is echter klein, omdat de lijst gesorteerd is volgens r_{max}. Het is mogelijk om aan *view dependant culling* te doen, maar ook hierbij is de performantiewinst klein.

4.3.4 Deferred Splatting

De technieken om met hoge kwaliteit punt-gebaseerd te renderen op de grafische hardware zijn een factor 5 tot 10 keer trager dan de ongefilterde versies, waarbij er kleine vierkantjes gerenderd worden. Dit komt vooral door de meerdere rendering passes (meestal 3) en het zware fragment shaden. Om die reden stelt Guennebaud in [GBP04] voor om het zware filtering enkel te doen op de zichtbare punten. Ze voegen de volgende selectie-algoritmen toe aan het begin van het renderen :

- view frustum culling
- back-face culling
- occlusion culling
- level-of-detail

Hun methode vereist geen voorbereiding en past perfect bij de eerder besproken hardware-methoden. In [GBP04] passen ze *deferred splatting* toe op de eerder besproken methode beschreven in [GP03].

Het algoritme is gebaseerd op accurate puntselectie en de tijdelijke samenhang tussen de achtereenvolgende beelden. Tussen de stap van de *visibility splatting* en het *EWA splatting* wordt er in een extra stap enkel gerenderd naar het kleurgedeelte van de schermbuffer. In plaats van de surfel kleur wordt de surfel index opgeslagen in deze buffer. Na deze stap bevat de kleurbuffer alle zichtbare surfels (omdat de onzichtbare pixels weggefilterd zijn door het zichtbaarheidssplatten). Deze deelverzameling van surfels noemen we B_i, waarbij de i verwijst naar het huidige frame. De volgende stap is het EWA splatting, enkel op de zichtbare surfels, wat al een aanzienlijke snelheidsverbetering is.

Om de snelheid van de allereerste stap (de *visibility splatting*) nog te verbeteren, wordt de tijdelijke samenhang uitgebuit. Er wordt dan namelijk enkel *visibility splatting* gedaan op de zichtbare surfels van de vorige frame, op B_{i-1} dus. Dit geeft ook een aanzienlijke snelheidsverbetering, omdat de *visibility splatting* een zware dieptecorrectie moet doen in de fragment shader.

Met deze methode zal de *depth buffer* gaten bevatten, door het uitbuiten van de tijdelijke samenhang. Om dit op te lossen wordt er nog een tweede maal aan visibility splatting gedaan, vlak voor het EWA splatting, maar dan

enkel op de punten die sinds de vorige frame zichtbaar geworden kunnen zijn (B_i - B_{i-1}).

Samengevat ziet het algoritme er dus als volgt uit :

1. Visibility splatting met B_{i-1}
2. Surfels renderen naar kleurbuffer (dit resultaat noemen we B_i)
3. Visibility splatting met B_i - B_{i-1}
4. EWA splatting met B_i
5. Normalisatie

Door de precieze puntselectie (er blijft een surfel per pixel over) treden er artefacten als aliasing en flikkerend beeld op. Om dit op te lossen worden er voorgefilterde textuursamples of surfel mipmaps gemaakt. Uit deze surfel mipmaps wordt dan tijdens het EWA splatting met lineaire interpolatie de juiste pixelkleur bepaald.

4.3.5 Phong Splatting

De splatting technieken die tot hiertoe besproken zijn bieden een weergavekwaliteit die niet perfect is. Dit komt omdat aan elke splat een constante normaalvector gekoppeld wordt, wat als resultaat vergelijkbaar is met flat of Gouraud shading bij polygonen. Om het equivalent van Phong shading bij polygonen te bereiken zou er normaal-interpolatie tussen splats moeten gebeuren, waarvoor er op het eerste zicht connectiviteitsinformatie nodig is.

In [BSK04] stelt Botsch voor om, in plaats van een constante normaalvector, een lineair variërend normaalveld per splat bij te houden. Op deze manier blijft het grootste voordeel van punt-gebaseerd weergeven behouden : er hoeft geen connectiviteitsinformatie bijgehouden te worden. Om het normaalveld te vinden wordt er een lineaire functie gezocht bij de normaalvectoren met de *least squares*-methode of *kleinste kwadraten*-methode (zie [Kol05]). Deze methode tracht een lineair functie te maken voor deze normalen, door de som van de kwadraten van de afwijkingen (tussen gegeven normaalwaarde en functiewaarde) zo klein mogelijk te houden.

Het weergeven kan efficiënt gedaan worden door gebruik van vertex en fragment shaders, waarbij tot 4 miljoen splats per seconden weergegeven

konden worden. Rekening houdende met het feit dat er bij Phong splatting veel minder splats nodig zijn om eenzelfde kwaliteit te verkrijgen, is Phong splatting dus ook sneller. In tegenstelling tot vroegere methoden van Botsch (zie 4.2.1 en 4.3.1), wordt er geen perspectieve benadering gebruikt. Hierdoor is deze methode wel projectief correct voor elke pixel en zal het uiteindelijke beeld preciezer zijn en minder artefacten bevatten (vooral rond de randen).

4.4 Algemene vergelijking

De hardware-methoden uit dit hoofdstuk zullen hier kort vergeleken worden met elkaar.

In 4.2.1 en 4.3.1 wordt de methode van Botsch besproken. Het accent van deze methode ligt op het efficiënt renderen met hoge kwaliteit en niet zozeer op de datastructuur. Er wordt gebruikt gemaakt van een splattingsalgoritme bestaande uit drie stappen, met Gaussiaanse filtering. De kwaliteit is vergelijkbaar met bestaande softwaremethoden. Het voordeel van deze methode is de snelheid van de vertex en fragment shaders, die hoger ligt dan de methode van Guennebaud. Het nadeel is de perspectiefbenadering die gebruikt wordt, wat voor artefacten (in de vorm van gaten) kan zorgen.

In 4.2.2 en 4.3.2 wordt de methode van Guennebaud besproken. Deze methode is vergelijkbaar met de methode van Botsch, met het verschil dat deze gebaseerd is op het *Surface splatting*-algoritme van Zwicker ([ZPvBG01]). Het maakt gebruik van een octree-gebaseerde datastructuur. Het voordeel is de hoge weergavekwaliteit. Het nadeel is dat deze methode nogal intensief is en dus iets trager zal presteren dan de methode van Botsch.

In 4.2.3 en 4.3.3 worden de *Sequential Point Trees* voorgesteld. Deze hiërarchische datastructuur is geschikt voor gebruik op grafische hardware en is zeer performant. Er wordt aan *level-of-detail*-selectie en *culling* gedaan door de CPU (ruwe selectie) en de GPU (fijne selectie, met een simpele vertex shader). Het grote voordeel is de snelheid : In [DVS03] worden er snelheden gehaald boven 50 miljoen punten per seconden. Hierbij wordt de CPU amper belast. De visuele kwaliteit is niet zo goed, maar dit kan gecompenseerd worden door deze methode te combineren met een van de andere methoden.

In 4.2.4 en 4.3.4 wordt het principe van *Deferred Shading* voorgesteld. Dit houdt in dat de zware berekeningen van de fragment shader enkel worden gedaan op de punten die uiteindelijk ook echt zichtbaar zullen zijn. Deze methode kan als een uitbreiding gezien worden op de andere methoden en kan voor een aanzienlijke snelheidswinst zorgen, aangezien de fragment shader vaak de bottleneck is bij algoritmen met een hoge kwaliteit.

In 4.2.5 en 4.3.5 wordt *Phong Splatting* van Botsch besproken. Deze methode bouwt verder op de eerdere methode van Botsch ([BK03]) en zorgt voor een betere visuele kwaliteit, door een lineair normaalveld aan de splats te koppelen, waarmee de belichting berekend wordt. Ook deze methode kan als een uitbreiding gezien worden om een betere kwaliteit te bekomen.

Hoofdstuk 5

Implementatie

5.1 Inleiding

In dit hoofdstuk wordt de implementatie van mijn thesis besproken. Hiervoor heb ik gekozen om twee technieken te combineren:

- De Sequential Point Trees van Dachsbacher besproken in 4.2.2 en 4.3.2
- De methode van Botsch om in hoge kwaliteit punt gebaseerd te renderen, besproken in 4.2.1 en 4.3.1

Mijn motivatie hiervoor is dat de Sequential Point Trees een zeer efficiëntie datastructuur bieden en dat de weergavemethode van Botsch zowel op een snelle manier als in hoog detail kan renderen op de nieuwe grafische hardware. Omdat de SPT-methode voor de datastructuur zorgt en de andere methode voor het weergeven, kunnen deze twee perfect samengevoegd worden. Dit werd al gesuggereerd in het discussie-gedeelte en de conclusie van de paper van Botsch ([BK03]). In de volgende twee secties wordt het implementeren van deze twee methoden besproken.

5.2 Datastructuur

In deze sectie zal de datastructuur besproken worden. Er zullen eerst een paar voorbereidingsstappen doorlopen moeten worden, vooraleer de Sequential Point Tree klaar is. De stappen worden besproken in de volgorde dat ze doorlopen worden wanneer de applicatie gebruikt wordt.

5.2.1 Point-struct

Om een punt voor te stellen wordt een simpele struct gebruikt. Deze struct bevat voor elk punt 3 floats die de posities voorstellen, 3 floats die de normaal voorstellen, een float voor de straal en 2 floats voor de foutwaarden, gebruikt door het Sequential Point Trees-algoritme.

5.2.2 Inlezen punten

Om met puntenwolken te werken moet er eerst geometrisch data ingelezen of gegenereerd worden. Hiervoor werd gekozen voor het PLY-formaat. Het PLY-formaat is ontwikkeld aan de Stanford University en biedt een simpele manier om objecten voor te stellen. Elk PLY-bestand bevat een header, gevolgd door een lijst van vertices en een lijst van polygon faces. De header geeft aan hoeveel vertices en polygon faces er zijn en welke attributen (coordinaten, normalen, kleurwaarden, ...) er in het bestand zijn aangegeven. Het voordeel van het gebruik van dit veelgebruikt formaat is dat er veel modellen beschikbaar zijn en dat er ook source-code te vinden is om modellen in te lezen.

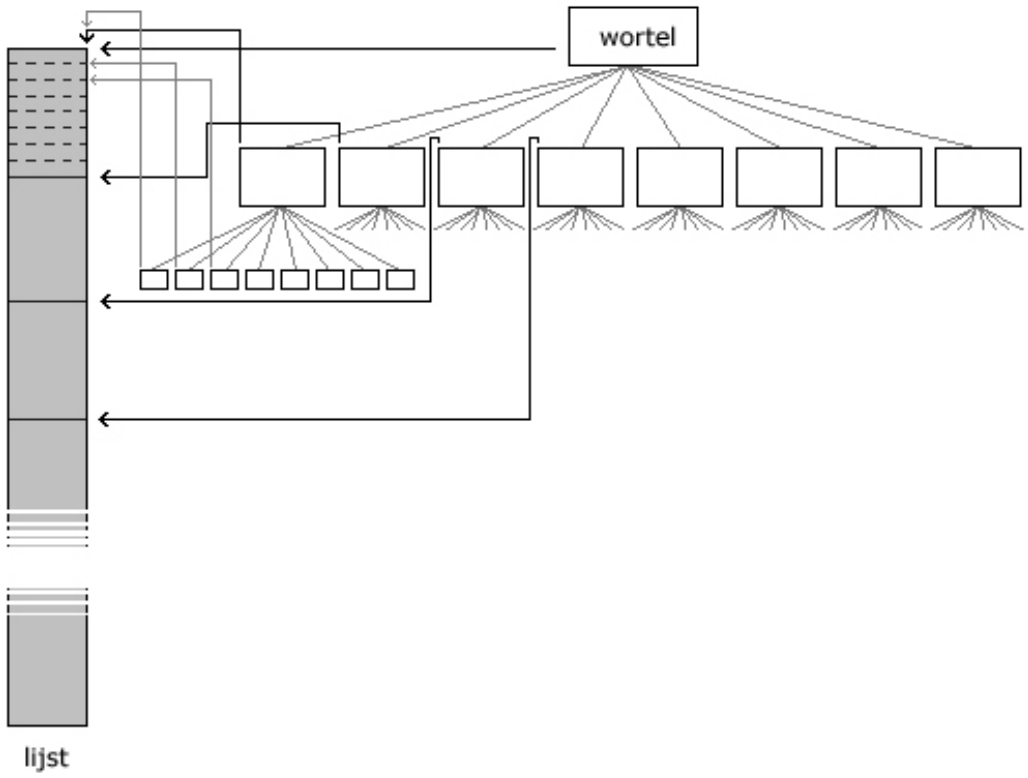
Voor het inlezen van de bestanden wordt gebruik gemaakt van stukken code van het Q-Splat raamwerk ([RL00]). Deze code heb ik aangepast en aangevuld zodat ik een simpele *CPointCloud*-klasse heb die een puntenwolk voorstelde en het volgende kan :

- Inlezen vertices en polygon faces uit bestand en in een array stoppen.
- Alle puntcoördinaten scaleren naar waarden binnen $[-1, 1]$ (door de minimum -en maximumcoördinaat te zoeken en dan te delen door de grootste absolute coördinaat).
- Normalen en splatgroottes bepalen (door gebruik te maken van de polygon faces).
- Het doorlopen van de array en het weergeven van elk punt met de `GL_POINTS`-primitief.

De volgende stap is het maken van een octree uit deze puntenarray die daarna omgezet zal worden naar een Sequential Point Tree.

5.2.3 Omzetting naar tijdelijk octree

Bij het opbouwen van de octree worden alle punten verdeeld in 8 deelbomen, die dan recursief verder onderverdeeld worden totdat het aantal punten in een deelboom onder een drempelwaarde zit. De grootte waarbij gestopt wordt moet klein genoeg zijn, anders heeft het algoritme weinig zin. Langs de andere kant, een te kleine drempelwaarde (1 of 2 punten) zorgt ervoor dat de octree heel groot wordt. Elke knoop van de octree wijst naar een positie in een grote array, die alle punten bevat, zoals getoond in figuur 5.1. Omdat later voor elke knoop van de octree attributen (normaal, positie, SPT-foutwaarden, ...) bepaald moeten worden, moet elke knoop precies weten welke punten deze omvat. Daarom wordt ook het aantal punten dat onder de knoop ligt bijgehouden.



Figuur 5.1: Structuur van de tijdelijke octree

Het algoritme voor het opbouwen van de octree ziet er als volgt uit :

1. Begin met de puntenarray van de *CPointCloud*-klasse.
2. Zoek het gemiddelde punt van de array door alle coördinaten op te tellen en te delen door het aantal punten.
3. Maak 8 suboctrees aan en verdeel alle punten van de puntenvector over de suboctrees (P is het huidige punt en M is het zonet bepaalde middelpunt):
   - SubOctree 1 : P.X $\gtrsim$ M.X EN P.Y $\gtrsim$ M.Y EN P.Z $\leq$ M.Z
   - SubOctree 2 : P.X $\gtrsim$ M.X EN P.Y $\gtrsim$ M.Y EN P.Z $\gtrsim$ M.Z
   - SubOctree 3 : P.X $\gtrsim$ M.X EN P.Y $\leq$ M.Y EN P.Z $\leq$ M.Z
   - SubOctree 4 : P.X $\gtrsim$ M.X EN P.Y $\leq$ M.Y EN P.Z $\gtrsim$ M.Z
   - SubOctree 5 : P.X $\leq$ M.X EN P.Y $\gtrsim$ M.Y EN P.Z $\leq$ M.Z
   - SubOctree 6 : P.X $\leq$ M.X EN P.Y $\gtrsim$ M.Y EN P.Z $\gtrsim$ M.Z
   - SubOctree 7 : P.X $\leq$ M.X EN P.Y $\leq$ M.Y EN P.Z $\leq$ M.Z
   - SubOctree 8 : P.X $\leq$ M.X EN P.Y $\leq$ M.Y EN P.Z $\gtrsim$ M.Z
4. Stop per suboctree alle punten in de array (die een membervariabele is van deze octree) en stel de lengte-variabele in.
5. Voer nu recursief voor elke suboctree stap 2 tot 4 uit, tot elke suboctree klein genoeg is (kleiner dan de drempelwaarde).
6. Maak een Point-array aan met dezelfde grootte als de puntenarray (`Point* pOctree = new Point[arraySize]`) en stel de lengte in op `arraySize`.
7. Doorloop de net aangemaakte octree recursief en kopieer alle punten uit de array naar de nieuwe Point-array. Stel ook per octree de pointer naar de juiste plaats in de Point-array in.
8. Verwijder alle tijdelijke puntarrays.

Op dit moment beschikken we over een Octree-datastructuur met een onderliggende array, zoals in figuur 5.1. Een simpele weergavefunctie is aanwezig bij deze Octree-klasse die alle punten met omvattende boxen uittekent. Dit was om te controleren of het algoritme goed werkte. De volgende stap is het omzetten van de Octree naar de Sequential Point Tree.

5.2.4 Omzetting naar Sequential Point Tree

Het omzetten naar de Sequential Point Tree bestaat uit 3 grote subtaken :

- Berekenen van positie, normaal en straal voor elke knoop van de octree
- Berekenen van SPT-foutwaarden (r_{min} en r_{max}) voor elke knoop van de octree
- Sequentialisatie

Deze taken zullen nu stap voor stap behandeld worden.

Berekenen positie, normaal en straal

Voor de berekening van deze attributen wordt de octree op een depth-first manier doorlopen. Voor de positie en normaal wordt er simpelweg uitgemiddeld : de posities en normalen worden opgeteld en gedeeld door het aantal. Voor de straal wordt voor elk punt/kind de som van de straal van dat punt/kind en afstand tot dat punt/kind berekend. De straal wordt dan ingesteld op het maximum van deze sommen.

Omdat de posities gekend moeten zijn om de stralen te kunnen berekenen, is deze berekening in twee functies onderverdeeld.

Berekenen van SPT-foutwaarden

Om voor elke knoop de foutwaarden r_{min} en r_{max} te berekenen moeten we de *loodrechte foutwaarde* (e_p) en de *divergerende foutwaarde* (e_t) berekenen, zoals besproken in hoofdstuk 4.2.2. De berekening van e_p is makkelijk te implementeren, met formule 4.1 :

```
for (each suboctree of the current node)
{
  if (suboctree.size > 0)
  {
     crossprod = crossproduct(this.normal,
                 suboctree.normal);
     di = suboctree..radius
                 * sqrt(1 - pow(crossprod,2));
     crossprod = crossproduct(this.normal,
                 (suboctree.pos - this.pos));

     if ((crossprod + di) > maxterm)
     {
```

```
        maxterm = crossprod + di;
      }

      if ((crossprod - di) < minterm)
      {
        minterm = crossprod + di;
      }
    }
}
ep = maxterm - minterm;
```

Bij de bladknopen van de octree gaat de for-loop niet over de suboctree, maar over de punten onder deze bladknoop.

De implementatie van de berekening van e_t is niet zo evident. e_t geeft ruwweg het teveel aan oppervlak weer dat door de huidige knoop wordt bedekt. Om deze berekening zo goed mogelijk te benaderen, wordt een grijze hoofdsplat getekend op het scherm, die de huidige knoop voorstelt. Deze hoofdsplat wordt zo weergegeven dat deze het volledige scherm bedekt (op de hoeken na, want het is een cirkel). Daarna worden alle punten/suboctrees direct onder deze knoop als witte splats getekend. Als de benadering van de knoop niet perfect is en er dus teveel oppervlak bedekt is (wat in de meeste gevallen ook zo is), dan zal er een deel van het scherm grijze pixels bevatten. Met de OpenGL-functie `glReadPixels` wordt berekend hoeveel witte, grijze en zwarte pixels er zijn. De zwarte pixels stellen de hoeken van het scherm voor, dit aantal is in principe altijd constant. De witte pixels stellen de punten/suboctree voor en de grijze pixels stellen de hoofdsplat voor. Hoe meer grijze pixels, hoe groter het teveel aan oppervlak dat deze hoofdsplat bedekt en hoe slechter de benadering dus. De waarde van e_t wordt nu bepaald met :

```
ratio = (#grijze pixels /
           (#witte pixels + #grijze pixels));
ep = this.radius * ratio;
```

Deze stappen worden voor alle knopen van de octree recursief uitgevoerd (op een depth-first manier). Alle splats worden op het scherm weergegeven en na elke stap wordt de schermbuffer leeggemaakt.

Na de berekening van e_p en e_t wordt e_g uit deze twee berekend. r_{min} mag ten hoogste de straal zijn van de huidige knoop :

```
eg = sqrt(pow(ep,2)+pow(et,2));
```

```
if (this.average.radius < eg)
        this.average.r_min = this.average.radius;
else
        this.average.r_min = eg;
```

e_g zou ook nog gedeeld kunnen worden door $\varepsilon \geq 1$. Als deze ε groter is dan 1, dan zal de weergavekwaliteit dalen en het aantal beelden per seconde stijgen.

Als alle r_{min}-waarden bepaald zijn, dan worden de r_{max}-waarden ingevuld. Dit gebeurt met een recursieve functie die de octree doorloopt en r_{min} van de huidige knoop als parameter meekrijgt. r_{max} wordt dan telkens op deze parameter ingesteld. Er wordt ook nog een kleine interval aan deze waarden toegevoegd om artefacten (gaten) te voorkomen, zoals vermeld op p.36.

Kort samengevat ziet de foutwaarden-berekening er zo uit :

1. Geef de huidige knoop weer als een witte hoofdsplat op het scherm
2. Geef alle kinderen weer als grijze subsplats op het scherm
3. Bereken e_p zoals op p.46
4. Bereken e_t door naar de verhouding witte op grijze pixels te kijken
5. Bereken e_g uit e_p en e_t
6. Bereken r_{min} uit e_g
7. Doe stap 1 tot 6 tot de volledige octree doorlopen is
8. Doorloop de octree nogmaals, om r_{max} overal in te vullen

Sequentialisatie

De volgende stap is de sequentialisatie, oftewel het klaarmaken van de datastructuur voor het gebruik op de GPU. Hierbij worden alle punten in een tijdelijke array gekopieerd. De volledige octree wordt doorlopen en de punten bij de knopen worden in de array gekopieerd. Na deze stap zal de tijdelijke array alle punten uit de octree en uit de onderliggende lijst bevatten. De volgende stap is om de array te sorteren, volgens de r_{max}-waarde. Tenslotte worden de punten uit de array weer in een nieuwe lijst gekopieerd en wordt de tijdelijke array uit het geheugen verwijderd. De nieuwe lijst kan nu naar het AGP-geheugen gekopieerd worden en gebruikt worden om weer

te geven. Tijdens het weergeven wordt er door de CPU een deel van deze lijst geselecteerd, wat dan door de GPU gerenderd zal worden. I

De data wordt in een vertex buffer object opgeslagen. Een vertex buffer object is te vergelijken met een gewone vertexlijst, met het verschil dat een vertex buffer object in het AGP-geheugen wordt opgeslagen, waardoor er veel minder datatransfer nodig is en de performantie dus veel zal stijgen.

5.3 Rendering

In dit deel wordt het implementatie-gedeelte besproken dat met het weergeven te maken heeft. Dit kan opgedeeld worden volgens de 3 weergave-stappen : een eerste weergave-stap naar de dieptebuffer, een tweede stap (het eigenlijke weergeven) en een normalisatiestap. Het weergeven gebeurt met OpenGL. Om gebruik te maken van de programmeerbare grafische hardware, wordt de CG Toolkit van NVidia gebruikt. CG is een hoge-niveau shading taal die werkt op verschillende platformen en API's ([MGAK03]).

Omdat het weergeven sterk afhangt van het afstand van de camera in de wereld en om een simpele navigatie door de wereld toe te laten, wordt er een simeple camera-klasse gebruikt, gebaseerd op de camera-klasse van mijn thesisbegeleider, Erik Hubo. In het begin van elke nieuwe frame wordt deze klasse gebruikt om de modelviewmatrix aan te passen.

5.3.1 Eerste weergavestap

In deze eerste stap wordt er enkel naar de dieptebuffer gerenderd. Deze dieptebuffer zal dan in de tweede stap gebruikt worden, zodat enkel de splats die het dichtst bij de camera liggen met elkaar geblend worden. Met `glColorMask(GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE);` wordt ervoor gezorgd dat er niet naar de kleurbuffer en dus enkel naar de dieptebuffer geschreven wordt. `glEnable(GL_DEPTH_TEST);` en `glDepthMask(GL_TRUE);` zorgen ervoor dat de dieptebuffer ook daadwerkelijk gebruikt en vernieuwd wordt.

Om zo weinig mogelijk dataoverdracht te hebben, wordt er gebruikt gemaakt van pointsprites (door de `GL_POINT_SPRITE_ARB`-extensie te gebruiken). Hierbij moet er maar 1 vertex per punt doorgestuurd worden. Pointsprites worden weergegeven als vierkanten, die naar de camera gericht zijn. Door de `GL_VERTEX_PROGRAM_POINT_SIZE_NV`-extensie te gebruiken, kan in de vertex shaders de grootte van deze pointsprites ingesteld worden.

Voor de OpenGL-oproepen moeten de shaders en het vertex buffer object ook nog gebonden en geconfigureerd worden. De pointers van het vertex buffer object moeten ingesteld worden, zodat deze werken op de data die we in het AGP-geheugen hebben opgeslagen. De shaders gebruiken enkele parameters die zodadelijk beschreven zullen worden.

Vertex shader

De vertex shader van de 2e stap begint met backface culling, waarbij splats die van de viewer weg wijzen, niet weergegeven worden. Dit wordt gedaan door de splatsize op 0 in te stellen. Voor de punten die de selectie halen, wordt allereerst de positie getransformeerd naar screen-space, door een vermenigvuldiging met de ModelViewProjectionMatrix (concatenatie van de ModelViewMatrix en de ProjectionMatrix). Vervolgens wordt de afstand r van de splat tot de camerapositie berekend. Als deze r binnen $[r_{min}, r_{max}]$ valt, dan wordt er verdergegaan met het weergeven. Deze selectie is de fijne selectie door de GPU zoals beschreven in hoofdstuk 4.3.3. Voor punten die het backface cullen en de fijne selectie halen, wordt de puntgrootte ingesteld volgens de formule 4.5. De grootte wordt ook nog eens vermenigvuldigd met een kleine constante (groter dan 1), zodat de splats groot genoeg zijn. Verder worden er nog 2 parameters (Dz_Value_1) en (Dz_Value_2) berekend, die in de fragment shader gebruikt zullen worden. Dz_Value_1 is de x-coördinaat van de normaal gedeeld door de z-coördinaat en Dz_Value_2 is de y-coördinaat van de normaal ook gedeeld door de z-coördinaat. Deze twee delingen worden in de vertex shader uitgevoerd, omdat fragment shaders veel vaker uitgevoerd wordt dan vertex shaders en de berekening op een efficiënte manier hier gedaan kan worden door de textuur-coördinaten te gebruiken. In de eerste stap wordt er niets op de schermbuffer weergegeven, dus hoeft er geen kleur ingesteld te worden.

Fragment shader

Tot nu toe worden de splats als vierkanten weergegeven. De fragment shader zal nu voor de vorm van de splat zorgen. Voor elke pixel van het vierkant moet bepaald worden of deze tot de splat behoord of niet. Met behulp van formule 4.6 wordt de diepte-afstand δz berekend. Wanneer de afstand van de pixel tot het midden ($=\|(x, y, \delta z)^T\|_2$) kleiner is dan 1, dan behoort deze pixel tot de splat en wordt de kleur op wit ingesteld. Als de pixel erbuiten valt, dan wordt met het CG-commando `discard`, waarmee output onderdrukt wordt.

5.3.2 Tweede weergavestap

In de tweede stap wordt de dieptebuffer uit de eerste stap gebruikt om op een correcte manier op de schermbuffer weer te geven. Het resultaat hiervan zal een beeld zijn van het object, met nog wat blendingartefacten die in de normalisatiestap weggefilterd zullen worden.

Vertex shader

De vertex shader van de tweede stap is bijna identiek aan de vertex shader van de eerste stap. Het enige verschil is dat deze shader ook de kleur moet berekenen omdat er in de tweede stap wel naar de schermbuffer weergegeven moet worden. Er wordt belichting berekend voor elke splat. Hierbij wordt de diffuse component en speculaire component berekend door door gebruik te maken van de lichtvector. Die laatste is meegegeven aan de vertex shader als parameter. Deze twee componenten wordt dan vermenigvuldigd met de kleur van de splat. Er wordt ook nog een beetje ambient licht toegevoegd, opdat de achterkant van het object zichtbaar als het licht op dezelfde plaats blijft staan en de camera rond het object draait. De kleur van de splat wordt doorgegeven aan de fragment shader, die de RGB-waarden gewoon zal overnemen.

Fragment shader

Deze fragment shader is bijna identiek aan de vertex shader van de eerste stap. Het enige verschil is dat deze shader ook de kleur moet berekenen. Waar de fragment shader uit de eerste stap gewoon de kleur op wit instelde, moet hier de juiste kleur gekozen worden. De RGB-waarden zijn berekend in de vertex shader en kunnen gewoon overgenomen worden. De alpha-waarde wordt bepaald door de afstand tot het midden van de splat te nemen en hiermee een textuuropzoeking te doen in een Gaussiaanse textuur (die meegegeven is aan de fragment shader als parameter). Deze afstand is al berekend om te bepalen of een pixel tot de splat behoord en ligt (in deze fase) altijd tussen 0 en 1. Het resultaat van deze textuuropzoeking is een waarde tussen 0 en 255 die als alpha-waarde van de pixel wordt ingesteld. Als de fragment shader met een splat klaar is, dan hebben pixels dichtbij het midden van de splat een hoge alpha-waarde en pixels verder van het midden een steeds lagere alpha-waarde en dus een hogere transparantie.

5.3.3 Normalisatiestap

In deze laatste stap worden de blendingartefacten weggefilterd door bij elke pixel de rgb-waarde te delen door de alpha-waarde. In de tweede stap werd de rgb-waarde van de framebuffer voor elke pixel namelijk vermenigvuldigd met de alpha-waarde, door het gebruikt van de `EXT_blend_func_separate`-extensie. De deling zorgt ervoor dat de kleur van de pixel het gewogen gemiddelde is van de geblende pixels op die pixel. Deze deling kan op een heel simpele manier gebeuren, door de inhoud van de schermbuffer van de vorige

stap naar een textuur te kopiëren en deze textuur dan over het volledige scherm weer te geven. De deling op elke pixel kan dan door een simpele en korte fragment shader gedaan worden. Door de `GL_TEXTURE_RECTANGLE_NV`-extensie te gebruiken, kan een textuur ter grootte van het scherm gebruikt worden (en hoeven de hoogte en breedte geen macht van 2 te zijn).

Hoofdstuk 6

Resultaat

In dit hoofdstuk worden de concrete resultaten van de geïmplementeerde algoritmen besproken. Het accent ligt op de weergavesnelheid en de visuele kwaliteit. Ook zal de snelheid van het voorbereiden kort besproken worden, hetgeen vooral van belang is bij het gebruik van de Sequential Point Trees. De meetingen zijn gedaan op een Pentium 4 3.2Ghz met een NVIDIA GeForce FX 5700, op een schermresolutie van 640 bij 480.

In sectie 6.3 zullen de modes in lage kwaliteit heel kort besproken worden. In sectie 6.4 komen de modes in hoge kwaliteit aan bod. Deze zullen in meer detail besproken worden.

6.1 Modellen

Er worden drie verschillende modellen gebruikt:

1. de Stanford bunny (34834 punten)
2. het draakmodel van Stanford (437645 punten)
3. het boeddha-model van Stanford (543652 punten)

Deze modellen zijn opgeslagen in zogenaamde .ply-bestanden en te vinden op de *Stanford 3D Scanning Repository*[1]. Zoals al vermeld in 5.2.2, bevat een .ply-bestand een lijst van vertices en een lijst van polygon faces. Deze polygon faces enkel worden gebruikt om de normalen van elke splat te bepalen tijdens de voorbereidingsfase. De vertices worden overgenomen als de posities.

[1] http://graphics.stanford.edu/data/3Dscanrep/

6.2 Performantie

De weergavesnelheid of framerate wordt uitgedrukt in het aantal beelden of frames per seconde (fps). Ook wordt telkens het aantal punten (bij lage kwaliteit) of splats (bij hoge kwaliteit) per seconde gegeven. Dit is het aantal beelden per seconde vermeningvuldigd met het aantal punten in het weergegeven model. Voor elk model wordt de renderingssnelheid gegeven voor verschillende renderingmodes :

1. *Software-mode* : De punten worden een voor een doorgestuurd door middel van `glVertex3f`. Er wordt gebruikt gemaakt van de `GL_POINT`-primitief en de weergave is van lage visuele kwaliteit (en zal daardoor niet verder besproken worden).

2. *Hardware-mode, Lage kwaliteit* : Deze manier van renderen is hetzelfde als de software-mode, behalve dat er gebruikt wordt gemaakt van *vertex buffer object*(5.2.4) om de punten snel beschikbaar te stellen voor de GPU.

3. *Hardware-mode, Hoge kwaliteit* : Bij deze manier bouwt verder op de vorige hardware-mode en maakt gebruik van de weergavemethode van Botsch ([BK03]) om een hoge visuele kwaliteit te verkrijgen.

4. *Hardware-mode, Hoge kwaliteit en Sequential Point Trees* : Deze manier bouwt verder op de vorige hardware-mode en maakt gebruik van de Sequential Point Trees van Dachsbacher ([DVS03]).

6.3 Weergave in lage kwaliteit

In tabel 6.1 worden de performanties van de eerste twee modes met elkaar vergeleken : software-mode en hardware-mode in lage kwaliteit. De weergave ziet er voor deze twee modes hetzelfde uit, maar is van zeer lage kwaliteit en niet relevant. Uit de performantietabel kunnen we afleiden dat het plaatsen van de geometrische data in de AGP-geheugen door middel van *vertex buffer objects* voor een snelheidswinst van ongeveer 40% zorgt. Een bijkomend voordeel is dat de CPU vrij is voor andere taken. Dit voordeel geldt ook voor alle andere hardware-modes.

6.4 Weergave in hoge kwaliteit

De volgende stap is het bespreken van weergave in hoge kwaliteit. Voor elk model wordt de weergave met en zonder Sequential Point Trees (SPT)

Model	Hard/software	Framerate	Punten/sec
Bunny	software	590fps	20,5 miljoen
Bunny	hardware	775fps	27 miljoen
Draak	software	64fps	28 miljoen
Draak	hardware	89fps	38,9 miljoen
Boeddha	software	49fps	26,6 miljoen
Boeddha	hardware	69fps	37,2 miljoen

Tabel 6.1: Performantie bij weergave in lage kwaliteit

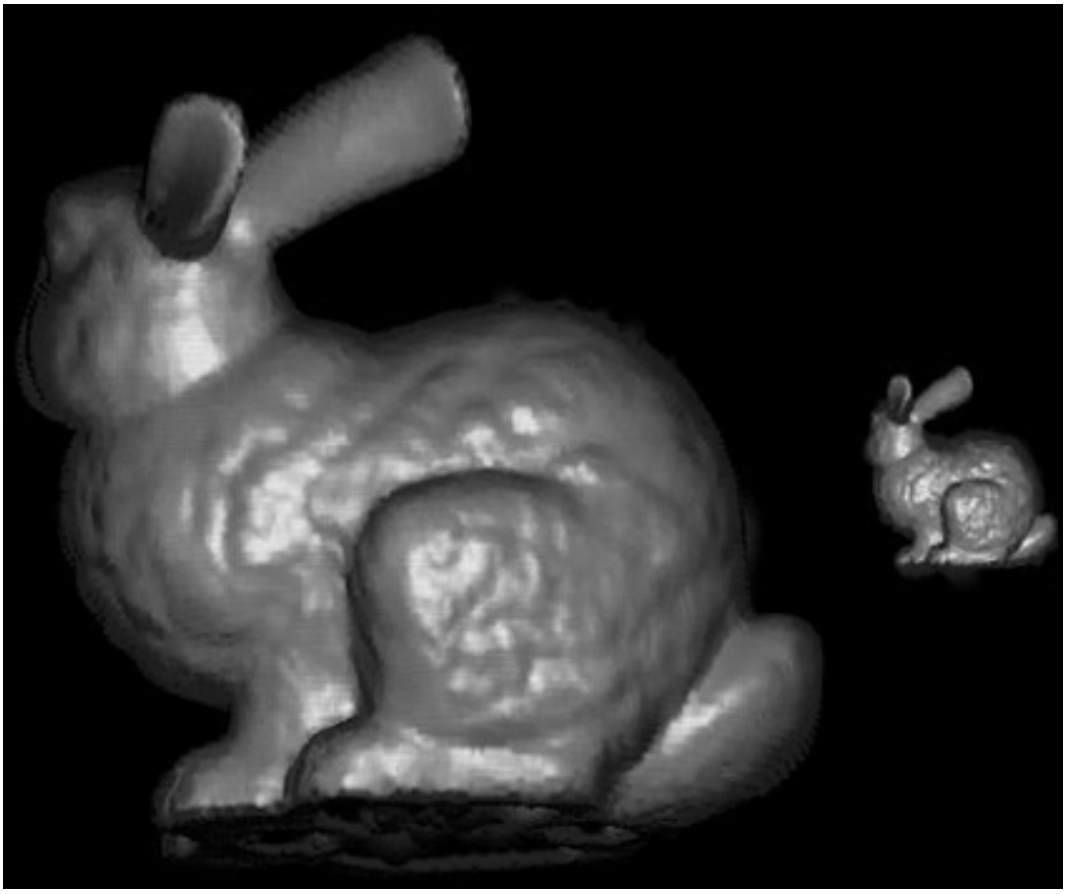
bekeken. In tabel 6.2 staan de resultaten van deze metingen. Figuur 6.1 toont de schermafbeelding voor de meeting voor de SPT-waarden bij de Stanford Bunny. De schermafdruk in het geval dat er geen SPT gebruikt wordt ziet er exact hetzelfde uit. Hetzelfde geldt voor het draakmodel (figuur 6.2) en het boeddha-model (figuur 6.3). De reden waarom deze schermafdrukken hetzelfde zijn wordt uitgelegd in 6.4.4.

Model	SPT	Framerate	Splats/sec
Bunny dichtbij	nee	21fps	0,7 miljoen
Bunny dichtbij	ja	21fps	0,7 miljoen
Bunny veraf	nee	99fps	3,4 miljoen
Bunny veraf	ja	284fps	10,5 miljoen
Draak dichtbij	nee	6fps	2,6 miljoen
Draak dichtbij	ja	6fps	2,6 miljoen
Draak veraf	nee	11fps	4,8 miljoen
Draak veraf	ja	46fps	21,6 miljoen
Boeddha dichtbij	nee	7fps	3,8 miljoen
Boeddha dichtbij	ja	7fps	3,8 miljoen
Boeddha veraf	nee	8fps	4,3 miljoen
Boeddha veraf	ja	52fps	28,2 miljoen

Tabel 6.2: Performantie bij weergave in hoge kwaliteit

6.4.1 Splatgrootte

De visuele kwaliteit bij het renderen met splats hangt sterk af van de gebruikte Gaussiaanse textuur en vooral van de grootte van deze splats. Om

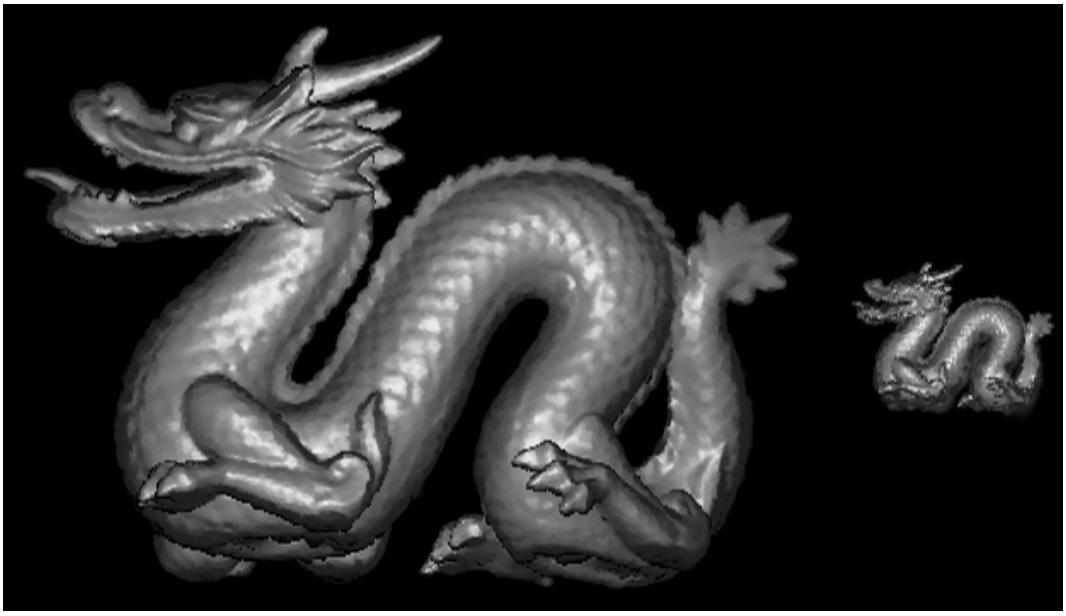


Figuur 6.1: Stanford Bunny gerenderd op kleine en grote afstand van camera

een mooi, glad oppervlak te krijgen, moeten deze parameters optimaal ingesteld worden. Het vergroten van de splatgrootte heeft een sterke negatieve impact op de renderingperformantie. Dit komt omdat de fragment shader, de bottleneck in het verwerkingsproces, zware berekeningen moet doen en door grotere splats te renderen, zal deze fragment shader veel vaker uitgevoerd moeten worden. Dit is ook de reden waarom de Stanford Bunny minder splats per seconde haalt, dan de Draak of de Boeddha : de splats zijn groter. Het negatieve effect hiervan („blurriness") wordt besproken in de volgende subsectie.

6.4.2 Schermresolutie en modelgrootte

Naast splatgrootte vormen ook de schermresolutie en de modelgrootte een belangrijke factor. De schermresolutie waarin de schermafbeeldingen geren-



Figuur 6.2: Draakmodel gerenderd op kleine en grote afstand van camera

derd zijn is 640 bij 480. Indien er een hogere schermresolutie gekozen zou worden, dan zou ook de splatgrootte aangepast moeten worden, anders zouden er gaten zichtbaar worden in het oppervlak.

De Stanford Bunny bestaat uit „maar" 34834 punten. Hierdoor moeten er grote splats worden getekend, omdat de punten ver uit elkaar liggen. Indien de camera dichtbij het object staat (zoals bij figuur 6.1 links), dan zullen deze grote splats zorgen voor een licht onscherp, „blurry" beeld. Dit negatief effect is vooral zichtbaar en storend wanneer er speculaire belichting wordt gebruikt (zoals bij de getoonde schermafbeeldingen). Dit is typisch voor een punt-gebaseerde weergave : Het verschil tussen punt-gebaseerde rendering en polygoonrendering is analoog met het verschil tussen bitmap-afbeeldingen en vector-afbeeldingen. De degradatie die zichtbaar is wanneer je op een punt-gebaseerd model inzoomt, is vergelijkbaar met de degradatie die je hebt wanneer je een bitmap-afbeelding vergroot.

De andere twee modellen hebben hier geen last van (althans niet bij de gekozen schermresolutie), omdat ze uit veel meer punten bestaan (437645 punten bij het draakmodel en 543652 punten bij het boeddha-model). Een mogelijke oplossing hiervoor zou een vorm van herbemonstering zijn, in plaats van het overnemen van de vertices uit de .ply-bestanden.

6.4.3 Perspectiefbenadering

Een nadeel aan het renderingsalgoritme van Botsch ([BK03]) is dat er een perspectieve benadering gebeurt bij de projectie naar schermruimte (*screenspace*). Botsch vermeldt dat het hierdoor mogelijk is dat er gaten optreden in het weergegeven object. Dit kan opgevangen worden door grotere splats te gebruiken. Een andere oplossing wordt voorgesteld door Botsch in het Phong Splatting-algoritme ([BSK04]). Hierin wordt een perspectief correcte projectie gebruikt.

Figuur 6.4 is een schermafdruk van de dieptebuffer waarop deze gaten zichtbaar zijn.

6.4.4 Sequential Point Trees

Wanneer het model dichtbij de camera staat, dan zal het gebruik van SPT niet veel voordeel bieden. In dit geval zal de framerate en de kwaliteit hetzelfde zijn als wanneer er geen SPT gebruikt worden. Pas als het object zich verder van de camera bevindt, zal er een snelheidswinst duidelijk worden. Naarmate deze afstand stijgt, stijgt de snelheidswinst steeds sneller. De visuele kwaliteit en de snelheidswinst bij het gebruik van SPT hangt heel sterk af van de grootte van de octree tijdens de voorbereiding (*preprocessing*). Dit wordt besproken in de volgende sectie (6.4.5).

Bij de SPT beslissen de twee foutwaarden (r_{min} en r_{max}) van elke knoop wanneer kleine splats worden vervangen door een grotere en vice versa. Dit hangt ook af van de afstand r van de camera tot het object. De mapping van deze r naar r_{min} en r_{max} is echter niet uniform gedefinieerd, waardoor er eerst een aantal parameters ingesteld moeten worden. Pas als deze parameters optimaal zijn ingesteld, zal de rendering een optimale balans tussen performantie en visuele kwaliteit bereiken.

In de SPT-schermafdrukken van dit hoofdstuk (figuur 6.1, 6.2 en 6.3) zijn deze parameters zo ingesteld, dat de visuele kwaliteit dezelfde was als wanneer er geen SPT gebruikt werd. Indien visuele kwaliteit minder belangrijk is dan performantie, dan kunnen deze parameters versoepeld worden en zal de performantie stijgen.

6.4.5 Preprocessing bij SPT

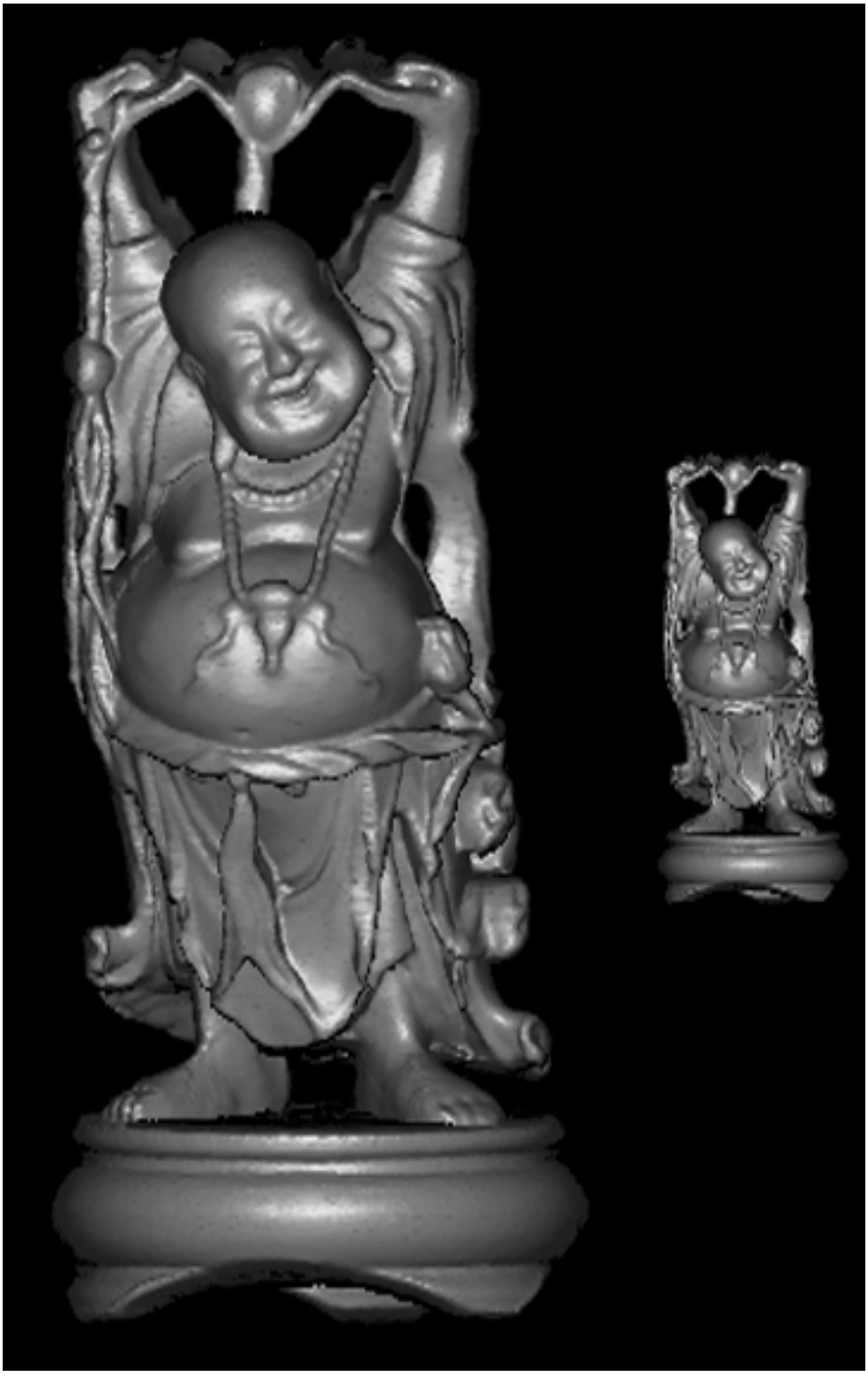
De efficiëntie van de SPT hangt heel sterk af van de grootte van de octree tijdens de voorbereiding. Deze grootte wordt bepaald door het maximaal aantal punten per bladknoop in de octree. De duur van deze voorbereiding per model wordt getoond in tabel 6.3. In de tabel staan twee voorbeelden per model, met telkens dit maximaal aantal achter elke modelnaam in de bovenste rij van tabel 6.3.

Hoe kleiner dit aantal, hoe groter de uiteindelijke octree (en dus ook de SPT) zal worden. Een grotere octree zal voor een betere visuele kwaliteit en een hogere performantie zorgen. Een SPT opgebouwd uit een kleine octree zal namelijk veel minder vlug kleine splats door grote splats vervangen. Het nadeel hiervan is dat de voorbereiding veel langer zal duren en de hoeveelheid data zal groeien (de niet-bladknopen maken de puntenverzameling groter).

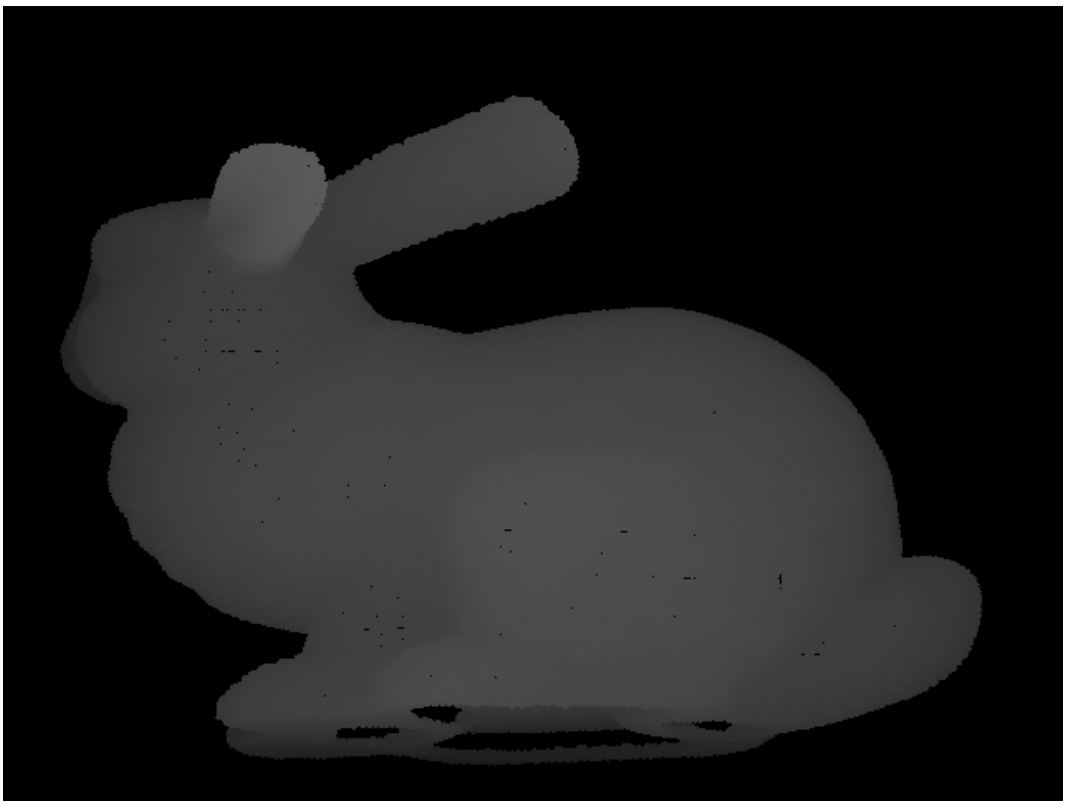
Een mogelijke oplossing voor de lange voorbereiding is het opslaan van een kant-en-klare SPT in een bestand, zodat de voorbereiding niet meer elke keer hoeft te gebeuren.

	Bunny (1000)	Draak (1000)	Boeddha (1000)
Inladen ply-file	2,2 sec	31,1 sec	39,2 sec
Opbouwen octree	0,06 sec	1,04 sec	1,34 sec
Omzetten naar SPT	5,9 sec	108,5 sec	140,3 sec
Totaal	8,1 sec	140,6 sec	180,8 sec
# punten in SPT	34933 (+99)	439580 (+1935)	546116 (+2464)
	Bunny (50)	Draak (50)	Boeddha (50)
Inladen ply-file	2,2 sec	31,1 sec	39,2 sec
Opbouwen octree	0,08 sec	1,6 sec	2,71 sec
Omzetten naar SPT	109,9 sec	1532,8 sec	1981 sec
Totaal	112,2 sec	1565,5 sec	2022,9 sec
# punten in SPT	37304 (+2470)	470558 (+32913)	585188 (+41536)

Tabel 6.3: Performantie van de voorbereidingsfase



Figuur 6.3: Boeddha-model gerenderd op kleine en grote afstand van camera



Figuur 6.4: Gaten in dieptebuffer door perspectieve benadering

Hoofdstuk 7

Conclusie

De laatste jaren is de interesse in een punt-gebaseerde weergave sterk gestegen, vooral door de groeiende complexiteit van de modellen en het beter worden van de 3D-scanning apparaten. Het doel van deze thesis was om een studie te doen naar punt-gebaseerde weergave en hiervan een implementatie te maken.

Het grote voordeel van punten tegenover polygonen is de conceptuele simpliciteit ervan : er hoeft geen connectiviteitsinformatie of andere speciale informatie bijgehouden te worden. Er is al veel onderzoek verricht naar software-matige punt-gebaseerde weergave en het is al mogelijk om in hoge kwaliteit en op een efficiënte manier te renderen. Toch zijn deze methoden nog erg gelimiteerd op het gebied van performantie. De huidige grafische hardware is geöptimaliseerd voor polygoon-rendering, maar recente grafische kaarten zijn programmeerbaarder en flexibeler geworden. Hierdoor is het mogelijk geworden om punt-gebaseerd te renderen op de grafische hardware in hoge kwaliteit en tegen hoge snelheid. Het nadeel is echter dat er 3 *passes* nodig zijn om tot een visueel aantrekkelijke en correcte weergave te komen. Een bijkomend algemeen nadeel van punt-gebaseerde weergave is dat punt-gebaseerde rendering zich verhoudt tot polygoonrendering als bitmaps tot vector afbeeldingen : vergroting bij een punt-gebaseerde weergave leidt tot een kwaliteitsvermindering.

In deze thesis is een implementatie gemaakt van de methode van Botsch, gecombineerd met de Sequential Point Trees van Dachsbacher. Deze combinatie blijkt goed te werken, als de verschillende parameters optimaal zijn ingesteld. Het is echter cruciaal dat deze parameters goed op elkaar zijn afgestemd, anders is de performantie en/of visuele kwaliteit niet goed. Zo zal een model dat uit weinig punten bestaat een „blurry" resultaat geven

wanneer de camera er dichtbij staat. Bij het inzoomen op een punt-gebaseerd model bestaande uit weinig punten, moeten de splats sterk vergroot worden, wat een visueel slecht resultaat geeft. Ook hebben grotere splats een negatief effect op de framerate, omdat de zware fragment shaders (de bottleneck van het weergaveproces) dan vaker moeten uitgevoerd worden.

Een nadeel van de methode van Botsch is de perspectiefbenadering, die gaten in het oppervlak kan veroorzaken. Een oplossing hiervoor is de splats groter maken om zo de gaten op te vullen, maar dit heeft een negatief effect op de performantie. Een andere oplossing wordt voorgesteld door Botsch in [BSK04], een algoritme dat een perspectief correcte projectie gebruikt.

Bij het gebruik van de Sequential Point Trees zijn de grootte van de octree en de mapping die beslist wanneer kleine splats door een grotere splat vervangen worden heel belangrijk. Pas als de octree groot genoeg is, zal het gebruik van de Sequential Point Trees nut hebben. Een grote octree vereist echter een lange voorbereidingsfase. Als de mapping bij de Sequential Point Trees te soepel is, dan zullen er te vlug grote splats gebruikt worden om veel kleine splats te vervangen, wat een storend en visueel slecht resultaat geeft. Is deze mapping te streng, dan worden er niet vlug grote splats gebruikt en heeft het gebruik ervan niet zo'n groot effect op de performantie.

Punten vormen een goede renderingprimitief en de grafische hardware is in staat is om een hoog kwalitatieve weergave te verzorgen, maar er is nog veel ruimte voor verbetering, zowel op het gebied van algoritmen als op het gebied van hardware-ondersteuning. Naar de toekomst toe zal dit ongetwijfeld verbeteren.

Bibliografie

[BK03] Mario Botsch and Leif Kobbelt. High-quality point-based rendering on modern gpus, 2003.

[BSK04] Mario Botsch, Michael Spernat, and Leif Kobbelt. Phong splatting. In S. Rusinkiewicz M. Alexa, editor, *Eurographics Symposium on Point-Based Graphics*. The Eurographics Association, 2004.

[BWK02] M. Botsch, A. Wiratanaya, and L. Kobbelt. Efficient high quality rendering of point sampled geometry, 2002.

[Cat74] Edwin Earl Catmull. *A Subdivision Algorithm for Computer Display of Curved Surfaces*. PhD thesis, Dept. of CS, U. of Utah, December 1974.

[CH02] Liviu Coconu and Hans-Christian Hege. Hardware-oriented point-based rendering of complex scenes, 2002.

[Dee93] Michael F. Deering. Data complexity for virtual reality: Where do all the triangles go?. In *VR*, pages 357–363, 1993.

[DVS03] Carsten Dachsbacher, Christian Vogelgsang, and Marc Stamminger. Sequential point trees. *ACM Trans. Graph.*, 22(3):657–662, 2003.

[GBP04] Gael Guennebaud, Loc Barthe, and Mathias Paulin. Deferred Splatting . In *(EG2004 Proceedings)*, volume 23, pages 653–660. (The Eurographics Association and Blackwell Publishing), septembre 2004.

[GP03] Gael Guennebaud and Mathias Paulin. Efficient screen space approach for Hardware Accelerated Surfel Rendering. In *VMV 20003*, pages 1–10, 2003.

[Hec86] Paul Heckbert. Survey of texture mapping. In *IEEE Computer Graphics and Applications*, pages 56–67, November 1986.

[Kol05] Ravikrishna Kolluri. Provably good moving least squares. In *Proceedings of ACM-SIAM Symposium on Discrete Algorithms*, pages 1008–1018, August 2005.

[LPC+00] Marc Levoy, Kari Pulli, Brian Curless, Szymon Rusinkiewicz, David Koller, Lucas Pereira, Matt Ginzton, Sean Anderson, James Davis, Jeremy Ginsberg, Jonathan Shade, and Duane Fulk. The digital michelangelo project: 3D scanning of large statues. In Kurt Akeley, editor, *Siggraph 2000, Computer Graphics Proceedings*, pages 131–144. ACM Press / ACM SIGGRAPH / Addison Wesley Longman, 2000.

[LW85] M. Levoy and T. Whitted. The use of points as a display primitive. Technical report, University of North Carolina at Chapel Hill, 1985.

[MGAK03] William R. Mark, R. Steven Glanville, Kurt Akeley, and Mark J. Kilgard. Cg: a system for programming graphics hardware in a c-like language. *ACM Trans. Graph.*, 22(3):896–907, 2003.

[MPN+02] Wojciech Matusik, Hanspeter Pfister, Addy Ngan, Paul Beardsley, Remo Ziegler, and Leonard McMillan. Image-based 3d photography using opacity hulls. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 427–437, 2002.

[PZvBG00] Hanspeter Pfister, Matthias Zwicker, Jeroen van Baar, and Markus Gross. Surfels: Surface elements as rendering primitives. In Kurt Akeley, editor, *Siggraph 2000, Computer Graphics Proceedings*, pages 335–342. ACM Press / ACM SIGGRAPH / Addison Wesley Longman, 2000.

[RL00] Szymon Rusinkiewicz and Marc Levoy. QSplat: A multiresolution point rendering system for large meshes. In Kurt Akeley, editor, *Siggraph 2000, Computer Graphics Proceedings*, pages 343–352. ACM Press / ACM SIGGRAPH / Addison Wesley Longman, 2000.

[RPZ02] L. Ren, H. Pfister, and M. Zwicker. Object space ewa surface splatting: A hardware accelerated approach to high quality point rendering, 2002.

[Smi99] Alvy Ray Smith. Smooth operator. *The Economist*, pages 73–74, 1999.

[ZPvBG01] Matthias Zwicker, Hanspeter Pfister, Jeroen van Baar, and Markus Gross. Surface splatting. In Eugene Fiume, editor, *SIGGRAPH 2001, Computer Graphics Proceedings*, pages 371–378. ACM Press / ACM SIGGRAPH, 2001.

[ZRB+04] M. Zwicker, J. Rsnen, M. Botsch, C. Dachsbacher, and M. Pauly. Perspective accurate splatting, 2004.