Voorwoord.

Graag zou ik mijn begeleider ir. Bart Adams willen bedanken voor de goede begeleiding en raad die hij mij gegeven heeft tijdens het werken aan deze thesis.

Vervolgens zou ik Dirk Hellemans willen bedanken, een medestudent informatica en tevens erg goede vriend die mij tijdens het schrijven van deze tekst vaak geholpen heeft met dingen die ikzelf moeilijk kon met mijn gebroken arm.

Als laatste dank aan Jonas Demeulenaere voor het nalezen van mijn tekst en de goede suggesties.

Heverlee, 13 mei 2005

Inhoudsopgave

Hoofdstuk 1

Inleiding.

Deze thesis behoort tot het domein computer graphics binnen informatica. Maar zelfs binnen computer graphics heb je nog een hele waaier aan sub-domeinen waarbinnen onderzoek gedaan wordt. Een paar van van die domeinen zijn het modelleren van 3D figuren en objecten en het renderen ervan. Tegenwoordig verschijnt de ene na de andere digitale animatiefilm in de bioscoop en worden we overspoeld door computer games en special effects in allerhande films. We zien virtuele werelden tot leven komen en zien computer gegenereerde wezens daarin de meest heldhaftige en grappigste dingen doen.

Om veel van deze dingen te verwezenlijken, moeten we beginnen met het modelleren van al deze 3D figuren en objecten, vanaf nu *modellen* genoemd. We bedoelen hiermee dat we deze modellen eerst moeten creëren, vorm geven op de één of de andere manier. Vaak worden hiervoor professionele software pakketten gebruikt zoals Maya of 3D Studio Max. Een andere manier is om bestaande echte-wereld modellen in te scannen met scan-technologie en op basis van de ingescande data het model virtueel te reconstrueren.

Een tweede punt van interesse is op welke manier we een model voorstellen. Traditioneel worden hier polygonen en splines voor gebruikt, maar sinds enkele jaren wordt er steeds meer gebruik gemaakt van een andere manier om modellen voor te stellen, namelijk punt-gebaseerde voorstellingen (*surfels*). In sectie 1.2 wordt er dieper ingegaan op de verschillende manieren om een model voor te stellen en wat hun belangrijkste kenmerken en voor- en nadelen zijn.

Vaak bestaat de voorstelling van ons model uit niet meer dan een verzameling punten. Dit kan zijn omdat we niet meer gegevens hebben (het model is bijvoorbeeld ingescand), maar vaak willen we net met zo'n verzameling punten werken. Een voorbeeld van een applicatie die uitsluitend met

puntenmodellen werkt, is bijvoorbeeld pointshop 3D. Andere toepassingen zijn bijvoorbeeld CSG (Constructive Solid Geometry, zie bijvoorbeeld [7]) of fysisch gebaseerde animatie. In sectie 1.3 zien we dat er een hele reeks technieken bestaan om een model dat voorgesteld wordt door bijvoorbeeld polygonen, te *bemonsteren*, te herleiden naar een verzameling punten (*monsters*).

Het reduceren van dergelijke puntenverzamelingen is vaak één van de belangrijkste voorverwerkingsstappen voor daaropvolgende visualisatie methodes. De tijd nodig om een model voorgesteld door een verzameling punten te renderen, is zo bijvoorbeeld rechtstreeks afhankelijk van het aantal punten in de verzameling. Het spreekt vanzelf dat hier dan ook reeds heel wat onderzoek naar gebeurd is. Een overzicht hiervan wordt eveneens in sectie 1.3 gegeven.

We zullen zien dat de meeste van deze simplificatie technieken zich concentreren op de geometrie van het model, ze proberen met andere woorden de puntenverzameling te reduceren zodat er zo weinig mogelijk detail van de geometrie van het model verloren gaat. In deze thesis zal een bestaand algoritme aangepast en uitgebreid worden zodat het niet enkel met de geometrie van het model rekening houdt, maar ook met de fysische eigenschappen van het model, in dit geval kleur. Ook zullen we niet vertrekken van een initiële verzameling punten, maar van een polygoonmodel zodat we de initiële bemonstering als deel van ons algoritme uitvoeren in functie van het behoud van kleur.

Eerst echter een overzicht van belangrijke termen die doorheen deze tekst gebruikt zullen worden.

1.1 Overzicht van veel gebruikte termen.

In deze sectie worden belangrijke begrippen verklaard die in deze tekst op verschillende plaatsen gebruikt zullen worden. Het is belangrijk om vertrouwd te zijn met deze termen om de inhoud van de volgende hoofdstukken goed te begrijpen. De meeste begrippen zijn echter basisbegrippen binnen computer graphics en mensen met enige achtergrond hierin zullen de meeste van deze termen reeds kennen, maar aangezien deze thesis echter begrijpbaar moet zijn voor een laatstejaarsstudent informatica die niet noodzakelijk een vak omtrent graphics gekregen heeft, toch even dit overzicht.

Polygoon: een polygoon is een vlakke figuur die uit verschillende lijnsegmenten bestaat. De lijnsegmenten kruisen elkaar niet en exact twee

lijnsegmenten raken elkaar bij elk hoekpunt. Een polygoon waarvoor geldt dat eender welke lijn, die in het vlak waarin de polygoon ligt, maximum 2 lijnsegmenten van de polygoon kruist, is convex; zo niet, is hij concaaf. Alle polygoonmodellen waar in deze thesis mee gewerkt wordt, worden verondersteld convex te zijn. (Engels: (convex/concave) polygon.)

Polygoonmodel: een 3D model dat volledig opgebouwd is uit aaneensluitende polygonen. Traditioneel worden hiervoor driehoeken en/of vierhoeken gebruikt. Bij een gesloten model hebben we een deel van de ruimte dat afgesloten is van de rest van de ruimte. (Engels: polygon model or mesh.)

Textuur: een afbeelding die op een model geplakt wordt. (Engels: texture.)

Monster: een punt gelegen op het oppervlak van een model. (Engels: surface point or point sample.)

Bemonstering: een manier om monsters te nemen van een bepaald model. (Engels: point sampling.)

Surfel (splat): afkorting voor 'surface element'. In principe een gewoon punt in 3D, maar aangezien we een visueel continue afbeelding willen, generaliseren we de pure punt-gebaseerde representatie naar een schijfgebaseerde representatie. Een punt wordt dus vervangen door een cirkel, een ellips of iets anders, een vlakke figuur met een oppervlakte dus. Een surfel kan verschillende attributen hebben zoals een normaal en een bepaalde kleur. In deze tekst zal met ellipsen gewerkt worden en de termen surfel en splat zullen vrij door elkaar gebruikt worden. (Engels: surfel or surface splat.)

Surfelmodel: een 3D model dat volledig voorgesteld wordt door surfels. Om het model er vloeiend en continu uit te laten zien kan er met splats gewerkt worden in de objectruimte, maar het is even goed mogelijk om de pure punt-gebaseerde voorstelling pas in de beeldruimte om te zetten naar splats. (Engels: surfelmodel.)

1.2 Manieren om modellen voor te stellen.

We willen een model kunnen gebruiken in applicaties, dus we moeten een praktische voorstelling van ons model vinden, een soort formele taal, waarin we een model kunnen beschrijven of op z'n minst benaderen. Traditioneel wordt het *oppervlak* van een model beschreven, vaak met behulp van wiskundige formules.

De volgende lijst geeft een overzicht van manieren om het oppervlak van een model te representeren:

- Expliciet:
  - Polygonen.
  - Splines.
  - Parameter oppervlak.
  - Surfels.
- Impliciet:
  - De 0-set van een impliciete functie.

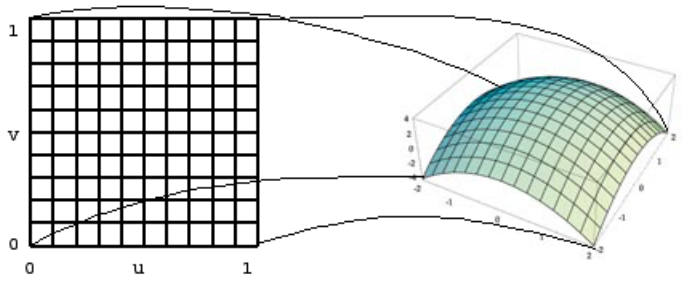
Allereerst is er een verschil tussen een expliciete en een impliciete voorstelling. Een expliciete voorstelling gaat een model rechtstreeks beschrijven door middel van bijvoorbeeld hoekpunten en vlakken (polygoonmodel) of wiskundige functies zoals splines of parameter functies, terwijl een impliciete voorstelling dit niet doet. Een expliciete representatie geeft ons als directe uitvoer welke punten (of verzamelingen van punten) op het oppervlak van het model liggen, bij een impliciete representatie moeten we zelf gaan kijken welke punten in de ruimte op het oppervlak van ons model liggen.

Een voorbeeld van een *expliciete* representatie is een *parameter functie*, die op basis van de waarden van de parameters een punt berekent dat deel uitmaakt van het oppervlak van het model. Zo zie je in figuur 1.1 een oppervlak op basis van de parameters u en v. Een voorbeeld van een *impliciete* representatie is de *0-set van een impliciete functie*, dit zijn alle punten die ervoor zorgen dat de functiewaarde van deze functie 0 is. De impliciete vergelijking van een bol is

$$f(x) = (x-a)^2 + (y-b)^2 + (z-c)^2 - r^2$$

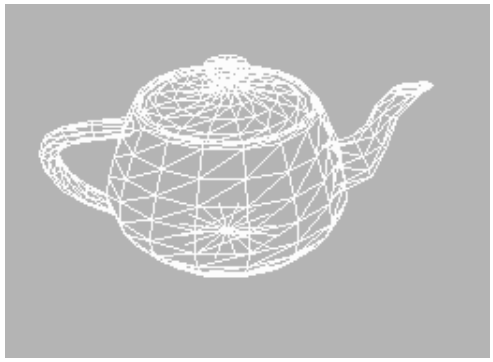
en deze heeft als 0-set alle punten (x_1, y_1, z_1) die op een afstand r van het punt (a, b, c) liggen. Deze impliciete vergelijking kan dus gebruikt worden om bolvormige modellen mee voor te stellen. Het nadeel van zowel deze als de parameter voorstelling is dat het vaak erg lastig is om willekeurige modellen exact voor te stellen met zulke functies.

Van de expliciete voorstellingswijzen is het *polygoonmodel* wellicht het meest bekende en gebruikte model. Het bestaat uit een verzameling aaneensluitende vlakke polygonen, meestal drie- en/of vierhoeken. Een voorbeeld is de theepot die te zien is in figuur 1.2. Polygoonmodellen zijn erg bekend en elk belangrijk 3D modelleringsprogramma biedt op z'n minst ondersteuning voor deze representatie. De reden hiervoor is dat het een erg



Figuur 1.1: Een parameter oppervlak.

simpele voorstellingswijze is die erg efficiënt gevisualiseerd kan worden met behulp van grafische hardware. Bovendien kan zowat elk oppervlak, hoe complex de vorm ook is, met deze voorstellingswijze benaderd worden. Het nadeel is dat we, afhankelijk van de complexiteit van het model, vaak erg veel polygonen nodig hebben. Een simpele bol kan bijvoorbeeld door één impliciete functie voorgesteld worden, terwijl we in theorie oneindig veel polygonen nodig hebben om een bol voor te stellen.



Figuur 1.2: Polygoonmodel.

Splines zijn simpele polynomiale functies die gebruikt worden om een verzameling punten te interpoleren of te benaderen. Splines worden meestal gebruikt om vloeiende oppervlakken te genereren, zoals je kan zien in figuur 1.3. Het is duidelijk te zien dat het oppervlak van de theepot hierbij veel zachter en ronder is vergeleken met figuur 1.2. Meestal worden zowel polygonen als splines aangeboden door grote modelleerprogramma's omdat beide technieken vrij complementair zijn: splines zijn zeer geschikt voor het modelleren van vloeiende curves, polygonen zijn meer geschikt voor de meer

'hoekige' delen van het model. Als je gans je model opgebouwd hebt, heb je vaak de optie om gans je model om te zetten naar polygonen waarbij de splines benaderd worden door honderden polygonen.

Figuur 1.3: Model opgebouwd uit splines.

Huidige gedetailleerde polygoonmodellen bevatten duizenden, vaak zelfs miljoenen polygonen. Door de steeds toenemende mate van detail, worden de polygonen ook steeds kleiner en kleiner. Daardoor worden verschillende polygonen vaak op slechts één beeldpixel geprojecteerd, bijvoorbeeld als het model meer op de achtergrond staat. Dit heeft geleid tot een aantal alternatieve rendering methodes zoals bijvoorbeeld ray tracing, waarbij er voor elke beeldpixel een straal ("ray") in de scene geschoten wordt en op basis van de intersectie van deze straal met de scene (en eventuele gereflecteerde stralen . . .) een kleur berekend wordt voor die pixel. Op deze manier worden enkel de noodzakelijke polygonen verwerkt.

In plaats van het render-mechanisme te vervangen, kunnen we ook de representatie van het model aanpassen. In [3] bespreekt men surfels als rendering primitieven, waarbij men een hiërarchische datastructuur van punten, gelegen op het oppervlak van het model, bijhoudt, wat toelaat om het aantal geprojecteerde surfels per beeldpixel te schatten en zo dus ook de snelheid en de kwaliteit van het renderen te bepalen. Sindsdien hebben punt-gebaseerde representatie methodes een hoop aandacht gekregen als nieuwe manier om modellen voor te stellen. Dit heeft geleid tot een hele hoop algoritmes om oppervlakken voorgesteld door surfels efficiënt te renderen en verschillende algoritmes om polygoonmodellen en surfelmodellen van de ene naar de andere representatie om te zetten.

Een belangrijk besluit omtrent surfels en polygonen vinden we in [6]. Markus Gross onderzoekt of punten een beter grafisch primitief zijn dan polygonen en hij geeft het volgende besluit:

> [..] points are simple and compact, scale to large datasets, support rendering features, and have built-in level of detail (LOD). On the down side, points are purely geometry (no topology), and a fast surface estimation to create points on the fly is needed. However, points are a good alternative graphics primitive, complementing triangles and splines.

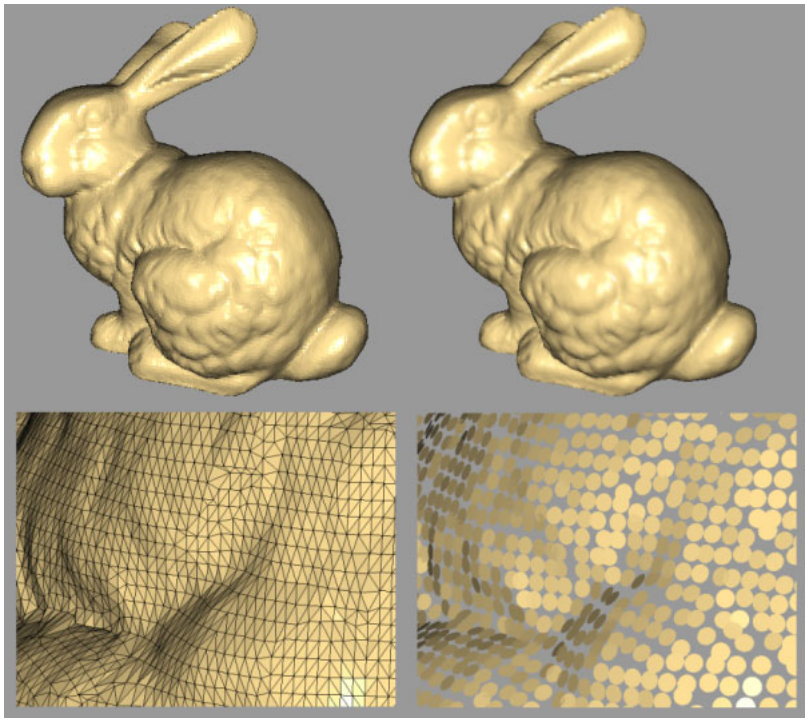
Het zal ook uit de resultaten van deze thesis blijken dat surfels vooral geschikt zijn voor de representatie van zeer gedetailleerde modellen, terwijl polygonen best gebruikt kunnen worden voor modellen met minder detail. Zo kan een kubus met texturen door 6 polygonen voorgesteld worden, terwijl we, afhankelijk van de texturen, honderden of zelfs duizenden surfels voor hetzelfde doeleinde nodig hebben. Surfels willen dus niet de traditionele grafische pipeline die gebaseerd is op polygonen vervangen, maar wel complementeren.

In figuur 1.4 zie je een vergelijking van polygonen en surfels. Hetzelfde model wordt daarin weergegeven in beide representaties.

1.3 Bemonstering en simplificatie.

Zoals reeds gezegd is er een groeiende interesse in punt-gebaseerde voorstellingen van modellen (zie bijvoorbeeld [7]) en punt-gebaseerde rendering en dus is er vaak de nood om een polygoonmodel om te zetten naar een verzameling punten. We noemen dit proces bemonsteren. Hier zijn verschillende manieren voor. Zo hebben we de optie om het model te bemonsteren met in het achterhoofd een bepaalde mate van detail die we willen bewaren of een maximum aantal monsters dat we in de uiteindelijke verzameling willen. We kunnen natuurlijk even goed een erg dichte bemonstering nemen en de initiële verzameling monsters gaan herleiden tot een kleinere verzameling. Dit herleiden van een grote hoeveelheid gegevens tot een kleinere hoeveelheid noemen we simplificatie. Simplificatie is ook belangrijk omdat het vaak een erg belangrijke voorverwerkingsstap voor het renderen van een model is. De tijd die nodig is om een model te renderen, is namelijk rechtstreeks afhankelijk van het aantal polygonen of surfels waaruit het model opgebouwd is.

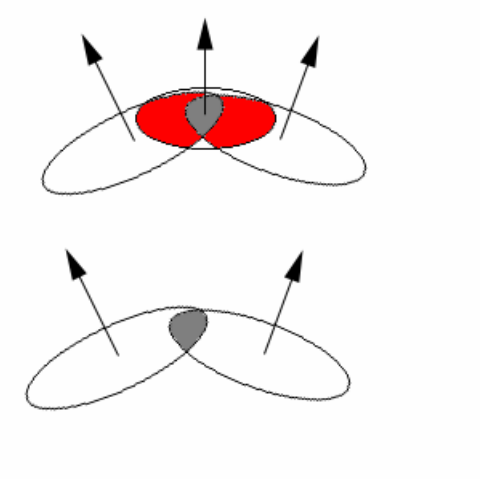
Een tweede reden waarom we aan simplificatie willen doen, is wegens de manier waarop splats op het scherm gerenderd worden (*splatting*). De exacte manier waarop dit gebeurt is niet zo belangrijk, maar wel het feit dat het renderen langer duurt voor pixels die overlappende splats weergeven.



Figuur 1.4: Hetzelfde model voorgesteld met polygonen (links) en met surfels (rechts). Bron: Bart Adams.

In figuur 1.5 zien we bovenaan drie splats. De rode gebieden en het grijze gebied zijn de plaatsen waar twee respectievelijk drie splats elkaar overlappen en de pixels die deze gebieden weergeven, vragen dus meer tijd om berekend te worden (doordat bijvoorbeeld de normalen van alle betrokken splats geïnterpoleerd moeten worden). Vandaar dat we bij het simplificeren vaak ook de overlappingen tussen splats willen verkleinen. Onderaan figuur 1.5 zien we twee splats die dezelfde oppervlakte bedekken, maar met minder overlappingen.

In [4] worden verschillende manieren besproken en vergeleken voor simplificatie van punt-gebaseerde oppervlakken. Zo wordt er een groeperingsmethode besproken die de puntenverzameling in een aantal deelverzamelingen opsplitst en per deelverzameling één punt kiest als vertegenwoordiger die alle punten in de deelverzameling vervangt. Als tweede worden er een paar iteratieve simplificatie methodes besproken die gebruik maken van verschillende atomische decimerende operatoren. Verschillende mogelijke operaties worden in een prioriteitsrij gesorteerd op basis van een maat voor de fout die door die operatie veroorzaakt wordt en de decimerende operatie die de

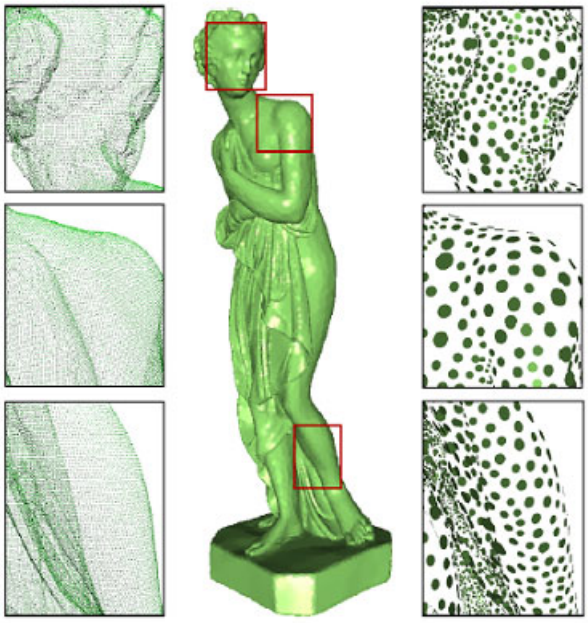


Figuur 1.5: Overlappende delen van splats vragen meer tijd om op het scherm weer te geven.

kleinste fout introduceert, wordt dan eerst toegepast. Als laatste methode wordt er een deeltjes simulator besproken waarbij men het begeerde aantal punten over het oppervlak verspreid en hierop een punt-afstotingsalgoritme laat werken zodat de deeltjes zichzelf optimaal verspreiden over het oppervlak.

Hiernaast bestaan er nog talrijke methodes om puntenverzamelingen te simplificeren, maar zowat alle methodes houden enkel rekening met de geometrie van het model. Dit wil zeggen dat ze op de één of andere manier proberen de geometrie van het oorspronkelijke model te benaderen met een bepaalde bovengrens of maat voor de fout. De resulterende punten hebben natuurlijk allemaal een kleur en indien er veel punten overblijven, zal er wellicht ook veel detail van de kleuren van de oorspronkelijke punten bewaard blijven. Helaas is het net zo goed mogelijk dat de resulterende verzameling veel minder punten bevat dan de oorspronkelijke verzameling en dat er heel veel detail betreffende de kleur verloren gaat.

We willen dus concreet een simplificatiealgoritme construeren dat een model voorstelt met zo weinig mogelijk surfels met zo weinig mogelijk overlappingen als voorverwerkingsstap voor het renderen van het model. We willen hierbij niet enkel de geometrie in beschouwing nemen, maar ook de kleur. In [1] wordt een algoritme beschreven voor het simplificeren van dichte puntenverzamelingen zoals getoond wordt in figuur 1.6. Dit algoritme bleek een goede basis te zijn voor het algoritme dat we willen bekomen en dus werd de keuze gemaakt om dit algoritme te implementeren met de nodige



Figuur 1.6: Illustratie van het algoritme beschreven in [1].

aanpassingen en uitbreidingen. In tegenstelling tot [1] vertrekken we niet van een dichte puntenverzameling die het model voorstelt, maar van een polygoonmodel met texturen. Zo kunnen we ook een bemonstering kiezen in functie van het behouden van zoveel mogelijk detail van de texturen.

1.4 Overzicht van de overige hoofdstukken.

In hoofdstuk 2 wordt een algoritme besproken dat een polygoonmodel omzet in een verzameling surfels. Het algoritme begint met het polygoonmodel te bemonsteren en vervolgens de verzameling monsters te reduceren tot een (veel) kleinere verzameling surfels. We zien zowel een basis algoritme, dat ervoor zorgt dat de geometrie van het gereduceerde model niet te ver afwijkt van de geometrie van het polygoonmodel, als een uitgebreid algoritme dat er eveneens voor zorgt dat er zoveel mogelijk detail van de textuur van het originele model bewaard blijft.

Hoofdstuk 3 toont resultaten. Polygoonmodellen worden er vergeleken met de surfelmodellen geproduceerd door beide algoritmes. Ook de invloed van verschillende parameters zullen aangetoond worden.

Hoofdstuk 4 geeft een kritische analyse van het algoritme, wat mogelijke beperkingen zijn en waar er ruimte is voor uitbreidingen. Ook zal er een tweede uitbreiding van het basis algoritme besproken worden.

Hoofdstuk 5 geeft een algemeen besluit.

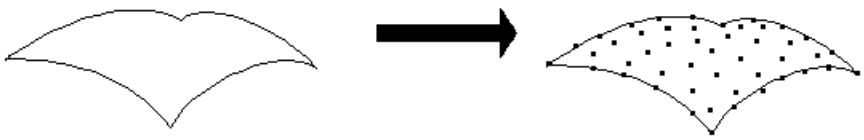
Hoofdstuk 2

Algoritme.

In dit hoofdstuk wordt een algoritme beschreven dat een model bestaande uit polygonen, omzet in een verzameling surfels. Zoals in de inleiding reeds beschreven, willen we een surfelmodel bekomen met zo weinig mogelijk surfels die zo weinig mogelijk overlappen zodat het model zo snel mogelijk kan gerenderd worden.

We beginnen met het bemonsteren van het polygoonmodel zoals beschreven wordt in sectie 2.1. In sectie 2.2 wordt het basis algoritme besproken. We zullen zien hoe de initiële verzameling monsters gereduceerd wordt tot een veel kleinere verzameling surfels op basis van een fouttolerantie zodat het uiteindelijke surfelmodel niet te hard afwijkt van het originele polygoonmodel. In sectie 2.3 wordt het basis algoritme uit sectie 2.2 uitgebreid om ook kleur en texturen in beschouwing te nemen.

2.1 Bemonstering van het model.

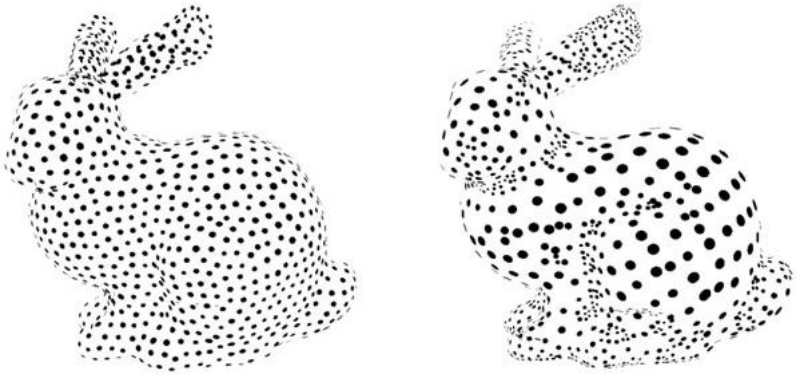


Figuur 2.1: Bemonsteren van een oppervlak.

De kern van het algoritme gaat in principe gewoon een grote verzameling initiële punten omzetten naar een kleinere groep punten met een bepaalde oppervlakte (splats). Als we vertrekken van een polygoonmodel, is het dus

belangrijk om daar eerst een goede initiële bemonstering op toe te passen. Figuur 2.1 toont deze stap. Twee dingen zijn belangrijk bij het kiezen van een bemonstering.

Ten eerste hebben we de keuze tussen een uniforme en een niet-uniforme bemonstering. Een uniforme bemonstering zal proberen om het ganse oppervlak van het polygoonmodel te bemonsteren met een gegeven regelmaat. De punten liggen met andere woorden steeds even ver van elkaar verwijderd. Ongeacht hoeveel detail het model bevat in zijn geometrie, er zullen ongeveer even veel monsters genomen worden op even grote regio's van het model. Bij een niet-uniforme bemonstering is dit niet zo. Typisch worden er meer monsters genomen in gebieden met een rijke geometrie dan in gebieden met een arme geometrie. Figuur 2.2 toont duidelijk het verschil aan.
Ten tweede is er de dichtheid van de bemonstering. Zowel bij een uniforme als bij een niet-uniforme bemonstering heb je een maat nodig voor hoe dicht er bemonsterd moet worden en hoeveel punten je dus in totaal zal bekomen.



Figuur 2.2: Uniform vs niet-uniform. Bron: Mark Pauly

Voor ons algoritme kiezen we voor een (min of meer) uniforme bemonstering aangezien een niet-uniforme bemonstering meer kans heeft om gaten in ons uiteindelijk surfelmodel te introduceren. Hoe we exact zullen bemonsteren, wordt uitgelegd in sectie 2.1.1.

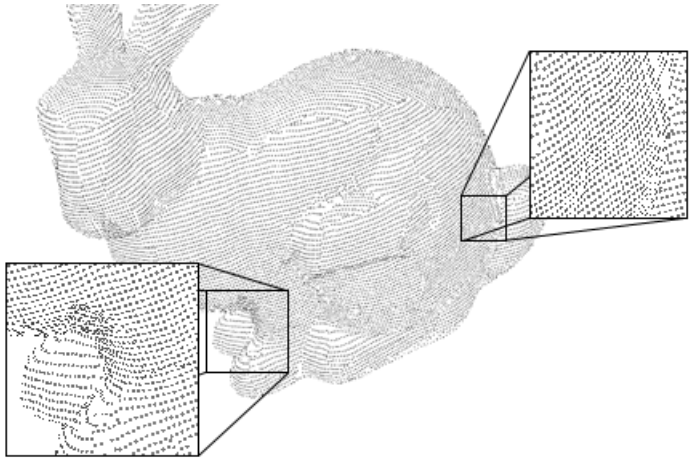
2.1.1 Uniforme bemonstering van het model.

Voor het testen van het algoritme maak ik onder andere gebruik van modellen zoals de bunny, de dragon en andere modellen die gratis te downloaden zijn op de site van Stanford University[1]. Deze modellen zijn zeer gedetailleerde triangle meshes bestaande uit duizenden hoekpunten en vlakken.

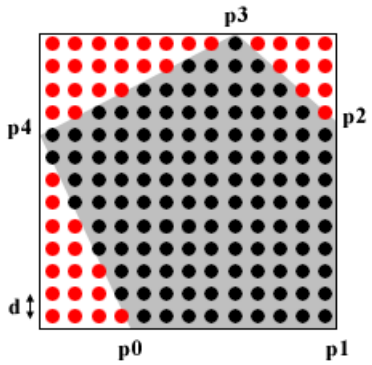
[1] http://graphics.stanford.edu/data/3Dscanrep/

Gezien de grote mate van detail van deze modellen volstaat het om gewoon de hoekpunten als initiële monsters te nemen.

Als voorbeeld zien we in figuur 2.3 deze bemonstering toegepast op de bunny. We zien duidelijk dat voor dit model de hoekpunten een dichte bemonstering vormen.



Figuur 2.3: Initiële bemonstering van de bunny.

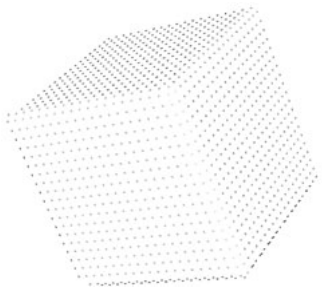


Figuur 2.4: Uniforme bemonstering.

Bij meer simpele modellen of meer algemeen wanneer we merken dat we te weinig initiële monsters hebben, wordt er uniform bemonsterd op basis

van een afstand d. Per polygoon wordt de omliggende rechthoek berekend en die wordt rij per rij bemonsterd zoals je kan zien in figuur 2.4. Het is duidelijk dat de waarde d bepaalt hoe dicht de bemonstering is. De zwarte punten behoren tot de initiële monsters, de rode niet aangezien deze buiten het vlak vallen dat bemonsterd wordt.
Deze manier van bemonsteren is meer geschikt wanneer we werken met modellen die texturen bevatten. Elke monster zal een kleur krijgen overeenkomstig met een pixel uit de textuur die op het face geplakt wordt en aldus zal de initiële bemonstering zo in grote mate bepalen hoeveel detail uit de texturen er maximum bewaard zal blijven.

Een manier om de waarde d te kiezen is om deze te baseren op de lengte van de kleinste rand van het model. We stellen voorop dat er een bepaald aantal monsters n moeten genomen worden op deze afstand. In figuur 2.5 zie je een afbeelding met daarop een kubus die bemonsterd is volgens de hierboven beschreven methode. Er werd vooropgesteld dat er 20 monsters moesten genomen worden voor de kleinste rand.

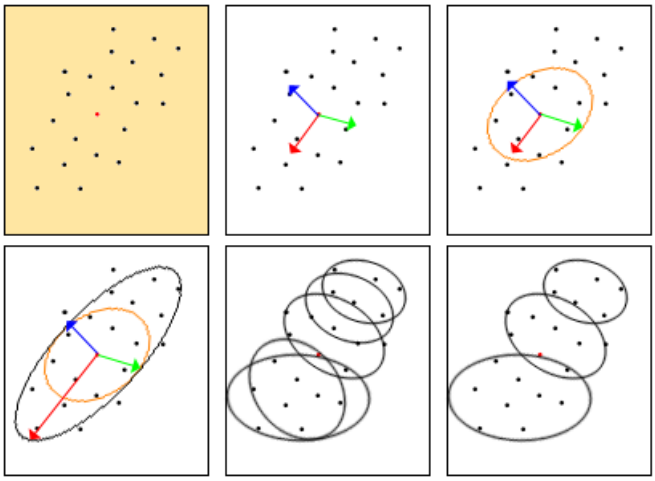


Figuur 2.5: Initiële bemonstering van een kubus.

2.2 Geometrie gebaseerde simplificatie.

In deze sectie wordt het basis algoritme uitgelegd. Dit is de kern van mijn thesis en zal in sectie 2.3 verder uitgebreid worden om ook rekening te houden met kleur en texturen. Dit algoritme is grotendeels gebaseerd op de techniek beschreven in [1].

Het algoritme vertrekt van een initiële verzameling van punten $P = \{p_i\}$, gelegen op het oppervlak van het model. Dit zijn de monsters uit de vorige sectie. In figuur 2.6 zien we een grafisch overzicht van de belangrijkste stappen in het algoritme en in de eerste tekening van figuur 2.6 zien we een aantal van deze punten. Voor elk punt wordt er een omgeving opgesteld die de k dichtstbijliggende punten bevat, genaamd $N_k(p_i)$. We zullen zien dat



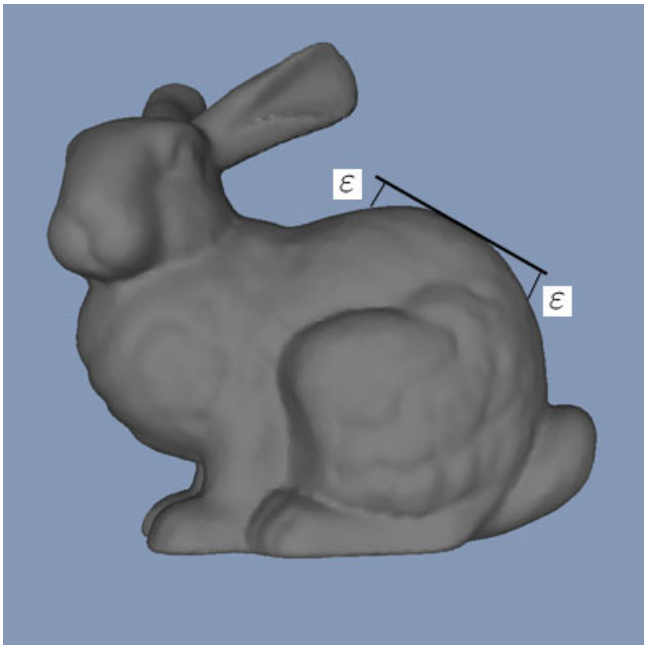
Figuur 2.6: Overzicht van het algoritme.

deze omgevingen nadien op verschillende plaatsen in het algoritme gebruikt worden, met als belangrijkste toepassing het genereren van de splats (sectie 2.2.2). Er wordt namelijk een splat t_i gegenereerd voor elk punt p_i. Het punt p_i noemen we het startpunt van de splat t_i. In de afbeelding wordt dit geïllustreerd voor het rode punt.

Het idee is dat deze splat een deel van het oppervlak van ons model bedekt en dat de fout (de afstand tussen de splat en het stuk van het oppervlak dat bedekt wordt) niet groter is dan een vooropgestelde fouttolerantie ε zoals in figuur 2.7 te zien is. We zijn vrij om met circulaire of elliptische splats te werken. Circulaire splats zijn eenvoudiger om voor te stellen en om mee te werken, elliptische splats hebben de mogelijkheid om zich beter aan te passen aan de kromming van het oppervlak en dit is meteen ook de reden waarom er met elliptische splats gewerkt zal worden. Een elliptische splat wordt voorgesteld door een centrum c_i, een normaal n_i en twee richtingsvectoren u_i en v_i welke de grote respectievelijk de kleine as voorstellen.

We beginnen dus met het zoeken van deze richtingsvectoren zoals je kan zien in de tweede tekening van figuur 2.6. Daarna genereren we een circulaire splat $t_i = (c_i, n_i, r_i)$ en deze wordt nadien mogelijk nog uitgebreid tot een elliptische splat $t_i = (c_i, n_i, u_i, v_i)$. (Stap 3 en 4 in figuur 2.6). Meer uitleg over de manier waarop de splats gegenereerd worden, vind je in sectie 2.2.2.

We hebben nu voor elk punt p_i uit de initiële verzameling punten een splat t_i. Het is meteen duidelijk dat veel van deze splats overbodig zijn. De bedoeling is dus om uit de verzameling van alle splats $T = \{t_j\}$ een



Figuur 2.7: Illustratie van de fouttolerantie ε.

minimale set T' te kiezen die gans het model bedekt. (Tekening 5 en 6 van figuur 2.6). In praktijk zullen we een set T' zoeken die gans P bedekt en vandaar dat het zo belangrijk is dat de initiële bemonstering dicht genoeg is, anders kunnen er gaten in het surfelmodel zitten. Hoe dit in zijn werk gaat, vind je in sectie 2.2.3 en 2.2.4.

2.2.1 Dichtstbijliggende buren zoeken.

We hebben dus een verzameling punten $P = \{p_i\}$. Voor elk punt p_i wordt er een omgeving $N_k(p_i)$ opgesteld die de k dichtstbijliggende buren bevat. De afstand $d_i = \|p_i - p_k\|$ tot de k-de dichtste buur kan gebruikt worden als een maat voor de lokale dichtheid van de bemonstering. De grootte van deze omgeving is per definitie gelijk aan $\omega_i = \pi d_i^2$.

De meest efficiënte manier om voor elk punt zijn k dichtstbijliggende punten te vinden, is door middel van een *kd-tree*. De geïnteresseerde lezer vindt meer informatie over kd-trees en hun werking in [10].

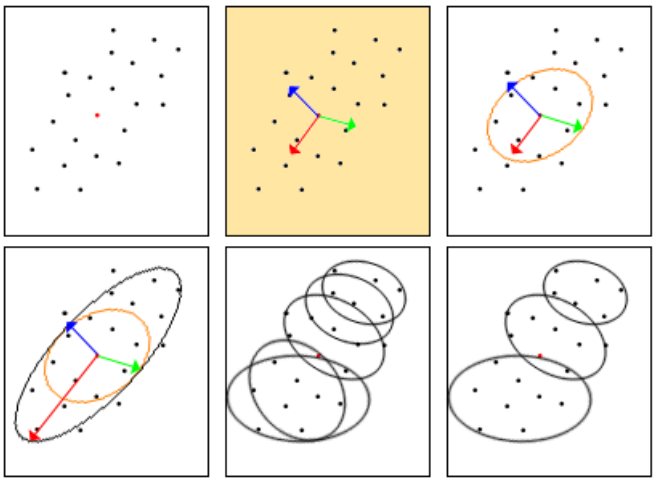
2.2.2 Generatie van initiële splats.

We gaan nu voor elk punt p_i een splat t_i genereren. Het punt p_i noemen we het startpunt van de splat t_i. We beginnen met het genereren van een

circulaire splat $t_i = (c_i, n_i, r_i)$ en deze wordt nadien mogelijk nog uitgebreid tot een elliptische splat $t_i = (c_i, n_i, u_i, v_i)$. Twee dingen moeten we bekomen, ten eerste de richtingen van de normaal, van de grote en van de kleine as van de splat en ten tweede de positie van het centrum van de splat en de grootte van de drie daarnet genoemde richtingen.

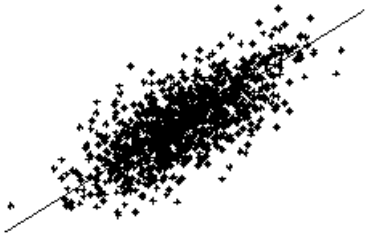
Berekenen van de richtingsvectoren.

We gaan dus nu voor elk punt $\{p_i\}$ drie genormaliseerde richtingsvectoren zoeken op basis van de ligging van p_i en zijn k dichtstbijliggende punten. Dit wordt voor één punt getoond in de tweede tekening van figuur 2.8.



Figuur 2.8: Overzicht van het algoritme. Berekenen van de richtingsvectoren.

Het zoeken van deze richtingen is eigenlijk een standaard probleem dat opgelost wordt door gebruik te maken van Principal Component Analysis (afgekort tot PCA). PCA is een statistische techniek die gebruikt wordt voor het analyseren van data, met andere woorden het vinden van patronen (gelijkheden en ongelijkheden) in data. Het doel is het vinden van een set van m loodrechte vectoren in de data ruimte die zoveel mogelijk de variantie van de data voor hun rekening nemen. Deze m hoofdrichtingen zijn dus de richtingen volgens dewelke de data het meest gespreid is. De data is het meest gespreid volgens de eerst gevonden richting.
Een voorbeeld van een twee-dimensionale data wolk zie je in figuur 2.9. De lijn toont de richting van de eerste principale component. Het is duidelijk zichtbaar dat de data het meest gespreid is volgens deze richting.



Figuur 2.9: PCA van een twee-dimensionale data wolk.
Bron: http://www.cis.hut.fi/aapo/papers/NCS99web/node5.html

PCA werkt als volgt: we stellen de covariantiematrix C op die als element c_{ij} de covariantie tussen dimensie i en dimensie j bevat. In de Euclidische ruimte ziet deze matrix er dus als volgt uit:

$$C = \begin{pmatrix} cov(x,x) & cov(x,y) & cov(x,z) \\ cov(y,x) & cov(y,y) & cov(y,z) \\ cov(z,x) & cov(z,y) & cov(z,z) \end{pmatrix}$$

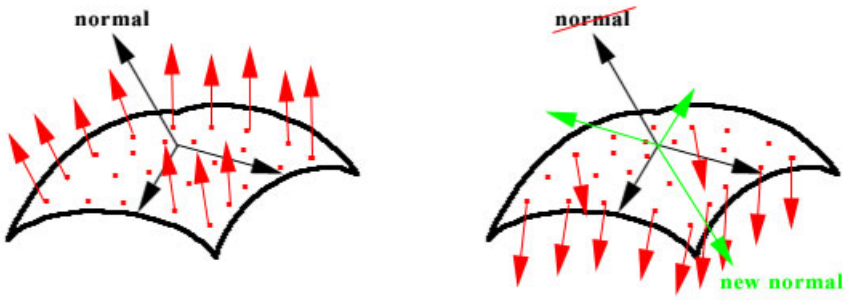
Van deze matrix berekenen we de eigenvectoren en sorteren deze volgens hun bijhorende eigenwaarde. De eigenvector horende bij de grootste eigenwaarde geeft de grootste spreiding weer en we stellen dus de grote as van de ellips gelijk aan deze richting (zie de rode pijl in de tweede afbeelding van figuur 2.8). De eigenvector horende bij de tweede grootste eigenwaarde komt overeen met de richting van de kleine as (zie de groene pijl) en de eigenvector horende bij de kleinste eigenwaarde is de normaal (de blauwe pijl).

Er is nog één kleine bedenking en die wordt geïllustreerd in figuur 2.10. Als we een oppervlak hebben dat bol is, dan zal PCA de normaal altijd laten wijzen in de richting van de bolle kant. Dit kan echter de binnenkant van ons model zijn, zodat we eigenlijk willen dat de normaal de andere richting op wijst (de normaal wijst standaard naar de buitenkant van modellen in graphics). Wanneer we dus de vectoren hebben berekend met behulp van PCA, doen we de volgende test: we berekenen een tweede normaal op basis van de normalen van de punten (rode pijlen in figuur 2.10) met de volgende formule

$$\overline{n}_i \quad + \sum_{p_j \in N_k(p_i)} \overline{n}_j \cdot e^{\frac{-3\|d_{ij}\|}{\|d_{ik}\|}}$$

Hierbij zijn $\overline{n}_i$ en $\overline{n}_j$ de genormaliseerde normalen van p_i respectievelijk p_j. Dit is in feite een gewogen gemiddelde van de normalen van p_i en zijn k dichtstbijliggende buren waarbij de normalen van punten die dicht bij p_i liggen, harder meetellen dan normalen van punten die ver van p_i liggen. De e-factor zorgt hiervoor. Als een punt dicht bij p_i ligt, zal $\|d_{ij}\|$ klein zijn en zal de exponent naar 0 gaan; een punt dat dicht bij de k-de buur ligt, zal een afstand hebben dicht bij $\|d_{ik}\|$ en dus zal de exponent naar -3 gaan. De waarde -3 is een parameter die bepaald in welke mate de normalen van verafgelegen punten van belang zijn in de berekening van de gewogen normaal. Hoe groter deze waarde, hoe minder de invloed van deze verafgelegen normalen. De exacte waarde is in deze toepassing echter niet zo van belang.

Wanneer we deze gewogen normaal berekend hebben, kijken we of deze een hoek maakt die groter is dan 90° met de normaal berekend door PCA en indien dit het geval is, draaien we de richting van zowel de normaal als de twee richtingsvectoren om. Dit is te zien in het rechter gedeelte van figuur 2.10. De zwarte vectoren zijn de resultaten van PCA, de groene vectoren zijn de inverses van de zwarte vectoren.
Voor degenen die meer willen weten over de werking van PCA en enkele van zijn toepassingen, zie [5].

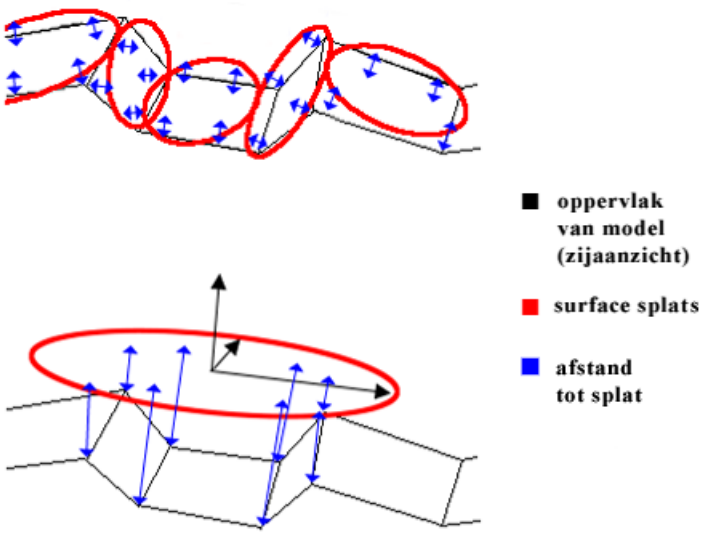


Figuur 2.10: Draai de vectoren om indien de PCA normaal en de gemiddelde normaal van de punten een hoek maken van meer dan 90°.

Berekenen van de grootte van de splat.

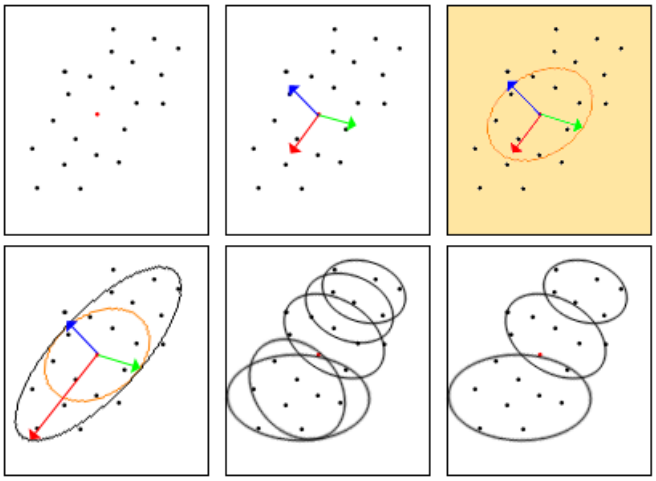
Nu we de drie hoofdrichtingen van onze splat hebben, is de volgende stap het bepalen van de exacte positie en de grootte van de splat. We willen

de splat natuurlijk zo groot mogelijk maken zodat we het model met zo weinig mogelijk splats kunnen benaderen. Anderzijds mogen we de splat niet te groot maken als we niet te veel detail willen verliezen. Hier komt de fouttolerantie ε naar voor wat de belangrijkste parameter van het algoritme is. Deze bepaalt namelijk hoe ver de splat verwijderd mag zijn van het oppervlak van het model en is dus meteen ook een maat voor de hoeveelheid detail die bewaard zal blijven.

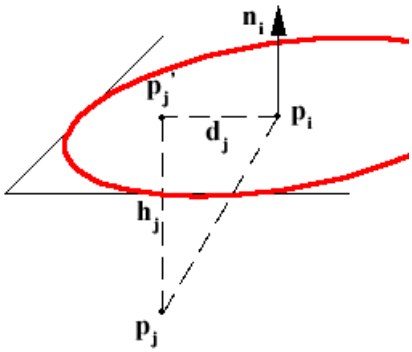


Figuur 2.11: Illustratie van de functie van de fouttolerantie parameter.

Figuur 2.11 verheldert dit. De onderste afbeelding toont duidelijk aan dat indien de fouttolerantie groot is, de splat een groot deel van het oppervlak bedekt en dat er dus veel detail uit de geometrie verloren gaat; zo worden de drie geïllustreerde vlakken uit het oppervlak herleid tot één splat.
De bovenste afbeelding verduidelijkt wat er gebeurt indien de fouttolerantie kleiner is. Het oppervlak wordt nu benaderd door meerdere kleinere splats die dichter bij het oppervlak liggen.



Figuur 2.12: Overzicht van het algoritme. Berekenen van de grootte van de splat.



Figuur 2.13: Projectie van punt p_j.

We willen dus een splat t_i genereren voor het punt p_i. We kennen inmiddels de genormaliseerde normaal $\overline{n}_i$ en de richtingsvectoren u_i en v_i van deze splat en willen nu de positie van het centrum c_i van de splat en de grootte van de twee richtingsvectoren bekomen. Een ellips wordt namelijk volledig gedefinieerd door zijn positie en zijn grote en kleine as. Zoals reeds gezegd in het begin van deze sectie beginnen we met het genereren van een circulaire splat. (Zie tekening 3 van figuur 2.12). We gaan punten p_j die in de omgeving van p_i liggen, projecteren op het vlak dat gedefinieerd wordt door p_i en de normaal $\overline{n}_i$. De formule

$$h_j = \overline{n}_i \cdot (p_j - p_i)$$

geeft de afstand van punt p_j tot dit vlak en de formule

$$d_j = \|(p_j - p_i) - (\overline{n}_i \cdot (p_j - p_i))\overline{n}_i\|$$

geeft de afstand binnen dit vlak van het geprojecteerde punt p'_j tot p_i zoals in figuur 2.13 te zien is. Belangrijk om op te merken is dat h_j geen absolute afstand is en dus zowel positief of negatief kan zijn afhankelijk van langs welke kant van het vlak p_j ligt.

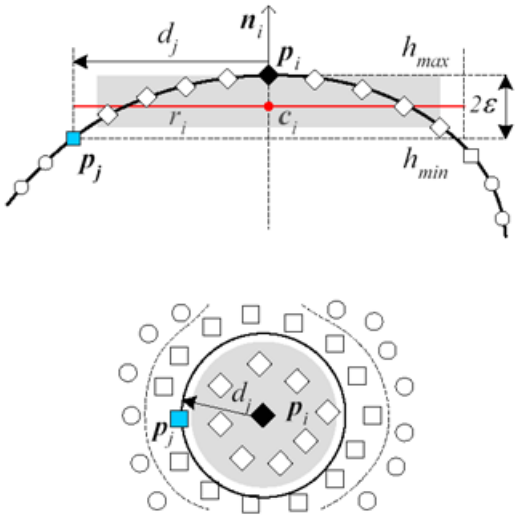
We gaan nu alle naburige punten doorlopen in volgorde van oplopende waarde d_j. Dit heeft als effect dat de straal van de circulaire splat groter en groter wordt. We houden ook twee waarden h_{min} en h_{max} bij die de kleinste respectievelijk de grootste waarde van h_j bijhouden en het groeien stopt wanneer het interval $[h_{min}, h_{max}]$ groter wordt dan 2ε zoals te zien is in figuur 2.14. We kiezen het centrum dan in het midden van dit interval onder het punt p_i zodat alle punten die we tot dan toe beschouwd hebben op een afstand kleiner dan ε verwijderd liggen van de splat.

We kunnen het centrum en de straal van de splat gemakkelijk berekenen met de volgende formules

$$c_i = p_i + \frac{h_{min} + h_{max}}{2}\overline{n}_i$$

$$r_i = \max_{p_j} \|(p_j - c_i) - (\overline{n}_i \cdot (p_j - c_i))\overline{n}_i\|$$

Hierbij heeft r_i de waarde van de grootste geprojecteerde afstand d_j van alle punten p_j die beschouwd werden voordat de fouttolerantie overschreden werd.

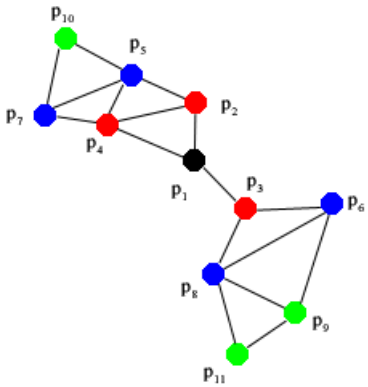


Figuur 2.14: Splat groei procedure. Afbeelding afkomstig uit [1].

Nu rest nog de vraag hoe we precies de buren doorlopen en hierbij wordt duidelijk dat de keuze van de parameter k (het aantal dichtstbijliggende buren waarmee gewerkt wordt) toch belangrijk is. Indien deze slecht gekozen wordt, kan deze voor subtiele problemen zorgen.
Doordat we voor elk punt de k dichtstbijliggende punten opgeslagen hebben, vormen alle punten een gerichte grafe waarbij elk punt verbonden is met zijn k dichtstbijliggende punten. We gaan nu gewoon deze grafe breedte-eerst doorlopen vanaf het punt p_i waarvoor we een splat aan het genereren zijn. Bij het berekenen van de grootte van splat t_i van punt p_i beschouwen we eerst de k punten uit $N_k(p_i)$ die we aflopen volgens oplopende waarde d_j (herinner dat d_j de projectie van het punt p_j op het vlak gevormd door p_i en $\overline{n}_i$ is, zoals hierboven reeds uitgelegd). Tijdens het verwerken van deze k punten houden we een lijst bij van al hun buren die tot dan toe nog niet beschouwd geweest zijn (we houden enkel buren bij die nog niet beschouwd zijn om lussen te voorkomen). Wanneer alle k punten uit $N_k(p_i)$ verwerkt zijn, sorteren we al de punten uit de lijst van nog niet beschouwde buren, opnieuw volgens de waarde d_j en beginnen we deze lijst van punten te doorlopen. Bij het verwerken van deze punten, houden we opnieuw een lijst bij van hun nog niet beschouwde buren enzovoort; dit tot het interval $[h_{min}, h_{max}]$ groter wordt dan 2ε.

Figuur 2.15 illustreert deze procedure. We zijn een splat aan het genereren voor p_1. Stel dat de waarde van k gelijk is aan 3. We beschouwen dus eerst

de 3 dichtstbijliggende buren van p_1 (de rode punten dewelke in $N_k(p_1)$ zitten) in volgorde van hun projectie d_j. We verwerken dus p_2, dan p_3 en daarna p_4. Ondertussen houden we de nog niet verwerkte buren uit $N_k(p_2)$, $N_k(p_3)$ en $N_k(p_4)$ bij voor de volgende iteratie.
$N_k(p_2)$ bevat p_1, p_4 en p_5, $N_k(p_3)$ bevat p_1, p_6 en p_8 en $N_k(p_4)$ bevat p_2, p_5 en p_7. Alle nog niet verwerkte buren uit deze verzamelingen zijn p_5, p_6, p_8 en p_7, dewelke gesorteerd worden volgens hun waarde d_j, wat de volgende lijst oplevert: p_5, p_8, p_6 en p_7. Dit zijn de blauwe punten op figuur 2.15 en zullen nu verwerkt worden.

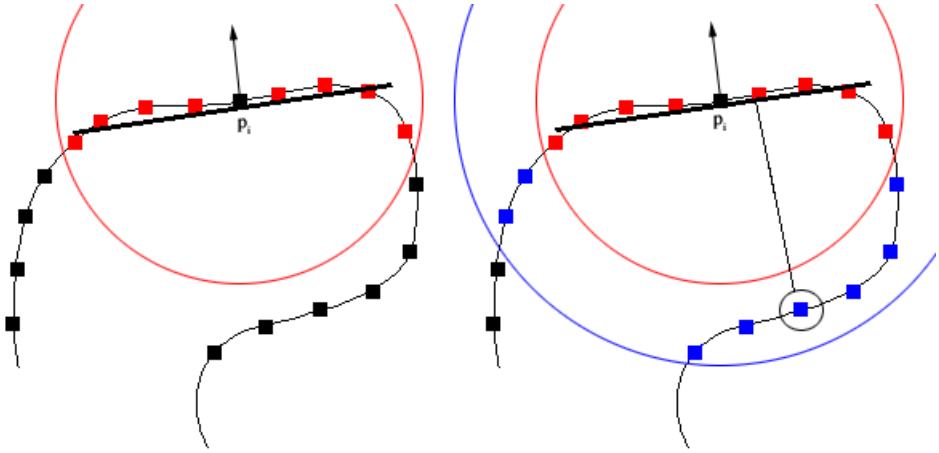


Figuur 2.15: Breedte eerst doorlopen van de buren-grafe.

Een subtiel punt van attentie hierbij is dat de keuze van k belangrijk is. Als deze te groot gekozen wordt, bestaat het gevaar dat er een punt tot $N_k(p_i)$ behoort dat ver verwijderd ligt van p_i, maar waarvan de projectie wel dicht bij p_i ligt, zodat dit punt reeds vroeg beschouwd wordt, maar waardoor het interval $[h_{min}, h_{max}]$ wel meteen 2ε overschrijdt, zodat de groei-procedure stopt en de splat veel kleiner is dan hij had kunnen zijn. Figuur 2.16 verduidelijkt dit.

In het linkerdeel zie je een goede keuze voor k. In het rechterdeel is k te groot gekozen en zien we dat het omcirkelde blauwe punt voor problemen zorgt. Dit punt ligt namelijk vrij ver van p_i verwijderd, maar zijn projectie d_j is wel het kleinst zodat het als eerste punt beschouwd zal worden. De kans is dan groot dan het interval $[h_{min}, h_{max}]$ meteen 2ε overschrijdt en het groeien al stopt. Onze splat is dan zeer klein en bedekt dan enkel het punt p_i terwijl hij in feite al de rode punten had kunnen bedekken zoals in het linkerdeel van de figuur getoond wordt. Natuurlijk is dit afhankelijk van de

waarde van ε. Als deze groot is, is de keuze van k iets minder van belang, maar belangrijker, wanneer ε klein is, is k best niet te groot.
k mag echter ook niet te klein gekozen worden, want anders bestaat het gevaar dat een groep van $k+1$ dicht bij elkaar liggende punten enkel elkaar als k dichtstbijliggende buren hebben en dat de splats van deze punten nooit groter kunnen worden dan de grootte van het gebied waarin deze $k+1$ punten liggen zodat er gaten kunnen ontstaan. Figuur 2.17 toont dit geval voor $k = 3$. Dit probleem zal echter minder snel voorkomen bij een uniforme bemonstering en een keuze van een k die niet erg klein is zoals 2, 3 of 4. Uit experimenten bleek een waarde variërend van 10 tot 20 steeds goed te werken.

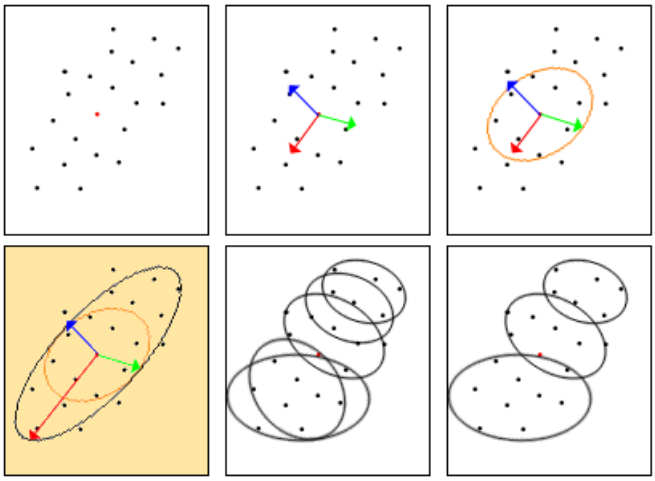


Figuur 2.16: Mogelijk probleem bij keuze van een te grote k.

Figuur 2.17: Mogelijk probleem bij keuze van een te kleine k. $k = 3$ in dit geval.

Uitbreiding tot ellips.

We hebben nu een circulaire splat $t_i = (c_i, n_i, r_i)$ en deze kan nu nog uitgebreid worden tot een elliptische splat $t_i = (c_i, n_i, u_i, v_i)$ zodat de splat



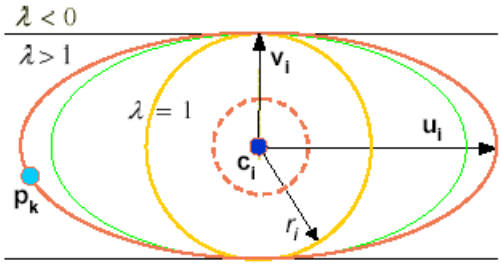
Figuur 2.18: Overzicht van het algoritme. Uitbreiding tot ellips.

zich beter aanpast aan de lokale kromming van het oppervlak zoals te zien is in afbeelding vier in figuur 2.18. We kennen reeds de richting van de beide assen (zie pagina 20) en gaan de splat nu verder uitbreiden in de richting van zijn grote as. De grootte van de kleine as stellen we gelijk aan de grootte van de straal die we hierboven reeds gevonden hebben; de grootte van de grote as gaan we nu berekenen.

Voor het genereren van de circulaire splat hebben we de buren-grafe doorlopen, waarbij we naburige punten telkens sorteerden volgens de grootte van hun projectie d_j, zodat we een steeds grotere straal r_i kregen voor onze splat. We doen nu hetzelfde, enkel sorteren we de naburige punten nu volgens de verhouding van de assen van de ellips die begrensd zou worden door de projectie van dit punt. We noemen deze verhouding $\sqrt{\lambda}$ (λ is dus de verhouding van de kwadraten van de assen van de ellips; deze keuze werkt iets gemakkelijker verderop). Figuur 2.19 verduidelijkt dit.

We zien op figuur 2.19 dat we reeds een oranje circulaire splat hebben en dat we deze nu uitbreiden naar een ellips met een steeds groter wordende $\sqrt{\lambda}$. Zo breiden we eerst uit naar de groene ellips en dan naar de rode ellips. We hebben dus, gegeven een naburig punt, de verhouding van de assen van de ellips begrensd door dit punt nodig. Op de afbeelding is p_k zo'n punt en we willen dus de verhouding van de rode ellips weten. Aangezien we aannemen dat p_k op de rand van de ellips ligt, geldt volgens de formule van een ellips

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} = 1$$



Figuur 2.19: Uitbreiding van een circulaire splat naar een elliptische splat. Afbeelding afkomstig uit [1].

waarbij x en y de afstanden zijn van de projectie van p_k op de genormaliseerde richtingsvectoren u en v.

$$x = \overline{u}_i \cdot (p_k - c_i)$$

$$y = \overline{v}_i \cdot (p_k - c_i)$$

en a en b zijn gelijk aan de lengte van de grote respectievelijk de kleine as van de ellips. De grootte van de kleine as kennen we reeds ($\|v_i\| = r_i$) en we kunnen nu dus een formule voor de grote as en dus ook voor de verhouding λ_k van beide assen bekomen.

$$\frac{x^2}{a^2} = \frac{b^2 - y^2}{b^2}$$

$$\frac{x^2}{b^2 - y^2} = \frac{a^2}{b^2} = \lambda_k$$

We kunnen dus voor het punt p_k de verhouding van de assen van de ellips begrensd door dit punt (de rode ellips in figuur 2.19) berekenen en dit geldt voor alle naburige punten.

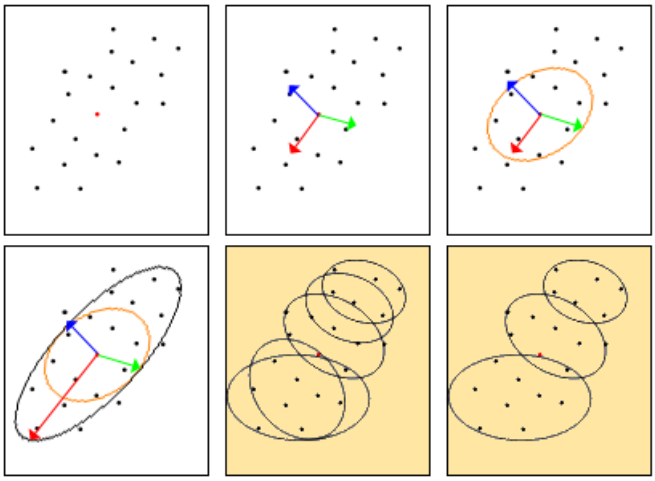
Wanneer we de groeiprocedure voor de initiële cirkel beëindigd hebben, zetten we de groeiprocedure gewoon verder, maar we sorteren de naburige punten nu volgens hun waarde λ_j in plaats van hun waarde d_j. Om correct te zijn, zouden we de punten moeten sorteren volgens $\sqrt{\lambda_j}$, maar aangezien de positieve vierkantswortel een stijgende functie is, kunnen we de punten evengoed volgens λ_j sorteren. Een belangrijke opmerking die hierbij nog wel gemaakt moet worden, is dat we enkel rekening houden met punten p_j die een waarde λ_j hebben die groter of gelijk is aan 1. We willen namelijk de ellips laten toenemen in de richting van zijn grote as en zoals figuur 2.19 aantoont, zijn punten met een λ_j kleiner dan 1, niet geschikt hiervoor.

Punten met een λ_j kleiner dan 0 liggen boven of onder de cirkel (c_i, n_i, r_i), punten met een λ_j gelijk aan 0 liggen op de twee horizontale strepen en kunnen dus niet deel zijn van een ellips en punten met een λ_j tussen 0 en 1 liggen in de cirkel (c_i, n_i, r_i).

We blijven deze groeiprocedure weer net zo lang verder zetten tot het interval $[h_{min}, h_{max}]$ de waarde 2ε overschrijdt en we nemen als verhouding van de uiteindelijke ellips de grootste λ_j van alle punten die beschouwd werden voordat de fouttolerantie 2ε overschreden werd. De uiteindelijke assen zijn dan gelijk aan

$$u_i = \|r_i\|\sqrt{\lambda_j}\overline{u}_i \qquad v_i = \|r_i\|\overline{v}_i$$

2.2.3 Splat selectie.



Figuur 2.20: Overzicht van het algoritme. Splat selectie.

We hebben nu een verzameling T van splats. Deze verzameling bevat evenveel splats als het aantal punten in P waar we van vertrokken zijn. We willen nu een deelverzameling T' van T die gans het model bedekt. De vijfde en zesde tekening van figuur 2.20 tonen dit aan. Nagaan of gans de oppervlakte van het model bedekt wordt is echter een moeilijk probleem en daarom zullen we in praktijk een set T' zoeken die gans P (de initiële monsters) bedekt. Dit is wel gemakkelijk na te gaan. We moeten namelijk gewoon kijken of elk punt p_i uit P door ten minste één splat uit de selectie T' bedekt wordt. De meest efficiënte manier om dit na te gaan is om voor elke splat t_j te onthouden welke punten p_i onder de splat vallen en daarna

voor een deelverzameling T' te itereren over alle splats $t_j \in T'$. Voor elke splat markeren we de punten p_i die onder de splat liggen en nadien moeten we dan nog kijken of er punten uit P zijn die niet gemarkeerd zijn.

Veilige punten.

Er wordt bij de generatie van een splat t_j dus gekeken welke van de initiële punten p_i onder de splat vallen. Een punt p_i valt onder de splat t_j onder de volgende voorwaarden: wanneer we p_i projecteren op het vlak waarin de splat $t_j = (c_j, n_j, u_j, v_j)$ ligt, moet het geprojecteerde punt binnen de omtrek van de splat vallen

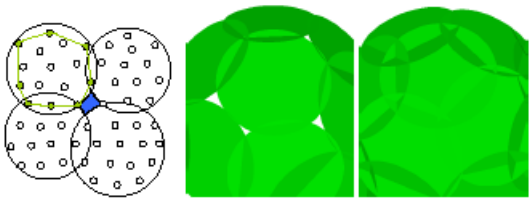
$$\frac{(\overline{u}_j \cdot (p_i - c_j))^2}{\|u_j\|^2} + \frac{(\overline{v}_j \cdot (p_i - c_j))^2}{\|v_j\|^2} \leqslant 1$$

en de afstand van p_i tot zijn projectie moet kleiner of gelijk zijn aan ε

$$\|\overline{n}_i \cdot (p_i - c_j)\| \leqslant \varepsilon$$

De verzameling van punten p_i die onder de splat t_j vallen, noemen we de verzameling Q_j. Deze verzameling kunnen we incrementeel opbouwen tijdens het doorlopen van de buren grafe. Elk punt dat we tegenkomen en dat beschouwd wordt en de fouttolerantie niet overschrijdt, wordt toegevoegd aan de verzameling Q_j.

Als de initiële bemonstering dicht genoeg is, is de kans klein dat er gaten in het model zullen zitten. Toch zijn er situaties waarbij we gaten kunnen hebben. Figuur 2.21 toont een dergelijke situatie. We zien dat alle punten p_i bedekt worden door één of meerdere splats, maar dat er toch een gat in het model zit (het blauwe gebied in het midden). We moeten onze definitie van punten die bedekt worden door de splat t_j dus wijzigen.



Figuur 2.21: Situatie waarbij er een gat in het model zit ondanks het feit dat alle punten p_i bedekt zijn. Afbeelding afkomstig uit [1].

We genereren zoals voorheen de verzameling Q_j van alle punten p_i die onder de splat t_j vallen. Wanneer we deze verzameling hebben, gaan we echter slechts een deelverzameling van Q_j als veilige punten beschouwen. We noemen deze deelverzameling $\overline{Q}_j$. We verkrijgen $\overline{Q}_j$ door alle punten uit Q_j op de splat t_j te projecteren en de punten wiens projectie op de convex hull van al deze projecties liggen, te verwijderen. In figuur 2.21 wordt voor de splat links boven de punten die niet tot $\overline{Q}_j$ behoren in het groen voorgesteld en verbonden met een groene lijn. Alle punten die binnen deze lijn liggen, zijn veilige punten en behoren dus tot $\overline{Q}_j$. Het rechterdeel van figuur 2.21 toont een oppervlak bestaande uit splats met en zonder gaten gegenereerd voor dezelfde fouttolerantie.

Indien Q_j 3 of minder punten bedekt, dan stellen we $\overline{Q}_j$ gelijk aan Q_j. Zo niet, zou $\overline{Q}_j$ geen punten bevatten en is de splat eigenlijk zinloos. Ook zorgen we ervoor dat elk punt p_i tot de verzameling $\overline{Q}_i$ van zijn eigen splat t_i behoort. Zo zijn we zeker dat elk punt door minstens één splat bedekt wordt en dat elke splat op z'n minst één punt (zijn startpunt) bedekt.

Gulzige selectie strategie.

We willen nu een deelverzameling T' van T die zo weinig mogelijk splats bevat en gans P bedekt. Door het genereren van de verzamelingen van veilige punten $\overline{Q}_j$, is het simpel om te zien of een deelverzameling T' van $T = \{t_j\}$ gans P bedekt of niet. Elk punt p_i uit P moet door minstens 1 splat t_j uit T' bedekt worden. Of anders verwoord: de unie van alle punten uit de verzamelingen $\overline{Q}_j$ behorende bij de splats uit de verzameling T' moet gelijk zijn aan P.

$$P = \bigcup_{t_j \in T'} \overline{Q}_j$$

Helaas is dit een moeilijk optimalisatie probleem, dus zullen we een benadering van het optimum zoeken. We maken hiervoor gebruik van een gulzige selectie strategie.

Voor een gulzige selectie strategie hebben we natuurlijk een bepaalde maat nodig om twee splats te vergelijken. Wanneer is een splat t_i groter dan een andere splat t_j? We zouden hiervoor het aantal punten dat beide splats bedekken (met andere woorden, $\|\overline{Q}_i\|$ en $\|\overline{Q}_j\|$) als criterium kunnen nemen, maar deze zeggen weinig over de oppervlakte van de splat zelf. Als de initiële bemonstering niet-uniform was, kan een splat die minder punten bedekt, gemakkelijk veel groter zijn dan een splat die meer punten bedekt. Anderzijds zouden we op basis van de lengte van de grote en de kleine as van een splat de oppervlakte van die splat kunnen berekenen ($\pi\|u_i\|\|v_i\|$) en

dit als criterium nemen. Helaas zegt dit dan weer niets over de hoeveelheid punten die bedekt worden door de splat.

We kiezen dus een middenweg die zowel met het aantal punten die onder de splat liggen als met de oppervlakte rekening houdt. We hebben in sectie 2.2.1 een definitie gegeven van de grootte van de omgeving $N_k(p_i)$ van het punt p_i. Deze was gelijk aan $\omega_i = \pi d_i^2$ waarbij $d_i = \|p_i - p_k\|$. We gebruiken deze definitie bij het opstellen van een definitie voor de grootte van een splat. We stellen de grootte van een splat t_j namelijk gelijk aan de som van de ω_i geassociëerd met de punten $p_i \in \overline{Q}_j$ die nog niet bedekt worden door eerder geselecteerde splats. Dit heeft als gevolg dat telkens we een splat t_j selecteren, de groottes van bepaalde andere splats wijzigen. Concreet zijn dit alle splats die een punt p_i bedekken dat ook door t_j bedekt wordt. Als we dus voor elk punt bijhouden door welke splats het bedekt wordt (net zoals we voor elke splat bijhouden welke punten de splat bedekt), dan kunnen we snel en efficiënt de groottes van deze splats aanpassen wanneer er een splat geselecteerd wordt.

We gaan dus concreet als volgt te werk: we sorteren alle splats volgens hun grootte. Aangezien er nog geen splats geselecteerd zijn, is de grootte van elke splat gewoon gelijk aan de som van de groottes van de omgevingen van alle punten die bedekt worden door de splat. De grootte van splat t_j is dus gelijk aan

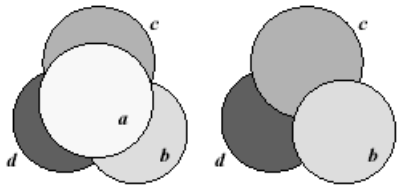
$$\sum_{p_i \in \overline{Q}_j} \omega_i$$

We selecteren nu de grootste splat. We stellen een lijst samen van alle splats die een punt bedekken dat ook door de geselecteerde splat bedekt wordt en passen de groottes van al deze splats aan. We sorteren alle nog niet geselecteerde splats opnieuw volgens hun grootte en kiezen weer de grootste splat. We passen weer de groottes aan van de nodige splats enzovoorts. Dit blijven we doen totdat alle punten $p_i \in P$ bedekt worden door een geselecteerde splat. Aangezien we ervoor gezorgd hebben dat elk punt p_i op z'n minst tot de verzameling $\overline{Q}_i$ van zijn eigen splat t_i behoort, zullen alle punten zeker bedekt worden.

2.2.4 Optimalisaties.

We hebben nu een deelverzameling T' van T die gans P bedekt. Zoals reeds aangehaald, is dit geen optimale oplossing en is er dus nog ruimte voor optimalisaties. Kijk bijvoorbeeld eens naar figuur 2.22. Deze toont vier splats die geselecteerd zijn door het gulzige selectie algoritme uit sectie 2.2.3 volgens hun oppervlakte bijdrage (eerst a, dan b, vervolgens c en als

laatste d). Uiteindelijk is de oppervlakte van a volledig bedekt door de andere drie splats en is a dus overbodig geworden.



Figuur 2.22: De gulzige selectie strategie produceert meestal overbodige splats. Afbeelding afkomstig uit [1].

We bespreken hieronder twee optimalisaties die we na de initiële selectiestap zullen uitvoeren. Ten eerste een relaxatie stap waarbij we de overlappingen tussen de geselecteerde splats zullen proberen te verkleinen en een verwijder stap waarbij we overbodige geselecteerde splats van de selectie zullen verwijderen.

Allereerst hebben we weer enkele definities nodig om de twee optimalisatie stappen uit te leggen. we beginnen met de notie van de omgeving van een splat. Dit is in feite gewoon een uitbreiding van de omgeving van een punt p_i. De omgeving van een splat t_i wordt voorgesteld door $N_k(t_i)$ en bevat de k splats die de k punten uit $N_k(p_i)$ als startpunt hebben waarbij p_i het startpunt is van splat t_i.
Een belangrijke tweede definitie is die van de kern van een splat $t_i \in T'$. De kern van de splat t_i bevat de punten die door geen enkele andere geselecteerde splat buiten t_i bedekt worden.

$$K_i = \overline{Q}_i \setminus \bigcup_{t_j \in T', j \neq i} \overline{Q}_j$$

De kern van een splat t_i is gemakkelijk te verkrijgen aangezien we voor elk punt bijhouden door welke splats het bedekt wordt en we kunnen dus kijken voor welke punten $p_j \in \overline{Q}_i$, t_i de enige geselecteerde splat is waardoor ze bedekt worden.
Ten derde hebben we een maat nodig voor de overlapping van twee splats. Dit is analoog aan de grootte van een splat en is simpelweg de som van de ω_i van alle punten p_i die door beide splats bedekt worden

$$overlap(t_i, t_j) = \sum_{p_l \in \overline{Q}_i \cap \overline{Q}_j} \omega_l$$

Relaxatie.

In deze eerste optimalisatie stap gaan we proberen om de regulariteit van de geselecteerde splats te verbeteren. Dit doen we door de overlappingen tussen de geselecteerde splats te verkleinen. We kijken voor elke geselecteerde splat of er een niet-geselecteerde splat bestaat die een kleinere overlapping heeft met de andere geselecteerde splats. Het is duidelijk dat de kern hierbij een belangrijke rol speelt. Als we geen gaten willen krijgen, moeten we ervoor zorgen dat de kern van de splat die we vervangen terug bedekt wordt door de nieuwe splat.

Formeel zullen we elke splat $t_i \in T'$ proberen te vervangen door een andere splat $t_j \in T \setminus T'$. We kiezen een kandidaat t_j uit de volgende verzameling

$$U_i = \{t_j \in N_k(t_i) | K_i \subset \overline{Q}_j\} \quad \subset \quad T \backslash T'$$

Door voorop te stellen dat alle splats in U_i de kern van t_i moeten bevatten, zijn we zeker dat de nieuwe splat geen gaten zal introduceren. Merk op dat de verzameling U_i geen reeds geselecteerde splats kan bevatten door de definitie van de kern. De verzameling U_i kunnen we gemakkelijk verkrijgen door voor elke $t_j \in N_k(t_i)$ te kijken of het geselecteerd is en zo niet, of $K_i \subset \overline{Q}_j$.

We vervangen t_i nu door een splat $t_j \in U_i$ die de maximale overlapping met eender welke andere geselecteerde splat minimaliseert. (Indien t_i reeds een lokaal minimum is, blijft deze splat gewoon geselecteerd en gebeurt er verder niets.) Hiertoe berekenen we voor zowel t_i als voor elke $t_j \in U_i$ de maximale overlapping met eender welke andere splat uit de verzameling

$$W_j = \bigcup_{p_m \in \overline{Q}_j} V_m$$

waarbij V_j de verzameling is van alle geselecteerde splats die het punt p_j bedekken. De verzameling W_j bevat dus alle geselecteerde splats die minstens één punt bedekken dat door t_j ook bedekt wordt, of anders verwoord, W_j bevat alle geselecteerde splats die een niet-lege overlapping hebben met t_j.

We kennen nu voor elke kandidaat splat $t_j \in U_i$ de maximale overlapping met eender welke andere geselecteerde splat en kiezen nu de splat t_l met de minimale maximale overlapping. Indien deze kleiner is dan de maximale overlapping van splat t_i, wordt splat t_l aan T' toegevoegd en t_i uit T' verwijderd.

Verwijderen.

De tweede optimalisatie is erg simpel. We kijken voor elke geselecteerde splat of zijn kern leeg is en zo ja, verwijderen hem uit T' zonder het gevaar dat we gaten krijgen in ons model. Een lege kern betekent immers dat elk punt dat door de splat bedekt wordt door minstens één andere geselecteerde splat bedekt wordt.

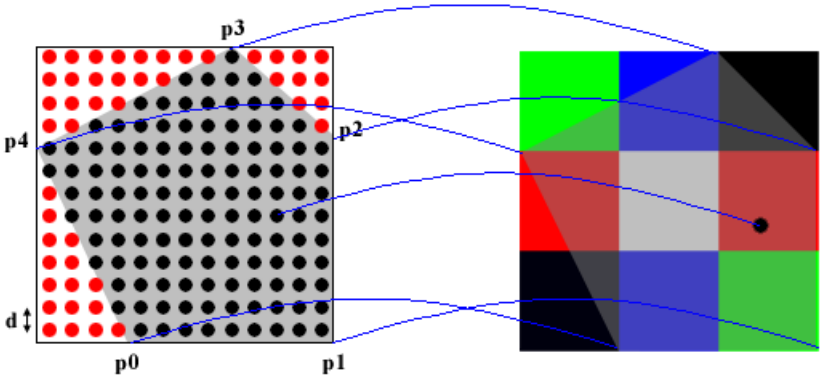
We voeren deze optimalisatie stappen meerdere keren na elkaar uit. We kunnen dit in principe doen tot we convergentie hebben (er wordt tijdens een bepaalde iteratie geen enkele splat vervangen of verwijderd), maar de praktijk toont aan dat er na ongeveer 5 iteraties nog maar weinig optimalisaties plaatsvinden. Aangezien zo'n iteratie vrij lang duurt, is het dus meestal niet de moeite om nog verder te zoeken naar optimalisaties na de eerste 5 iteraties.

2.3 Geometrie en textuur gebaseerde simplificatie.

We gaan het basis algoritme nu uitbreiden zodat het ook rekening houdt met eventuele texturen die op het initiële polygoonmodel plakken. We willen enerzijds, wanneer er geen texturen op het polygoonmodel plakken, dat het aangepaste algoritme dezelfde uitvoer genereert als het basis algoritme en anderzijds, wanneer er wel texturen op het polygoonmodel plakken, dat de gebruiker dan vrij is om de hoeveelheid detail die bewaard zal blijven te kiezen. Hiervoor zullen we een tweede fouttolerantie introduceren voor de kleur, de parameter φ. Deze zal bepalen hoeveel de kleur van een splat die een punt p_i bedekt mag verschillen van de kleur van het punt p_i zelf. Meer uitleg hierover vind je in sectie 2.3.1. Sectie 2.3.2 verduidelijkt hoe het algoritme zelf uitgebreid zal worden.

2.3.1 Texturen en kleur.

Bij het bemonsteren van het model geven we elk monster een bepaalde kleur. Indien het model texturen bevat, kan de kleur simpelweg verkregen worden door de textuurcoördinaten van de hoekpunten van het vlak waarin het monster ligt, te interpoleren en daarna de kleur uit de textuur af te lezen. (Zie figuur 2.23.) Indien er geen textuur plakt op het vlak waaruit het monster genomen wordt, geven we het een standaard kleur (grijs). In deze thesis wordt er gewerkt met de RGB kleurruimte. Elk punt p_i zal dus 3 kleurcomponenten bevatten die elk variëren van 0 tot en met 255.



Figuur 2.23: De textuurcoördinaten van de monsters verkrijgen we door interpolatie van de textuurcoördinaten van p_0, p_1, p_2, p_3, p_4 en p_5.

Gelijkaardige kleuren.

We willen nu gelijkaardige kleuren definiëren. We hebben reeds aangehaald dat we daarvoor de parameter φ zullen gebruiken. Dit is een parameter die de gebruiker kan opgeven en deze kan variëren van 0 tot en met 255. Stel dat we twee punten hebben met kleurcomponenten (r_1, g_1, b_1) respectievelijk (r_2, g_2, b_2). Kleur 2 is gelijkaardig aan kleur 1 indien

$$\begin{cases} r_1 - \varphi \leqslant r_2 \leqslant r_1 + \varphi \\ g_1 - \varphi \leqslant g_2 \leqslant g_1 + \varphi \\ b_1 - \varphi \leqslant b_2 \leqslant b_1 + \varphi \end{cases}$$

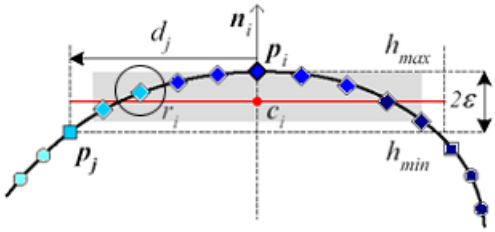
Deze definitie van gelijkaardige kleuren is een symmetrische relatie.

2.3.2 Generatie van initiële splats.

In de vorige paragraaf hebben we een definitie gegeven van gelijkaardige kleuren. Dit zullen we nu gebruiken bij het genereren van de initiële splats. Een splat t_i zal de kleur krijgen van haar startpunt p_i en enkel punten p_j met een kleur die gelijkaardig is aan de kleur van p_i mogen bedekt worden door t_i. Op deze manier zullen de kleuren van het gereduceerde surfelmodel binnen een afstand φ van de kleuren van de initiële verzameling monsters liggen.

Concreet hoeven we enkel de methode die de initiële splats genereert, aan te passen. Net als voordien genereren we eerst een circulaire splat waarvoor

we de buren-grafe doorlopen. Het stop criterium wordt echter aangepast. We stoppen nu niet alleen wanneer het interval $[h_{min}, h_{max}]$ de fouttolerantie 2ε van de geometrie overschrijdt, maar ook wanneer we een punt tegenkomen dat niet gelijkaardig is aan de kleur van het startpunt van de splat die we aan het genereren zijn. Wanneer één van deze twee situaties voorkomen, gaan we net zoals voorheen verder met het uitbreiden van de circulaire splat naar een elliptische splat, waarbij we de buren-grafe weer doorlopen tot één van de twee stop criteria voorkomen. Figuur 2.24 toont een situatie waar we voor het punt p_i een splat aan het genereren zijn. p_i is donkerblauw, meteen ook de kleur die de splat zal krijgen. Tijdens de groeiprocedure zullen eerst de andere vier donkerblauwe punten beschouwd worden zoals voorheen. Vervolgens komen we bij het omcirkelde lichtblauwe punt en afhankelijk van de waarde φ stopt de groeiprocedure (indien de lichtblauwe kleur teveel verschilt van de donkerblauwe kleur van de splat) of gaan we gewoon verder met het volgende punt.



Figuur 2.24: Splat groei procedure die rekening houdt met kleur. Oorspronkelijke afbeelding afkomstig uit [1].

Op deze manier zal het uiteindelijke model zowel de geometrie als de kleuren van de verzameling initiële monsters benaderen met een fouttolerantie ε respectievelijk φ. Natuurlijk is de bemonstering van het polygoonmodel ook belangrijk aangezien deze ook in grote mate bepaalt hoeveel detail er maximum (indien $\varphi = 0$) bewaard zal blijven.

Hoofdstuk 3

Resultaten.

In sectie 3.1 wordt allereerst een overzicht gegeven van de parameters van het algoritme. Daarna bekijken we in sectie 3.2 een aantal resultaten van het basis algoritme. We zullen onder andere de invloed van de parameter ε bekijken voor enkele modellen. In sectie 3.3 uiteindelijk zien we het uitgebreide algoritme aan het werk. We bekijken de invloed van de parameter φ voor verschillende soorten texturen en geven enkele besluiten.

3.1 Overzicht van de parameters.

We hebben in totaal 4 parameters waar de resultaten van zullen afhangen. We bespreken ze alle vier nog eens kort en geven richtlijnen wat goede en minder goede keuze's zijn.

ε : Dit is veruit de belangrijkste parameter van het algoritme. De fouttolerantie ε bepaalt hoe ver de geometrie van de gereduceerde surfel set verwijderd mag zijn van de geometrie van de initiële verzameling punten $\{p_i\}$. De keuze van de waarde van deze parameter bepaalt dus onrechtstreeks ook hoeveel surfels er nodig zullen zijn om gans het model te bedekken. Een goede keuze van deze parameter is belangrijk als we het model goed willen omzetten. Indien deze (veel) te groot gekozen wordt, zal er veel detail uit het model verloren gaan en kan het model zelfs onherkenbaar worden. Indien deze te klein gekozen wordt, zullen de splats meestal erg klein zijn en zullen er erg veel splats nodig zijn om gans het model voor te stellen. Een goede keuze kan meestal bekomen worden op basis van de diagonaal van de bounding box van het model. $\varepsilon = diag(bounding\ box)$ / 1000 bleek tijdens mijn experimenten steeds een goede richtlijn. Natuurlijk kan je deze waarde nog verkleinen of vergroten afhankelijk van het feit of je meer of minder detail en splats in het uiteindelijke model wil. In dit hoofdstuk zal ε steeds gelijk genomen worden aan $diag(bounding\ box)$ / 1000 tenzij anders vermeld.

φ **:** De fouttolerantie voor de kleur. Wanneer we het model simplificeren op basis van de geometrie en de kleur, bepaalt deze het bereik van kleuren die onder een splat met een bepaalde kleur mogen vallen. Bij het basis algoritme wordt deze parameter genegeerd. De keuze van deze parameter is volledig vrij en kan door de gebruiker gekozen worden in functie van het detail dat hij wil behouden. Een waarde 0 betekent dat een splat enkel punten bedekt die dezelfde kleur hebben als de splat zelf, een waarde 255 negeert de kleur en levert hetzelfde resultaat op als het basis algoritme. De standaard waarde is 10.

k **:** De grootte van de omgeving $N_k(p_i)$ van een punt p_i. Deze parameter is reeds uitvoerig besproken in sectie 2.2.2 op pagina 28. Deze parameter zal in alle experimenten in dit hoofdstuk de waarde 10 hebben.

d **:** Parameter voor de dichtheid van de uniforme bemonstering die gebruikt kan worden om het polygoonmodel te bemonsteren. Deze waarde is echter afhankelijk van de grootte van het model: een model met een bounding box tussen $(0, 0, 0)$ en $(1, 1, 1)$ heeft een andere waarde d nodig dan datzelfde model, gescaleerd met een waarde 100. Om de gebruiker niet te belasten met een absolute keuze voor d (de gebruiker moet dan de absolute grootte van het model kennen), gaan we in praktijk de gebruiker een relatieve parameter laten kiezen, namelijk het aantal monsters n die genomen moeten worden op de kortste rand van het model. Dit is ook reeds besproken in sectie 2.1.1 op pagina 15.

3.2 Geometrie gebaseerde simplificatie resultaten.

In deze sectie bekijken we enkele resultaten van het basis algoritme. In figuur 3.1 zie je zo bijvoorbeeld de dragon. Bovenaan zie je het originele polygoonmodel bestaande uit meer dan 400000 hoekpunten, onderaan zie je het gesimplificeerde surfelmodel bestaande uit slechts 13336 surfels, wat de mogelijke maat van reductie aantoont. We hebben enkel de hoekpunten als initiële monsters genomen en ε was gelijk aan 0.0002. We zien dat het gereduceerde model iets minder scherp en iets minder gedetailleerd is dan het polygoonmodel. Dit is natuurlijk volledig te wijten aan de keuze van ε. Indien we deze kleiner gekozen hadden, zou er meer detail bewaard gebleven zijn, maar zouden we natuurlijk ook meer surfels nodig gehad hebben in het gereduceerde model.

Experimenteel is gebleken dat de uitvoeringstijd van het algoritme vrij lang is als het aantal initiële monsters meer dan 100000 bedraagt. Natuurlijk is dit een voorverwerkingsstap voor het renderen en is de uitvoeringstijd niet

zo belangrijk, maar toch kan deze waarde als een richtlijn genomen worden om te lange uitvoeringstijden te vermijden.

Laten we de de fouttolerantie ε nu eens wat grondiger bekijken, meer bepaald de invloed ervan op de kwaliteit van het gesimplificeerde model. Hiervoor hebben we natuurlijk een maat nodig om de fout van het gesimplificeerde model te meten. We definiëren deze fout als volgt: voor elk punt p_i uit de verzameling initiële monsters bepalen we de afstand tot elke splat die dat punt overlapt en we nemen hiervan de minimale afstand. Zo bekomen we voor elk punt p_i de afstand tot zijn dichtstbijliggende punt op het oppervlak van het surfelmodel. We tellen al deze minimale afstanden op en delen dit door het aantal punten p_i. Deze waarde $\overline{\epsilon}$ is dus de gemiddelde afstand van een punt p_i tot zijn dichtstbijliggende splat. ε is natuurlijk een bovengrens voor deze fout en het is interessant om na te gaan hoe deze fout zich verhoudt tot de fouttolerantie ε. We kunnen $\overline{\epsilon}$ berekenen vlak na het genereren van de splats (als er dus nog geen splats geselecteerd zijn) en na de selectie van de splats. Het is interessant om te kijken met welke mate de fout toeneemt door een subset T' te kiezen van de verzameling T van alle splats. Merk op dat de gemiddelde fout vóór de selectiestap niet noodzakelijk 0 is aangezien een splat t_i niet noodzakelijk door zijn startpunt p_i gaat.

Laten we dit eens bekijken voor de bunny. Tabel 3.1 geeft verschillende reducties weer op basis van verschillende waarden voor de fouttolerantie ε. We zien hoeveel splats er overblijven na de simplificatie en wat de gemiddelde fout is van de verzameling splats voor en na de selectiestap. De waardes voor ε zijn gekozen op basis van de diagonaal van de bounding box van het model. Uit experimenten is gebleken dat een keuze voor ε die in de buurt ligt van de bounding box van het model gedeeld door 1000 doorgaans de beste resultaten oplevert. In het geval van de bunny was dit een ε in de buurt van 0.00025. De uitvoeringstijd voor één waarde van ε variëert van 30 seconden tot enkele minuten, afhankelijk van de grootte van ε. Bij een grotere ε, zullen de splats groter worden en zal de groeiprocedure langer duren. Ook liggen er dan meer punten onder een splat waardoor de optimalisatiestap veel langer duurt aangezien deze kwadratisch afhankelijk is van het aantal punten die onder een splat liggen.

Allereerst zien we meteen dat ε de mate van reductie bepaalt en dus hoeveel surfels er overblijven. Een toenemende ε zorgt voor een sterkere reductie en dus voor minder splats in het omgezette surfelmodel.
Een tweede interessante bemerking is dat ε inderdaad een bovengrens is voor de fout. We zien duidelijk in de laatste kolom dat de gemiddelde fout slechts iets meer is dan een derde van de fouttolerantie ε. De precieze verhouding $\overline{\epsilon}/\varepsilon$ neemt af naarmate ε groter wordt.

ε	# monsters	# splats	$\overline{\epsilon}$ voor selectie		$\overline{\epsilon}$ na selectie	
			$(.10^{-5})$	$\overline{\epsilon}/\varepsilon$	$(.10^{-5})$	$\overline{\epsilon}/\varepsilon$
0.0001	34801	17836	3.08	30.8%	4.27	42.7%
0.0002	34801	7888	3.17	15.9%	7.76	38.8%
0.0003	34801	4734	2.98	9.9%	11.4	38.0%
0.0004	34801	3252	2.77	6.9%	14.6	36.5%
0.0005	34801	2367	2.57	5.1%	18.1	36.2%

Tabel 3.1: Invloed van de fouttolerantie ε voor de bunny.

Een derde interessante bemerking is dat de gemiddelde fout van het omgezette model groter wordt, maar dat de gemiddelde fout van het surfelmodel voor de selectiestap net kleiner wordt, voor een toenemende ε. Dit heeft wellicht te maken met het feit dat voor een grotere ε de splats groter worden en elk punt dus ook door meerdere splats bedekt wordt, waardoor de kans groter is dat de afstand van een punt tot zijn dichtstbijliggende splat kleiner wordt. Een laatste bemerking is dat de fout opmerkelijk groter wordt na de selectiestap, zoals we al konden vermoeden. We selecteren in de selectiestap namelijk een subset uit de verzameling T van alle splats, zodat elk punt door minder splats bedekt wordt dan voor de selectiestap. De afstand van dit punt tot zijn dichtstbijliggende splat wordt dus groter of blijft gelijk aan de afstand van dat punt tot zijn dichtstbijliggende splat voor de selectiestap.

In tabel 3.2 doen we een soortgelijk experiment voor de armadillo. We reduceren het model weer voor verschillende waardes voor de fouttolerantie ε die in de buurt liggen van een duizendste van de diagonaal van de bounding box van het model, wat in dit geval overeenkomt met een ε in de buurt van 0.229. De uitvoeringstijd voor één waarde van ε variëert van een paar minuten tot een half uur.

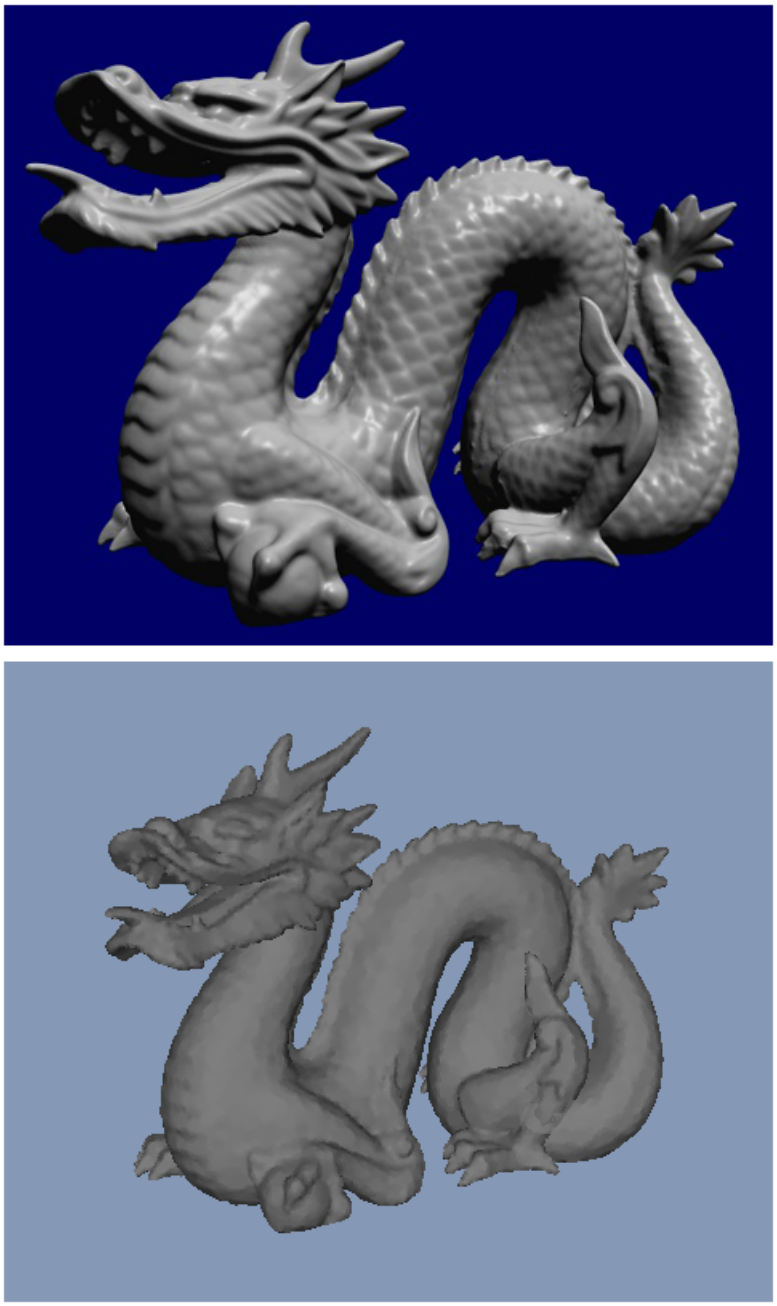
ε	# monsters	# splats	$\overline{\epsilon}$ voor selectie		$\overline{\epsilon}$ na selectie	
			$(.10^{-3})$	$\overline{\epsilon}/\varepsilon$	$(.10^{-2})$	$\overline{\epsilon}/\varepsilon$
0.10	172974	29395	13.3	13.3%	3.80	38.0%
0.15	172974	14927	10.3	6.9%	5.33	35.5%
0.20	172974	9216	8.43	4.2%	6.84	34.2%
0.25	172974	6495	7.42	3.0%	8.38	33.5%
0.30	172974	4850	6.86	2.3%	9.94	33.1%

Tabel 3.2: Invloed van de fouttolerantie ε voor het armadillo model.

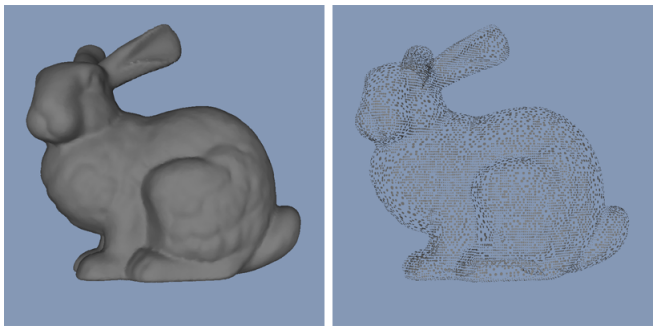
We kunnen dezelfde conclusies trekken als uit de resultaten van de bunny. Ten eerste bepaalt de fouttolerantie weer de mate van reductie. We zien duidelijk dat een hogere ε voor een grotere reductie zorgt. Ten tweede is de gemiddelde fout van het gereduceerde surfelmodel slechts ongeveer een derde is van de fouttolerantie. Vervolgens hebben we ook hier weer het verschijnsel dat voor een toenemende ε de gemiddelde fout na de selectiestap groter wordt, maar dat de gemiddelde fout voor de selectiestap net kleiner wordt. Een laatste bemerking is dat de fout weer opmerkelijk groter wordt na de selectiestap. De verklaringen voor deze conclusies zijn dezelfde als bij het experiment met de bunny.

Het is moeilijk, zo niet onmogelijk, om op basis van enkel de waarde van ε te schatten hoeveel surfels er zullen nodig zijn om een model voor te stellen en dus een waarde voor ε te kiezen in functie van hoeveel surfels je wil bekomen. Dit komt doordat het aantal surfels niet enkel afhankelijk is van de waarde van ε, maar ook van de geometrie van het model. Stel in het extreme geval dat ons model een plat vlak is, dan kunnen we, ongeacht de waarde van ε, gans dit model met slechts één surfel voorstellen. Hetzelfde geldt voor gedetailleerde modellen als de bunny en de armadillo. Stel dat we dubbel zoveel monsters genomen hadden bij onze experimenten met de bunny in tabel 3.1, dan zou het aantal surfels nodig om de bunny voor te stellen voor een bepaalde ε niet erg verschillen van het aantal dat je in tabel 3.1 vindt, aangezien de geometrie in grote mate hetzelfde is, maar de reductiefactor zou dan wel veel groter zijn. Vandaar dat het bijna onmogelijk is om zowel het aantal surfels dat zal overblijven, als de reductiefactor te voorspellen op basis van enkel de waarde van ε.

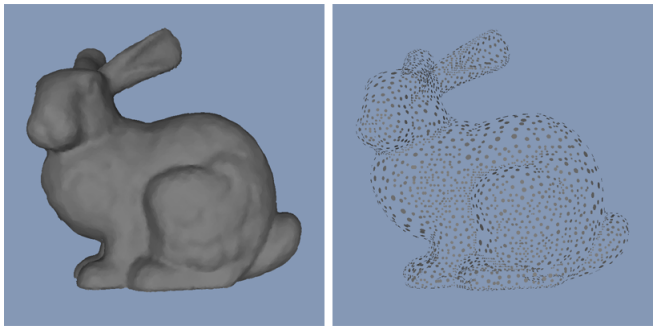
Op de volgende bladzijden tonen we enkele figuren van de hierboven beschreven resultaten om aan te tonen dat de simplificatie wel degelijk werkt. In figuur 3.2 t.e.m. figuur 3.4 zien we het gesimplificeerde bunny model met een fouttolerantie van respectievelijk 0.0001, 0.0003 en 0.0005. Figuren 3.5 t.e.m. 3.7 tonen het gesimplificeerde armadillo model met een fouttolerantie van respectievelijk 0.10, 0.20 en 0.30. We zien dat de modellen voor een groeiende ε met steeds minder splats voorgesteld worden, maar dat er langs de andere kant meer en meer detail verloren gaat en dat de modellen steeds minder scherp afgelijnd zijn. Bij de figuren die de armadillo weergeven, zie je telkens een close-up van het schildpadachtige schild op zijn rug. Dit schild bevat groeven die in figuur 3.5 duidelijk zichtbaar zijn (we zien vele kleine surfels binnen deze groeven liggen), maar naarmate de fouttolerantie stijgt, zien we dat deze groeven steeds minder duidelijk worden in het gesimplificeerde model (onderaan figuur 3.6 en 3.7).



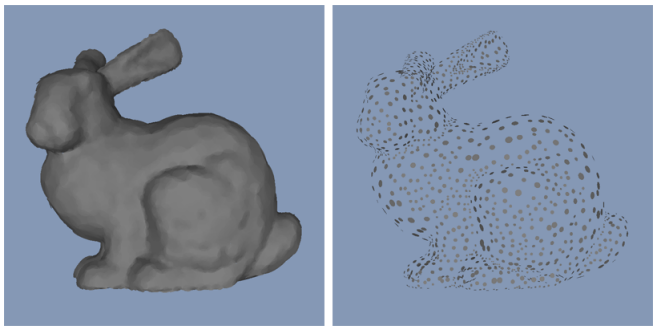
Figuur 3.1: Simplificatie van de dragon.
Bovenaan: polygoonmodel. Onderaan: gesimplificeerd surfelmodel.



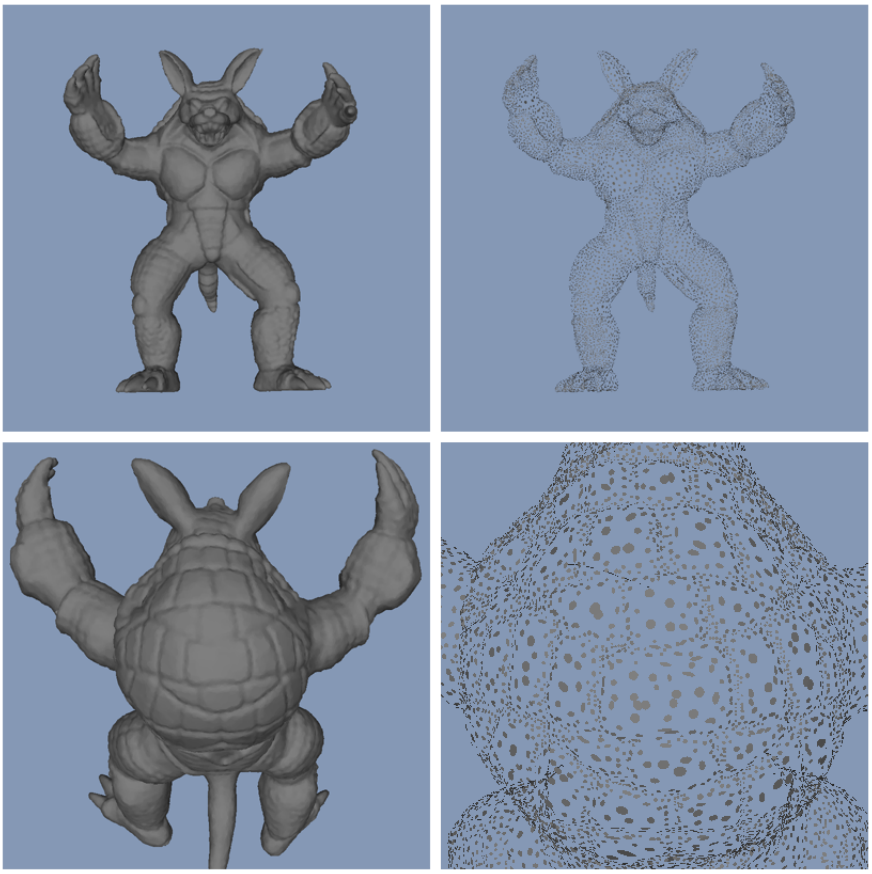
Figuur 3.2: Simplificatie van de bunny.(17836 splats, $\varepsilon = 0.0001, k = 10$)



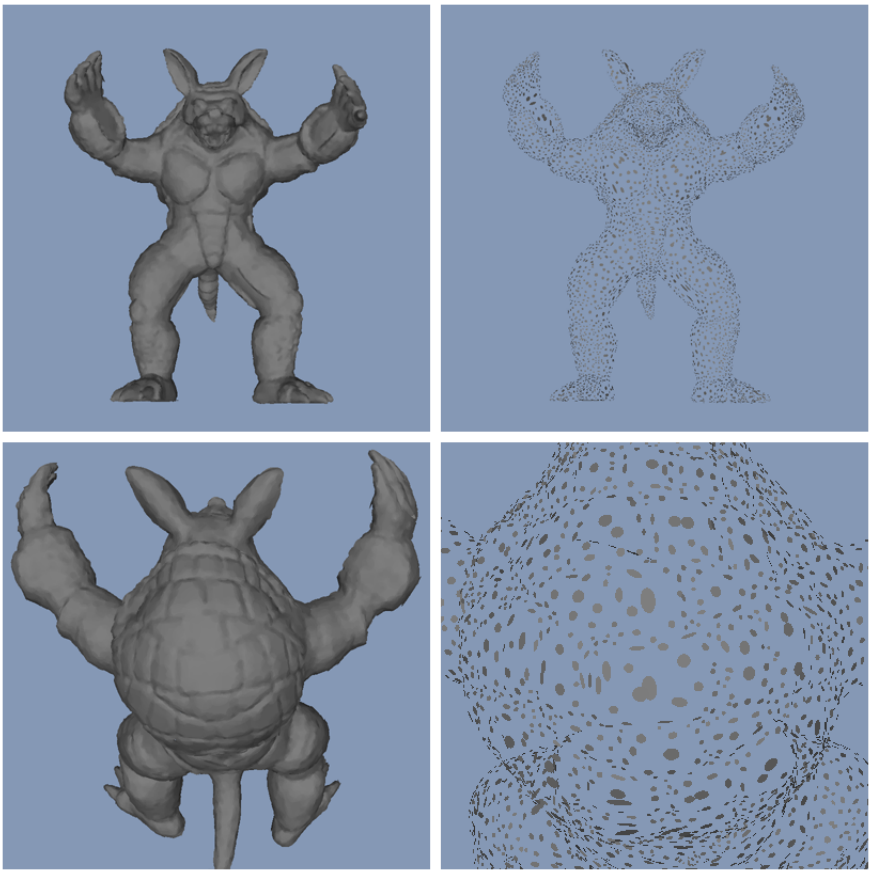
Figuur 3.3: Simplificatie van de bunny.(4734 splats, $\varepsilon = 0.0003, k = 10$)



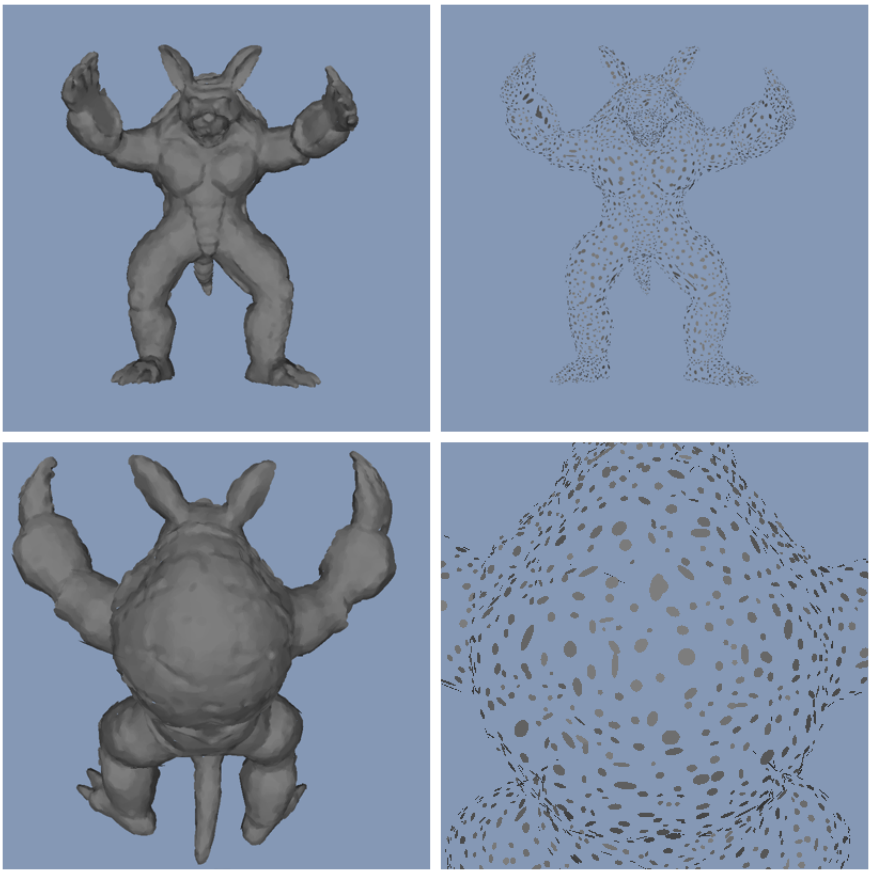
Figuur 3.4: Simplificatie van de bunny.(2367 splats, $\varepsilon = 0.0005, k = 10$)



Figuur 3.5: Simplificatie van de armadillo.(29395 splats, $\varepsilon = 0.10, k = 10$)



Figuur 3.6: Simplificatie van de armadillo.(9216 splats, $\varepsilon = 0.20, k = 10$)



Figuur 3.7: Simplificatie van de armadillo.(4850 splats, $\varepsilon = 0.30, k = 10$)

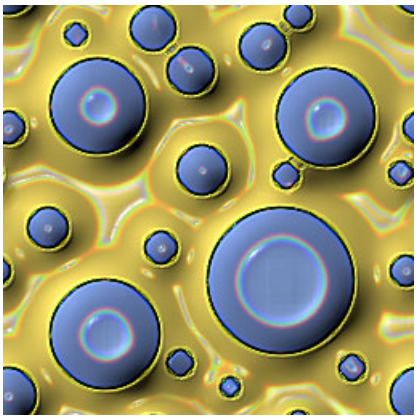
3.3 Geometrie en textuur gebaseerde simplificatie resultaten.

In deze sectie zullen we het uitgebreide algoritme aan een aantal testen onderwerpen. Meer bepaald zullen we de invloed van de fouttolerantie φ bekijken op het aantal surfels nodig om het model voor te stellen en op de kwaliteit van het gesimplificeerde model, en dit zowel voor een textuur met zachte als een textuur met harde kleurovergangen. We hebben deze keer een maat voor de fout van de kleur nodig en deze definiëren we als volgt: voor elk punt p_i uit de verzameling initiële monsters berekenen we de gemiddelde kleur van alle splats die het punt p_i overlappen en we berekenen daarna het verschil van deze gemiddelde kleur met de kleur van het punt p_i zelf. Nadien nemen we voor elk kleurkanaal het gemiddelde van al deze kleurverschillen en bekomen zo de gemiddelde fout voor elk kleurkanaal $(\overline{\phi}_r, \overline{\phi}_g, \overline{\phi}_b)$. Het is weer interessant om na te gaan hoe deze fout zich verhoudt tot de fouttolerantie φ. Ook berekenen we deze fout weer voor en na de selectiestap zodat we zien met welke mate de fout in de kleur toeneemt door een subset T' te kiezen van de verzameling T van alle splats.

We zullen gebruik maken van de bunny voor deze experimenten en we nemen een constante ε die gelijk is aan de bounding box van het model gedeeld door 1000, wat neerkomt op een ε van 0.00025. Uit de vorige sectie bleek dat deze waarde voor dit model voor goede resultaten zorgt. Bovendien nemen we niet enkel de hoekpunten van het model als initiële punten aangezien dit er slechts 34801 zijn en voor een textuur van 256 * 256 (= 65536 pixels) is dit aantal wellicht te weinig om geen detail te verliezen. We maken daarom gebruik van de uniforme sampling beschreven in sectie 2.1.1, pagina 15 en kiezen 0.3 voor het aantal monsters n voor de kleinste rand in het model. Dit levert ons 110174 monsters op. Voor de experimenten met een textuur met zachte kleurovergangen, maken we gebruik van de textuur uit figuur 3.8, voor de experimenten met een textuur met harde kleurovergangen, maken we gebruik van de textuur uit figuur 3.9.

In tabel 3.3 zie je de resultaten van de simplificatie van de bunny met de textuur uit figuur 3.8. In tabel 3.4 zie je de resultaten van de simplificatie van de bunny met de textuur uit figuur 3.9. Telkens is voor een oplopende φ het aantal initiële monsters gegeven, het aantal splats dat overblijft na de simplificatie en de gemiddelde fout per kleurkanaal $(\overline{\phi}_r, \overline{\phi}_g, \overline{\phi}_b)$ van de verzameling splats voor en na de selectiestap. De uitvoeringstijd was gemiddeld een twintigtal minuten per waarde van φ.

Figuur 3.8: Textuur met zachte kleurovergangen.



Figuur 3.9: Textuur met harde kleurovergangen.

φ	# monsters	# splats	$\overline{\phi}$ voor selectie			$\overline{\phi}$ na selectie		
			$\overline{\phi}_r$	$\overline{\phi}_g$	$\overline{\phi}_b$	$\overline{\phi}_r$	$\overline{\phi}_g$	$\overline{\phi}_b$
0	110174	77884	0	0.00	0.00	0	0.00	0.00
2	110174	10530	0	0.05	0.05	0	0.45	0.46
5	110174	5542	0	0.12	0.13	0	0.66	0.66
10	110174	5132	0	0.16	0.16	0	0.71	0.73
15	110174	5055	0	0.18	0.18	0	0.75	0.76
20	110174	5019	0	0.19	0.19	0	0.76	0.77
30	110174	5003	0	0.21	0.21	0	0.80	0.81
40	110174	5014	0	0.21	0.22	0	0.81	0.81
50	110174	5013	0	0.21	0.22	0	0.79	0.80
75	110174	4999	0	0.23	0.23	0	0.91	0.91
100	110174	4994	0	0.27	0.27	0	0.91	0.92
150	110174	4808	0	0.68	0.68	0	1.42	1.43
200	110174	4799	0	0.79	0.79	0	1.51	1.52
255	110174	4799	0	0.79	0.79	0	1.51	1.52

Tabel 3.3: Invloed van de fouttolerantie φ bij een zachte textuur.
$\varepsilon = 0.00025, n = 0.3$

φ	# monsters	# splats	$\overline{\phi}$ voor selectie			$\overline{\phi}$ na selectie		
			$\overline{\phi}_r$	$\overline{\phi}_g$	$\overline{\phi}_b$	$\overline{\phi}_r$	$\overline{\phi}_g$	$\overline{\phi}_b$
0	110174	104797	0.00	0.00	0.00	0.00	0.00	0.00
2	110174	95653	0.01	0.01	0.01	0.06	0.06	0.05
5	110174	76812	0.14	0.13	0.12	0.36	0.34	0.33
10	110174	56132	0.53	0.46	0.50	1.07	0.97	1.01
15	110174	45858	0.90	0.77	0.87	1.68	1.47	1.56
20	110174	39443	1.22	1.04	1.19	2.20	1.95	2.06
30	110174	30535	1.87	1.57	1.76	3.29	2.84	3.00
40	110174	23921	2.59	2.24	2.36	4.40	3.89	3.94
50	110174	19081	3.35	2.95	2.98	5.50	4.98	4.88
75	110174	12620	4.81	4.50	4.44	7.50	7.21	7.10
100	110174	9618	5.91	5.66	5.57	9.19	8.86	8.87
150	110174	6479	7.86	7.61	7.42	12.03	11.50	11.43
200	110174	5191	9.58	9.12	8.80	14.45	13.51	13.62
255	110174	4799	11.26	10.58	10.19	15.94	15.12	10.87

Tabel 3.4: Invloed van de fouttolerantie φ bij een harde textuur.
$\varepsilon = 0.00025, n = 0.3$

Ten eerste valt meteen op dat de reductiefactor voor dezelfde waarde van φ veel groter is bij een zachte textuur. Dit is vrij logisch aangezien er bij een zachte textuur meer gelijkaardige kleuren in dezelfde regio's liggen en de splats dus gemiddeld groter kunnen worden dan bij een harde textuur. Bij een waarde van 255 voor φ wordt er geen rekening gehouden met de kleur (elke kleur is toegelaten onder een splat met een bepaalde kleur) en dit zien we ook duidelijk in beide tabellen: er blijven telkens 4799 splats over, ongeacht de textuur. Het precieze aantal surfels (hier 4799) is volledig afhankelijk van de geometrie en ε.
Vervolgens zien we weer duidelijk dat de gemiddelde fout veel kleiner is dan de toegestane fouttolerantie. Voor de experimenten met de zachte textuur zien we dat we de kleur heel dicht kunnen benaderen, zelfs bij hoge fouttoleranties. Bij de experimenten met de harde textuur is de gemiddelde fout groter, maar toch nog steeds een factor 10 kleiner dan de toegestane fouttolerantie.
Ook neemt de gemiddelde fout weer toe tijdens de selectiestap. Deze toename is echter niet zo dramatisch als bij de geometrie het geval was. Het gaat hier gemiddeld maar over een factor 1.5 à 2 in plaats van 10.

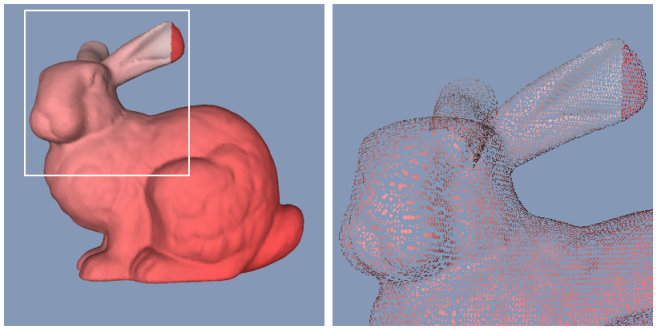
Merk ook op dat $\overline{\phi}_r$ in tabel 3.3 overal 0 is. Dit heeft natuurlijk te maken met de keuze van de kleuren in de gebruikte textuur. Figuur 3.8 is namelijk een gradiënt van rood ($r = 255, g = 0, b = 0$) naar wit ($r = 255, g = 255, b = 255$) zodat alle pixels in de textuur de waarde 255 voor het rode kanaal hebben. Idem voor de initiële monsters en de splats waardoor $\overline{\phi}_r$ overal gelijk is aan 0.

In figuur 3.10 t.e.m. 3.12 zie je enkele resultaten die overeenkomen met bepaalde rijen uit tabel 3.3. We zien duidelijk dat de zachte textuur in alle drie de figuren zeer goed bewaard blijft, maar dat de textuur in figuur 3.10 toch net iets scherper is dan in figuur 3.12 (kijk bijvoorbeeld naar de scheidingslijn op het linkeroor die in figuur 3.12 niet mooi afgelijnd is). Figuur 3.11 toont de gulden middenweg: voor een waarde 10 voor φ hebben we nog steeds een mooi afgelijnde textuur en we hebben er bovendien veel minder splats voor nodig dan bij een waarde 0 voor φ.

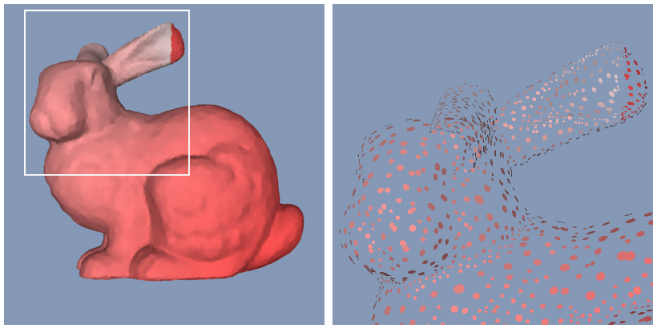
In figuur 3.13 t.e.m. 3.21 zie je verscheidene resultaten die overeenkomen met rijen uit tabel 3.4. We zien duidelijk dat de kwaliteit van de harde textuur in de gesimplificeerde modellen geleidelijk aan afneemt. Telkens is er een close-up gegeven die dit duidelijk aantoont. Kijk bijvoorbeeld naar de regenboogkleuren in het midden van de paarse vlek en de punten die de zwarte lijn afbakenen. In figuur 3.13 zie je dat er erg veel detail bewaard blijft door de veelheid aan splats, maar vanaf figuur 3.17 zien we dat de regenboogkleuren stilletjes beginnen te verdwijnen in het gesimplificeerde

model en vanaf figuur 3.19 begint ook de zwarte lijn steeds minder scherp te worden.

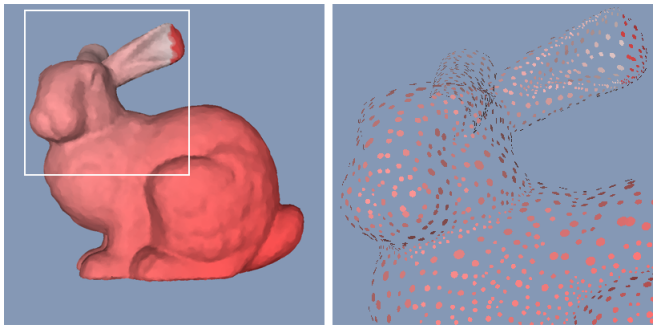
We zien dus duidelijk dat de fouttoleranties de gebruiker de vrijheid geven om het model om te zetten met de nadruk op reductie of op kwaliteit of een afweging van de twee. Meestal kan er mits een klein (soms zelfs weinig merkbaar) verlies in kwaliteit toch een grote reductie bekomen worden. Kijk bijvoorbeeld maar naar figuur 3.11 en figuur 3.15.



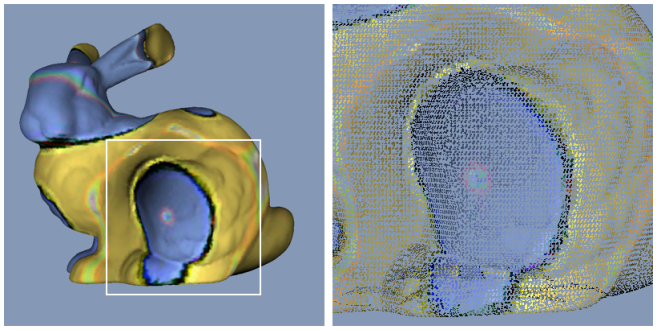
Figuur 3.10: Simplificatie van de bunny met de textuur uit figuur 3.8. (77884 splats, $\varepsilon = 0.00025, \varphi = 0, k = 10, n = 0.3$)



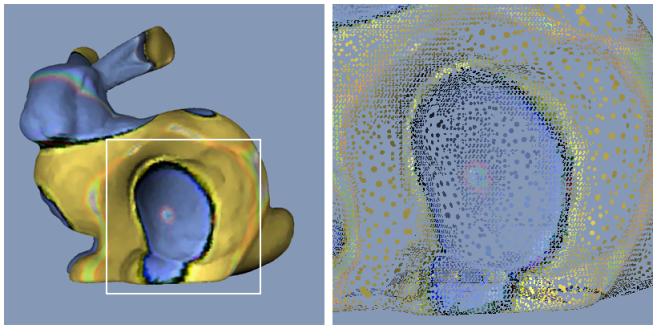
Figuur 3.11: Simplificatie van de bunny met de textuur uit figuur 3.8. (5132 splats, $\varepsilon = 0.00025, \varphi = 10, k = 10, n = 0.3$)



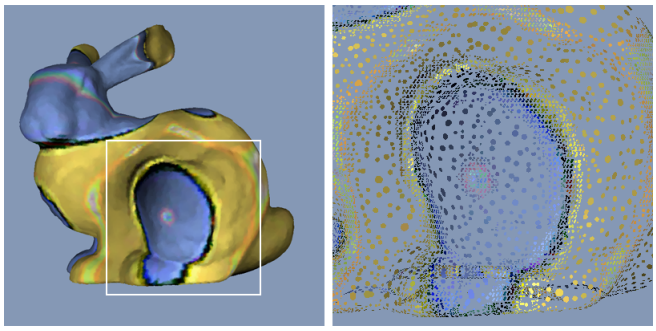
Figuur 3.12: Simplificatie van de bunny met de textuur uit figuur 3.8. (4799 splats, $\varepsilon = 0.00025, \varphi = 255, k = 10, n = 0.3$)



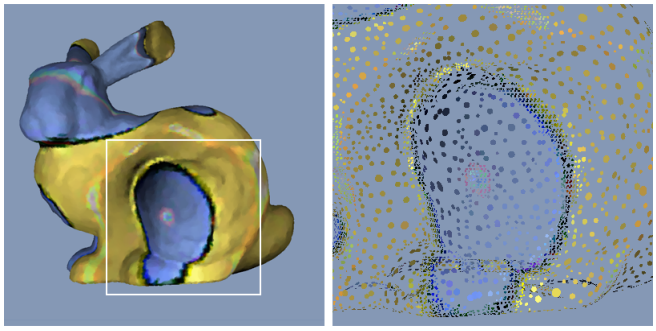
Figuur 3.13: Simplificatie van de bunny met de textuur uit figuur 3.9. (104797 splats, $\varepsilon = 0.00025, \varphi = 0, k = 10, n = 0.3$)



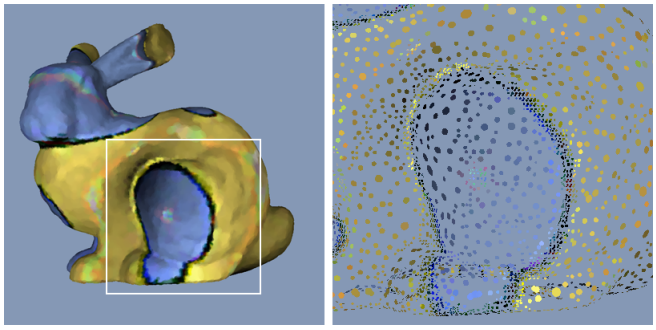
Figuur 3.14: Simplificatie van de bunny met de textuur uit figuur 3.9. (56132 splats, $\varepsilon = 0.00025, \varphi = 10, k = 10, n = 0.3$)



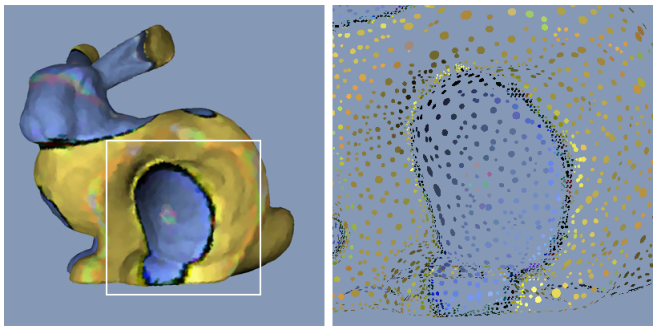
Figuur 3.15: Simplificatie van de bunny met de textuur uit figuur 3.9. (39443 splats, $\varepsilon = 0.00025, \varphi = 20, k = 10, n = 0.3$)



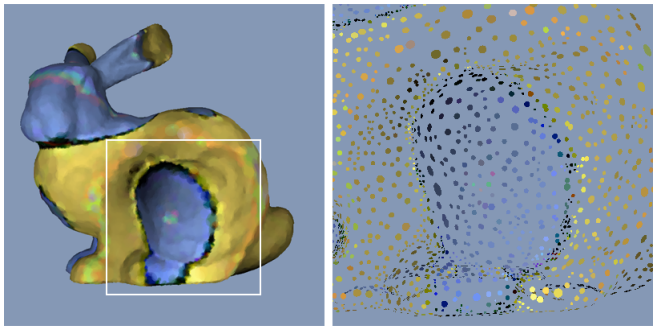
Figuur 3.16: Simplificatie van de bunny met de textuur uit figuur 3.9. (19081 splats, $\varepsilon = 0.00025, \varphi = 50, k = 10, n = 0.3$)



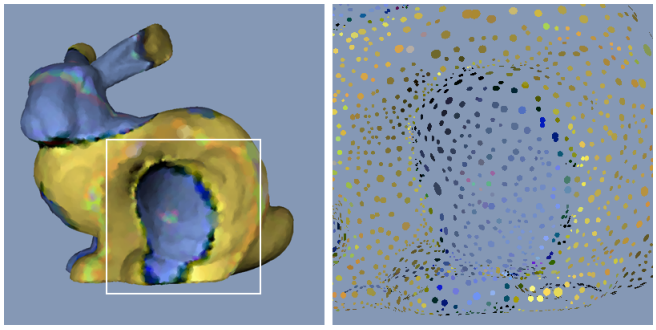
Figuur 3.17: Simplificatie van de bunny met de textuur uit figuur 3.9. (12620 splats, $\varepsilon = 0.00025, \varphi = 75, k = 10, n = 0.3$)



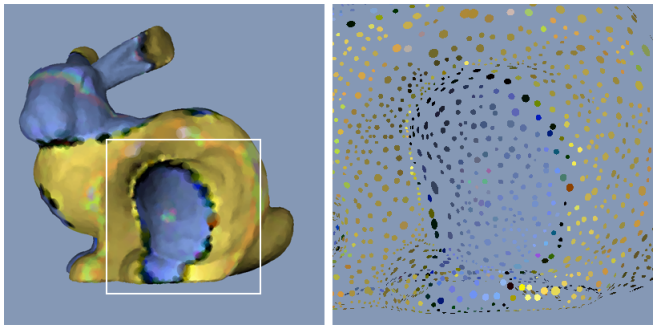
Figuur 3.18: Simplificatie van de bunny met de textuur uit figuur 3.9. (9618 splats, $\varepsilon = 0.00025, \varphi = 100, k = 10, n = 0.3$)



Figuur 3.19: Simplificatie van de bunny met de textuur uit figuur 3.9. (6479 splats, $\varepsilon = 0.00025, \varphi = 150, k = 10, n = 0.3$)



Figuur 3.20: Simplificatie van de bunny met de textuur uit figuur 3.9. (5191 splats, $\varepsilon = 0.00025, \varphi = 200, k = 10, n = 0.3$)



Figuur 3.21: Simplificatie van de bunny met de textuur uit figuur 3.9. (4799 splats, $\varepsilon = 0.00025, \varphi = 255, k = 10, n = 0.3$)

Hoofdstuk 4

Beperkingen en mogelijke uitbreidingen.

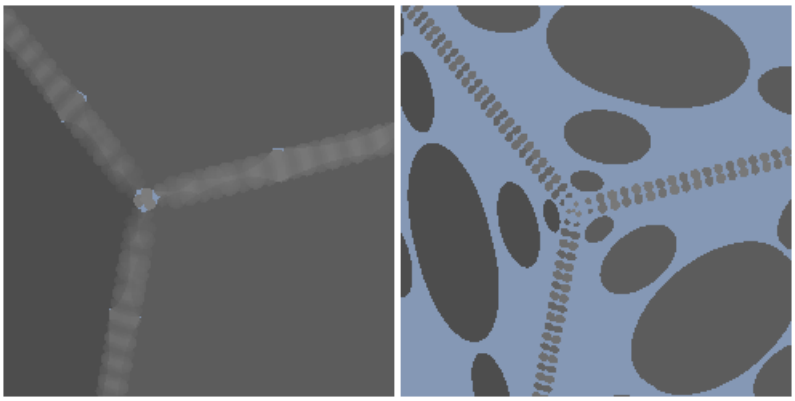
In de vorige hoofdstukken hebben we reeds een aantal puntjes kort aangehaald wat mogelijke beperkingen zijn en wat er nog verbeterd zou kunnen worden. Dit hoofdstuk gaat daar wat uitvoeriger op in. We bekijken het algoritme eerst op een kritische manier en tonen in welke situaties het goed/minder goed werkt (sectie 4.1) en wat daar de oorzaken van (kunnen) zijn. In sectie 4.2 bespreken we een aantal verbeteringen en mogelijke uitbreidingen.

4.1 Beperkingen.

4.1.1 Gaten te wijten aan harde randen in de geometrie.

Zoals reeds uit de resultaten gebleken is, werkt het algoritme het beste voor zachte modellen en minder goed voor modellen met veel harde randen. Hiermee bedoelen we modellen met polygonen die een grote hoek maken ten opzichte van elkaar zoals een kubus, waarbij de hoek tussen elk paar aangrenzende polygonen 90° bedraagt. Een voorbeeld hiervan zie je in figuur 4.1. Ondanks de dichte bemonstering en de protectie van het algoritme tegen gaten (denk terug aan de definitie van veilige punten), zijn er duidelijke gaten in het model. We zien in het rechtergedeelte van figuur 4.1 dat het algoritme dit probeert te voorkomen door verschillende kleine splats langs de randen te plaatsen, maar zelfs dat blijkt niet genoeg te zijn.

De verklaring hiervoor is simpel en wordt getoond in figuur 4.2. Als de polygonen een zachte overgang maken zoals in het linkerdeel van de afbeelding, dan zal een splat die de rand van een polygoon bedekt, ook een deel van de aangrenzende polygoon bedekken. De exacte grootte van deze overlapping is afhankelijk van de fouttolerantie ε en het is ook niet zo belangrijk

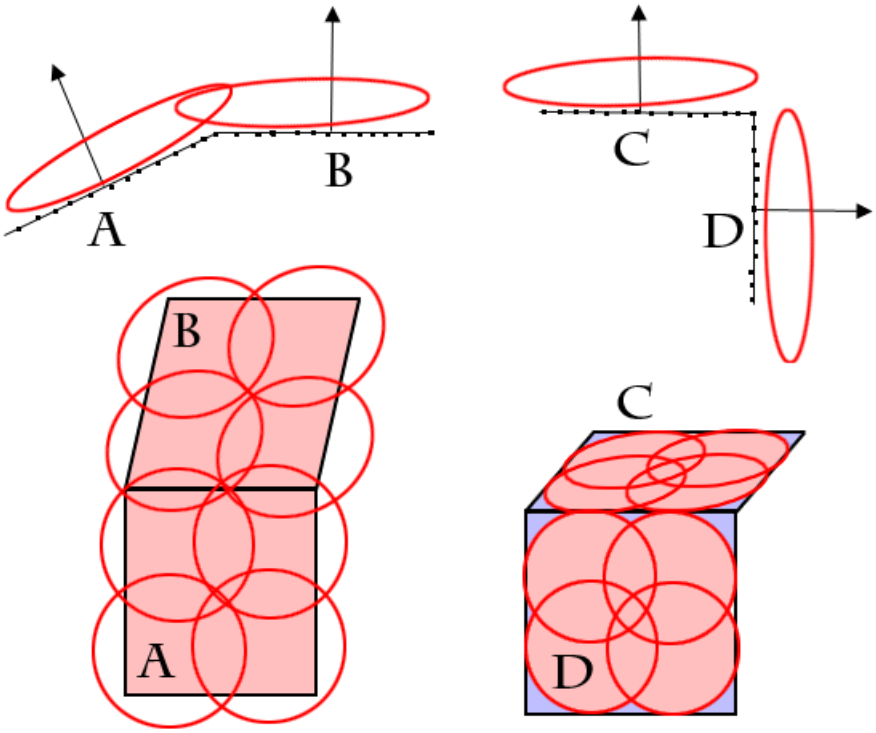


Figuur 4.1: Probleem bij figuren met harde randen in de geometrie. Hier wordt een hoekpunt van een kubus getoond.

hoe groot de overlapping precies is. Wel is belangrijk dat er een overlapping is. Zo zien we dat de twee splats die de bovenkant van polygoon A bedekken ook een deel van polygoon B bedekken en omgekeerd, dat de twee splats die het onderste deel van polygoon B bedekken, ook een deel van polygoon A bedekken. De doorsnede links bovenaan in figuur 4.2 toont duidelijk deze overlapping.

Aan de rechterkant van figuur 4.2 zien we wat er gebeurt als we twee aangrenzende polygonen hebben die een grote hoek maken met elkaar. De splats die gegenereerd worden voor punten gelegen in polygoon D zullen (bijna) niet voorbij de rand van polygoon D komen omdat de fouttolerantie ε overschreden wordt voor punten uit polygoon C. Hetzelfde geldt voor splats die gegenereerd worden voor punten gelegen in polygoon C. Zoals we in de doorsnede rechts bovenaan zien, zal de verticale splat niet voorbij de horizontale zwarte lijn komen en zal de horizontale splat niet voorbij de verticale zwarte lijn komen. Dit heeft als gevolg dat de kans op gaten in het model veel groter is. Figuur 4.2 toont duidelijk waar deze gaten zullen liggen (blauwe regio's). Deze gaten zijn precies degene die je ook in figuur 4.1 kon opmerken.

We kunnen deze gaten proberen zo klein mogelijk te maken door een dichtere bemonstering te nemen. Een tweede mogelijkheid, de oplossing waarvoor ik gekozen heb, is de grootte van de splats een beetje aan te passen. Normaal nemen we voor de kleine as van een splat de grootste waarde d_j van alle punten die beschouwd worden voordat de fouttolerantie 2ε overschreden wordt (laten we deze afstand even d_1 noemen). De waarde van d_j van het punt dat de fouttolerantie overschrijdt wordt in [1] niet in



Figuur 4.2: Bij een kleine hoek overlappen de splats de randen (links). Bij een grote hoek niet, waardoor delen van het oppervlak niet bedekt worden (rechts).

rekening genomen (laten we deze waarde van d_j even d_2 noemen). Ik heb een aantal experimenten gedaan om de kleine as van een splat te baseren op zowel d_1 als d_2 en experimenteel bleek

$$\|v_i\| = \begin{cases} d_1 & \text{indien } d_2 < d_1 \\ d_2 & \text{indien } d_2 \geqslant d_1 \end{cases}$$

de beste resultaten op te leveren. Hetzelfde doen we voor de grote as, die afhankelijk gemaakt wordt van de grootste waarde λ_j van alle punten die beschouwd worden voordat de fouttolerantie 2ε overschreden wordt (λ_1) en de waarde van λ_j van het punt dat de fouttolerantie overschrijdt (λ_2):

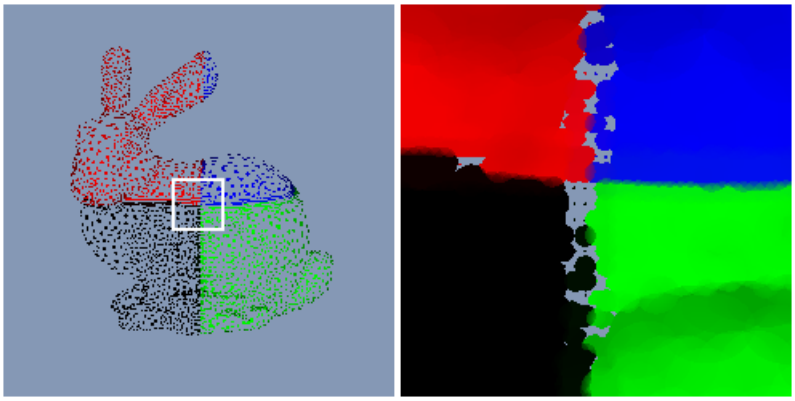
$$\|u_i\| = \begin{cases} \lambda_1\|v_i\| & \text{indien } \lambda_2 < \lambda_1 \\ \lambda_2\|v_i\| & \text{indien } \lambda_2 \geqslant \lambda_1 \end{cases}$$

Helaas elimineert deze methode niet altijd alle gaten. Een dichtere bemonstering kan hier een oplossing zijn. Algemeen genomen is het moeilijk

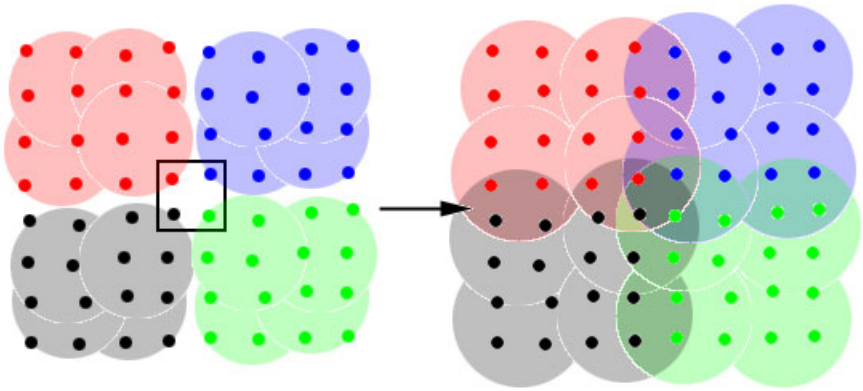
om gaten compleet te voorkomen bij modellen met veel harde randen.

4.1.2 Gaten te wijten aan harde kleurovergangen.

Een soortgelijk probleem doet zich voor bij texturen met veel harde kleurovergangen. Als we een textuur hebben met veel aan elkaar grenzende pixels met niet-gelijkaardige kleuren (bepaald door de parameter φ), dan zullen er (vaak grote) gaten rond de kleurovergangen zichtbaar zijn zoals getoond wordt in figuur 4.3.



Figuur 4.3: Texturen met harde randen zorgen voor gaten.



Figuur 4.4: Bij niet-gelijkaardige kleuren groeien de splats te weinig en krijgen we gaten (links). We wijzigen de groottes van de assen van de splats (rechts).

De verklaring is weer simpel en komt neer op de keuze van de groottes van de assen van de splats. Figuur 4.4 verduidelijkt dit. Bekijk in het bijzonder de vier omkaderde punten. Stel dat de parameter φ klein is, dan zijn dit vier niet-gelijkaardige kleuren. Stel dat we een splat voor het rode punt willen genereren, we zien dat het zwarte punt het dichtstbijgelegen punt is en behandelen dit punt dus eerst. Helaas is dit al meteen een punt dat de fouttolerantie φ overschrijdt en dus zal de splat voor het rode punt niet groter zijn dan het rode punt op zich (de straal r is 0). Hetzelfde geldt voor vele andere punten die op zulke kleurovergangen liggen waardoor er dus gaten in het model zitten.

We lossen dit weer op door de groottes van de assen te baseren op de maximale waardes voor d_j en λ_j van alle punten die beschouwd worden voordat de fouttolerantie φ overschreden wordt (d_1 respectievelijk λ_1) en de waardes van d_j en en λ_j van het punt dat de fouttolerantie overschrijdt (d_2 respectievelijk λ_2). Experimenteel bleek

$$\begin{aligned} \|v_i\| &= \begin{cases} d_1 & \text{indien } d_2 < d_1 \\ d_2 + (d_2 - d_1) & \text{indien } d_2 \geqslant d_1 \end{cases} \\ \|u_i\| &= \begin{cases} \lambda_1 \|v_i\| & \text{indien } \lambda_2 < \lambda_1 \\ \lambda_2 + (\lambda_2 - \lambda_1)\|v_i\| & \text{indien } \lambda_2 \geqslant \lambda_1 \end{cases} \end{aligned}$$

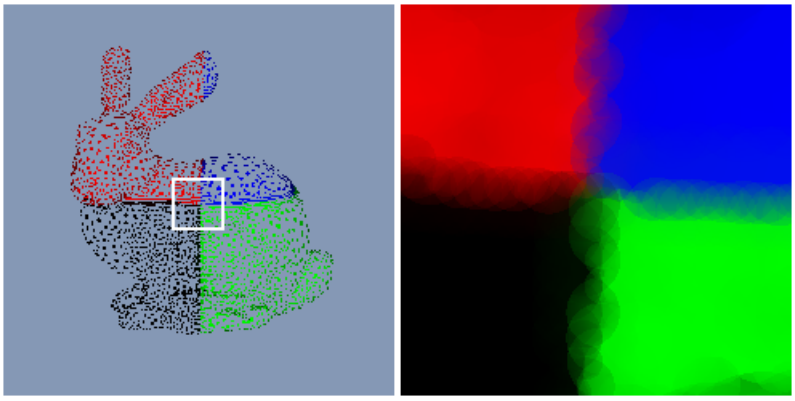
erg goede resultaten op te leveren. De rechterafbeelding in figuur 4.4 toont wat het effect hiervan is. De splats zijn groter waardoor er zo goed als geen gaten meer voorkomen rondom kleurovergangen. Helaas is deze oplossing, net zoals bij de geometrie, geen waterdichte oplossing en kunnen er nog af en toe gaten voorkomen. Een betere en meer specifiek, een meer uniformere, bemonstering kan hier een oplossing zijn.

Beide correcties die de grootte van de splats aanpassen om gaten, te wijten aan harde randen in het model of harde kleurovergangen in de textuur, te vermijden, maken deel uit van mijn algoritme en zitten in de implementatie. Alle resultaten uit het vorige hoofdstuk maken er dus gebruik van.

4.1.3 Zichtbare overlappingen.

De grotere splats zorgen voor meer overlappingen en dit contrasteert een beetje met het doel om zo weinig mogelijk overlappingen te hebben. Natuurlijk is dit niet zo erg aangezien we in de eerste plaats het model zo goed mogelijk willen voorstellen. Anderzijds leidt dit ons wel naar een ander nadeel van het algoritme vergeleken met de traditionele weergave van polygoonmodellen: bij een polygoonmodel zijn texturen duidelijk afgelijnd; bij surfels hebben we, zoals we daarnet besproken hebben, zichtbare overlappingen bij kleurovergangen. Dit komt door de manier waarop splats gerenderd worden. De kleuren van overlappende splats worden geïnterpoleerd

waardoor we een meer geleidelijke overgang krijgen, maar bij harde kleurovergangen valt dit natuurlijk wel vrij hard op. Figuur 4.5 toont dit duidelijk aan.



Figuur 4.5: Texturen met harde randen zorgen nu voor zichtbare overlappingen.

4.2 Mogelijke uitbreidingen.

4.2.1 Initiële bemonstering meer automatiseren.

Zoals reeds gezegd, vertrekt het algoritme van een dichte verzameling van punten gelegen op het oppervlak van het model. Aangezien de modellen waar we van vertrekken polygoonmodellen zijn, moeten we deze dus eerst bemonsteren. We hebben in sectie 2.1 hiervoor een methode gezien. De correcte werking van het algoritme is gebaseerd op een dichte bemonstering en ook in de voorgaande sectie zagen we dat bepaalde beperkingen van het algoritme verbeterd kunnen worden door de dichtheid van de bemonstering aan te passen.

Helaas is het nog steeds de taak van de gebruiker om voor een goede bemonstering te zorgen. De gebruiker moet immers bepalen of hij enkel de hoekpunten van het polygoonmodel als initiële monsters neemt of dat hij daarnaast ook nog elk vlak op zich gaat bemonsteren op basis van een afstand d. Indien hij voor deze laatste optie kiest, moet hij ook nog een goede waarde voor d zien te vinden. Uit voorgaande hoofdstukken bleek dit geen obstakel te zijn (indien we merken dat er een probleem is zoals een textuur die niet nauwkeurig genoeg wordt voorgesteld ondanks een goede keuze van de parameters of we hebben gaten op bepaalde plaatsen, dan nemen we gewoon een dichtere bemonstering), maar het zou handig zijn

als het algoritme zelf een dichtheid kiest voor de bemonstering waarvan het zeker is dat deze gewenste resultaten zal geven.

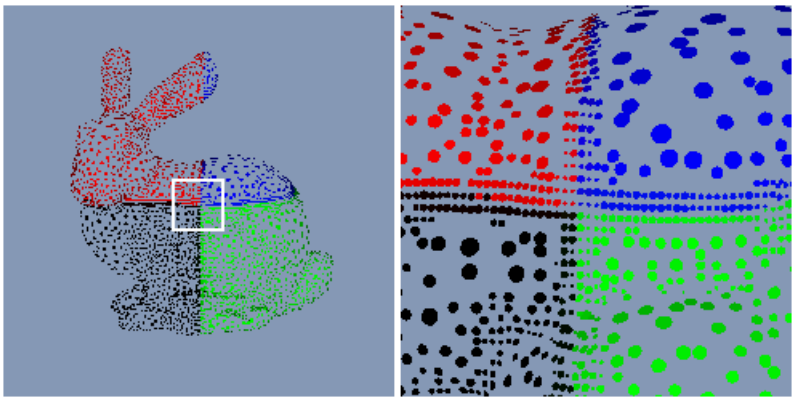
We willen dus een maat voor de dichtheid van een bemonstering. In [1] wordt verwezen naar een r-sample, die in [2] als volgt wordt omschreven:

> The *medial axis* of a (d - 1)-dimensional surface in $\mathbb{R}^d$ is the set of points with more than one closest point on the surface. This definition includes components on the exterior of a closed surface. [..] a sample S is an *r-sample* from a surface F when the Euclidean distance from any point $p \in F$ to the nearest sample point is at most r times the distance from p to the nearest point on the medial axis of F.

Helaas is deze definitie niet meteen praktisch bruikbaar en vandaar dat het in deze thesis aan de gebruiker wordt overgelaten om voor een goede initiële bemonstering te zorgen.[1]

4.2.2 Afgesneden splats.

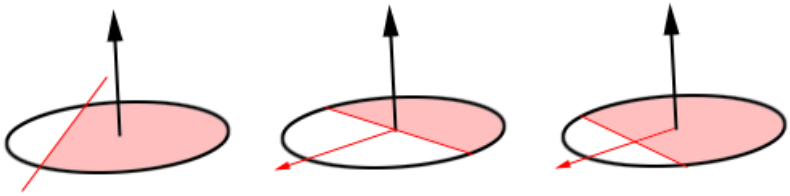
In sectie 4.1 is reeds getoond dat er vele kleine splats liggen rond harde randen in het model en bij harde kleurovergangen. Figuur 4.6 toont een model waar er duidelijk vele kleine splats zichtbaar zijn rond de kleurovergangen.



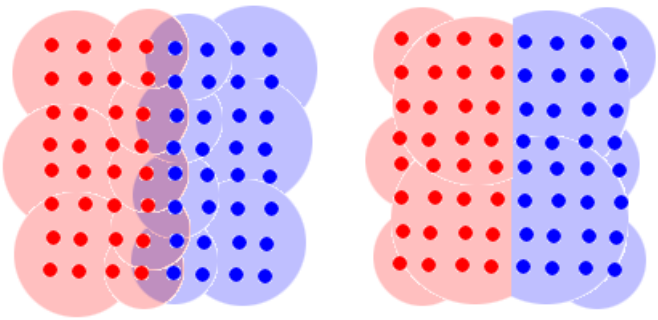
Figuur 4.6: Texturen met harde randen zorgen nu voor zichtbare overlappingen.

[1] Diegenen die meer willen weten over deze *r-sample*, verwijs ik door naar [2].

Een oplossing is simpel, namelijk *afgeknipte splats* (Engels: clipped splats). Dit zijn splats waar een deel van afgeknipt wordt, bijvoorbeeld op basis van een rechte of een richtingsvector. In [8] en [9] wordt bijvoorbeeld gebruik gemaakt van deze techniek. Een paar voorbeelden zijn te zien in figuur 4.7. Figuur 4.8 toont hoe afgeknipte splats een oplossing kunnen zijn voor de vele kleine splats in de buurt van een kleurovergang. Afgeknipte splats zijn ook een oplossing voor de gaten beschreven in sectie 4.1. Kijk bijvoorbeeld nog eens naar de rechtse tekening van figuur 4.2, pagina 61. De blauwe gebieden (gaten in het model) kunnen perfect vermeden worden door gebruik te maken van halve ellipsen, terwijl dit met gewone splats veel minder triviaal is.



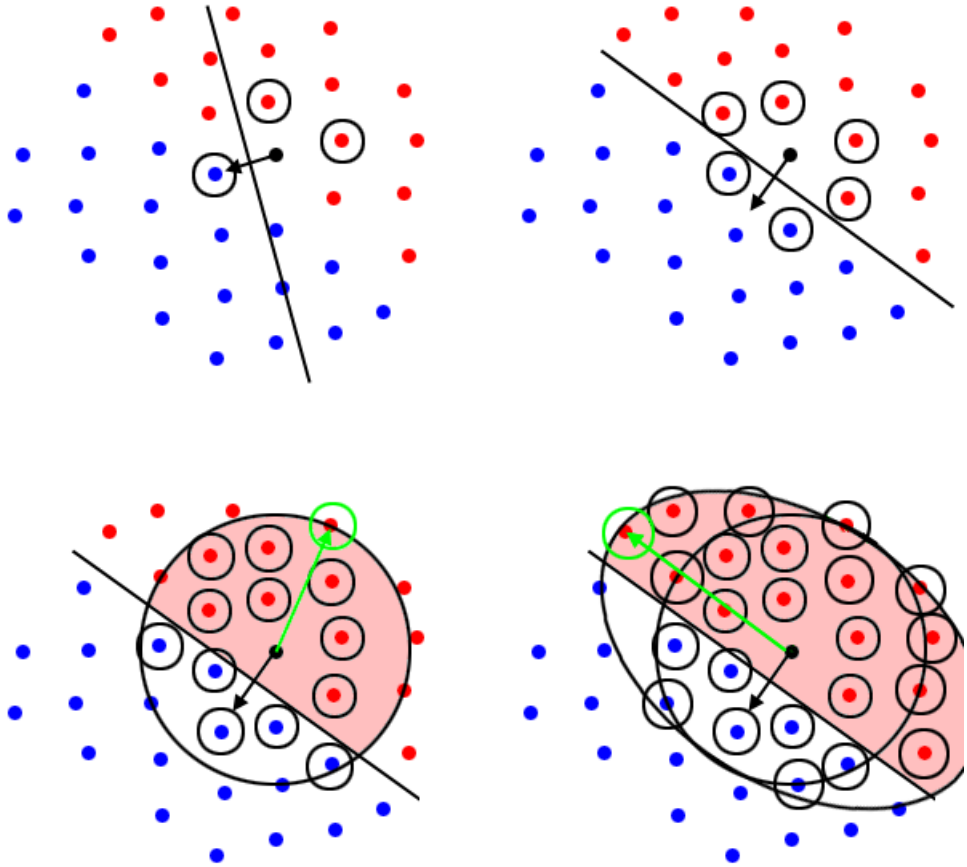
Figuur 4.7: Afgeknipte splats.



Figuur 4.8: Afgeknipte splats vervangen de vele kleine splats op kleurovergangen en harde randen.

Afgeknipte splats zijn vrij simpel toe te voegen aan mijn algoritme. Zo kunnen we de groeiprocedure bijvoorbeeld als volgt uitbreiden: tijdens het doorlopen van de burengrafe, houden we een richtingsvector v bij die weergeeft in welke richting de splat afgeknipt zal worden. Deze vector is het gemiddelde van de genormaliseerde vectoren gaande van het zwarte punt naar alle

punten die we zijn tegengekomen die een fouttolerantie overschreden. Wanneer we een punt tegenkomen dat een fouttolerantie overschrijdt en de hoek tussen de vector van het zwarte punt naar dit punt en de hiervoor vernoemde richtingsvector, is kleiner dan 90°, vernieuwen we de richtingsvector. Indien de hiervoor vermeldde hoek groter is dan 90°, stoppen we met groeien.



Figuur 4.9: Een mogelijke manier om een afgeknipte splat te genereren.

Figuur 4.9 geeft een aantal stappen van het algoritme weer. Bij het genereren van een splat voor het zwarte punt, beginnen we met een circulaire splat te genereren. De richtingsvector v stellen we initieel gelijk aan $(0, 0, 0)$. We doorlopen de burengrafe als voorheen: diepte per diepte, volgens groeiende d_j. Eerst beschouwen we de twee omcirkelde rode punten en vervolgens het omcirkelde blauwe punt. Dit punt overschrijdt een fouttolerantie (in dit geval die van de kleur) en we stellen v dus gelijk aan de vector

gaande van het zwarte punt naar het omcirkelde blauwe punt (afbeelding links boven). We doorlopen de burengrafe verder en komen na een tijd een tweede punt tegen dat een fouttolerantie overschreidt (het tweede omcirkelde blauwe punt). De vector van het zwarte punt naar dit tweede blauwe punt maakt een hoek van minder dan 90° met de huidige vector v en dus updaten we v, die nu het gemiddelde is van de twee genormaliseerde vectoren van het zwarte punt naar de twee omcirkelde blauwe punten (afbeelding rechts boven).
We blijven deze procedure verder zetten tot we een punt tegenkomen dat een fouttolerantie overschrijdt en waarvan de vector van het zwarte punt naar dit punt een hoek maakt met v die groter is dan 90°. Kijk naar de afbeelding links onder in figuur 4.9. Stel dat het groen omcircelde punt de fouttolerantie van de geometrie overschrijdt, we berekenen de hoek tussen de groene en de zwarte pijl en deze is groter dan 90°. De groeiprocedure stopt en we hebben onze initiële afgeknipte circulaire splat. v is op dit moment het gemiddelde van de vijf genormaliseerde vectoren gaande van het zwarte punt naar de vijf omcirkelde blauwe punten.

We gaan de circulaire splat nu net als voorheen verder proberen uit te breiden tot een ellips door de burengrafe verder te doorlopen, diepte per diepte, volgens groeiende λ_j en enkel punten beschouwend met een λ groter of gelijk aan 1. We blijven v ook weer updaten, net zoals voorheen en stoppen ook weer als we een punt vinden dat een fouttolerantie overschrijdt en waarvan de vector van het zwarte punt naar dit punt een hoek maakt met v die groter is dan 90°. De afbeelding rechts onderaan in figuur 4.9 toont de uiteindelijke afgeknipte elliptische splat. We knippen een stuk van de splat langs de kant waarnaar v wijst. Hoe groot dit deel precies is (de afstand van het centrum van de splat tot de zwarte lijn die aangeeft welk deel afgeknipt wordt), kan afgeleid worden van de afstand van het zwarte punt tot het dichtstbijliggende punt dat een fouttolerantie overschreed tijdens het groeiproces, maar het kan best experimenteel getest worden welke afstand het beste werkt, zoals ik gedaan heb met de grootte van de splats in sectie 4.1.

Ik had deze uitbreiding graag geïmplementeerd aangezien het een mooie bijdrage is aan het algoritme, die zowel de kwaliteit van het model (mogelijkheid om veel meer gaten te vermijden) als de reductiefactor (veel minder splats langs harde randen en kleurovergangen) sterk verbeterd zou hebben. De reden dat ik het niet gedaan heb, is omdat ik gebruik maak van het surfelformaat[2] van Pointshop 3D[3] en dit formaat ondersteunt geen afgeknipte surfels. Het programma zelf biedt ook geen ondersteuning om

[2]http://graphics.ethz.ch/pointshop3d/sfldoc/html/pages.html
[3]http://graphics.ethz.ch/pointshop3d/index2.html

dergelijke surfels te bekijken. Als ik het dus geïmplementeerd had, had ik geen mogelijkheid gehad om te kijken of deze afgeknipte surfels wel degelijk werken en dat ze in praktijk ook daadwerkelijk gaten vermijden zoals in theorie zou moeten. Meer nog, aangezien Pointshop de surfels in hun geheel zou tonen (ook de stukken die normaal afgeknipt worden), zouden al mijn resultaten er wellicht minder goed uit zien dan nu het geval is. Vandaar dat ik het algoritme enkel op papier heb uitgewerkt.

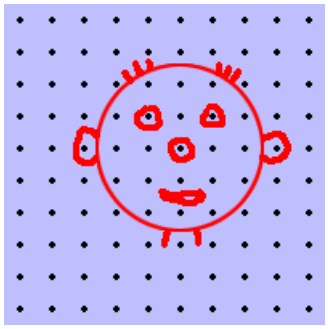
4.2.3 Tweede uitbreiding van het basis algoritme.

In sectie 2.3 heb ik een uitbreiding van het basis algoritme geschreven zodat het algoritme ook rekening houdt met kleur. In deze sectie wordt een tweede uitbreiding van het basis algoritme beschreven. Het zou zeker interessant zijn om beide uitbreidingen in de praktijk met elkaar te vergelijken op gebied van uitvoeringstijd, resultaten, ...

De uitbreiding die ik wel geïmplementeerd heb, gaat reeds tijdens het groeien van de splats rekening houden met de kleuren van de beschouwde punten en het genereren van de splats is dus, net zoals in het basis algoritme, een incrementeel proces. We hebben gezien dat de bemonstering van het polygoonmodel bij deze manier van werken, erg belangrijk is omdat die bepaald hoeveel detail er maximaal bewaard kan worden. Kijk een keer naar figuur 4.10. We zien één vlak van een polygoonmodel met daarop een textuur met een mannetje op. Tevens zien we welke pixels hun kleur verlenen aan een monster. Dit voorbeeld is misschien wat extreem gekozen, maar we zien dat de monsters zo gelegen zijn dat ze allemaal dezelfde blauwe kleur krijgen. Het gezicht in de textuur zal dus gans verdwenen zijn in het omgezette model. Dit is omdat het algoritme enkel werkt met de initiële monsters en geen rekening meer houdt met de textuur na het bemonsteren van het polygoonmodel. Als we dus meer detail willen behouden, moeten we bijgevolg een dichtere bemonstering nemen.

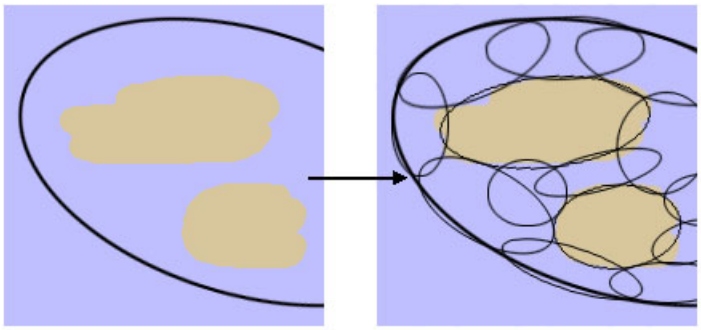
Voor de tweede uitbreiding wou ik dit obstakel overkomen. Het zou namelijk handig zijn als het uiteindelijke resultaat minder afhankelijk was van de bemonstering van het polygoonmodel. Ideaal zou zijn dat we enkel de monsters nemen die we nodig hebben om de geometrie van het model goed om te zetten en dat we geen extra monsters nodig hebben om geen detail uit de texturen te verliezen. Een impliciet voordeel hierbij is dat de uitvoeringstijd van het algoritme waarschijnlijk veel sneller zal zijn omdat we van veel minder punten vertrekken.

De tweede uitbreiding gaat als volgt: we genereren de splats eerst volgens het basis algoritme, enkel rekening houdend met de geometrie, en selecteren



Figuur 4.10: Bij de eerste uitbreiding van het basis algoritme bepaalt de bemonstering hoeveel detail er maximaal bewaard kan worden.

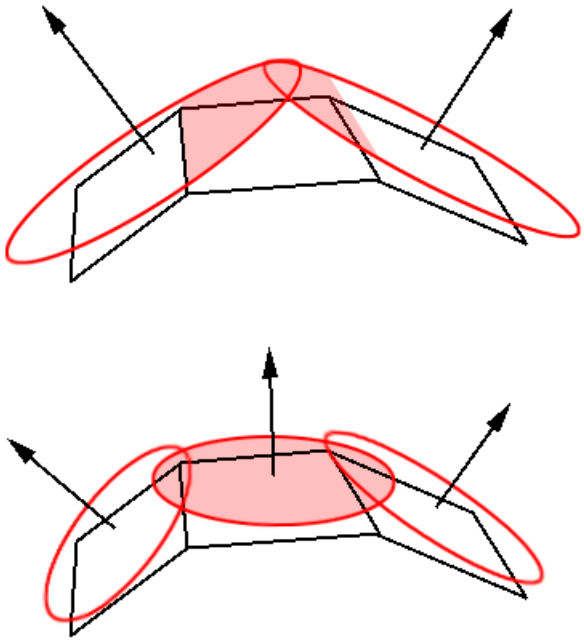
de splats die nodig zijn om gans het model te bedekken. Met andere woorden, we voeren gewoon het basis algoritme uit. We hebben nu een omgezet surfelmodel dat het polygoonmodel benaderd met een fouttolerantie ε, maar de splats bedekken mogelijk wel niet-gelijkaardige kleuren. We projecteren dus elke splat t_i uit de verzameling T' van geselecteerde splats op de vlakken van het polygoonmodel (en hun overeenkomstige texturen) die helemaal of gedeeltelijk door de splat bedekt worden en splitsen de splat t_i daarna op in kleinere splats afhankelijk van de kleuren in de texturen die door de splat bedekt worden. Figuur 4.11 toont het idee aan.



Figuur 4.11: De tweede uitbreiding van het basis algoritme genereert eerst splats op basis van de geometrie en splitst die dan op in kleinere splats op basis van de kleuren van de onderliggende texturen.

De echte uitdaging bij deze uitbreiding is de manier waarop we deze opsplitsing gaan doen. In figuur 4.11 hadden we maar twee gele regio's en het was reeds een hele karwij om de grote splat op te splitsen in zo weinig mogelijk splats met zo weinig mogelijk overlappingen zonder dat er gaten tussen de splats liggen. Het is dan ook verre van triviaal hoe we dit kunnen veralgemenen voor willekeurige texturen.

Ik vermoed dat de uitbreiding waarvoor ik gekozen heb een meer optimale oplossing geeft dan deze uitbreiding. Kijk bijvoorbeeld naar figuur 4.12 waar we bovenaan de tweede uitbreiding in werking zien. We zien twee splats die gegenereerd zijn door het basis algoritme op basis van de geometrie en deze zullen we nu dus opsplitsen in kleinere splats op basis van de texturen. Na de opsplitsing zullen er wellicht een aantal kleinere splats in de rode gebieden liggen. Bij de eerste uitbreiding zouden deze splats gegenereerd zijn tijdens het splat groeiproces en zouden deze splats dichter bij het oppervlak liggen zoals op de onderste afbeelding te zien is. Het is duidelijk dat de geometrie van het gesimplificeerde model in dat geval bij de eerste uitbreiding dichter bij het initiële model ligt dan bij de tweede uitbreiding het geval is.



Figuur 4.12: De tweede uitbreiding van het basis algoritme (bovenaan) zal wellicht een minder optimaal resultaat geven dan de eerste uitbreiding (onderaan) als de bemonstering tenminste goed gekozen wordt.

Een andere mogelijkheid is dat de rechterkant van de linkersplat en de linkerkant van de rechtersplat bovenaan in figuur 4.12 min of meer hetzelfde deel van de textuur bedekken en dat er dus zowel bij het opsplitsen van de linker- als de rechtersplat, kleinere splats gegenereerd worden die eigenlijk hetzelfde gebied bedekken. Dit kan nog met mogelijke optimalisaties opgelost worden, maar het toont aan dat er verscheidene problemen optreden die we niet hebben bij de eerste uitbreiding.

Algemeen vermoed ik dat de eerste uitbreiding betere resultaten zou geven dan de tweede uitbreiding (als de bemonstering tenminste goed gekozen wordt) omdat bij de eerste uitbreiding de splats gegenereerd worden op basis van de geometrie en de kleur te samen, terwijl ze in de tweede uitbreiding eerst op basis van de geometrie gegenereerd worden en daarna pas aangepast worden op basis van de kleur.

Natuurlijk zijn deze laatste paragrafen slechts vermoedens en zou het interessant zijn om ze daadwerkelijk in praktijk met elkaar te kunnen vergelijken.

Hoofdstuk 5

Besluit.

In deze thesis werd een algoritme beschreven dat een polygoonmodel omzet naar een verzameling *surfels*. Surfels zijn een punt-gebaseerde voorstellingswijze die geen connectiviteitsinformatie bevatten zoals polygonen. Allereerst wordt een basis algoritme beschreven dat enkel met de *geometrie* van het model rekening houdt. Nadien breiden we dit algoritme uit om bij de conversie ook rekening te houden met eventuele *texturen* die op met model geplakt zijn. Het algoritme probeert om zo weinig mogelijk surfels met zo weinig mogelijk overlappingen te behouden (als voorwerwerkingsstap voor het renderen van het model), maar stelt het zo correct mogelijk omzetten van het model op basis van fouttoleranties voor de geometrie en de kleur als voornaamste prioriteit.

In hoofdstuk 3 hebben we aangetoond dat het algoritme zijn doel vrij goed bereikt en dat de fouttoleranties ons toelaten om een afweging te maken tussen behoud van detail en reductie. In hoofstuk 4 werd onder andere een interessante uitbreiding beschreven, namelijk afgeknipte surfels, die een zeer mooie aanvulling zou zijn bij het huidige algoritme.

Bibliografie

[1] **Jainhua Wu, Leif Kobbelt.** *Optimized Sub-Sampling of Point Sets for Surface Splatting.* Computer Graphics Forum. 23(3), pp. 643-652, 2004.

[2] **Nina Amenta, Marshall Bern, Manolis Kamvysselis.** *A New Voronoi-Based Surface Reconstruction Algorithm.* Proceedings of SIGGRAPH 98. pp. 415-422, 1998. (page 3-4 of paper).

[3] **Hanspeter Pfister, Matthias Zwicker, Jeroen van Baar, Markus Gross.** *Surfels: Surface Elements as Rendering Primitives.* Proceedings of ACM SIGGRAPH 2000. pp. 335-342, 2000.

[4] **Mark Pauly, Markus Gross, Leif P. Kobbelt.** *Efficient Simplification of Point-Sampled Surfaces.* In IEEE Visualization '02 Proceedings, 2002.

[5] **Lindsay I. Smith.** *A tutorial on Principal Components Analysis.*

[6] **Markus Gross.** *Are Points the Better Graphics Primitives?* Computer Graphics Forum. 20(3), 2001.

[7] **Bart Adams, Philip Dutré.** *Interactive Boolean Operations on Surfel-Bounded Solids.* In Proc. of SIGGRAPH 2003, Annual Conference, San Diego, USA, 26-31 july 2003.

[8] **Mark Pauly, Richard Keiser, Leif P. Kobbelt, Markus Gross.** *Shape Modeling with Point-Sampled Geometry.* ACM Transactions on Graphics. 22(3), pp. 641-650, 2003.

[9] **Matthias Zwicker, Jussi Rsnen, Mario Botsch, Carsten Dachsbacher, Mark Pauly.** *Perspective accurate splatting.* Proceedings of the 2004 conference on Graphics interface.

[10] **Jon Louis Bentley.** *Multidimensional Binary Search Trees Used for Associative Searching.* Commun. ACM 18(9): 509-517 (1975).