Samenvatting

Het probleem dat in deze thesis bestudeerd wordt is het genereren van 3D objecten uit foto's. We zullen twee algoritmen die dit probleem trachten te oplossen, van naderbij bestuderen en implementeren. Bovendien zullen we ook een testopstelling bouwen om de twee algoritmen te testen.

Voorwoord

Voordat we met deze thesis beginnen, zou ik eerst enkele personen willen bedanken. Allereerst wil ik de promotor van deze thesis, professor doctor Philippe Bekaert bedanken voor zijn deskundig advies.

Mijn begeleider Jan Fransens moet ik zeker niet vergeten te bedanken. Hij heeft mij met raad en daad bijgestaan tijdens deze thesis. Verder wil ik ook nog Tom de Weyer bedanken voor het bouwen van het roterend platform, Erik Hubo en Koen Beets voor het maken van de opstelling, en het werken met de camera.

Ten laaste wil ik nog mijn familie bedanken voor de steun die ze mij gegeven hebben tijdens het maken van deze thesis.

Tongeren, 1 maart 2005

Inhoudsopgave

Voorwoord **1**

Lijst van figuren **5**

1 Inleiding **7**
1.1 Probleemstelling . . . 7
1.2 Motivatie . . . 8
1.3 Overzicht . . . 8

2 Literatuurstudie **10**
2.1 Camera calibratie . . . 10
2.1.1 Intrinsieke en extrinsieke parameters . . . 10
2.1.2 Camera calibratie . . . 11
2.2 3D scanning algoritmen . . . 12
2.2.1 Voxel Coloring . . . 13
2.2.2 Space Carving . . . 14
2.2.3 Image Based Visual Hulls . . . 15
2.2.4 Gekozen algoritmes . . . 17

3 Space Carving **18**
3.1 Datastructuren . . . 18
3.2 Overzicht . . . 19
3.3 Zichtbaarheid van een voxel . . . 20
3.3.1 Eliminatie van interne voxels . . . 21
3.3.2 Het traceer algoritme . . . 22
3.3.3 Voorwaarden . . . 22
3.3.4 De initialisatie fase . . . 23
3.3.5 Doorlopen van het grid . . . 23
3.4 Bepalen van de kleur van een voxel . . . 25
3.4.1 Puntprojectie . . . 25
3.4.2 Voxelprojectie . . . 26

3.4.3 Sprite projectie 27
3.4.4 Besluit 28
3.5 Consistentie van een voxel 29
3.6 Iteratie door het voxelgrid 30
3.6.1 Het plane-sweep algoritme 31
3.7 Rendering van resultaten 33
3.8 Besluit 34

4 Image Based Visual Hull 35
4.1 Overzicht 36
4.2 Berekenen van het silhouet 36
4.3 Intersectie van twee kegels 37
4.3.1 Edge-bin constructie 38
4.3.2 Intersectie met behulp van de edge-bins 40
4.3.3 Creëren van polygonen 42
4.4 Intersecties op een face 45
4.4.1 Quad-bin constructie 45
4.4.2 Opsplitsen van een polygoon in de quad-bins 46
4.4.3 Opsplitsen van de quad-bins 48
4.4.4 Intersectie met behulp van de quad-bins 49
4.5 Mesh constructie 52
4.6 Texture van het model 53
4.6.1 Textuur coördinaten 53
4.6.2 Zichtbaarheid van een polygoon 54
4.6.3 De blending gewichten 54
4.6.4 Renderen 55
4.7 Besluit 56

5 Resultaten 57
5.1 Space Carving 57
5.1.1 Computer gegenereerde objecten 58
5.1.2 Reële objecten 60
5.2 Image based visual hulls 67
5.2.1 Computer gegenereerde objecten 67
5.2.2 Reële objecten 69

6 De 3D scanner 72
6.1 Constructie 72
6.2 Werking 73

7 Besluit 75

7.1 Space Carving 75
7.1.1 Verbeteringen 76
7.1.2 Conclusie 76
7.2 Image Based Visual Hull 77
7.2.1 Conclusie 77

Bibliografie **78**

Lijst van figuren

2.1 Voorbeeld van een calibratie patroon 12
2.2 Silhouetten en kegels van 3 beelden van een zelfde scène, de intersectie van de kegels zal de visueel omhullende vormen . . 16

3.1 De voxel V is zichtbaar in camera $C2$ maar niet in $C1$ 20
3.2 Sommige voxels kunnen geëlimineerd worden, sommige niet . . 21
3.3 Het middelpunt a van voxel V wordt geprojecteerd op het cameravlak P in het punt p 26
3.4 Project van voxel V op het cameravlak P 27
3.5 Projectie van een sprite 28
3.6 Figuur uit [7] 32
3.7 We laten alleen de camera's toe die achter het sweep-plane S liggen (de zwarte camera's). De pijl geeft de richting aan waarin s bewogen wordt. 32

4.1 Silhouetten en kegels van 3 beelden van een zelfde scène, de intersectie van de kegels zal de visueel omhullende vormen (figuur uit [11]) 35
4.2 Mogelijke ligging van de epipool t.o.v. het silhouet 39
4.3 Constructie van de edge-bins. Het silhouet is het grijs gebied. 40
4.4 Intersectie van een geprojecteerd face (de blauwe lijnen) met het silhouet(het grijs gebied). Het resultaat is alles wat groen is. 41
4.5 Als het silhouet bestaat uit twee disjuncte delen (A en B)), dan kan dat resulteren in twee of meer polygonen, als we de intersectie berekenen met een face F (projectie wordt gevormd door de lijnen $F1$ en $F2$) 43
4.6 De doorsnede van twee polygonen op een face. 45
4.7 Indeling van een polygoon in quad-bins, de blauwe lijnen zijn de grenzen van de quad-bins. De twee buitenste lijnen zijn de grenzen van de face. 46

4.8 Opsplitsen van een niet convexe veelhoek in een quad-bin, levert meer dan twee intersectie punten op, op iedere straal ($a1$,..,$a4$ op de eerste en $b1$,..,$b4$ op de tweede). 47
4.9 De vierhoeken a en b die elkaar slechts overlappen op één straal. De intersectie moet het gebied c opleveren. We splitsen de bin in twee aan het intersectie punt. 48
4.10 De vierhoeken a (rood) en b (blauw) die elkaar overlappen op beide stralen. 50
4.11 De vierhoeken a,b en c die het resultaat zijn van de intersectie van de blauwe en de groene polygoon, kunnen samengevoegd worden tot één polygoon. De rode lijnen zijn de grenzen van de quad-bins. 52

5.1 Resultaten voor een kubus 58
5.2 Eén van de input foto's van de theepot 59
5.3 Resultaten voor de theepot op een resolutie van 50 bij 50 bij 50 60
5.4 Resultaten voor de theepot op een resolutie van 100 bij 100 bij 100 60
5.5 Pritt roller met resultaten op een resolutie van 100 bij 100 bij 100 61
5.6 Resultaten op 150 bij 150 bij 150 62
5.7 Resultaten op 200 bij 200 bij 200 62
5.8 Resultaten op 200 bij 200 bij 200 (spriteprojectie) 62
5.9 Kaktus met resultaten op een resolutie van 100 bij 100 bij 140 64
5.10 Kaktus met resolutie 150 bij 150 bij 175 65
5.11 Kaktus met resolutie 200 bij 200 bij 225 65
5.12 Kaktus met resolutie 200 bij 200 bij 225 (spriteprojectie) . . . 66
5.13 Resultaten voor de kubus 67
5.14 Beperking van het image-based visual hull algoritme 68
5.15 Resultaten voor de theepot 68
5.16 Pritt roller met resultaten 70
5.17 Kaktus met resultaten 71

6.1 Het roterend platform 73
6.2 De opstelling 74

Hoofdstuk 1

Inleiding

In dit hoofdstuk zullen we het onderwerp van deze thesis inleiden. We zullen eerst beginnen met het probleem van 3D objecten te genereren uit foto's, naderbij te bekijken. Dit wordt gevolgd door een motivatie, om daarna af te sluiten met een overzicht van de rest van deze thesis.

1.1 Probleemstelling

Computer graphics is haast onmisbaar geworden in industrieën zoals de film- en gamesindustrie. Het renderen van realistische 3D beelden is één van de belangrijkste toepassingen. Een probleem bij het renderen van realistische beelden is dat er gedetailleerde 3D modellen moeten bestaan.

Het creëren van complexe 3D modellen is echter niet gemakkelijk en bovendien nog duur ook omdat deze modellen meestal gemaakt worden door gespecialiseerde personen in een programma zoals 3D Studio Max of Maya, waaraan een duur prijskaartje hangt. Daarom is er onderzoek bezig om dit proces te automatiseren.

Eén van de belangrijkste onderzoekpistes hierbij is het genereren van 3D modellen met behulp van foto's. Van het in te scannen object worden een aantal foto's gemaakt vanuit verschillende hoeken, zodanig dat alle visuele eigenschappen van het object duidelijk zijn. Een 3D model wordt gegenereerd met de foto's.

Het doel van deze thesis is twee algoritmen die dit kunnen te implementeren en bovendien een test opstelling te bouwen waarmee we de algoritmen kunnen uittesten, we moeten in feite een 3D scanner bouwen.

1.2 Motivatie

Het genereren van een 3D model uit beelden is een zeer fascinerend probleem, omdat wij het onbewust doen. Onze ogen genereren beelden, die door onze hersenen gebruikt worden om ons een 3D beeld te vormen van onze omgeving. Er is dan ook al veel onderzoek verricht om dit te kunnen nabootsen, we zijn er echter nog niet in geslaagd om dit met dezelfde accuraatheid en detail te doen als onze hersenen.

Deze technologie maakt vele toepassingen mogelijk. Een voorbeeld hiervan is bij het ontwikkelen van computergames. In vele games zijn er heel wat gebruiksvoorwerpen die gemodelleerd moeten worden, dit vraagt uiteraard tijd en geld. Een 3D scanner zou hier heel wat tijd en geld kunnen besparen, bovendien laat het ook toe om de artiesten hun tijd in meer belangrijkere dingen te steken zoals bijvoorbeeld de modellen van de monsters (die niet ingescanned kunnen worden).

Producten verkopen via het internet kan verbeterd worden door een 3D model te laten zien van het product, zodanig dat de klant het van alle mogelijke hoeken kan bekijken, net zoals in een reële winkel.

Dit zijn slechts enkele mogelijke toepassingen die gerealiseerd kunnen worden door deze technologie. Foto-realistische reconstructie van reële objecten met behulp van foto's, is dan ook een interessant onderzoeksgebied.

1.3 Overzicht

Hoofdstuk 2 zal een overzicht geven van de literatuurstudie die we gemaakt hebben voor deze thesis. Daarna zullen we het space carving algoritme bespreken in hoofdstuk 3, gevolgd door het image-based visual hull algoritme in hoofdstuk 4.

Na de twee algoritmen zullen we een overzicht geven van de resultaten in

hoofdstuk 5, daarna zullen we de 3D-scanner van naderbij bekijken (hoofdstuk 6).

We zullen uiteindelijk afronden met het besluit in hoofdstuk 7.

Hoofdstuk 2

Literatuurstudie

In dit hoofdstuk zullen we een overzicht geven van de literatuurstudie die we voor deze thesis gemaakt hebben. We zullen starten met een sectie over camera calibratie, gevolgd met een overzicht van verschillende scan algoritmen.

2.1 Camera calibratie

Als men met beelden werkt van echte camera's dan komt onvermijdelijk het probleem van camera calibratie om de hoek piepen. Om nuttige 3D gegevens te verkrijgen uit foto's is het nodig om de intrinsieke en de extrinsieke parameters van een camera te kennen. Via camera calibratie kunnen we de intrinsieke parameters berekenen.

We zullen eerst de intrinsieke en extrinsieke parameters bespreken. Daarna zullen we verder gaan met camera calibratie.

2.1.1 Intrinsieke en extrinsieke parameters

Beschouw het punt $m = [u, v]^T$ in 2D en het punt $M = [x, y, z]^T$ in 3D. We zullen een vector x die we uitgebreid hebben met een extra 1, $\overline{x}$ noemen. Zodat $\overline{m}$ gelijk is aan $[u, v, 1]^T$ en $\overline{M}$ gelijk is aan $[x, y, z, 1]^T$.

Stel nu dat m de projectie is van het punt M op het camera vlak van een

echte camera, die we zullen modeleren met een gebruikelijke pinhole camera. De wiskundige relatie tussen deze twee punten wordt dan gegeven door de vergelijking :

$$s\,\overline{m} = A\ [R\,t]\ \overline{M} \tag{2.1}$$

In deze formule is s een arbitraire scaleer factor. De matrix $[Rt]$ bestaat uit de rotatie en translatie die het coördinaten systeem van de wereld relateren met dat van de camera. Deze twee parameters worden de extrinsieke parameters genoemd. De matrix A is de intrinsieke matrix en ziet er als volgt uit :

$$A = \begin{bmatrix} \alpha & \gamma & u_0 \\ 0 & \beta & v_0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.2}$$

Waarbij u_0 en v_0 de coördinaten van het principieel punt in de u en v assen van het beeld, α en β zijn scaleer factoren voor de twee assen van het beeld, ze worden ook wel de focale lengte genoemd. Tenslotte is γ de helling tussen de twee beeld assen, γ is meestal gelijk aan 0.

Naast de matrix A bestaan de intrinsieke matrix ook nog uit de lensdistortie. Lensdistortie zorgt ervoor dat lijnen die in de werkelijkheid recht zijn, op de foto krom zijn. De lensdistortie wordt beschreven met behulp van 5 coëfficiënten. Om nuttige gegevens te halen uit de foto's, wordt de distortie ongedaan gemaakt op de foto's.

2.1.2 Camera calibratie

Het calibratie algoritme wat we in deze thesis gebruikt hebben is het algoritme dat wordt beschreven in [16]. We hebben het zelf niet geïmplementeerd, maar gebruik gemaakt van de implementatie in de Intel Open Computer Vision Library [5] en de Camera Calibratie Toolbox voor Matlab [2].

Dit algoritme maakt gebruik van een dambord patroon (zie figuur 2.1), dat voor de camera wordt gehouden en waarvan verschillende foto's worden genomen. Het maakt geen verschil uit of het patroon of de camera wordt bewogen, de beweging moet zelfs niet gekend zijn. Het algoritme heeft minstens twee foto's nodig die uiteraard van een verschillend standpunt zijn genomen.

Figuur 2.1: Voorbeeld van een calibratie patroon

Eerst worden uit de foto's de hoekpunten van de vakjes van het dambord patroon gehaald. Met behulp van deze hoekpunten en de grootte van de vakjes zal het algoritme de intrinsieke parameters berekenen. De lensdistortie wordt met behulp van OpenCV ongedaan gemaakt. Voor verdere details zullen we verwijzen naar [16].

2.2 3D scanning algoritmen

In deze sectie zullen we enkele image based 3D scan algoritmen bekijken. Al deze algoritmen werken met een set foto's van de scène. Van iedere foto zijn de intrinsieke en extrinsieke parameters gekend.

Volgens [6] kunnen we image base 3D scan algoritmen indelen in volume gebaseerde algoritmen en oppervlakte gebaseerde algoritmen.

Volume gebaseerde algoritmen trachten de visual hull te benaderen door een collectie van elementaire cellen, ook wel voxels genoemd. De voxels vormen samen een volume die het object moet voorstellen.

Oppervlakte gebaseerde algoritmen proberen niet een volume te berekenen, maar proberen om een 3D mesh te construeren die het object moet benaderen.

We zullen nu een overzicht geven van enkele algoritmen, en hun voor- en nadelen bespreken.

2.2.1 Voxel Coloring

Een eerste algoritme is voxel coloring, het wordt beschreven in [13]. Bij voxel coloring wordt de ruimte opgedeeld in een drie dimensioneel grid van voxels, het is dus een volume gebaseerd algoritme.

Voor iedere voxel wordt de kleur berekend door de voxel te projecteren op de foto's waarin de voxel zichtbaar is. Als de voxel ongeveer dezelfde kleur heeft in al deze foto's, dan is hij consistent en zal hij toegevoegd worden aan de scène.

De voxels worden verdeeld in een serie van lagen met toenemende afstand tot de camera. Alle lagen zullen bezocht worden door het algoritme, in iedere laag zal voor alle voxels worden bepaald of ze tot de scène behoren.

Omdat we de lagen bezoeken in toenemende afstand tot de camera kunnen we occlusies behandelen door een één bit masker voor iedere pixel van de foto's. Dit masker is initieel 0. Als we een voxel vinden die geprojecteerd wordt op de pixel en ook nog eens tot de scène behoort, zetten we het masker op 1. Nadat het masker op 1 is gezet weten we dat alle volgende voxels die op deze pixel projecteren, bedekt zijn door een andere.

Voordelen

Een groot voordeel van dit algoritme is dat het in één pas werkt. En dus niet meerdere keren over alle voxels moet itereren.

Door de manier van itereren door de voxels worden occlusies op een handige en elegante manier afgehandeld met het bit masker.

Aangezien er slechts één voxel tegelijkertijd behandeld wordt, en voxels niet onderling vergeleken worden, hoeft de hele scène niet in het geheugen worden

gehouden. Alleen de foto's en het bit masker voor iedere foto moeten in het geheugen worden gehouden. Hierdoor is de plaatscomplexiteit van het algoritme lineair en kunnen dus grote hoeveelheden foto's gebruikt worden.

Nadelen

Het algoritme veronderstelt dat de scène en de belichting stationair is, bovendien moet de scène bestaan uit oppervlakken die lambertiaans zijn. Dit is noodzakelijk omdat het algoritme met kleuren werkt, speculaire en doorzichtige oppervlakten kunnen van kleur veranderen als het camerastandpunt wijzigt.

Een tweede nadeel is dat er met een drempelwaarde wordt gewerkt bij het bepalen of de voxel ongeveer dezelfde kleur heeft in verschillende foto's. Een te kleine drempelwaarde kan ervoor zorgen dat de scène niet goed wordt gereconstrueerd, een te grote kan dan weer zorgen voor voxels die fout zijn.

2.2.2 Space Carving

Net zoals voxel coloring is space carving een volume gebaseerd algoritme. Het werkt ook met voxels en zal op dezelfde manier bepalen of een voxel tot de scène behoort.

In tegenstelling tot voxel coloring vertrekken we niet van een lege scène, maar van een grid van voxels. Voxels worden ook niet toegevoegd, maar verwijderd van het grid. Het grid zal meerdere keren doorlopen worden, totdat er geen voxels meer kunnen verwijderd worden.

Dit algoritme werkt eigenlijk zoals een beeldhouwer, die delen van een stuk steen wegkapt om tot een beeld te komen.

Het algoritme is beschreven in [7], aangezien dit één van de twee algoritmen is voor deze thesis geïmplementeerd zijn, hebben we een heel hoofdstuk gewijd aan de implementatie, voor verdere details zullen we doorverwijzen naar hoofdstuk 3.

Voordelen

Een voordeel van het space carving algoritme is de vrij simpele implementatie.

Nadelen

Net zoals Voxel Coloring kan het algoritme niet overweg met speculaire en doorzichtige oppervlakten. Ook in dit algoritme worden kleuren vergeleken om de consistentie van een voxel te bepalen, dit heeft tot gevolg dat er ook met een drempel waarde moet gewerkt worden.

Nog een nadeel is het grote initiële geheugengebruik van het algoritme, naarmate er echter meer voxels worden verwijderd van het grid, zal het geheugengebruik dalen.

Het laatste nadeel, is dat dit algoritme meerdere passen nodig heeft om tot een eindresultaat te komen.

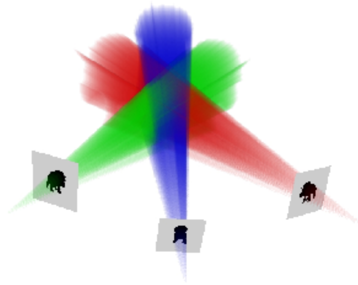
2.2.3 Image Based Visual Hulls

Dit algoritme wordt beschreven in [10]. Het is een oppervlakte gebaseerd algoritme, dat een exacte representatie tracht te berekenen. Dit is één van de twee algoritmen die we gekozen hebben om te implementeren, voor de volledige beschrijving van het algoritme zullen we verwijzen naar hoofdstuk 4.

Kort overzicht

Het algoritme werkt met het silhouet van het object in de verschillende camerastandpunten. Het silhouet vormt samen met de positie van de camera een soort van kegel (zie figuur 2.2.3). Dit algoritme zal de visueel omhullende berekenen, door de kegels met elkaar te intersecteren.

Het intersecteren van deze kegels is een 3D operatie maar volgens [11] kan deze intersecties gereduceerd kunnen worden naar simpelere intersecties in 2D. Dit komt omdat iedere kegel gedefinïeerd wordt door een 2D silhouet.



Figuur 2.2: Silhouetten en kegels van 3 beelden van een zelfde scène, de intersectie van de kegels zal de visueel omhullende vormen

Voordelen

Het grote voordeel van dit algoritme is dat een polygonale representatie wordt berekend. Dit heeft tot gevolg dat het renderen van het model kan gedaan worden door graphics hardware. Bovendien is een mesh standpunt onafhankelijk, ze moet dus maar één keer berekend worden.

Nog een groot voordeel van dit algoritme is dat het zeer snel kan berekend worden, mede door de reductie naar 2D van de intersectie berekeningen.

Nadelen

Een nadeel van dit algoritme is dat de texture die gebruikt wordt om het gegenereerde model te bekleden, afhankelijk is van de positie van de camera. Dit doet deels het voordeel van de polynomiale representatie teniet, alhoewel het niet onmogelijk moet zijn om een texture te berekenen die niet afhankelijk is van de positie van de camera.

Daarnaast heeft dit algoritme ook als nadeel dat het eigenlijk niet het object berekend, maar de visueel omhullende (visual hull). Het object bevindt zich in de visueel omhullende, maar de visueel omhullende kan meer zijn dan het object. Als er bijvoorbeeld een deuk in het object is en deze deuk op geen enkele manier tot uiting komt in het silhouet, zal deze deuk niet in het uiteindeljke resultaat zitten.

2.2.4 Gekozen algoritmes

Aangezien er twee soorten 3D scan algoritmes zijn, en we er twee moesten kiezen om te implementeren, hebben we gekozen voor één algoritme van iedere soort.

Voor de volume gebaseerde algoritmen hebben we het space carving algoritme gekozen, bij de oppervlakte gebaseerde algoritmen hebben we voor het image based visual hull algoritme gekozen.

Hoofdstuk 3

Space Carving

In dit deel van de thesis zullen we de implementatie van het space carving algoritme beschrijven. Space carving is een algoritme waarbij de ruimte rond het te scannen object wordt opgedeeld in kleine driedimensionale kubussen (voxels).

Voxels die vanuit verschillende camerastandpunten verschillende kleuren hebben, worden uit het voxelvolume verwijderd. Als een voxel in alle camerastandpunten ongeveer de zelfde kleur heeft, dan wordt deze kleur aan de voxel toegekend. Dit noemt men de consistentie test, een voxel is consistent als hij ongeveer dezelfde kleur heeft in alle camerastandpunten waar hij zichtbaar is.

De implementatie is vooral gebaseerd op [4] en [7]. In de volgendes secties, zullen we de gebruikte datastructuren en algoritmes verder toelichten.

3.1 Datastructuren

Aangezien we met een 3D grid van voxels te maken hebben, is de meest logische en simpele datastructuur een 3D matrix. Een 3D matrix verbruikt echter heel veel geheugen : een matrix van 100 bij 100 bij 100 voxels heeft in totaal 1.000.000 voxels. Dit aantal daalt naarmate er meer voxels inconsistent worden bevonden, waardoor er ook geheugen terug kan worden vrijgegeven.

Daarom wordt er in [7] gebruikt gemaakt van een 2D matrix van dubbel

gelinkte lijsten. Er wordt voor iedere rij voxels met coördinaten X en Y een reeks van Z-spannen bijgehouden, alle voxels tussen deze spannen zijn consistent bevonden.

$$V[X,Y] = \left\{ \left\{ Z_1^1, Z_2^1 \right\}, .., \left\{ Z_1^k, Z_2^k \right\} \right\} \tag{3.1}$$

Volgens de auteurs van [7] zou deze methode minder geheugen verbruiken, maar aangezien hier niet de kleur van iedere voxel in wordt opgeslagen, zouden we nog ergens anders de kleur voor iedere voxel moeten bijhouden. Dit lijkt mij dan weer het voordeel van minder geheugen verbruik teniet te doen.

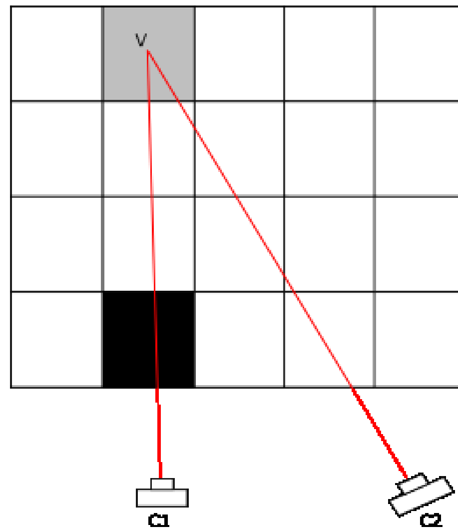
Uiteindelijk hebben we gekozen voor een 3D matrix. Van de twee mogelijke oplossingen leek deze het simpelste, en aangezien de methode van [7] niet het uiteindelijke geheugen verbruik lijkt te verminderen, valt het grote voordeel van deze methode weg.

3.2 Overzicht

We zullen eerst een overzicht geven van het algoritme. Daarna zullen we op ieder punt dieper ingaan, zien wat de specifieke problemen zijn en de oplossingen die we gekozen hebben verder toelichten.

- Laad de data van al de foto's en hun camera eigenschappen.
- Initialiseer het voxel volume.
- Itereer door het voxel volume totdat er geen voxels meer verwijderd kunnen worden :
  - Voor iedere voxel ga na bij welke camera hij zichtbaar is.
  - Zoek de kleur van de voxel op iedere foto waar hij zichtbaar is.
  - Als alle kleuren ongeveer gelijk zijn (de voxel is consistent), doen we niks.
  - Als de kleuren te veel verschillen, verwijderen we de voxel.

Eerst zullen we de zichtbaarheid van een voxel bespreken, gevold door het bepalen van de kleur van een voxel. Daarna zullen we de consistentietest aan



Figuur 3.1: De voxel V is zichtbaar in camera $C2$ maar niet in $C1$

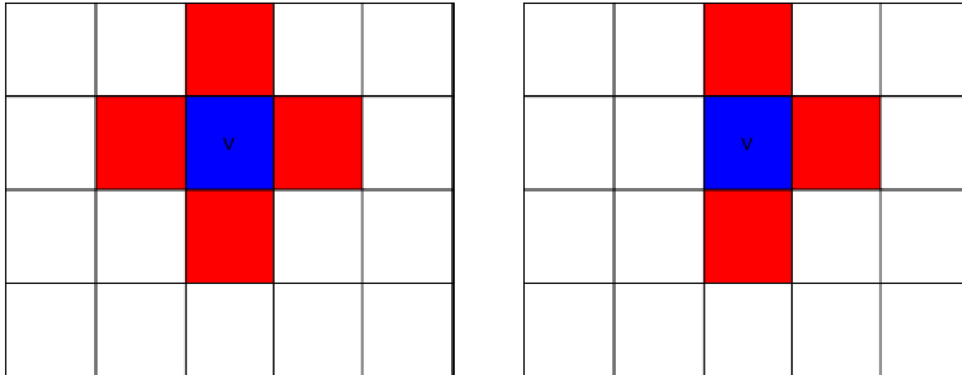
bod laten komen, om uiteindelijk af te sluiten met de manier van itereren door het grid.

3.3 Zichtbaarheid van een voxel

Stel dat we een verzameling camera's C hebben en een voxel v. We moeten bepalen in welke camera's van C, de voxel v zichtbaar is. Uiteraard gaan we voor iedere individuele camera van C bepalen of v al dan niet zichtbaar is, maar we kunnen echter al op een snelle manier beslissen of v voor geen enkele camera zichtbaar kan zijn.

Met deze snelle methode kunnen we heel wat tijd besparen doordat we voxels vroegtijdig kunnen elimineren. De voxels worden uiteraard niet verwijderd van het voxelgrid, we kunnen gewoon niet bepalen of ze al dan niet consistent zijn omdat geen enkele camera ze kan zien.

Als de eliminatie van een voxel v niet lukt, moeten we beslissen of v al dan niet zichtbaar is. De voxel v is zichtbaar in een camera C_i, als er zich geen andere voxel bevindt tussen v en C_i. Dit doen we door een aangepaste versie van een algoritme gebruikt bij raytracen, toe te passen. Dit algoritme traceert een straal vanuit v naar C_i, en gaat dan kijken of de straal door andere voxels dan v gaat.



(a) V is volledig omringd door andere voxels, en is dus niet zichtbaar

(b) V is niet volledig omringd door andere voxels, en kan dus zichtbaar zijn voor sommige camera's

Figuur 3.2: Sommige voxels kunnen geëlimineerd worden, sommige niet

In de volgende sectie zullen we de eliminatie methode bespreken, daarna zullen we doorgaan met het traceer algoritme in de daarop volgende secties.

3.3.1 Eliminatie van interne voxels

Als we door het voxelgrid itereren kunnen we in feite alleen maar voxels behandelen die zich op de rand bevinden van het object, met object bedoelen we hier de verzameling van voxels die nog in het grid zitten. Alle voxels die niet op deze rand zitten, kunnen niet zichtbaar zijn. We zullen dit de interne voxels noemen.

Een voxel is intern als al zijn naburige voxels ook in het grid zitten. Twee voxels zijn buren als ze één zijkant delen. Aangezien een voxel een kubus is, heeft een voxel dus zes buren. Om te bepalen of een voxel intern is, moeten we zien of alle zes buren in het grid zitten.

Stel dat de voxel v op positie (x, y, z) zit in het voxelgrid, dan moeten we kijken naar de voxels $(x + 1, y, z)$, $(x - 1, y, z)$, $(x, y + 1, z)$, $(x, y - 1, z)$, $(x, y, z + 1)$ en $(x, y, z - 1)$. Als deze allemaal in het grid zitten, dan is v niet zichtbaar.

We hebben hierboven vermeld dat een voxel zes buren heeft, dit gaat echter

niet op voor voxels die op de rand van het grid liggen. Om deze voxels te behandelen, doen we alsof het voxelgrid aan alle kanten een extra laag voxels heeft, die allemaal verwijderd zijn.

3.3.2 Het traceer algoritme

Om te bepalen of een voxel v zichtbaar is vanuit een gegeven camera standpunt C_i, moeten we controleren of er zich geen andere voxel tussen de camera en v bevindt. Een gelijkaardig probleem is het zoeken van intersecties bij een raytracer, waarbij de ruimte wordt onderverdeeld in een grid. Om het aantal intersectie berekeningen te verminderen zal men alleen intersecties doen met objecten die in gridcellen liggen waar de straal doorgaat.

We hebben het algoritme uit [1] om een straal door een voxelgrid te traceren, lichtjes aangepast om te beslissen of een voxel al dan niet zichtbaar is.

In de volgende sectie zullen we een aantal voorwaarden bespreken, waaraan het voxelgrid en de camera's moeten voldoen opdat het algoritme met succes kan toegepast worden. Daarna zullen we het algoritme zelf bespreken in secties 3.3.4 en 3.3.5.

3.3.3 Voorwaarden

Voordat we het algoritme dieper gaan bestuderen, moeten we opmerken dat het van belang is dat alle camera's buiten het voxelgrid liggen. Als een camera in het grid ligt dan zal enkel de voxel waarin de camera zich bevindt als zichtbaar beschouwd worden, alle andere voxels zullen onzichtbaar zijn, ze worden immers geblokkeerd door de voxel waarin de camera zit.

Als alle camera's nu in het grid liggen heeft dit tot gevolg dat geen enkele voxel verwijderd zal worden. De enigste voxels die verwijderd kunnen worden zijn de voxels waarin de camera's liggen, maar deze zijn maar zichtbaar vanuit één camera, en kunnen dus niet inconsistent bevonden worden (zie hoofdstuk 3.5 voor meer uitleg over consistentie).

Om deze problemen dus te vermijden moet de omvang van het voxelgrid zodanig gekozen worden dat geen enkele camera in het voxelgrid ligt.

3.3.4 De initialisatie fase

De voxel van waaruit we starten noemen we v, met positie (v_x, v_y, v_z). De camera C_i heeft als positie (c_x, c_y, c_z). De straal tussen C_i en v noemen we r , en zijn richtingsvector u.

We initialiseren de variabelen X, Y en Z op de coördinaten van v in het grid. De variabelen *stepX*,*stepY* en *stepZ* zijn ofwel -1 of 1, afhankelijk van het teken van respectievelijk u_x,u_y en u_z.

Vervolgens bepalen we de plaats t waar r naar een andere voxel gaat in de X-richting, en we slaan dit op in de variabele $tMaxX$. We doen hetzelfde voor de andere twee assen, en initialiseren zo de variabelen $tMaxY$ en $tMaxZ$.

Uiteindelijk berekenen we voor iedere as een *tDelta* waarde (*tDeltaX*, *tDeltaY* en *tDeltaZ*). De *tDelta* waarde geeft aan hoeveel we langs r moeten bewegen (in eenheden van t), zodanig dat de afgelegde afstand op de as gelijk is aan de grootte van een voxel langs dezelfde as.

3.3.5 Doorlopen van het grid

De doorloop fase van het algoritme is vrij simpel :

```
// resolutie geeft aan hoeveel voxels
// er in iedere richting zijn
Vector res = grid->getResolution();
while (true)
{
  if (tMaxX < tMaxY)
  {
    if (tMaxX < tMaxZ)
    {
      X += stepX;
      // Als we buiten het volume zitten,
      // dan is de voxel zichtbaar
      if (X < 0 || X >= res.x)
        return true;
      tMaxX += tDeltaX;
    }
    else
```

```
    {
      Z += stepZ;
      // Als we buiten het volume zitten,
      // dan is de voxel zichtbaar
      if (Z < 0 || Z >= res.z)
        return true;
      tMaxZ += tDeltaZ;
    }
  }
  else
  {
    if (tMaxY < tMaxZ)
    {
      Y += stepY;
      // Als we buiten het volume zitten,
      // dan is de voxel zichtbaar
      if (Y < 0 || Y >= res.y)
        return true;
      tMaxY += tDeltaY;
    }
    else
    {
      Z += stepZ;
      // Als we buiten het volume zitten,
      // dan is de voxel zichtbaar
      if (Z < 0 || Z >= res.z)
        return true;
      tMaxZ += tDeltaZ;
    }
  }
  // we vragen de Voxel op positie (X,Y,Z) op,
  // als hij bestaat (dus niet NULL is)
  // dan is de originele voxel niet zichtbaar
  Voxel* v = grid->getVoxel(X,Y,Z);
  if (v)
    return false;
}
```

Eerst wordt bepaald wat het minimum is van $tMaxX$, $tMaxY$ en $tMaxZ$. Voor de minimale $tMax$ tellen we de stapwaarde op bij de respectievelijke as

op, en we vergroten $tMax$ met de overeenkomstige $tDelta$ waarde.

Daarna gaan we kijken naar de positie (X, Y, Z), als deze buiten het grid ligt, kunnen we concluderen dan v zichtbaar is, we zijn immers buiten het grid geraakt zonder dat we een andere voxel zijn tegengekomen.

Als (X, Y, Z) binnen het grid ligt moeten we kijken of er zich een voxel bevindt op deze positie, in dat geval zal deze v bedekken, en moeten we dus het algoritme stoppen omdat v niet zichtbaar is, in het andere geval zullen we doorgaan met het algoritme.

3.4 Bepalen van de kleur van een voxel

In de vorige sectie hebben we besproken hoe we moesten bepalen of een voxel v zichtbaar is voor een camera C. Als de voxel zichtbaar is, dan moeten we de kleur van de voxel bepalen in camera C.

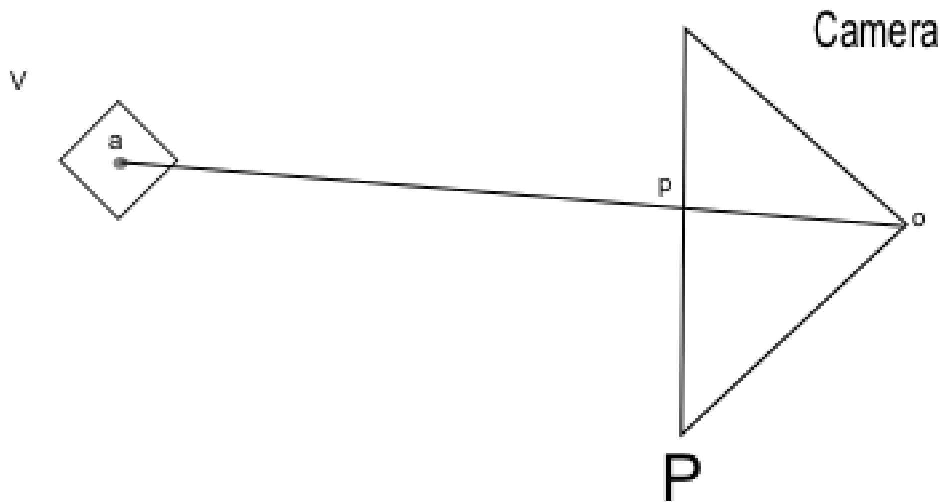
Om dit te doen, moeten we v projecteren op C en de kleur bepalen van de projectie. Aangezien v een voxel is en geen punt, is dit niet triviaal. Er zijn drie methoden om dit te doen :

- Puntprojectie : Projecteer het middelpunt van de voxel op het camera vlak.
- Voxelprojectie : Projecteer de hele voxel op het camera vlak.
- Spriteprojectie : Stel de voxel voor als een sprite en projecteer de sprite.

In de volgende secties zullen we ze alle drie bespreken.

3.4.1 Puntprojectie

Van de drie methoden is dit de eenvoudigste en de snelste. We projecteren het middelpunt van de voxel v op het camera vlak, zodat we nu een punt hebben in pixel coördinaten. Aangezien het hoogst onwaarschijnlijk is dat we rechtstreeks op een pixel van het beeld projecteren, nemen we de gemiddelde kleur van de vier omliggende pixels. Als er buiten het beeld geprojecteerd



Figuur 3.3: Het middelpunt a van voxel V wordt geprojecteerd op het cameravlak P in het punt p

wordt (de voxel is niet zichtbaar vanuit C), dan zullen we de voxel niet beschouwen bij de consistentie test.

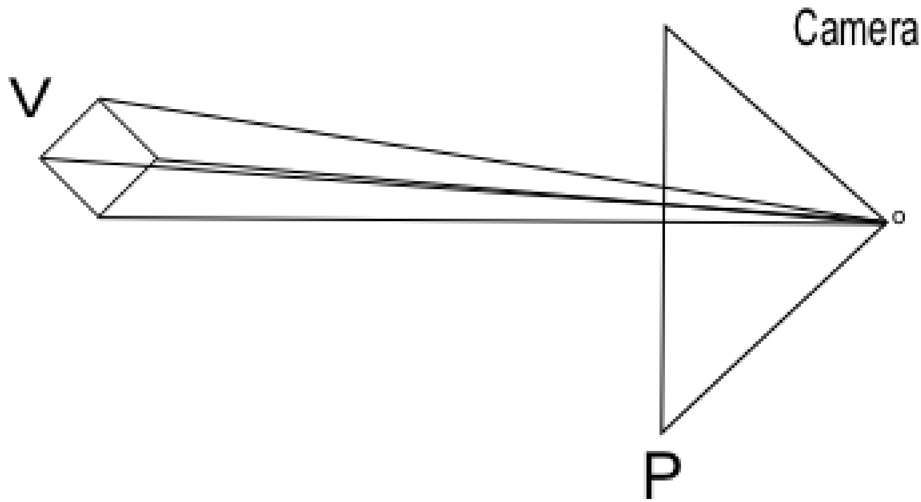
Deze methode is snel en eenvoudig maar voxels zijn geen punten. Een geprojecteerde voxel kan een groter gebied beslaan dan een pixel, voor een accurate kleurbepaling zouden we met al de pixels die in de projectie van de voxel vallen, moeten rekening houden.

Tegenover dit nadeel moeten we dan wel stellen dat naarmate de resolutie van het grid verhoogt, de voxels kleiner worden. Hoe kleiner de voxels zijn, hoe minder groot de projectie wordt van een voxel. Als ze klein genoeg zijn, zullen ze zelfs kleiner zijn dan een pixel, waardoor puntprojectie dan wel een correct resultaat teruggeeft.

3.4.2 Voxelprojectie

Waar we met puntprojectie alleen het middelpunt zullen projecteren op het vlak van de camera, zullen we met voxel projectie de volledige voxel projecteren.

Om dit te doen zullen we alle acht hoekpunten van de voxel moeten projecteren op het vlak van de camera. Dit levert uiteraard een algoritme op dat acht keer trager is dan punt projectie. Van deze acht geprojecteerde punten



Figuur 3.4: Project van voxel V op het cameravlak P

zullen we de convex omhullende moeten berekenen, om daarna de gemiddelde kleur in de convex omhullende te berekenen, die we als resultaat teruggeven.

Aangezien we hier acht punten gaan projecteren zullen we ook acht zichtbaarheidstesten (zie sectie 3.3) moeten doen voor al deze punten. Dit vertraagt het algoritme echter nog meer.

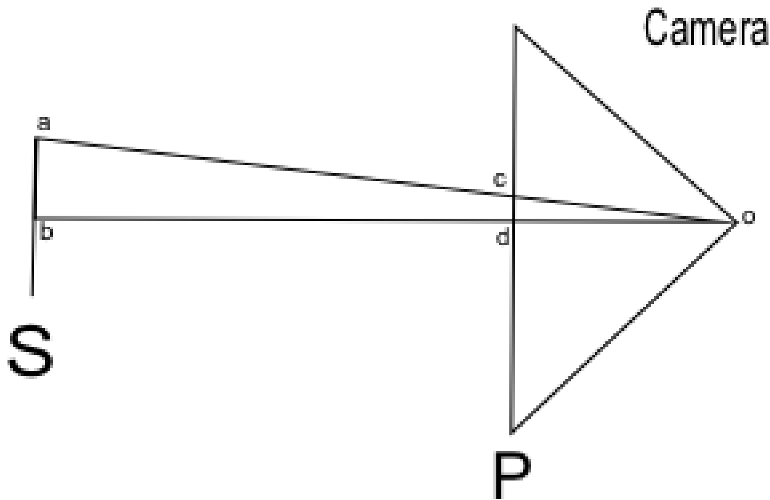
Dit algoritme is veel nauwkeuriger dan punt projectie, maar het is dan ook een stuk trager en het vergroot de complexiteit aanzienlijk.

3.4.3 Sprite projectie

Een derde algoritme is sprite projectie, een soort van gulden middenweg tussen de twee vorige algoritmes.

Als we naar de projectie van een voxel kijken dan merken we dat deze groter wordt naarmate de voxel dichter bij de camera staat. Om een accurate kleur te krijgen moeten we rekening houden met de grote van de projectie.

We stellen de voxel voor als een sprite, een vierhoekig vlak dat evenwijdig staat met het camera vlak. Het grote voordeel van deze voorstelling is dat de projectie een vierhoek is, wat de kleurberekening vergemakkelijkt. Het centrum van de sprite is wordt gelijk gesteld aan het centrum van de voxel, hoogte en de breedte van de sprite is gelijk aan de grootte van de voxel.



Figuur 3.5: Projectie van een sprite

Wat we hier in feite doen is de voxel gelijk stellen aan een voxel waarvan we alleen één zijkant kunnen zien.

Om deze sprite te projecteren, projecteren we eerst het middelpunt b van de sprite op het punt d in het camera vlak. Daarna moeten we de grootte van de sprite S bepalen op het camera vlak P (zie figuur 3.5). Stel dat b het middelpunt van de voxel is, en s de grootte van een voxel is. Het punt a kunnen we dan berekenen met behulp van de volgende formule :

$$a = b + (s/2) * up \tag{3.2}$$

Waarbij up de vector van de camera is die naar boven wijst. Nadat we a berekend hebben, projecteren we a op het camera vlak, dit geeft het punt c als resultaat. De afstand $2|cd|$ nemen we nu als grootte van de sprite in het camera vlak.

Dit algoritme is nauwkeuriger dan punt projectie en sneller dan voxel projectie. We kunnen het zien als een benadering van voxelprojectie.

3.4.4 Besluit

Puntprojectie is niet nauwkeurig genoeg, vooral omdat het een voxel beschouwt als een punt, terwijl een voxel een grotere projectie kan hebben dan

een punt. De hele voxel projecteren met behulp van voxelprojectie is dan weer te traag en te ingewikkeld.

Spriteprojectie is nauwkeuriger dan puntprojectie omdat het rekening houdt met het feit dat een voxel een grotere projectie heeft dan een punt. Het is bovendien ook sneller en simpeler dan voxel projectie.

We hebben besloten de drie algoritmen te implementeren en de gebruiker de keuze te laten. De gebruiker moet maar zelf de afweging maken tussen snelheid en accuraatheid.

3.5 Consistentie van een voxel

Nadat we eerst beslist hebben in welke camera's de voxel v zichtbaar is, en daarna voor iedere camera de kleur van de voxel berekend hebben, moeten we nog zien of de voxel v consistent is. Een voxel is consistent als de kleurwaarden niet te veel verschillen. Als een voxel inconsistent wordt bevonden, dan verwijderen hem van het voxelgrid.

De verzameling camera's van waaruit v zichtbaar is, zullen we CV noemen. Voor iedere camera c uit CV hebben we de kleur bepaald van voxel v. De kleuren houden we bij in de verzameling C. Nu moeten we nog zien of de kleurwaarden van C niet te fel verschillen.

We kunnen een kleur beschouwen als een punt in een 3D ruimte, waar de R,G en B-componenten van een kleur de assen voorstellen. De Euclidische afstand tussen twee punten(kleuren), kunnen we dan nemen om het verschil tussen twee kleuren te meten.

Een simpel algoritme voor de consistentie test zou dan zijn :

1. Neem twee kleuren $a = (a_r, a_g, a_b)$ en $b = (b_r, b_g, b_b)$ uit C.
2. Bereken de afstand $d = \sqrt{(a_r - b_r)^2 + (a_g - b_g)^2 + (a_b - b_b)^2}$.
3. Als $d < D$, waarbij D een drempelwaarde is, dan is de voxel v inconsistent, en breken we het algoritme af.
4. Blijf het bovenste herhalen totdat alle kleuren in C paarsgewijs vergeleken zijn of totdat de voxel v inconsistent wordt bevonden.

Het bovenstaande algoritme heeft echter een probleem waardoor het niet geschikt is voor de consistentie test. Stel dat de verzameling C in totaal n kleurwaarden bevat, en dat $n-1$ kleurwaarden niet te fel van elkaar verschillen, terwijl de n-de waarde wel te fel van een andere kleur verschilt. Door die ene kleur zal de voxel dan inconsistent bevonden worden, terwijl die ene uitschieter misschien door een afrondingsfout bij het bepalen van de kleur, veroorzaakt wordt.

We moeten dus een test hebben die alle kleurwaarden tegelijk vergelijkt. Daarom zullen we de standaardafwijking van alle kleuren berekenen. Als de standaardafwijking kleiner is dan de drempelwaarde D, zullen we de voxel consistent bevinden, als hij groter is dan D, zullen we de voxel inconsistent bevinden en verwijderen uit het voxelgrid.

Met de volgende formule zullen we de standaardafwijking σ_{RGB} berekenen :

$$\sigma_{RGB}^2 = \frac{\sum_{i=1}^{n} (\bar{c} - c_i)^2}{n} \tag{3.3}$$

Waarbij :

$$\bar{c} = \frac{\sum_{i=1}^{n} c_i}{n} \tag{3.4}$$

Uiteindelijk zullen we dan de voxel v consistent bevinden als $\|\sigma_{RGB}\| < D$.

3.6 Iteratie door het voxelgrid

In de vorige secties hebben we gezien hoe we de consistentie van een voxel moeten beslissen. Nu moeten we nog de manier van itereren over het voxelgrid bespreken.

We kunnen niet zomaar lukraak voxels op hun consistentie gaan testen, dit zou veel te veel nodeloze iteraties over het voxelgrid opleveren. Stel we hebben twee voxels (v_1 en v_2) en een camera c. De voxel v_1 bedekt de voxel v_2 in camera c, met andere woorden v_1 ligt tussen c en v_2. Stel dat v_1 inconsistent is, en v_2 consistent.

Als we tijdens een passage over het grid eerst v_2 behandelen dan merken we dat deze niet zichtbaar is. Tijdens dezelfde passage zal v_1 verwijderd worden,

waardoor we in de volgende passage v_2 wel kunnen behandelen. Is de volgorde echter omgekeerd dan zal v_1 verwijderd worden, zodat we in dezelfde passage nog v_2 kunnen behandelen.

De volgorde waarin we voxels bezoeken kan er dus voor zorgen dat werk dat in één passage kan gedaan worden, in twee of meer passages gedaan wordt. Dit is uiteraard niet al te efficiënt. Een algoritme dat rekening houdt met de occlusie relaties van de voxels kan dus veel tijd besparen.

Het algoritme dat deze eigenschap heeft is het plane-sweep algoritme. We zullen dit algoritme in de volgende sectie bespreken.

3.6.1 Het plane-sweep algoritme

Het plane-sweep algoritme maakt gebruik van meerdere passages over het voxelgrid. Tijdens iedere passage bewegen we een vlak door het voxelgrid en testen we de consistentie van de voxels die in het vlak liggen.

Het vlak staat altijd loodrecht op een as van de wereld. We kunnen het vlak zowel in positieve als negatieve richting langs de assen bewegen. Dit geeft dus uiteindelijk zes richtingen. Een passage langs de positieve X-as over een voxelgrid dat $N \times M \times K$ groot is, ziet er als volgt uit :

- $\forall y, z$ met $y \in \{1, \ldots, M\}$ en $z \in \{1, \ldots, K\}$: test de voxel $(1, y, z)$ op consistentie.
- $\forall y, z$ met $y \in \{1, \ldots, M\}$ en $z \in \{1, \ldots, K\}$: test de voxel $(2, y, z)$ op consistentie.
- $\ldots$
- $\forall y, z$ met $y \in \{1, \ldots, M\}$ en $z \in \{1, \ldots, K\}$: test de voxel (N, y, z) op consistentie.

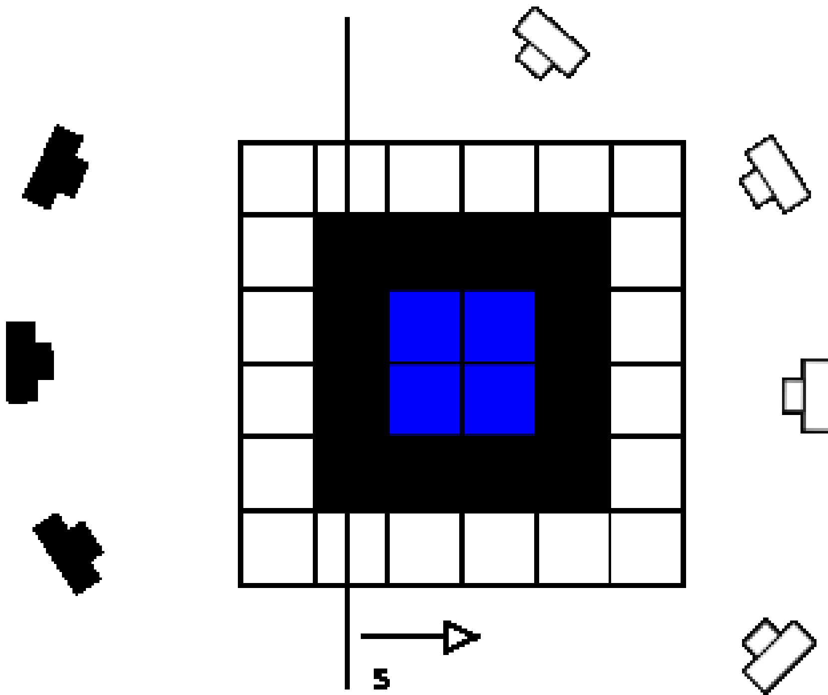
Bij een passage langs de negatieve X zal de X-coördinaat van N tot 1 gaan in plaats van 1 tot N. Passages langs de andere assen verlopen analoog.

Om ervoor te zorgen dat alle occlusie relaties tussen de voxels en de camera's gerespecteerd worden, zullen alleen de camera's die achter het sweep-plane

(a) De gele camera wordt gebruikt terwijl de andere twee niet gebruikt worden omdat ze zich voor het sweep-plane bevinden.

(b) Naarmate het sweep-plane wordt bewogen, worden meer en meer camera's gebruikt.

Figuur 3.6: Figuur uit [7]



Figuur 3.7: We laten alleen de camera's toe die achter het sweep-plane S liggen (de zwarte camera's). De pijl geeft de richting aan waarin s bewogen wordt.

liggen gebruikt worden bij het bepalen van de consistentie. Camera's worden als het ware geactiveerd als het sweep-plane er voorbij beweegt.

Als we dit algoritme nu toepassen op het bovenstaande voorbeeld, dan zijn er drie mogelijkheden :

- c ligt achter het sweep-plane. Dit heeft tot gevolg dat v_1 al bezocht is als we v_2 bezoeken, omdat v_1 tussen v_2 en de camera ligt. Het sweep-plane is dus v_1 al gepasseerd.
- c ligt voor het sweep-plane. Dit betekent dat c nog niet actief is, en dat er geen rekening wordt gehouden met deze camera. We behandelen dit geval best door een passage te doen in de omgekeerde richting, wat als resultaat het eerste geval heeft.
- c,v_1 en v_2 liggen alle drie in het sweep-plane. Om dit soort uitzonderlijke situaties te behandelen doen we passages over de andere assen, zowel in positieve als negatieve richting.

Een passage van het plane-sweep algoritme bestaat dus uit zes passages over het voxelgrid, over de drie assen zowel in positieve als negatieve richting. We stoppen het algoritme als er geen enkele voxel inconsistent wordt bevonden tijdens een volledige passage van het algoritme. Dus niet als er een passage over een as geen inconsistente voxels opleverd.

3.7 Rendering van resultaten

Om het finale object uit te tekenen maken we gebruik van het marching cubes algoritme. Het algoritme word beschreven in [14], dat zelf gebaseerd is op [8].

Het marching cubes algoritme dient om volumetrische modellen te renderen. Het doet dit door een polygonale mesh te genereren van het volumetrisch model. We gebruiken hier om het uiteindelijke voxelgrid om te vormen tot een polygonale mesh.

Deze polygonale mesh wordt dan uitgetekend met behulp van OpenGL.

3.8 Besluit

Ter afsluiting van dit hoofdstuk zullen we nog een overzicht geven van het volledige algoritme, zodanig dat we duidelijker zien hoe alle puzzelstukken van de vorige hoofdstukken in het geheel passen.

- Laad de data van al de foto's en hun camera eigenschappen.
- Initialiseer het voxel volume.
- Itereer door het voxel volume totdat er geen voxels meer verwijderd kunnen worden :
  - Doe een plane-sweep passage in de X,Y,Z,$-X$,$-Y$ en $-Z$ richting. (Sectie 3.6)

Waarbij iedere plane-sweep passage bestaat uit het volgende :

- Bereken het sweep-plane.
- Bepaal de verzameling C waarin alle camera's zitten die achter het sweep-plane liggen.
- Voor alle voxels v in het vlak van het sweep-plane :
  - Bepaal alle camera's van C, waarvoor v zichtbaar is, en sla deze op in de verzameling CV. (Sectie 3.3)
  - Projecteer v op alle camera's van CV, bereken de kleur van de projectie, en steek al de kleuren in een lijst. (Sectie 3.4)
  - Doe de consistentie test met alle kleuren in de lijst. (Sectie 3.5)
  - Als de consistentie test faalt, verwijderen we v, als ze niet faalt, dan geven we v de gemiddelde kleur van alle kleuren in de lijst.
- Verplaats het sweep-plane en ga terug naar het tweede punt als we nog niet alle voxels gehad hebben.

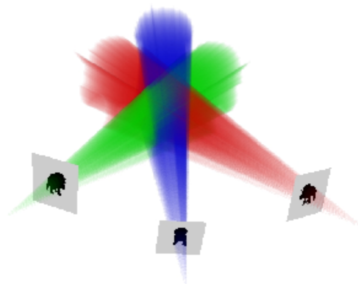
Hoofdstuk 4

Image Based Visual Hull

Het tweede algoritme dat geïmplementeerd werd voor deze thesis, is gebaseerd op [10] em [9]. Bij dit algoritme wordt een polygonale mesh berekend vanuit het silhouet van het object in verschillende foto's.

Het silhouet van een object zal bestaan uit een reeks van convexe of niet-convexe 2D polygonen. Deze polygonen kunnen zelfs gaten hebben, dit maakt geen verschil uit voor het algoritme. Iedere polygoon is een reeks van edges die twee opeenvolgende vertices met elkaar verbinden.

Als we nu een straal vanuit het camera standpunt schieten door de twee vertices van een edge, dan krijgen we een face. Doen we dit voor alle edges dan vormen al de faces samen een soort van kegel.



Figuur 4.1: Silhouetten en kegels van 3 beelden van een zelfde scène, de intersectie van de kegels zal de visueel omhullende vormen (figuur uit [11])

Het object zal in iedere foto een silhouet hebben (zie figuur 4), en dus ook een kegel. Wat dit algoritme in feite doet is de intersectie berekenen van al deze kegels.

We zullen nu een overzicht geven van het algoritme en zullen dan dieper ingaan op specifieke delen.

4.1 Overzicht

Een beknopt overzicht van het algoritme :

- Lees de data in (foto's)
- Voor iedere foto bereken het silhouet s en de bij behorende kegel c
- Neem twee verschillende kegels c_i en c_j, en bereken hun intersectie. Blijf dit doen totdat alle kegels met elkaar zijn geïntersecteerd.
- Voor ieder face van iedere kegel, intersecteer al de 3D polygonen die op dat face zijn geprojecteerd.

In de volgende sectie zullen we zien hoe we een silhouet genereren, daarna zullen we de intersectie van twee kegels behandelen. Gevold door een sectie over het intersecteren van de 3D polygonen, die op een face geprojecteerd zijn. We sluiten uiteindelijk af met het berekenen van de texture.

4.2 Berekenen van het silhouet

We berekenen het silhouet door voor iedere pixel aan te duiden of hij al dan niet deel is van het object. Als we te maken hebben met computergegeneerde objecten, vergelijken we iedere pixel met de achtergrondkleur. We moeten dan wel een achtergrondkleur kiezen die niet voorkomt op het object zelf.

Bij echte objecten zouden we hetzelfde kunnen toepassen met behulp van een blauw scherm, dit heeft echter het nadeel dat we voor een blauw object,

dan weer een scherm met een andere kleur moeten maken. Bovendien zal het blauw scherm het roterend platform niet verwijderen uit het uiteindelijk resultaat.

Een veel praktische en simpelere oplossing is eerst een reeks foto's nemen zonder object, gevolgd door een reeks foto's met het object. Voor iedere foto met object hebben we dan een foto zonder object. Als we nu deze twee foto's met elkaar vergelijken, dan zullen de pixels die verschillend zijn op de twee foto's, de pixels zijn waar het object zich bevindt.

Nu dat we weten welke pixels tot het silhouet behoren, creëren we één of meerdere polygonen met behulp van het marching squares algoritme.

Het marching squares algoritme is een 2D variant van het marching cubes algoritme. We zullen het algoritme hier niet uitleggen, aangezien het niets te maken heeft met het image based visual hull algoritme. Voor verdere details verwijzen we naar [12].

4.3 Intersectie van twee kegels

We hebben dus twee kegels c_i en c_j (met respectievelijk de silhouetten s_i en s_j), waarvan we de intersectie willen berekenen. Om dit te doen zullen we alle faces van c_i nemen en deze intersecteren met c_j.

Het intersecteren van een face van een kegel met een andere kegel, is een 3D operatie. Maar volgens [11] kan deze intersectie naar 2D worden gereduceerd, waardoor er minder complexere datastructuren gebruikt kunnen worden.

Om nu de intersectie van een face f van kegel c_i met een kegel c_j te berekenen, zullen we f eerst projecteren op het vlak van silhouet s_j, daarna berekenen we de intersectie van de geprojecteerde face f in 2D met silhouet s_j. Als we dit gedaan hebben projecteren we de geïntersecteerde polygoon terug op het vlak van face f.

Om het intersectie proces te versnellen zullen we eerst het silhouet preprocessen in een edge-bin datastructuur. De edge-bin structuur verdeelt het silhouet zodat we makkelijk de set van edges kunnen berekenen die een face f intersecteren.

We zullen in de volgende sectie beschrijven hoe we de edge-bin datastructuur

construeren, gevolgd door een sectie over het 2D intersectie proces met behulp van de edge-bin structuur, om dan af te sluiten met een sectie over het reconstrueren van polygonen uit een reeks edges die het resultaat zijn van het intersectie proces.

4.3.1 Edge-bin constructie

In het geval van perspectief projectie, zullen alle stralen op het oppervlakte van de kegel c_i projecteren tot een waaier van stralen op het camera vlak van silhouet s_j. Al deze stralen zullen door hetzelfde punt p_0 gaan, de epipool.

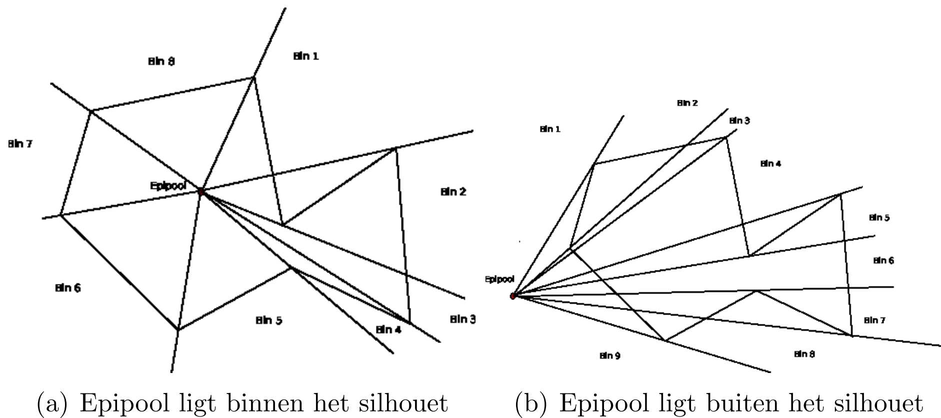
We kunnen nu alle lijnen in het camera vlak parametriseren met de waarde α, die de helling aangeeft met een referentie lijn (bv. een horizontale lijn). Met behulp van deze parametrisatie kunnen we nu het silhouet opdelen in een reeks van edge-bins.

Aan iedere vertex v van het silhouet s_j geven we een α-waarde, door de α-waarde van de lijn tussen p_0 en v te berekenen. Vervolgens sorteren we alle vertices op hun α-waarde. Voor alle twee opeenvolgende vertices v en w, definiëren we nu een edge-bin als een 3-tupel $(\alpha_{start}, \alpha_{end}, E)$, waarbij α_{start} de α-waarde van v is, α_{end} de α-waarde van w en E een lijst van edges is. De lijst van edges E bevat alle edges van het silhouet s_j die geheel of gedeeltelijk in de edge-bin liggen. Iedere vertex behalve de eerste en de laatste, vormt de grens tussen twee edge-bins.

De bins worden opgeslagen in een lijst, die gesorteerd is op de begin en eind α-waarden van de bins. Aangezien de vertices al gesorteerd zijn kunnen we de lijst gewoon gesorteerd opbouwen als we de vertices doorlopen.

Het berekenen van de edge lijst van iedere bin, hangt af van de positie van de epipool ten opzichte van het silhouet. De epipool kan ofwel buiten het silhouet liggen (figuur 4.2(b)), ofwel binnen (figuur 4.2(a)). We kunnen dit controleren door te kijken naar het bereik van de α-waarden te kijken. Is dit bereik groter dan 180 graden, dan ligt de epipool in het silhouet. Als het kleiner is dan 180 graden, dan ligt het buiten het silhouet.

Voor de duidelijkheid moeten we hierbij opmerken dat binnen het silhouet niet betekent dat de epipool werkelijk in het silhouet ligt. Als het silhouet bestaat uit een concave polygoon dan is het mogelijk dat de epipool niet in de concave polygoon ligt maar erbuiten, terwijl dit toch als binnen het silhouet



(a) Epipool ligt binnen het silhouet (b) Epipool ligt buiten het silhouet

Figuur 4.2: Mogelijke ligging van de epipool t.o.v. het silhouet

zal worden beschouwd. Met binnen bedoelen we eigenlijk er midden in.

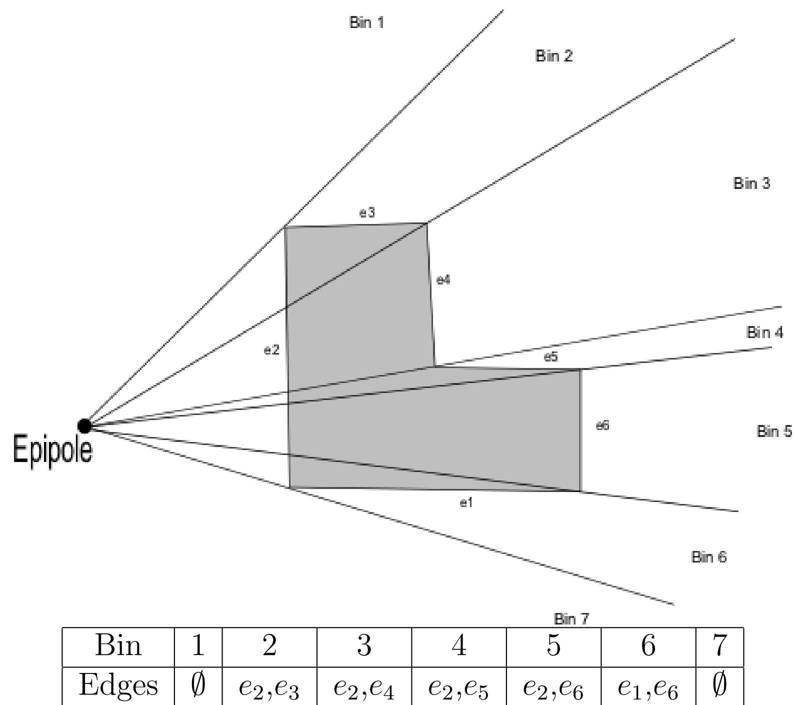
De epipool ligt erbuiten als we een rechte kunnen trekken tussen de epipool en het silhouet en deze rechte snijdt geen enkele zijde van een polygoon van het silhouet.

We zullen nu de twee gevallen apart bespreken in de volgende twee secties.

Epipool ligt buiten het silhouet

Om de edge lijst van iedere bin te berekenen, houden we eerst een set S van huidige edges bij, die in het begin leeg is. We itereren over de lijst van bins. Als we een bin b_k bezoeken dan voegen we alle edges aan S toe die beginnen op de begin vertex, en verwijderen we alle edges die eindigen op de begin vertex. Voor het begin van een edge nemen we de vertex met de kleinste α-waarde, het einde is de vertex met de grootste α-waarde. Uiteindelijk stellen we de lijst van edges E van bin b_k gelijk aan S en gaan we naar de volgende bin. We blijven dit doen totdat we alle bins gehad hebben. De hele procedure is geïllustreerd in figuur 4.3.

In iedere bin worden alle edges ook nog gesorteerd op de afstand tot de epipool. Dit vergemakkelijkt het berekenen van de intersecties.



Bin	1	2	3	4	5	6	7
Edges	$\emptyset$	e_2,e_3	e_2,e_4	e_2,e_5	e_2,e_6	e_1,e_6	$\emptyset$

Figuur 4.3: Constructie van de edge-bins. Het silhouet is het grijs gebied.

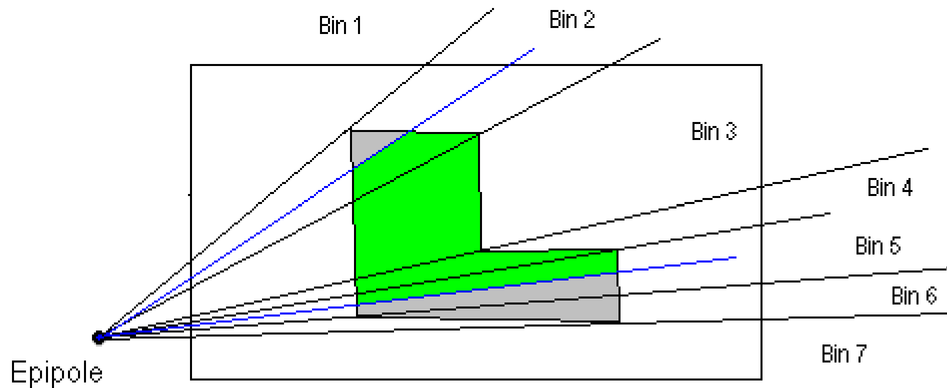
Epipool ligt binnen het silhouet

Als de epipool erbinnen ligt dan passen we grotendeels hetzelfde algoritme toe dat we in de vorige sectie hebben beschreven. Het verschil ligt in het feit dat we niet beginnen met een lege set S, maar met een set die de edges van de laatste bin al bevat.

We overlopen eerst alle bins van de eerste tot de laatste, en bepalen telkens de set van huidige edges zoals we hierboven beschreven hebben. We stellen de lijst van edges in iedere bin nog niet gelijk aan S. Na de eerste passage hebben we een set die de edges bevat van de laatste bin. Met deze set doen we nu een tweede passage, waarbij we wel de lijst van edges in iedere bin gelijkstellen aan S.

4.3.2 Intersectie met behulp van de edge-bins

Aangezien we nu perfect weten welke edges er in iedere bin zitten, kunnen we de intersectie berekenen van een geprojecteerd face f met het silhouet s_j.



Figuur 4.4: Intersectie van een geprojecteerd face (de blauwe lijnen) met het silhouet(het grijs gebied). Het resultaat is alles wat groen is.

Een face wordt in feite gevormd door twee stralen, als we deze op het camera vlak van s_j projecteren krijgen we twee lijnen l_1 en l_2 die door de epipool gaan. Voor deze twee lijnen berekenen we nu de α-waarde. Met de α-waarden kunnen we de twee bins b_i en b_j bepalen waarin l_1 en l_2 liggen.

We veronderstellen even dat $i \neq j$, het speciaal geval $i = j$ zullen we later behandelen. Ook veronderstellen we dat l_1 een kleinere α-waarde heeft dan l_2, als dit niet zo is dan kunnen we ze gewoon omwisselen.

We gaan alle bins doorlopen van b_i tot b_j. Terwijl we de bins doorlopen zullen we een set van edges (we zullen ze E noemen) bijhouden, deze edges zullen later tot één of meer polygonen verwerkt worden. Onder set verstaan we hier een verzameling waarin elementen niet twee keer of meer kunnen voorkomen.

In bin b_i weten we dat l_1 ligt, dus moeten we de intersectie berekenen van alle edges in b_i met de lijn l_1. Voor iedere edge e in bin b_i berekenen we het snijpunt s met l_1, en vormen we een nieuwe edge tussen s en het eindpunt van e, deze edge voegen we toe aan de set E.

We creëren ook nieuwe edges tussen de snijpunten, maar niet tussen alle snijpunten. Als de snijpunten bijvoorbeeld s_1,s_2 ... s_n zijn dan creëren we

de nieuwe edges (s_1, s_2), (s_3, s_4), (s_5, s_6) ... en voegen deze toe aan de set E. We slaan dus telkens een edge over. Dit doen we omdat als we aan s_1 het silhouet binnengaan, verlaten we aan s_2 het silhouet, om er aan s_3 dan weer terug binnen te gaan, ... Hierdoor is (s_2, s_3) dus geen edge van de intersectie tussen het silhouet s_j en de face f.

Het is mogelijk dat de epipool in een polygoon van het silhouet ligt. In dit geval moeten we ook een edge toevoegen tussen de epipool en het eerste snijpunt s_1. Om dit geval makkelijk te detecteren, werken we van buiten naar binnen. Dit betekent dus dat we eerst de edge (s_{n-1}, s_n) toevoegen, gevolgd door (s_{n-3}, s_{n-2}) ... . Uiteindelijk zullen we aan s_1 komen. Als we de edge (s_2, s_3) moeten toevoegen dan voegen we ook de edge $(s_1, epipool)$ toe.

Doordat de edges gesorteerd zijn op afstand tot de epipool, worden de snijpunten in een gesorteerde volgorde berekend, waardoor we ze gewoon aan een lijst kunnen toevoegen zonder ze te moeten sorteren op hun afstand tot de epipool.

Voor alle bins tussen b_i en b_j voegen we al hun edges toe aan de set E, behalve als ze er al in zitten of als ze al gesplitst zijn.

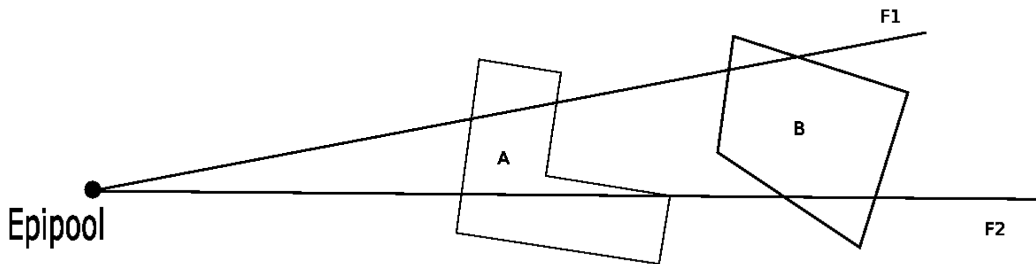
Uiteindelijk komen we dan aan de bin b_j, deze bin kunnen we analoog behandelen als b_i, op enkele kleine details na :

- We werken hier niet met l_1 maar met l_2.
- Als we een edge e splitsen met l_2 en we vinden het snijpunt s, dan vormen we een nieuwe edge met het beginpunt van e en s.

Voor het geval dat $b_i = b_j$, moeten we de snijpunten s en r berekenen van iedere edge met respectievelijk l_1 en l_2, en voegen we de edge (s, r) toe aan E. We vormen ook edges met alle snijpunten die op l_1 liggen zoals we dit hierboven gedaan hebben. Voor l_2 doen we identiek hetzelfde.

4.3.3 Creëren van polygonen

Met behulp van de set van edges E zullen we nu één of meer polygonen moeten vormen. We kunnen meerdere polygonen hebben als het silhouet bestaat uit twee of meer disjuncte delen (bv figuur 4.5).



Figuur 4.5: Als het silhouet bestaat uit twee disjuncte delen (A en B)), dan kan dat resulteren in twee of meer polygonen, als we de intersectie berekenen met een face F (projectie wordt gevormd door de lijnen $F1$ en $F2$)

Aangezien in iedere edge een pointer wordt bijgehouden naar de twee vertices die de edge verbindt, kunnen we makkelijk zien welke edges op elkaar aansluiten. Het algoritme voor de polygonen te construeren loopt als volgt :

- Neem een edge e_{start} van E en verwijder ze uit E.
- Steek e_{start} in een lege lijst die we de ketting gaan noemen.
- We blijven het volgende herhalen totdat we de ketting gesloten hebben :
- Voor iedere edge $e = (v_{start}, v_{end})$ uit E doe het volgende :
  - Als v_{end} gelijk is aan de start vertex van de eerste edge in de ketting en v_{start} gelijk is aan de eind vertex van de laatste edge van de ketting, dan hebben we de ketting gesloten. We verwijderen e uit E, creëren een polygoon uit de ketting, en we starten het algoritme opnieuw als er nog edges in E overblijven, anders stoppen we.
  - Als v_{end} gelijk is aan de start vertex van de eerste edge in de ketting, dan voegen we e vooraan toe aan de ketting en verwijderen we e uit E.
  - Als v_{start} gelijk is aan de eind vertex van de laatste edge van de ketting, dan voegen we e achteraan toe aan de ketting en verwijderen we e uit E.

Met dit algoritme kunnen we de polygonen creëren, die we gaan projecteren op het vlak van de face f. Het projecteren van een vertex doen we door een straal te schieten uit de camera door de vertex, en de intersectie met het vlak

te berekenen. Edges en polygonen worden geprojecteerd door al hun vertices te projecteren.

Problemen bij de projectie

Het projecteren van een polygoon p vanuit camera c_1 op een face f dat camera c_2 als oorsprong heeft, kan een probleem opleveren. Stel dat c_1 en c_2 bijna in stereo staan, met stereo bedoelen we dat de rechte tussen c_1 en c_2 evenwijdig is met de camera vlakken. Bij camera's die in stereo staan is er geen epipool, de epipool staat op oneindig.

We gaan nu de vertex v van p projecteren op f, dit doen we door een straal r te schieten vanuit c_1 door v en het snijpunt berekenen van r met het vlak van f. Het resultaat van deze intersectie is de afstand t die we in de parametrische vergelijking van r (vergelijking 4.1) moeten invullen om het intersectiepunt te berekenen.

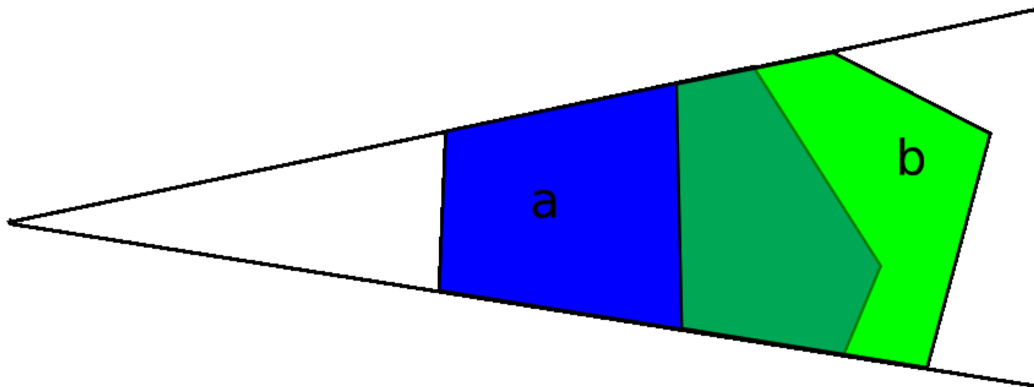
$$r_p = r_s + t * r_d \tag{4.1}$$

In vergelijking 4.1 is r_p een punt op de straal r, r_s het startpunt van r en r_d de richtingsvector van r.

In een bijna stereo situatie kan het mogelijk zijn dat t negatief is. Dit kan bijvoorbeeld gebeuren als c_1 en c_2 ongeveer boven elkaar staan, waarbij c_2 de bovenste camera is. Stel dan dat de straal r die uit c_1 vertrekt, naar beneden gaat terwijl de face f naar boven gaat. Het snijpunt van r en f zal achter de twee camera's liggen, waardoor de t waarde dus negatief is.

Zulke situaties kunnen we echter niet toelaten, als we negatieve t waarden tegenkomen, dan voegen we een punt toe dat dezelfde positie heeft als de camera van het face waarop we projecteren. We vermijden in feite punten die achter de camera liggen door deze te vervangen met punten die op de positie van de camera liggen.

We moeten nog opmerken dat c_1 en c_2 niet in stereo kunnen staan, we laten dit gewoon niet toe bij het genereren van de data.



Figuur 4.6: De doorsnede van twee polygonen op een face.

4.4 Intersecties op een face

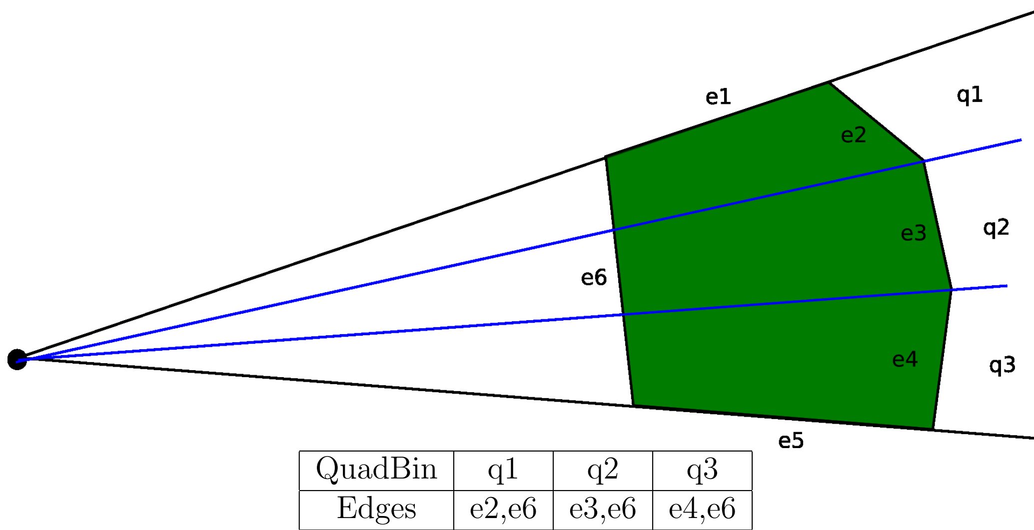
Nadat we iedere kegel met iedere andere kegel geïntersecteerd hebben, hebben we op ieder face van iedere kegel een reeks polygonen zitten. Wat we nu nog moeten doen is de intersectie van al deze polygonen doen, met intersectie bedoelen we hier de doorsnede van alle polygonen op een face (bv. Het donkergroene gedeelte op figuur 4.6).

Al de polygonen P liggen op het vlak van face f, bovendien liggen ze ook tussen de twee stralen r_1 en r_2 die de face f vormen. We zullen nu een gelijkaardige constructie toepassen als de edge-bins. We werken hier niet met edges maar met vierhoeken (quadrilaterals of quads).

4.4.1 Quad-bin constructie

Net zoals bij de edge-bins zullen we een α-waarde berekenen voor iedere vertex van iedere polygoon. Als α-waarde nemen we de hoek tussen r_1 en de straal gevormd door de vertex en het camera standpunt (wat ook de begin positie is van r_1 en r_2).

We sorteren opnieuw alle vertices op hun α-waarde, en creëren voor iedere twee opeenvolgende vertices v en w, een quad-bin $q = (\alpha_{start}, \alpha_{end}, Q)$, waarbij α_{start} de α-waarde is van v, en α_{end} die van w. Dit doen we alleen als de α-waarden van v en w verschillend zijn. Van de twee vertices van edge $e1$ in figuur 4.7, zullen we geen quad-bin maken.



QuadBin	q1	q2	q3
Edges	e2,e6	e3,e6	e4,e6

Figuur 4.7: Indeling van een polygoon in quad-bins, de blauwe lijnen zijn de grenzen van de quad-bins. De twee buitenste lijnen zijn de grenzen van de face.

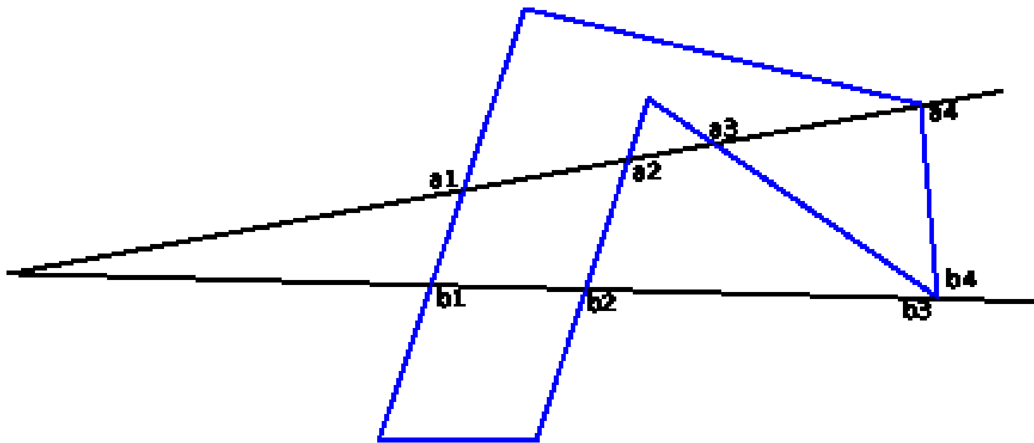
De verzameling Q bevat een reeks van vierhoeken, die in de bin zitten. Deze vierhoeken zullen we berekenen door alle polygonen op f in de quad-bins op te splitsen.

4.4.2 Opsplitsen van een polygoon in de quad-bins

Het opsplitsen van een polygoon p in de quad-bins, is geen triviale taak. We gaan eerst alle vertices af van p en slaan deze op in een lijst V, die we sorteren op de α-waarden van de vertices. Naast de lijst houden we ook een vertex v bij, v zal wijzen naar de huidige vertex en wordt geïnitializeerd op het eerste element van V.

Na de lijst van vertices maken we ook een lijst van edges van p, we zullen deze lijst LE noemen. We voegen echter niet alle edges toe aan de lijst, alle edges die volledig op één van de twee stralen van een quad-bin liggen, voegen we niet toe. In figuur 4.7 voegen we bijvoorbeeld $e1$ en $e5$ niet toe aan de lijst van edges.

Daarna passen we een algoritme toe dat nogal wat gelijkenissen bevat met het algoritme dat we gebruikt hebben bij het bepalen welke edges in welke edge-bin horen. We houden een set van actieve edges bij, die initieel leeg is,



Figuur 4.8: Opsplitsen van een niet convexe veelhoek in een quad-bin, levert meer dan twee intersectie punten op, op iedere straal ($a1$,..,$a4$ op de eerste en $b1$,..,$b4$ op de tweede).

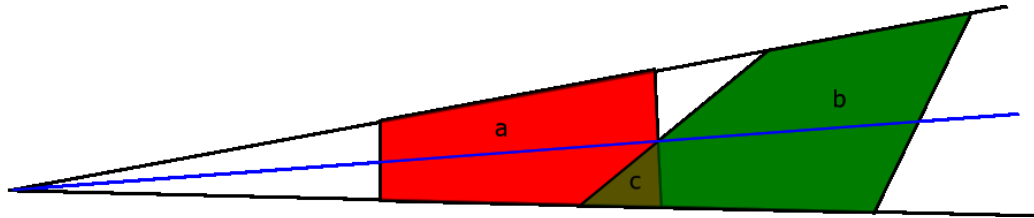
deze set zullen we E noemen.

We gaan nu de lijst van quad-bins overlopen. Stel we zitten aan quadbin q_i, dan kijken we naar de α-waarde van de begin straal van q_i. Als deze gelijk is aan de *alpha*-waarde van de vertex v, dan voegen we aan E alle edges van LE toe die beginnen in v, en verwijderen we alle edges die eindigen in v. Het begin van een edge is de vertex met de kleinste α-waarde. Na het aanpassen van E, schuiven we v op in de lijst V totdat de α-waarde van v verschillend is van de α-waarde van de begin straal van q_i.

Nadat we eventueel E aangepast hebben, maken we één of meer vierhoeken door E en q_i te intersecteren. Hoe dit gebeurt zullen we in de volgende sectie uitleggen. We blijven vierhoeken maken totdat we aan het einde van de lijst van quad-bins gekomen zijn.

Intersectie tussen E en q_i

De intersectie van E en q_i, berekenen we door de intersectie te berekenen van alle edges in E, met de twee stralen die de grens vormen van q_i. We slaan hier niet het intersectie punt zelf op in q, maar de afstand tot de oorsprong van de straal. Dit doen we omdat we dan makkelijk de intersectie kunnen berekenen, het punt kunnen we altijd terug berekenen door de afstand in te vullen in de parametrische vergelijking van de straal.



Figuur 4.9: De vierhoeken a en b die elkaar slechts overlappen op één straal. De intersectie moet het gebied c opleveren. We splitsen de bin in twee aan het intersectie punt.

Deze intersecties leveren voor iedere straal een even aantal afstanden op die we in twee lijsten stoppen en sorteren van klein naar groot. Het zijn er meer dan twee als we te maken hebben met niet convexe veelhoeken (zie figuur 4.8). Het aantal is even, omdat we als we op een punt de veelhoek binnengaan met een straal er noodzakelijkerwijs een ander punt moet zijn waar we de vierhoek verlaten. Dit geld mogelijk niet als we op een hoekpunt de veelhoek binnengaan, maar aangezien dat hoekpunt op twee edges ligt, die we allebei intersecteren met de straal, krijgen we dus twee samenvallende intersectie punten en hebben we dus toch een even aantal punten.

Van deze twee lijsten van punten, die we $I = \{i_1, \ldots, i_n\}$ en $J = \{j_1, \ldots, j_m\}$ zullen noemen, maken we nu een aantal vierhoeken, door telkens een interval te nemen uit iedere lijst te nemen en daarmee een vierhoek te vormen. We doen dit door i_1 en i_2 uit I, en j_1 en j_2 uit J te nemen en hiermee een vierhoek te maken die we in de lijst van vierhoeken van de quad-bin steken. De volgende vierhoek maken we niet met het tweede en derde element uit I en J, maar we nemen het derde en het vierde element, we slaan dus telkens een interval over.

4.4.3 Opsplitsen van de quad-bins

Om tijdens het intersecteren een hele reeks problemen te vermijden, zullen we nog sommige bins moeten opsplitsen in twee of meer bins.

Als twee vierhoeken in een bin overlappen, dan kan de overlap gebeuren op beide assen, maar het kan ook zijn dat de overlap slechts op één as plaatsvindt. Dit betekent dat twee edges van verschillende vierhoeken elkaar kruisen in de bin (zie figuur 4.9).

Als we dit niet vermijden kunnen we grote problemen krijgen bij het berekenen van de doorsnede van 2 of meer vierhoeken. In figuur 4.9 bijvoorbeeld zou de intersectie van a en b, c moeten opleveren. Dit is veel makkelijker te doen als we de quad-bin opsplitsen aan het snijpunt van de twee kruisende edges, de blauwe straal zal dan de worden tussen de twee nieuwe quad-bins.

4.4.4 Intersectie met behulp van de quad-bins

Nu dat we alle veelhoeken hebben ingedeeld in quad-bins, kunnen we met behulp van de quad-bins de intersectie berekenen. We gaan eerst van de veronderstelling uit dat alle vierhoeken in de quad-bin, van verschillende kegels komen. Voor iedere vierhoek houden we bij vanuit welke kegel hij geprojecteerd is geweest.

We zullen eerst het geval behandelen waarbij we twee overlappende vierhoeken hebben. Daarna zullen we dit uitbreiden naar n onderling overlappende vierhoeken, met $n > 2$. Gevolgd door het geval waarbij we n vierhoeken hebben die niet noodzakelijk overlappen.

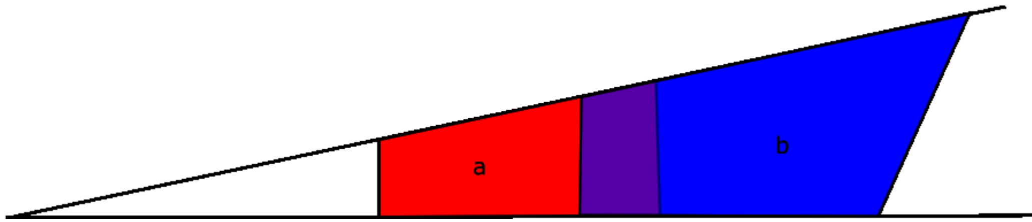
Uiteindelijk zullen we bespreken wat te doen als er meerdere vierhoeken vanuit één kegel geprojecteerd worden.

Twee overlappende vierhoeken

De vierhoeken in een quad-bin hebben we opgeslagen door voor ieder punt de afstand tot de oorsprong van de face op te slaan. Een vierhoek is dus in feite een 4-tupel $(r^1_{start}, r^1_{end}, r^2_{start}, r^2_{end})$, waarbij r^1_{start} de afstand aangeeft waar de vierhoek begint op de eerste straal van de quad-bin, en waarbij r^1_{end} de afstand aangeeft waar de vierhoek eindigt. De andere twee waarden r^2_{start} en r^2_{end} zijn het begin en het einde op de tweede straal.

Stel nu dat we twee vierhoeken a en b (zie figuur 4.10) hebben die elkaar overlappen, op beide stralen. Dan verkrijgen we een nieuwe vierhoek c door voor iedere as het maximum te nemen van de start waarden van a en b, en het minimum van de eindwaarden :

$$\begin{aligned} a &= \left(a^1_s, a^1_e, a^2_s, a^2_e\right) \\ b &= \left(b^1_s, b^1_e, b^2_s, b^2_e\right) \end{aligned}$$



Figuur 4.10: De vierhoeken a (rood) en b (blauw) die elkaar overlappen op beide stralen.

De intersectie van a en b is :

$$c = \left(max\left(a_s^1, b_s^1\right), min\left(a_e^1, b_e^1\right), max\left(a_s^2, b_s^2\right), min\left(a_e^2, b_e^2\right)\right)$$

Het geval dat a en b niet overlappen op beide stralen, maar slechts op één straal, kan niet meer voorkomen (zie sectie 4.4.3).

Meer dan twee overlappende vierhoeken

Nu dat we de intersectie kunnen berekenen van twee overlappende vierhoeken, kunnen we dit uitbreiden tot meerdere onderling overlappende veelhoeken. Stel dat we een reeks van vierhoeken $v_1, v_2, \ldots, v_n$ hebben met n een positief natuurlijk getal. Al deze vierhoeken overlappen met elkaar, dus $\forall i, j$ met $i \neq j$: v_i overlapt met v_j.

Dit probleem kunnen we oplossen door herhaaldelijk het algoritme van de vorige sectie toe te passen. We maken een nieuwe vierhoek s waarin we de intersectie zullen bijhouden. Eerst stellen we s gelijk aan v_1, daarna intersecteren we s met v_2 en de slaan de intersectie terug op in s. We blijven dit herhalen totdat we v_n gehad hebben en de volledige intersectie berekend hebben.

Het algoritme berekent telkens het maximum van de startwaarden en het minimum van de eindwaarden van twee vierhoeken. Als we dit herhaaldelijk toepassen houden we dus uiteindelijk het maximum van de startwaarden van $v_1, v_2, \ldots, v_n$ en het minimum van de eindwaarden over. Dit minimum en maximum kunnen we uiteraard rechtstreeks berekenen, maar aangezien beide methoden even snel zijn (beide zijn $O(n)$), maakt het niet veel uit welke methode we kiezen.

Niet overlappende vierhoeken

Het is mogelijk dat sommige vierhoeken niet overlappen. Als dit gebeurt dan kunnen we onmogelijk de intersectie berekenen van al de vierhoeken, we voegen dan ook geen polygoon toe aan de mesh. Tijdens de implementatie bleek dit de beste oplossing te zijn.

Meerdere vierhoeken vanuit dezelfde kegel

Stel we hebben n verzamelingen van vierhoeken $V_1, V_2, \ldots, V_n$. Voor iedere set geldt dat de vierhoeken die in de set zitten vanuit dezelfde kegel geprojecteerd werden. We moeten nu de intersectie berekenen van al deze verzamelingen. Dit is echter niet triviaal.

Dit probleem hebben we opgelost door voor iedere verzameling een vierhoek te maken die alle vierhoeken in de verzameling omvat. Stel dat in de verzameling V_i de vierhoeken a en b zitten, de vierhoek m_i berekenen we dan op de volgende manier :

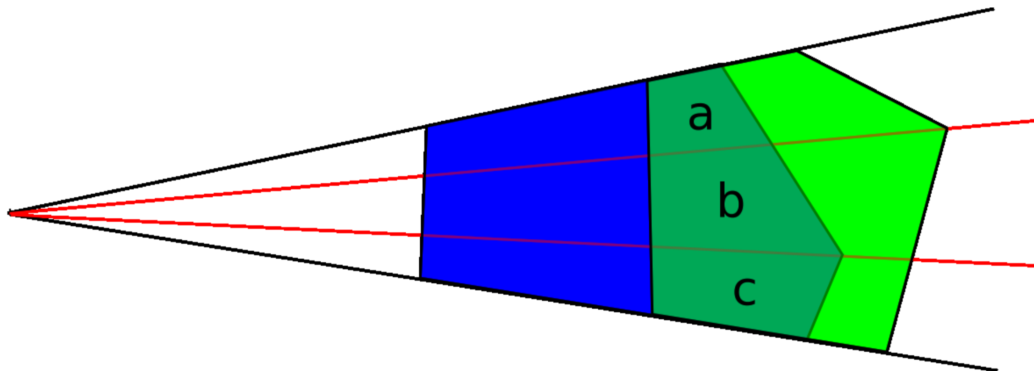
$$
\begin{aligned}
a &= \left(a_s^1, a_e^1, a_s^2, a_e^2\right) \\
b &= \left(b_s^1, b_e^1, b_s^2, b_e^2\right) \\
m_i &= \left(min\left(a_s^1, b_s^1\right), max\left(a_e^1, b_e^1\right), min\left(a_s^2, b_s^2\right), max\left(a_e^2, b_e^2\right)\right)
\end{aligned}
$$

We nemen in feite het minimum van de startwaarden en het maximum van de eindwaarden. De uitbreiding naar meer dan 2 vierhoeken is triviaal.

Als resultaat krijgen we dan n vierhoeken $m_1, m_2, \ldots, m_n$. Daarna intersecteerden we al deze vierhoeken zoals we beschreven hebben in de vorige drie secties, wat de vierhoek m tot gevolg heeft.

De vierhoek m is echter geen correct resultaat, aangezien deze vierhoek delen zal bevatten die zeker niet in het uiteindelijk resultaat zitten. Als we teruggrijpen naar de vierhoeken a en b van hierboven, dan weten we dat het stuk tussen a en b zeker niet, deel mag uitmaken van m. Deze tussenstukken zullen we dan ook uit m wegsnijden.

Voor iedere verzameling V_i berekenen we de tussenstukken, en snijden we deze tussenstukken weg uit de vierhoek m. Dit zal één of meerdere vierhoeken



Figuur 4.11: De vierhoeken a,b en c die het resultaat zijn van de intersectie van de blauwe en de groene polygoon, kunnen samengevoegd worden tot één polygoon. De rode lijnen zijn de grenzen van de quad-bins.

opleveren voor de mesh. Als m volledig in een tussenstuk ligt, dan kunnen we uiteraard geen vierhoeken toevoegen aan de mesh.

4.5 Mesh constructie

In iedere quad-bin hebben we nu één of meerdere vierhoeken berekend. We zouden deze vierhoeken rechtstreeks aan de mesh kunnen toevoegen, maar aangezien dit vele kleine vierhoeken zou opleveren en omdat de vierhoeken in de quad-bins al het resultaat zijn van het opdelen van een polygoon over de quad-bins kunnen we deze vierhoeken nog samenvoegen tot enkele polygonen. In figuur 4.11 kunnen we bijvoorbeeld de vierhoeken a,b en c samenvoegen tot één enkele polygoon.

We doen dit door een vierhoek te nemen in de eerste quad-bin, daarna gaan we in de volgende quad-bin kijken of er een vierhoek is die aansluit. Als we er één vinden dan voegen we deze toe aan de veelhoek. Dit blijven we herhalen totdat we aan de laatste quad-bin komen, of totdat we geen aansluitende vierhoek meer kunnen vinden. Al de vierhoeken die we toevoegen aan de polygoon worden verwijderd uit hun quad-bin. We herhalen deze procedure totdat er geen enkele vierhoek meer te vinden is in de quad-bins.

Nadat we de vierhoeken samengevoegd hebben tot polygonen moeten we nog de 3D coördinaten berekenen van de vertices van iedere polygoon. De punten van de vierhoeken hebben we immers opgeslagen als afstanden tot het

camerastandpunt over de straal die de grens vormt van de quad-bin waarin de vierhoek ligt. De 3D coördinaten van ieder punt kunnen we makkelijk berekenen door de afstand in te vullen in de vergelijking van de straal waarop het punt ligt.

4.6 Texture van het model

Nu dat we een mesh hebben berekend voor het model, moeten we deze mesh nog van een texture voorzien. Dit doen we door de originele foto's die als input zijn gebruikt, te gebruiken als texture. Het algoritme is gebaseerd op het Unstructured Lumigraph Rendering algoritme.

De invloed van een foto op een vertex van een polygoon is afhankelijk van de hoek tussen de kijkvector (van het huidig camerastandpunt naar de vertex) en de kijk richting van de foto (camerastandpunt van de foto naar de vertex). De foto's met de kleinste hoeken zullen meer invloed krijgen op het uiteindelijke resultaat.

De $k-1$ foto's die de kleinste hoeken hebben, zullen een blending gewicht krijgen. Dit gewicht is groter naarmate de hoek dichter bij 0 ligt. De som van de $k-1$ gewichten is samen 1.

We zullen eerst bespreken hoe we de textuur coördinaten van een vertex berekenen. Gevolgd door het bepalen van de zichtbaarheid van een polygoon. Daarna gaan we verder met het berekenen van de blending gewichten. Uiteindelijk sluiten we af met het renderen.

4.6.1 Textuur coördinaten

Als we N foto's hebben, dan heeft iedere vertex van iedere polygoon N textuur coördinaten, voor iedere foto één. We kunnen deze textuur coördinaten makkelijk berekenen door de vertex terug te projecteren op de originele foto's. Dit levert ons de pixel coördinaat (x, y) op. De uiteindelijke textuur coördinaat bekomen we dan door x te delen door de breedte van de foto en y te delen door de hoogte.

Deze methode heeft wel het nadeel dat als de projectie niet accuraat genoeg is, dat de textuur coördinaten net buiten het object kunnen vallen, waar-

door kleine delen van de achtergrond zullen te zien zijn op het uiteindelijke resultaat.

4.6.2 Zichtbaarheid van een polygoon

Stel dat we willen bepalen of de polygonen die liggen op de face f zichtbaar zijn vanuit foto k. De face f wordt gevormd door de edge i in het silhouet van foto j. Al de polygonen die op f liggen zijn zichtbaar vanuit k als i zichtbaar is vanuit de epipool p_0 (de projectie van camera k of foto j).

Dit reduceert het probleem van 3D naar 2D. Bovendien kunnen we dit makkelijk bepalen met behulp van de edge-bin datastructuur. Aangezien de edges in iedere edge-bin gesorteerd zijn op afstand tot de epipool, is alleen de eerste edge in de edge-bin zichtbaar. Als i dus als eerste te vinden is in een edge-bin, dan zijn alle polygonen op f zichtbaar vanuit k. Tijdens het intersecteren van de kegels kunnen we dus gemakkelijk de zichtbaarheids informatie berekenen.

Het is mogelijk dan een edge slechts gedeeltelijk zichtbaar is, we behandelen dit geval door de edge (en dus ook de face) op te splitsen in zichtbare en onzichtbare delen.

De zichtbaarheids berekening die we hier beschreven hebben, is conservatief. We labelen dus nooit een onzichtbare edge, zichtbaar. Het is echter te conservatief, vooral voor objecten die silhouetten hebben met gaten erin. De theepot in hoofdstuk 5.2.1 is een duidelijk voorbeeld hiervan.

4.6.3 De blending gewichten

De blending gewichten berekenen we voor iedere vertex van iedere polygoon. Stel we hebben de vertex v van polygoon p, en de huidige camera is gelegen op positie c. Stel dat de cameras van waaruit p zichtbaar is, $c_1, ..., c_n$ zijn. Voor iedere c_i met $i = 1, ..., n$, berekenen we dan de hoek tussen de rechte $\{v, c\}$ en $\{v, c_i\}$. Deze hoek zullen we h_i noemen.

We nemen dan de k kleinste hoeken. Deze hoeken mappen we dan naar een getal tussen 0 en 1 dat groter zal zijn naarmate de hoek kleiner is. Hiervoor gebruiken we de functie $gewicht\,(h) = 1 - h/h_k$. Dit levert ons $k - 1$ gewichten op die niet 0 zijn. We hernormalizeren deze gewichten nog, voor de

uiteindelijke gewichten.

In [10] wordt voor k het getal 3 gebruikt, wat dus 2 gewichten oplevert per vertex.

4.6.4 Renderen

Met behulp van de blending gewichten en de originele foto's als texture zullen we nu de mesh renderen. We doen dit met behulp van multitexturing en de volgende pixel shader :

```
uniform sampler2D texture0;
uniform sampler2D texture1;
uniform float alpha0;
uniform float alpha1;

void main()
{
  vec4 texval0 = texture2D(texture0,
    vec2(gl_TexCoord[0]));
  vec4 texval1 = texture2D(texture1,
    vec2(gl_TexCoord[1]));

  gl_FragColor = alpha0*texval0 + alpha1*texval1;
}
```

In de bovenstaande pixel shader, zijn $alpha0$ en $alpha1$ de twee blending gewichten, $texval0$ en $texval1$ zijn de kleuren die we bekomen van de originele foto's. De uiteindelijke kleur $gl_FragColor$ is dan de gewogen som van $texval0$ en $texval1$. De waarden $alpha0$,$alpha1$,$texture0$ en $texture1$ worden voor iedere polygoon aangepast.

Dit is niet de enige manier om de mesh te renderen, in [10] maakt men gebruikt van een multipass algoritme, dat de resultaten in de framebuffer accumuleert. Dit algoritme heeft echter het nadeel dat iedere polygoon twee keer dient gerendert te worden. Bovendien is deze pixel shader makkelijker te implementeren.

Dit algoritme vereist wel dat de graphics hardware de OpenGL shading language ondersteunt. Alle recente kaarten ondersteunen dit, voor oudere kaarten

kan eventueel het multipass rendering algoritme nog soelaas brengen.

4.7 Besluit

Na al deze berekeningen zouden we een mesh moeten hebben die min of meer de vorm van het object heeft. Aangezien het algoritme werkt met het silhouet van het object zullen zaken die niet zichtbaar zijn in het silhouet, niet in het uiteindelijke resultaat zitten. Een voorbeeld hiervan is een deuk in het object, de deuk zal niet in de resulterende mesh zitten.

Hoofdstuk 5

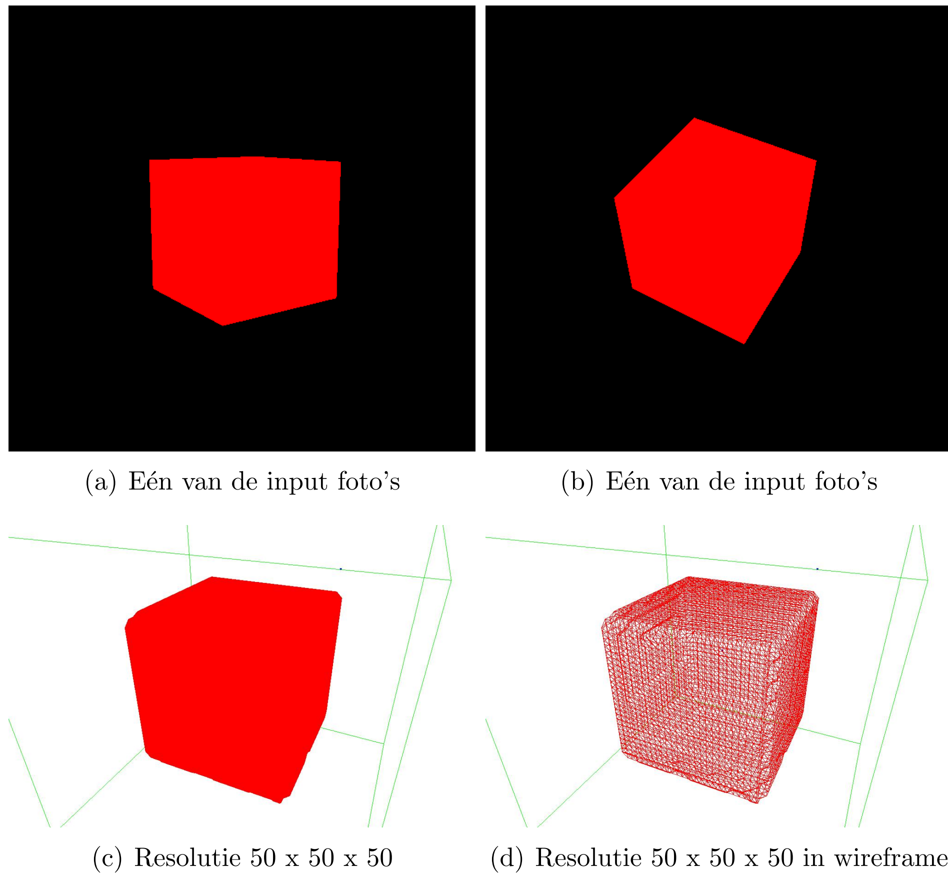
Resultaten

Dit hoofdstuk geeft een overzicht van de resultaten die bereikt zijn met beide algoritmen. Beide algoritmes hebben we uitgebreid getest met zowel reële als virtuele objecten. We zullen eerst de resultaten van het space carving algoritme bespreken gevolgd door de resultaten van het image based visual hull algoritme.

Voor ieder algoritme hebben we gebruikt gemaakt van echte en virtuele data. De echte data werd gemaakt met behulp van de 3D scanner. De virtuele data werd met behulp van een programma gegenereerd. Dit programma tekent een object uit en laat de gebruiker toe om de camera vrij rond het object te roteren met behulp van de muis. De gebruiker kan zo een aantal camera standpunten kiezen om een foto te nemen. Als de gebruiker tevreden is kan deze de foto's en de camera data (positie, rotatie en projectie voor ieder standpunt) opslaan in een directory naar keuze. De directory zal de foto's bevatten en een xml bestand met de camera data.

5.1 Space Carving

We zullen eerst de resultaten bespreken op computer gegenereerde objecten, gevolgd door de resultaten op echte objecten.



(a) Eén van de input foto's (b) Eén van de input foto's

(c) Resolutie 50 x 50 x 50 (d) Resolutie 50 x 50 x 50 in wireframe

Figuur 5.1: Resultaten voor een kubus

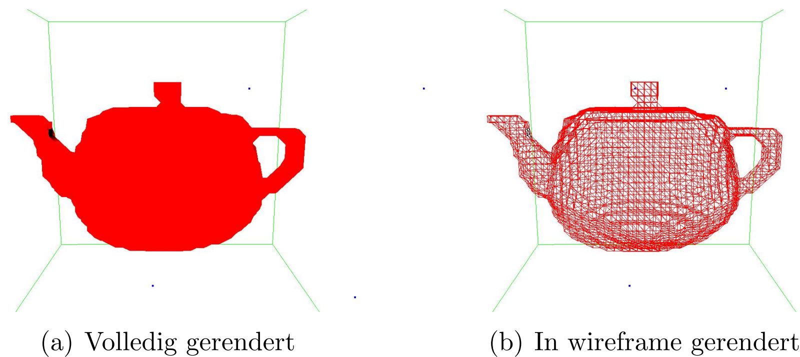
5.1.1 Computer gegenereerde objecten

Het eerste object waarop we het space carving algoritme hebben uitgetest, is een kubus. De kubus heeft een rode kleur en er is geen belichting in de scène aanwezig. We hebben de meest ideale omstandigheden gecreëerd voor het algoritme.

Van de kubus hebben we 26 foto's genomen. Een overzicht van de resultaten is te vinden in figuur 5.1. De kubus is duidelijk herkenbaar bij een vrij lage resolutie van 50 bij 50 bij 50. We gebruikten voor de consistentie drempel de waarde 0.01.

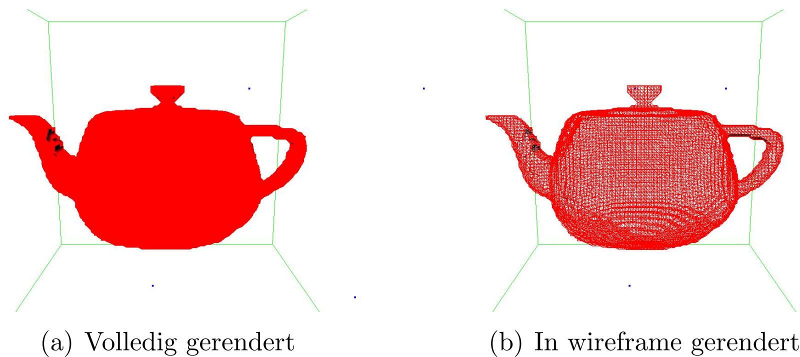
Natuurlijk is een kubus een heel simpel object en daarom hebben we een meer gecompliceerder object gekozen bij de tweede test : een theepot. De theepot is net zoals de kubus gerenderd zonder belichting. We hebben 23

Figuur 5.2: Eén van de input foto's van de theepot



(a) Volledig gerendert (b) In wireframe gerendert

Figuur 5.3: Resultaten voor de theepot op een resolutie van 50 bij 50 bij 50



(a) Volledig gerendert (b) In wireframe gerendert

Figuur 5.4: Resultaten voor de theepot op een resolutie van 100 bij 100 bij 100

foto's genomen van de theepot, één van de foto's is te zien in figuur 5.2.

Eerst hebben we gescanned op een grid resolutie van 50 bij 50 bij 50 met consistentie drempel van 0.01. Twee screenshots zijn hiervan te zien in figuur 5.3. De vorm van de theepot is overduidelijk aanwezig, alhoewel hij nog vrij ruw is. Dit ruw zijn is te wijten aan de lage resolutie.

Een tweede scan op een resolutie van 100 bij 100 bij 100, geeft al veel betere resultaten. De screenshots zijn te vinden in figuur 5.4. We gebruiken ook hier een consistentie drempel van 0.01.

5.1.2 Reële objecten

Als eerste object hebben we een pritt roller gekozen. Het echte object samen met de resultaten op een resolutie van 100 bij 100 bij 100 is te zien in figuur 5.5. We hebben 0.15 als consistentie drempel gebruikt. De vorm van de pritt

(a) Het echte object

(b) Zijkant van het gegenereerde object

(c) Het object van bovenaf gezien

(d) Andere zijkant van het object

Figuur 5.5: Pritt roller met resultaten op een resolutie van 100 bij 100 bij 100

roller zit er goed in, de tekst op de zijkant is echter een beetje wazig. Dit wordt vooral veroorzaakt door de vrij lage resolutie.

In de figuren 5.6 en 5.7, zijn resultaten te zien op hogere resoluties. De grote tekst aan de zijkant is al veel duidelijker, maar de kleine tekst blijft wazig. Hoe hoger de resolutie, hoe beter de kleine details zichtbaar zullen zijn.

Bij al de vorige resultaten hebben we telkens puntprojectie gebruikt, als we spriteprojectie gebruiken op een resolutie van 200 bij 200 bij 200, krijgen we iets betere resultaten. De resultaten zijn te zien in figuur 5.8.

Het tweede object is een kaktus. Net zoals de pritt roller gebruiken we ook

Figuur 5.6: Resultaten op 150 bij 150 bij 150

Figuur 5.7: Resultaten op 200 bij 200 bij 200

Figuur 5.8: Resultaten op 200 bij 200 bij 200 (spriteprojectie)

hier een consistentie drempel van 0.15. Het echte object en de resultaten op een resolutie van 100 bij 100 bij 140 zijn te zien in figuur 5.9. De vorm zit er goed in, en de tekeningen op de zijkant van het potje zijn ook goed te zien, alhoewel een beetje wazig. De wazigheid komt vooral door de lage resolutie.

Als we de resolutie verhogen, dan krijgen we betere resultaten. In figuren 5.10 en 5.11 is dit te zien. De tekeningen op de zijkant van het potje zijn veel beter.

Al de resultaten van de kaktus tot nu toe werden telkens gegenereerd met puntprojectie. In figuur 5.12 hebben we spriteprojectie gebruikt.

(a) Het echte object

(b) Zijkant van het object

(c) Andere kant van het object

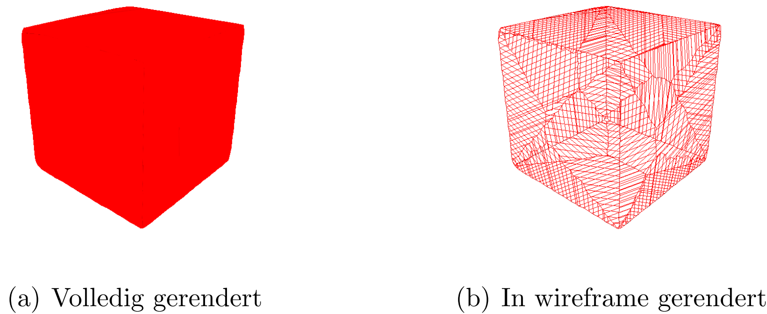
(d) Nog een andere kant

Figuur 5.9: Kaktus met resultaten op een resolutie van 100 bij 100 bij 140

Figuur 5.10: Kaktus met resolutie 150 bij 150 bij 175

Figuur 5.11: Kaktus met resolutie 200 bij 200 bij 225

Figuur 5.12: Kaktus met resolutie 200 bij 200 bij 225 (spriteprojectie)



(a) Volledig gerendert (b) In wireframe gerendert

Figuur 5.13: Resultaten voor de kubus

5.2 Image based visual hulls

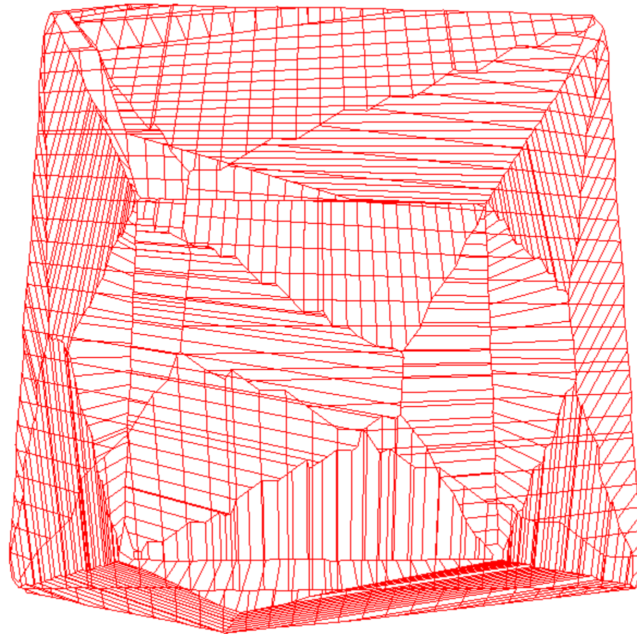
Net zoals het vorige hoofdstuk gaan we eerst de resultaten op computer gegenereerde objecten bespreken, gevolgd door de resultaten op echte objecten.

5.2.1 Computer gegenereerde objecten

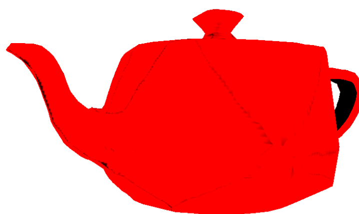
Zoals in hoofdstuk 5.1.1 hebben we ons algoritme eerst uitgetest op een kubus, en een theepot. De resultaten zijn te vinden in figuur 5.13 en 5.15. Van beide objecten werden 6 foto's genomen.

De resultaten voor de kubus laten een mesh zien, die het object goed benadert. Aan de onderkant van de kubus is te zien dat de resulterende mesh meer bevat dan het object. Dit is duidelijker te zien in figuur 5.14. Aan de kubus is ook mooi te zien hoe de polygonen van verschillende kegels netjes op elkaar aansluiten.

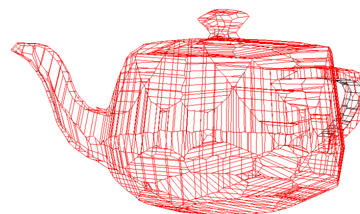
In figuur 5.15 kunnen we zien dat het algoritme een goede benadering produceert van een theepot. De kleine zwarte randjes op sommige polygonen worden veroorzaakt door onnauwkeurigheden bij het projecteren tijdens het berekenen van de textuur coördinaten. De textuur coördinaten vallen net buiten het object, waardoor we een kleine zwarte rand krijgen. De binnenkant van de oor die volledig zwart is, wordt veroorzaakt door het feit dat de zichtbaarheids bepaling te conservatief is.



Figuur 5.14: Beperking van het image-based visual hull algoritme



(a) Volledig gerendert



(b) In wireframe gerendert

Figuur 5.15: Resultaten voor de theepot

5.2.2 Reële objecten

Net zoals bij het space carving algoritme hebben we als eerste object een pritt roller gekozen. Het object is te zien in figuur 5.16. Van de pritt roller hebben we 8 foto's genomen. Zoals de resultaten laten zien produceert het algoritme een realistisch uitziend object. Een deel van de schaduw die veroorzaakt wordt door het object is ook nog te zien. Het schaduw gedeelte is uiteraard verschillend van de achtergrond, waardoor dit ook nog wordt meegenomen in de uiteindelijke mesh.

Als tweede object kozen we voor een kaktus plantje. Dit is te zien in figuur 5.17. Ook van de kaktus hebben we 8 foto's genomen. De witte lijnen in de vierde foto van figuur 5.17, zijn het gevolg van de onnauwkeurigheden bij het berekenen van de textuur coördinaten.

(a) Het echte object

(b) Zijkant van het gegenereerde object

(c) Andere zijkant van het object

(d) Het object van bovenaf gezien

Figuur 5.16: Pritt roller met resultaten

(a) Het echte object

(b) Het gegenereerde object

(c) Andere kant van het object

(d) Nog een andere kant

Figuur 5.17: Kaktus met resultaten

Hoofdstuk 6

De 3D scanner

In dit hoofdstuk zullen we de 3D-scanner bespreken. Deze scanner werd gebouwd om de algortimen die we in hoofdstukken 3 en 4 hebben beschreven, te testen.

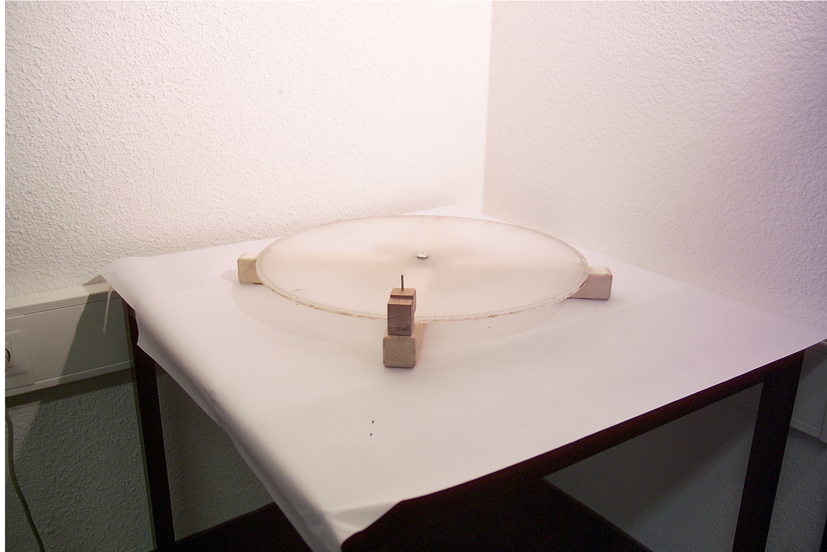
De scanner is niet volautomatisch, je kan er dus geen object in plaatsen, op een knop drukken en wachten tot de scanner een 3D model produceert. Alhoewel dit niet onmogelijk is om te bouwen, is dit niet de bedoeling van deze thesis.

We zullen eerst bespreken hoe de scanner opgebouwd is, om daarna de werking ervan te bespreken.

6.1 Constructie

De scanner bestaat uit ronde schijf waarop het object dient geplaatst te worden. De schijf wordt handmatig gedraaid. Op de schijf staan de graden aangegeven, zodanig dat we weten met hoeveel graden het object geroteerd is. Het roterend platform is te zien in figuur 6.1.

Op enige afstand van de schijf plaatsen we een digitale camera op een statief. De tafel waarop het roterend platform staat omringen we met twee witte tafels die op hun zijkant zijn gezet. Achter de tafels zetten we twee spots die niet rechtstreeks op het object schijnen. Deze hele opstelling heeft tot doel om harde schaduwen te vermijden. Een foto van de opstelling is te zien in



Figuur 6.1: Het roterend platform

figuur 6.2.

6.2 Werking

Het eerste wat moet gedaan worden, is de camera calibreren met behulp van een dambord patroon en de camera calibratie toolbox van Matlab.

Na de calibratie, plaatsen we het object in het midden op de schijf. Eerst nemen we een reeks foto's zonder object, zodanig dat we voor iedere foto, een achtergrond foto hebben. De achtergrond foto's dienen vanuit dezelfde hoeken te worden genomen als de foto's van het te scannen object.

Daarna plaatsen we het object op de schijf. We nemen telkens een foto en draaien de schijf een aantal graden. Hoeveel graden hangt af van hoeveel foto's we willen nemen, 30 foto's betekent om de 12 graden een foto nemen.

Via een Matlab script dat de calibratie data als input neemt, berekenen we dan de posities en rotaties van iedere individuele foto. Dit Matlab script zal ook een XML bestand genereren dat zal gebruikt worden als input tesamen met de foto's, voor de twee algortimen.

Voordat we het 3D model genereren moeten alle foto's eerst van lens distortie

Figuur 6.2: De opstelling

ontdaan worden. Dit doen we met behulp van een klein programma dat gebruik maakt van OpenCV. Daarna voeren we een algoritme uit op de foto's en het XML bestand. Dit moet dan een 3D-model als resultaat opleveren.

Hoofdstuk 7

Besluit

Het probleem dat we in deze thesis bestudeerd hebben is het genereren van 3D objecten uit foto's. We hebben dit probleem trachten op te lossen door twee algoritmen te implementeren.

De twee algoritmen hebben we uitgebreid getest op zowel echte als virtuele data. Om de echte data te genereren hebben we een test opstelling gebouwd die bestaat uit een roterend platform waarop we een object plaatsen. We gebruikten een camera om foto's te nemen van het object, door het roterend platform te draaien konden we foto's nemen vanuit verschillende hoeken.

7.1 Space Carving

Het eerste algoritme is space carving. Space carving maakt gebruik van een voxel volume rond het object. Voxels die niet dezelfde kleur hebben vanuit verschillende camerastandpunten worden uit het volume verwijderd. Het object wordt uit het volume gebeeldhouwd.

Omdat het algoritme voxels elimineert door de kleur van de voxel te vergelijken vanuit verschillende camerastandpunten, kan space carving niet overweg met speculaire oppervlakten. Op een speculaire oppervlakte is immers niet alleen de kleur van de oppervlakte te zien, maar ook de reflectie van andere oppervlakten. En deze reflectie verschilt voor ieder camerastandpunt, waardoor de kleur verschillend is, wat leidt tot de verkeerde eliminatie van

voxels.

Dit is een groot nadeel voor de praktische toepasbaarheid van space carving. Er bestaan echter technieken waardoor dit nadeel kan weggewerkt worden.

7.1.1 Verbeteringen

Het space carving algoritme kan mogelijk verbeterd worden door de techniek beschreven in [15]. Ieder beeld is het product van de eigenschappen van een scène. Twee van de meest belangrijke eigenschappen van een scène zijn shading en reflectie. Met shading bedoelen we de interactie van oppervlakten in de scène met de belichting. De reflectie van een scène is beschrijft hoe ieder punt van de scène, licht reflecteert (welke kleur dus).

Het algoritme in [15] berekend voor een beeld nu de reflectie en shading in ieder punt. Dit levert als resultaat een intrinsiek beeld op voor de shading en een intrinsiek beeld voor de reflectie. Een intrinsiek beeld is een beeld dat één intrinsieke eigenschap van een beeld bevat.

Als we nu space carving toepassen op de reflectie beelden in plaats van de echte beelden, zou het mogelijk moeten zijn dat space carving ook werkt op speculaire oppervlakten.

7.1.2 Conclusie

Zoals de resultaten in hoofdstuk 5 laten zien, het space carving algoritme kan onder omstandigheden die rekening houden met de beperkingen van het algoritme, realistische resultaten opleveren. Dit zijn echter omstandigheden die niet al te veel voorkomen in de realiteit.

Uit de resultaten met echte objecten, kunnen we ook concluderen dat voor goede resultaten te krijgen, we hoge resoluties nodig hebben. Dit komt de uitvoeringstijd echter niet ten goede.

7.2 Image Based Visual Hull

Als tweede algoritme hebben we het image-based visual hull algoritme gekozen, dit algoritme maakt gebruik van het silhouet van het object in de foto's.

Het silhouet van het object en de camera, vormen samen een kegel. Wat dit algoritme in feite doet, is de intersectie van al de kegels berekenen. Dit is een 3D operatie, maar deze kan volledig in 2D uitgevoerd worden.

Als we een kegel met een andere kegel willen intersecteren, projecteren we ieder face van de kegel op het cameravlak van de andere kegel. We berekenen de intersectie tussen de face en het silhouet op het cameravlak. Deze intersectie projecteren we dan terug op het vlak van de face.

Nadat we zo paarsgewijs alle kegels met elkaar geïntersecteerd hebben, hebben we op iedere face een aantal polygonen zitten. Voor iedere face moeten we nu nog de doorsnede van alle polygonen berekenen. De uiteindelijke mesh renderen we met de orignele foto's als textures.

7.2.1 Conclusie

Het algoritme produceert goede resultaten, en heeft een stuk minder beperkingen dan space carving. De resulterende mesh is uiterst geschikt voor het renderen op moderne graphics hardware.

De standpunt afhankelijke texturing is echter het grootste nadeel. Het is waarschijnlijk niet onmogelijk om een algoritme te bedenken dat onafhankelijk is van de huidige positie van de camera.

Bibliografie

[1] John Amanatides and Andrew Woo. A fast voxel traversal algorithm for ray tracing. http://www.cs.yorku.ca/~amana/research/grid.pdf, 1987.

[2] Jean-Yves Bougue. Camera calibration toolbox for matlab. http://www.vision.caltech.edu/bouguetj/calib_doc/.

[3] Chris Buehler, Michael Bosse, Leonard McMillan, Steven Gortler, and Michael Cohen. Unstructured lumigraph rendering. http://graphics.csail.mit.edu/pubs/siggraph2001_ulr.pdf, 2001.

[4] Sven Charleer. 3d objecten uit foto's. http://users.pandora.be/sickb0y/thesis/files/thesis_svencharleer_2003.pdf, 2003.

[5] Intel corporation. Intel opencv library. http://www.intel.com/research/mrl/research/opencv/.

[6] Jean Sbastien Franco and Edmond Boyer. Exact polyhedral visual hulls. http://www.inrialpes.fr/movi/people/Franco/Papers/franco_boyer_bmvc03.pdf, 2003.

[7] Kiriakos N. Kutulakos and Steven M. Seitz. A theory of shape by space carving. http://www-2.cs.cmu.edu/~seitz/course/SIGG99/papers/kutu98.pdf, 1998.

[8] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. In *Computer Graphics (Proceedings of SIGGRAPH '87)*, pages 163–169, 1987.

[9] Wojciech Matusik. Image-based visual hulls. http://www.cs.ualberta.ca/~vis/papers/geometry/silhouette/matusikthesis.pdf, 2001.

[10] Wojciech Matusik, Chris Buehler, and Leonard McMillan. Polyhedral visual hulls for real-time rendering. http://graphics.lcs.mit.edu/~wojciech/vh/pvh_egrw2001.pdf, 2001.

[11] Wojciech Matusik, Chris Buehler, Ramesh Rashkar, Steven J. Gortler, and Leonard McMillan. Image-based visual hulls. http://graphics.lcs.mit.edu/~wojciech/vh/IBVH2000.pdf, 2000.

[12] Jonathan Roberts. Marching squares - an overview. http://www.cs.kent.ac.uk/people/staff/jcr/vova/WebTutorial/squares/overview/.

[13] Steven M. Seitz and Charles R. Dyer. Photorealistic scene reconstruction by voxel coloring. http://www.cs.cmu.edu/~seitz/course/SIGG99/papers/seitz97.pdf, 1997.

[14] James Sharman. The marching cubes algorithm. http://www.exaflop.org/docs/marchcubes/ind.html.

[15] Marshall F Tappen, William T Freeman, and Edward H Adelson. Recovering intrinsic images from a single image, 2002.

[16] Zhengyou Zhang. A flexible new technique for camera calibration. http://research.microsoft.com/~zhang/calib/, 1998.