2020•2021
Faculteit Industriële Ingenieurswetenschappen
master in de industriële wetenschappen: energie

Masterthesis

Vulgraaddetectie van dynamische volumes in dieptebeelden: optimalisatie met OpenCL

PROMOTOR :
Prof. dr. ir. Eric DEMEESTER

PROMOTOR :
Dhr. Stijn DEBRUYCKERE

COPROMOTOR :
ir. Kim RUTTEN

Nick Hamers, Luca Lupo

Scriptie ingediend tot het behalen van de graad van master in de industriële wetenschappen: energie, afstudeerrichting automatisering

Gezamenlijke opleiding UHasselt en KU Leuven

2020•2021

Faculteit Industriële Ingenieurswetenschappen

master in de industriële wetenschappen: energie

Masterthesis

Vulgraaddetectie van dynamische volumes in dieptebeelden: optimalisatie met OpenCL

PROMOTOR :

Prof. dr. ir. Eric DEMEESTER

PROMOTOR :

Dhr. Stijn DEBRUYCKERE

COPROMOTOR :

ir. Kim RUTTEN

Nick Hamers, Luca Lupo

Scriptie ingediend tot het behalen van de graad van master in de industriële wetenschappen: energie, afstudeerrichting automatisering

UHASSELT KU LEUVEN

Woord vooraf

Voor het mogelijk maken van deze masterproef willen de auteurs de volgende personen bedanken: Stijn Debruyckere; promotor van Arkite NV, Kim Rutten; copromotor van Arkite NV, prof.dr.ir. Eric Demeester; promotor van UHasselt.

De masterproef is gekozen door Nick Hamers en Luca Lupo omwille van hun interesse in software development voor concrete toepassingen binnen het kader automatisering.

Computervisie is een interessant onderwerp omwille van de versatiliteit dat het biedt bij het automatiseren van taken. Goede computervisie programma's zijn krachtig, maar vaak complex voor diegene die hiermee weinig ervaring hebben. De auteurs zagen deze masterproef als een opportuniteit om kennis te maken met computervisie programmatie en hiervan ervaring op te doen in de vorm van een betekenisvolle en concrete toepassing. De mogelijkheid dat het project werkelijk gebruikt wordt door Arkite, ter verbetering van haar product, vormde een additionele bron van motivatie.

Inhoudsopgave

Lijst van tabellen

Lijst van figuren

Verklarende woordenlijst

HIM	"Human interface mate", een systeem ontwikkeld door Arkite NV voor het bijstaan van operatoren binnen hun taken.
CPU	Central processing unit, een component die algemeen bewerkingen doet binnen een computer.
GPU	Graphical processing unit, een component die toestaat om grote aantallen stukken data te verwerken in parallel.
Kernel	Code geschreven ter uitvoering van de GPU in parallel.
Libraries	Reeds bestaande, extern geschreven code die ingeladen kan worden in projecten ter gebruik.
Overhead	Bijkomende bewerkingen en vergelijkingen die uitgevoerd worden om een gewenste actie uit te voeren.
Assenstelsel	Oorsprong van een enkel- of multidimensionale ruimte. Een Punt worden gerelateerd aan een assenstelsel door de afstand tussen dat punt en de oorsrong van het assenstelsel in alle dimensies. Ook een vlak/ruimte kan gerelateerd worden aan een assenstelsel met het gekende oriëntatieverschil tussen het assenstelsel en het vlak/ruimte, gedefinieerd als verschil in rotatie rond alle dimensies.
Transformatiematrix	De wiskundige voorstelling van de positie van een punt ten opzichte van een assenstelsel. Deze bepaalt, bij een vlak/ruimte, ook de oriëntatie hiervan ten opzichte van een assenstelsel.
Wereldruimte	3D-ruimte van de echte wereld, gedefinieerd door een assenstelsel.
Beeldruimte	2D-vlak van het gecapteerde camerabeeld van een wereldruimte, gedefinieerd door een assenstelsel.
Augmented reality	Het tonen van relevante projecties en het meten van relevante grootheden in een fysieke omgeving door middel van een computer om die computer te laten interacteren met de omgeving.

Abstract

NL

Arkite NV produceert de Human Interface Mate, een toestel dat operatoren assisteert met taken door instructies te projecterenen en te detecteren of deze gevolgd worden. Detecties van operator-handelingen gebeuren met behulp van een dieptesensor (Microsoft Kinect V2). De vereiste voor het systeem is dat 500 volumes met 30 beelden per seconde gecontroleerd worden met betrouwbare nauwkeurigheid. Het huidige systeem haalt dit niet, dus is een nieuw systeem ontworpen in deze thesis.

Ter detectie van handelingen worden volumes, operator-gedefinieerd in de cartesiaanse werkomgeving, gevuld met een 3D-puntenwolk. De beelden, waargenomen door de dieptecamera, bestaan uit 2D-pixelcoördinaten met een bijkomend diepte-component. Elk punt van de puntenwolk is gelinkt met een pixel in het sensorbeeld. Indien de dieptewaarde van voldoende punten van een volume gelijk zijn aan die van de bijhorende pixel, is een detectie gemeten. Elk volume is toegewezen aan 1 operator-gedefinieerd assenstelsel in de werkomgeving. Om elk 3D-punt te linken aan bijbehorende 2D-pixel, beschikt elk assenstelsel over een coördinatentransformatie.

Het doel is bereikt door gebruik te maken van het OpenCL-platform in combinatie met de *'graphics processing unit'*, waardoor puntenwolken parallel verwerkt worden per volume. Met verdere optimalisaties van het detectie-proces is een systeem gecreëerd die voldoet aan de vooropgestelde voorwaarden tot aan 19664 punten per volume in de *'worst-case'* en tot vele malen meer in een *'good-case'*.

EN

Arkite NV produces the Human Interface Mate, a device that assists operators with tasks by projecting instructions and detecting if these are being followed. Detections of the operator's actions happen with the help of a depth-sensor (Microsoft Kinect V2). The prerequisite of the system is that 500 volumes are checked at 30 frames per second with reliable accuracy. The current system does not respect this requirement, so a new system is designed in this thesis.

To detect actions, volumes, operator-defined in the cartesian workspace, are filled with a 3D-point-cloud. Frames, captured by the depth camera, consist of 2D-pixel-coordinates with an additional depth-component. Each point of the point-cloud is linked with a pixel in the sensor's frame. If the depth-value of enough points of a volume are equal to that of the matching pixel, a detection is measured. Every volume is assigned to 1 operator-defined coordinate system in the workspace. To link each 3D-point with the matching pixel, every coordinate system has a coordinate transformation.

The goal is achieved by utilizing the OpenCL-platform in combination with the graphics processing unit, which processes point-clouds in parallel per volume. With further optimizations of the detection process, a system is created that complies with the requirements until a maximum of 19664 points per volume in the worst-case and until many times more in a good-case.

Hoofdstuk 1

Inleiding

Het centrale thema van deze thesis is de Human Interface Mate (HIM). Dit is een *'augmented reality'*-systeem (AR-systeem) dat productiewerknemers assisteert in *'real-time'*.

Specifiek wordt er gefocust op het softwareaspect van deze toepassing. Door gebruik te maken van gepaste algoritmes, die steeds in het belang van efficiëntie worden uitgedacht, is de HIM verbeterd. Het verbeteren van de toepassing bevordert niet enkel de wendbaarheid, maar verlaagt eveneens de instapdrempel voor nieuwe werknemers.

1.1 Situering

Deze masterproef is uitgevoerd bij Arkite NV. Arkite, met haar hoofdkantoor op C-Mine Crib te Genk, ontwikkelt en verkoopt de HIM. De HIM-technologie maakt gebruik van een 3D-sensor (Microsoft Kinect V2) en een projector om de arbeider in het werkgebied te begeleiden tijdens het productieproces. De projector geeft instructies en informatie, terwijl de 3D-sensor meet of deze instructies gevolgd worden. Dankzij de HIM kan, binnen productieprocessen, een werknemer complexe handelingen uitvoeren met evenveel zekerheid als een robot. Voor de analyse van de menselijke handelingen wordt 3D-sensor-data verwerkt en gecombineerd met de diverse communicatiestromen binnen het productieapparaat. De "virtual twin" van de werkpost, die ontstaat met behulp van de 3D-sensoren en projector, bestaat uit videobeelden van 6.5 miljoen pixels. De beelden dienen altijd verwerkt te worden met een minimale snelheid van 30 beelden per seconden. Arkite heeft deze technologie binnen verschillende bedrijven in de praktijk gebracht. Eén van deze bedrijven is Atlas Copco.

Atlas Copco was op zoek naar een systeem om haar werknemers te ondersteunen tijdens de complexe handmatige montage van verschillende apparaten. De HIM biedt een oplossing die de verkeerde montage van componenten, als gevolg van menselijke fouten, minimaliseert. Dit is een oplossing die in lijn ligt met de visie van Atlas Copco. De HIM voldeed aan de verwachtingen en verlaagde eveneens de instapdrempel voor nieuwe werknemers door haar methodologie. Het werd namelijk veel eenvoudiger om nieuwe werknemers aan het werk te krijgen door een aanzienlijke reductie in trainingstijd. Atlas Copco is gekozen als voorbeeld omdat het een goede integratie is van de HIM in de industrie.

De 3D-sensoren bestaan uit een infraroodcamera en een RGB-camera. De infraroodcamera stuurt een infrarood signaal uit en ontvangt deze vervolgens terug. De tijd tussen het uitsturen van

het signaal en het terug opvangen ervan is gekend als de *'Time-of-Flight'* (ToF). Deze ToF is de maat voor het bepalen van de afstand tussen de sensor en het waargenomen punt.

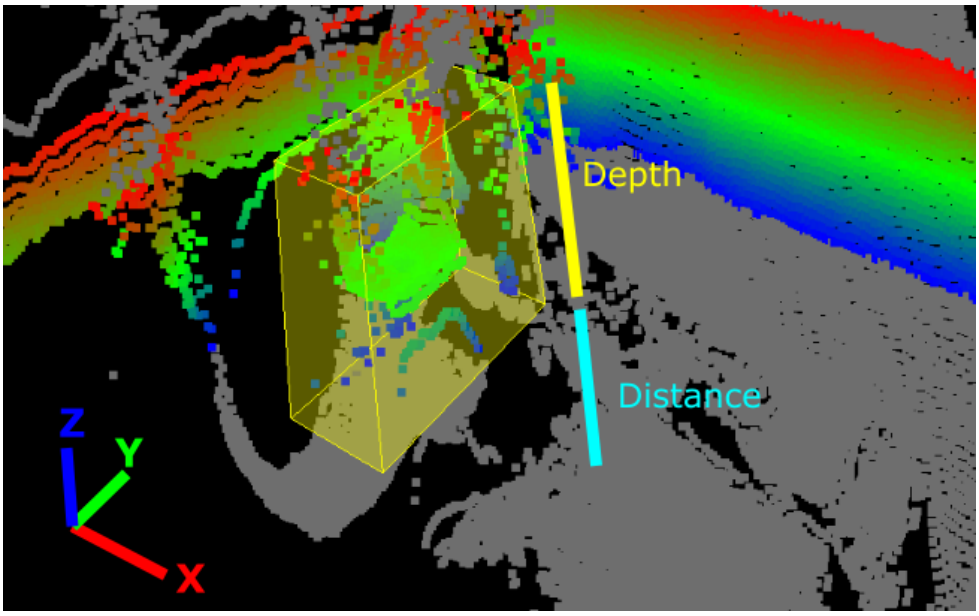
Door de samenwerking van de RGB-camera en de infraroodcamera van de sensor, wordt het volledige gezichtsveld van de sensoren waargenomen in dieptewaarde. Hierdoor ontstaat een 3D-puntenwolk van de oppervlakte van de werkomgeving. Op basis van deze puntenwolk worden interacties waargenomen. De variaties binnen deze puntenwolk zijn bijvoorbeeld het gevolg van een actie uitgevoerd door een werknemer of gereedschap.

Door op voorhand een volume te definiëren, dat bestaat uit een aantal punten in de driedimensionale ruimte, en de beschikbare sensordata, kan de evaluatie plaatsvinden van eventuele interacties met dit volume. Zodra voldoende punten van het sensorbeeld overeenkomen met punten binnen het gedefinieerd volume, is er sprake van een detectie. Dit gebeurt geheel in de beeldruimte van de sensor die bestaat uit pixels die elk een dieptewaarde bezitten. Hiervoor dient het gedefinieerd volume wel vertaald te worden naar pixels en diepte waarden in het sensorbeeld.

Een *'graphical user interface'* (GUI) is opgebouwd gebruikmakend van de programmeertaal C#. De werking van de hardware is in C++ geschreven, met de toevoeging van verschillende *'libraries'* (bibliotheken) zoals Qt en OpenCV. C++ is een programmeertaal die zeer snel code uitvoert, omdat de geschreven code, in tegenstelling tot vele andere programmeertalen, wordt omgezet in instructies voor de CPU met behulp van slechts 1 vertaalslag zonder de bijkomst van veel overhead.

1.2 Probleemstelling

Het originele systeem kent een aantal beperkingen. De voornaamste beperking is het gebrek aan transformaties tussen de tweedimensionale ruimte (het sensorbeeld) en de driedimensionale ruimte. Concreet wil dit zeggen dat de verschillende punten op het beeld altijd in beeldcoördinaten gedefinieerd zijn. Dit resulteert in het ontbreken van een verband tussen het beeld en de werkelijke positie in de driedimensionale ruimte van de werkplaats, zoals ervaren door de gebruiker. Ook creëert dit een vervorming van volumes naar de sensor toe, zoals te zien is op figuur 1.1.



Figuur 1.1: Beperkingen huidige volumeopbouw.

Verder wordt enkel gebruik gemaakt van de *'central processing unit'* (CPU) om alle nodige berekeningen te doen. De CPU moet gelijktijdig met het detectiealgoritme ook verschillende andere processen uitvoeren. Dit wil zeggen dat bij een hoge werkdruk niet genoeg beelden verwerkt kunnen worden om de operatoren te voorzien van betrouwbare feedback in *'real-time'*. Dit omwille van de beperkte beschikbaarheid en kracht van de CPU. Er is een *'graphical processing unit'* (GPU) geïntegreerd in de CPU, waar nog geen gebruik van wordt gemaakt.

Een eigenschap die de verdere uitbreiding van het systeem in de weg staat, is de statische aard van gedefinieerde volumes. De HIM functioneert optimaal in een omgeving waar er geen sprake is van verplaatsende volumes. Het uitbreiden van dit systeem naar een systeem met een meer dynamisch karakter zorgt ervoor dat het breder inzetbaar is. Dit kan klanten bijstaan in toepassingen waar bijvoorbeeld een transportband aan te pas komt waarop de desbetreffende volumes 'getransporteerd' worden.

1.3 Doelstellingen

Het eerste doel bestaat uit het introduceren van zowel een tweedimensionale beeldruimte als een driedimensionale wereldruimte. Beide ruimtes beschikken over een coördinatensysteem die iedere positie voorziet van een uniek punt. Binnen deze ruimtes dienen alle volumes en iedere punt gedefinieerd te zijn.

De punten gecapteerd door de 3D-sensor en de punten van de gedefinieerde volumes dienen vertaald te worden tussen beeldruimte en wereldruimte. De voorgedefinieerde volumes zullen, in tegenstelling tot het origineel systeem, in de wereldruimte gedefinieerd worden. Bij het definiëren van de volumes in de wereldruimte moet er steeds de keuze zijn tussen een absolute positie en een relatieve positie ten opzichte van een assenstelsel naar keuze. Figuur 1.2 toon het gelabelde absolute assenstelsel en enkele relatieve assenstelsels. De punten die de volumes vormen worden, eens getransformeerd naar de beeldruimte, vergeleken met het sensorbeeld in de beeldruimte om acties te kunnen waarnemen. Op basis van deze evaluatie dient een gepaste feedback gevormd

te worden.

Er moet echter een zekere betrouwbaarheid gegarandeerd worden, onafhankelijk van de werkdruk opgelegd aan het systeem. Het te behalen doel dicteert dat er simultaan 500 volumes, van ongespecificeerde grootte, gecontroleerd moeten worden, die op hun beurt verdeeld kunnen zijn over maximaal 10 verschillende relatieve assenstelsels. Er wordt gesproken van succes als dit doel bereikt is met een minimale snelheid van 30 beelden per seconde.

Verder dienen aanpassingen, verwijderingen en toevoegingen van volumes en assenstelsels in *'real-time'* te gebeuren zonder dat dit het systeem te zeer vertraagt. Dit is van groot belang bij de aanwezigheid van een groot aantal objecten, aangezien de rekenintensiteit evenredig toeneemt met het aantal objecten.

De extra uitbreiding op het origineel systeem bestaat, zoals al vermeld, uit het introduceren van dynamische assenstelsels en de *'real-time'* translatie- en orientatieveranderingen van volumes. Het moet steeds mogelijk zijn om acties binnen volumes, die getransporteerd worden, te detecteren. De voorwaarden van het afvragen van 500 volumes tegen 30 beelden per seconde met 10 assenstelsels blijven gelden, waardoor er in de laatste stap nog meer aandacht aan optimalisatie geschonken wordt.

Figuur 1.2: Werkomgeving voorzien van relatieve en een absolute, driedimensionale assenstelsels.

1.4 Methode

In eerste instantie is een studie nodig van de gebruikte software om een verbeterde versie van de HIM tot stand te brengen. In termen van software wordt er gefocust op C++, omdat het deel waar deze thesis op focust voornamelijk in C++ geschreven is. De inzichten verkregen omtrent C++ worden verder uitgediept met een verkenning van de OpenCV bibliotheek. Deze bestaat uit verschillende computervisie gerelateerde algoritmen en hulpmiddelen.

Het is belangrijk om volumes te kunnen definiëren in een driedimensionale ruimte. Dit is echter niet voldoende, want een volume vertelt het systeem niet uit welke punten met bijhorende coordinaten het bestaat. Het volume dient gediscretiseerde te worden vooraleer er bruikbare informatie

uit gehaald kan worden. De discretisatie zorgt ervoor dat een puntenwolk binnen het volume gegenereerd wordt die in een volgende stap gebruikt wordt voor de detectie van handelingen.

Dit wordt verder uitgebreid met een studie van bronnen die betrekking hebben tot de technieken die gebruikt worden bij het realiseren van het systeem, zoals projectie. Projectie is een techniek die, aan de hand van wiskundige operaties, een volume op een tweedimensionaal beeld weergeeft. Dit is geheel analoog aan het natekenen van een driedimensionaal voorwerp op een vlakke oppervlakte. Dit wordt gerealiseerd door een transformatiematrix op te stellen die zowel de intrinsieke als de extrinsieke eigenschappen van de sensor bevat. Deze matrix wordt met ieder punt binnen de volumes vermenigvuldigd, waardoor punten uit de wereldruimte naar de beeldruimte getransformeerd worden.

Eens de volumes en hun respectievelijke punten naar de beeldruimte getransformeerd zijn, kan de detectie van acties plaatsvinden. Elk punt van het volume is nu toegewezen aan een pixel in de beeldruimte. De dieptewaarden van de punten en de pixels worden vergeleken. Indien deze hetzelfde zijn, is een lichaam gedetecteerd op dat bepaalde punt. Vanaf een bepaald aantal gedetecteerde punten, afhankelijk van de ingestelde vulgraad waaraan een volume moet voldoen om detectie te meten, wordt verondersteld dat een actie uitgevoerd wordt binnen dit volume. Deze vergelijking dient echter zeer snel te gebeuren om een robuuste werkomgeving te creëren en behouden. Om dit proces te versnellen en de CPU te ontlasten, wordt er gebruik gemaakt van de GPU. De GPU maakt gebruik van het OpenCL-platform, een toepassing van de programmeertaal C, die vooral gericht is op acceleratie van processen door parallelle dataverwerking. Alvorens begonnen wordt met het ontwikkelen van de *kernels*, wordt het platform onderzocht. Met de kennis die verworven wordt uit dit onderzoek, wordt een kernel ontwikkeld die de CPU geheel ontlast van de detectie.

Verder wordt er naar een methode gezocht die, met een zo klein mogelijke impact op performantie, de mogelijkheid creëert om dynamische volumes te creëren en controleren.

Eens een model bekomen is die in *'real-time'* voldoet aan de voorwaarden, wordt gezocht naar andere opties omtrent het optimaliseren van het systeem. Zo wordt er gekeken naar alternatieven voor het controleren op detecties van de volumes en verdere optimalisaties van de *kernel*.

Er wordt dus een systeem ontworpen dat, op basis van de vulgraad van een gedefinieerd, eventueel bewegend volume en een dieptecamera, de aanwezigheid van een lichaamsonderdeel waarneemt. Dit systeem wordt geoptimaliseerd gebruikmakend van het OpenCL-platform dat toestaat om data in parallel te verwerken. Dit verklaart de titel: "Vulgraaddetectie van dynamische volumes in dieptebeelden: optimalisatie met OpenCL".

1.5 Overzicht

- Hoofdstuk 2: In de bronnenstudie worden relevante bevindingen besproken uit bestaande literatuur.
- Hoofdstuk 3: De gebruikte hardware van de HIM en de testomgeving wordt besproken. Dit beslaat zowel de sensoren als de computatiehardware.
- Hoofdstuk 4: De onderzochte elementen in verband met het creëren van volumes en assenstelsels worden samengebracht tot een werkend systeem.

- Hoofdstuk 5: Met behulp van de resultaten in stap 4 wordt een algoritme ontworpen dat volumes linkt aan de sensordata en deze vergelijkt voor het bepalen of er detectie is of niet. Verdere optimalisaties, in verband met snelheid, worden besproken en getest.
- Hoofdstuk 6: Het *'real-time'* aanpassen, verwijderen en toevoegen van volumes en assenstelsels wordt hier uitgelegd. Optimalisaties, in verband met snelheid, worden besproken.
- Hoofdstuk 7: De resultaten van verschillende tests op snelheid worden gegeven en besproken.

Hoofdstuk 2

Bronnenstudie

2.1 Inleiding

De bronnenstudie bestaat uit 4 delen die elk een gesteld probleem proberen op te lossen.

In hoofdstuk 2.2.1 wordt uitgelegd hoe, aan de hand van een set opgegeven hoekpunten, een vlak gevormd wordt. Door dit vlak als basis te beschouwen die geëxtrudeert wordt, wordt een volume tot stand gebracht. Dit volume speelt een vitale rol in het detecteren van bepaalde handelingen verricht door de arbeider of gereedschap. Er worden mogelijke opties overwogen om zo efficiënt mogelijk het volume op te bouwen. Door naar de efficiëntie te peilen wordt de snelheid, waarop volumes gecreëerd worden, en de hoeveelheid geheugen die deze nodig hebben in kaart gebracht.

In hoofdstuk 2.2.2 wordt uitgelegd hoe de vertaling tussen wereldruimte en beeldruimte bepaald wordt. Omdat de volumes in de wereldruimte gegenereerd worden, dient de operator werkelijke ruimtematen (metrisch) op te geven. De beeldruimte werkt niet op basis van deze metrische eenheden, maar berust geheel op de positie van pixels in het beeld. Hier komt het belang van de transformatie aan het licht. Door middel van de transformatie zijn punten uniek te definiëren in beide ruimtes, waardoor omzetting in beide richtingen mogelijk is.

In hoofdstuk 2.2.3 zijn verklaringen te vinden voor de versnelling die de GPU biedt bij het uitvoeren van het detectiealgoritme.

In hoofdstuk 2.2.4 worden enkele onderzochte optimalisatiemethoden aangehaald.

2.2 Resultaten van de bronnenstudie

2.2.1 Volumebepaling

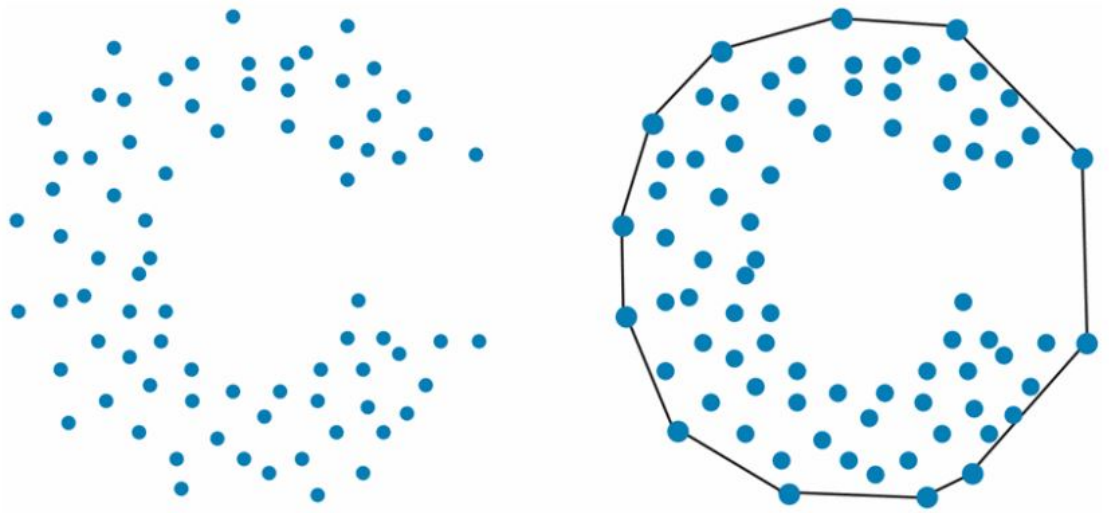
Met een minimum van 3 niet uitgelijnde punten ontstaat, door deze punten te verbinden met elkaar, een oppervlakte. Deze oppervlakte kan zowel convex als concaaf zijn. Omdat de voorkeur binnen een systeem steeds naar gestandaardiseerde entiteiten gaat, is gekozen om met convexe oppervlakten te werken. Dit is mogelijk door gebruik te maken van het *'Quickhull'*-algoritme [1].

Het *'Quickhull'*-algoritme is een variatie op het *'Quicksort'*-algoritme [2], die veelal gebruikt wordt om, op recursieve wijze, snel data te sorteren. Dit algoritme neemt een verzameling

punten en vormt een omhulsel op basis van de meest extern gelegen punten. Dit resulteert altijd in een convex vlak.

Deze eigenschap wordt, in een volgende stap, gedeeld met het volume dat ontstaat bij extrusie van de oppervlakte volgens de hoogte, of z-as. Het grote voordeel dat dit algoritme biedt, is de minimalisatie van omtrekpunten, waardoor het tekenen van het volume op ieder beeld minder rekenintensief is. Dit is te zien in figuur 2.1, waar enkel de buitenste hoekpunten worden gebruikt om het omhulsel te vormen en de interne hoekpunten verwijderd worden. Echter limiteert dit systeem de variatie van mogelijke oppervlakten tot eenvoudigere oppervlakten.

Dit algoritme behoud, in het slechtste geval, het initieel aantal hoekpunten. Concreet komt het erop neer dat het aantal hoekpunten nooit zal toenemen. Overigens zal dit algoritme maar eenmalig per vlak uitgevoerd worden, waardoor de *'overhead'* te verwaarlozen is.



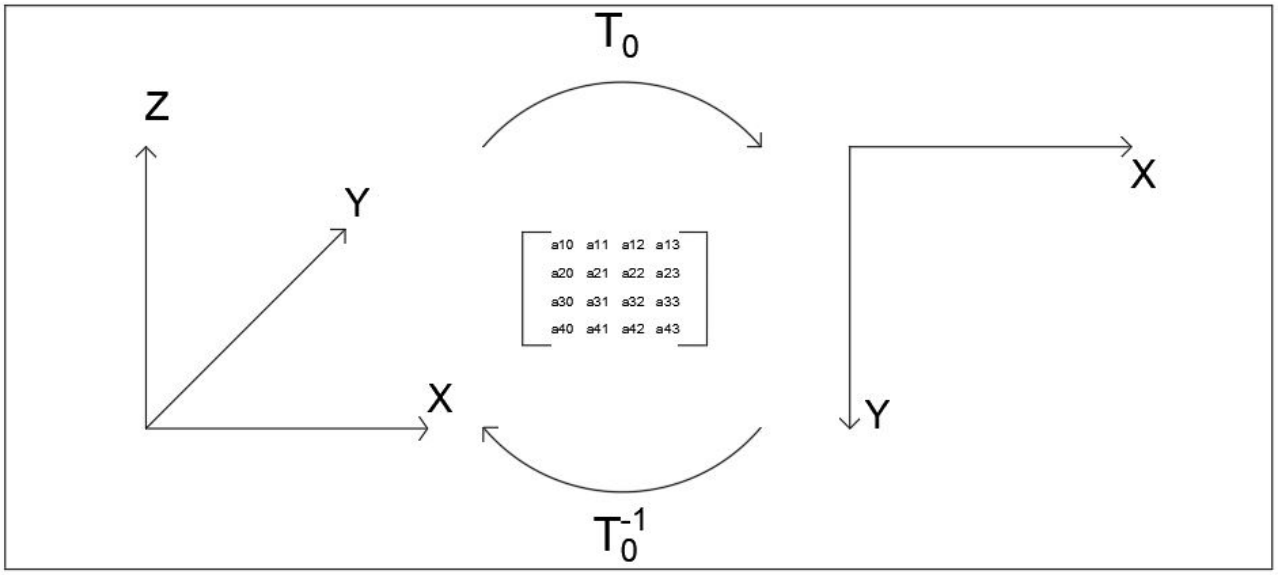
Figuur 2.1: *Quickhull*-algoritme in werking [1].

2.2.2 Volumetransformatie

In [3] is een methode uitgelegd om volumes, op basis van posities en afmetingen in de wereldruimte, te vertalen naar beeldcoördinaten. Figuur 2.2 is een visuele voorstelling van de vertaling van een punt tussen de driedimensionale wereldruimte en de tweedimensionale beeldruimte.

De vertaling wordt bereikt door gebruik te maken van lineaire transformaties. Deze transformaties worden voorgesteld door 4x4 matrices, welke eveneens getoond zijn op figuur 2.2 als T_0. In eerste instantie wordt ieder punt, waaruit het volume opgebouwd is, voorgesteld als een 4x1-matrix. Deze punten worden vervolgens vermenigvuldigd met de gepaste transformatiematrix.

De transformatiematrices tussen de wereldruimte en de beeldruimte bestaan uit de extrinsieke en de intrinsieke eigenschappen van de opstelling. Onder de intrinsieke eigenschappen worden de eigenschappen van de sensor zelf verstaan. Eigenschappen zoals de brandpuntsafstand, het optisch centrum en het scheeftrek-coëfficiënt brengen namelijk distorties met zich mee, die gecompenseerd worden door de intrinsieke matrix. De extrinsieke eigenschappen beschrijven de opstelling van de camera ten opzichte van de te capteren omgeving. Deze eigenschappen worden



Figuur 2.2: Transformatie tussen wereld- en beeldruimte met transformatiematrix.

in 2 4x4-matrices gegoten, waarvan het matrixproduct genomen wordt. Dit product zorgt voor de unieke conversie tussen de ruimtes en is beschreven in formule 2.1. Op te merken is dat de z-waarde niet bestaat in de beeldruimte, maar nodig is om terug naar de wereldruimte te converteren. Consequent wordt deze waarde onthouden voor elk punt.

$$P_{(x_{wrld},y_{wrld},z_{wrld})} * T_0 <=> P'_{(x_{img},y_{img},z_{cam})} * T_0^{-1} \tag{2.1}$$

De transformatiematrices binnen dezelfde ruimte zijn het product van een aantal verschillende matrices. In dit voorbeeld zijn er 4 transformatiematrices die ieder een bepaalde transformatie voorstellen. Er zijn 3 rotatiematrices, welke respectievelijk de rotaties voorstellen rond de z-as, x-as en y-as. De 4_{de} transformatiematrix stelt de translatie voor volgens alle assen simultaan. De volgorde van de rotaties speelt een belangrijke rol en wordt bepaald door de volgorde van de transformatiematrices in de matrixvermenigvuldiging.

Volumes kunnen absoluut gedefinieerd zijn ten opzichte van de oorsprong van de wereldruimte. Dit is echter niet altijd het geval aangezien er ook altijd de mogelijkheid is om een volume relatief te definiëren ten opzichte van een assenstelsel naar keuze. Ieder assenstelsel beschikt, in eerste instantie, over een matrix die zijn relatie tot de oorsprong beschrijft. Door gebruik te maken van deze matrix is het steeds mogelijk om volumes, die relatief gedefinieerd zijn, steeds te voorzien van globale coördinaten. Om dit te doen worden wederom alle punten, waaruit het volume bestaat, vermenigvuldigd met een transformatiematrix. In dit geval bevat de matrix haar oriëntatie en positie ten opzichte van de oorsprong. In formule 2.2 is deze relatie verder uitgelicht met T_1 als de transforamtiematrix die de transformatie beschrijft van het relatieve naar het absolute assenstelsel.

$$P_{(x_r,y_r,z_r)} * T_1 <=> P'_{(x_a,y_a,z_a)} * T_1^{-1} \tag{2.2}$$

Als gevolg kan gesteld worden dat de transformatie, van een relatief gedefinieerd volume in de wereldruimte naar de beeldruimte, bekomen wordt door vermenigvuldiging van matrices zoals weergegeven in formule 2.3. Waarbij T_1 de relatie van het relatief assenstelsel t.o.v. de oorsprong, en T_0 de intrinsieke en extrinsieke eigenschappen van de opstelling beschrijft.

$$P_{(x_r,y_r,z_r)} * T_1 * T_0 <=> P'_{(x_{img},y_{img},z_{wrld})} * T_0^{-1} * T_1^{-1} \quad (2.3)$$

Het bepalen van de extrinsieke en intrinsieke parameters van een sensor wordt besproken in [3]. Deze parameters zijn afhankelijk van de constructie en eventuele onnauwkeurigheden van de sensor. Hierdoor kunnen verschillende sensoren, zelfs meerdere sensoren van hetzelfde type en van dezelfde fabrikant, andere parameters bezitten. Deze parameters dienen achterhaald te worden. Hiervoor is een methode in [3] uitgelegd. Het principe van deze methode berust op het kalibreren van het beeld ten opzichte van een gekend, eenvoudig ijkingsfiguur. Dit figuur heeft herkenbare patronen, bijvoorbeeld de zwarte en witte vierkanten van een schaakbord. Wanneer de eigenschappen van het figuur bekend zijn, worden deze vergeleken met de overeenkomstige gemeten eigenschappen in de gecapteerde beelden van het figuur. Indien een groot aantal, liefst zo veel mogelijk, beelden gecapteerd zijn van het figuur in verschillende oriëntaties en posities, wordt het kalibratiealgoritme uitgevoerd en de interne parameters bepaald. Een bestaand algoritme binnen OpenCL, met bijhorend patroon, zorgt ervoor dat geen eigen kalibratiealgoritme wordt ontwikkeld.

2.2.3 Volumes en detectie

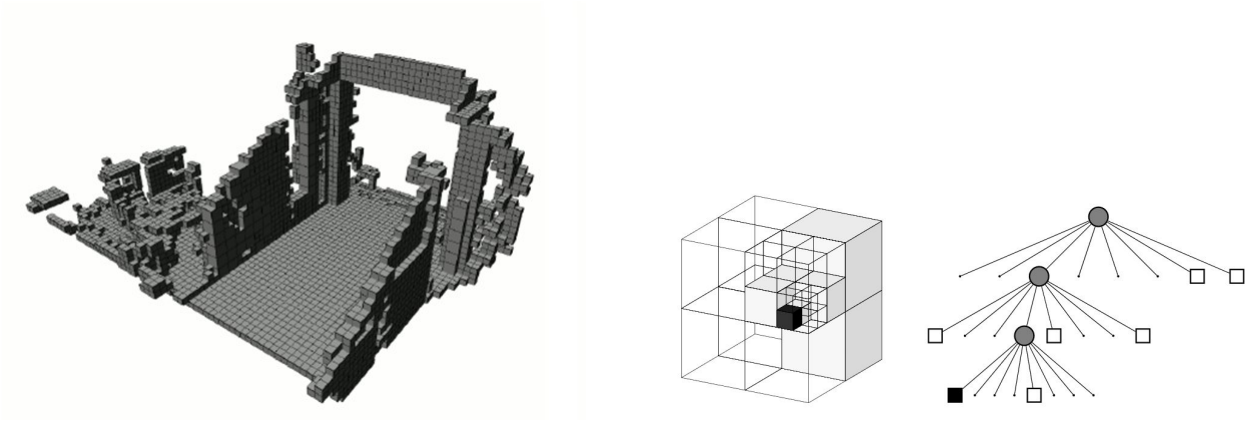
Een GPU kan, dankzij haar grote hoeveelheid *'cores'* (kernen), op grote hoeveelheden verschillende data simultaan bewerkingen uitvoeren. Dit concept heet *'data parallelism'* en is veelal gebruikt om het verwerken van grote datapakketten te versnellen [4]. De prestatie is, in vergelijking met een CPU, beter op voorwaarde dat de dataset groot genoeg is en dat de kernel de sterke eigenschappen van de GPU benut. Bij het verwerken van beelden in combinatie met het afvragen van volumes ter detectie van acties, kan een GPU op een hoog tempo een groot deel volumes evalueren.

2.2.4 Verdere optimalisaties

Bepaalde elementen kunnen een vitale rol spelen in het verbeteren van de prestaties van een systeem. Zo wordt er gekeken naar alternatieven methodes om kernels te schrijven en om volumes weer te geven.

Octomap

De volumes in deze toepassing zijn opgebouwd uit puntenwolken die bekomen worden door discretisatie van de omhulde ruimtes. Dit geeft een significante hoeveelheid punten die afgevraagd worden bij ieder beeld voor ieder volume. Er is echter een probabilistische methode beschreven in [5] die op basis van knooppunten een volume beschrijft. Deze methode creëert *'octomap'*, die een volume of ruimte indeelt in kubussen die op hun beurt verder onderverdeeld zijn. Op figuur 2.3 (a) is een kaart afgebeeld die opgebouwd en opgedeeld is volgens deze methode. Op figuur



(a) Octomap kaart (b) Octomap Volume

Figuur 2.3: Weergave octomap [5].

2.3 (b) is te zien hoe deze kubussen intern verder verdeeld zijn. Dit kan het beste gezien worden als een waaier die van het grootste object doorloopt tot het kleinste onderdeel van dit object.

De octomap is een snelle methode om te controleren in welke maten een ruimte ingenomen is. Concreet kan dit toegepast worden in deze thesis om de detectie te versnellen.

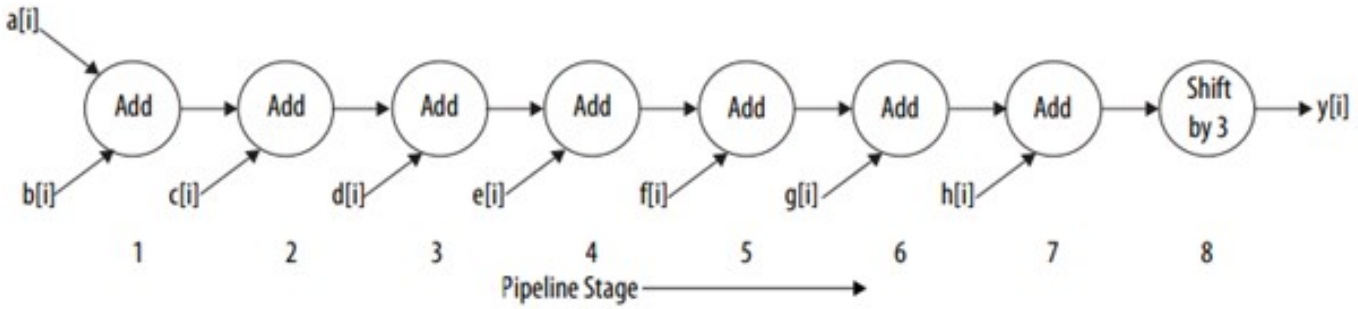
OpenCL met Intel

Verder is er nog de mogelijkheid om de kernel te optimaliseren. Om efficiënt om te gaan met de GPU, is het belangrijk om te weten hoe code wordt omgezet in fysieke hardware. Gids [6] verteld hoe een *'field-programmable gate array'* (FPGA) best geprogrammeerd wordt om er zoveel mogelijk performantie uit te halen. Verder worden relevante optimalisatiemethoden besproken voor kernels die kunnen dienen voor de doeleinden van deze masterproef.

Het verschil tussen een FPGA en GPU is dat de GPU parallelle dataverwerking verricht door dezelfde generieke computatie-hardware te dupliceren. De FPGA bereikt parallelle dataverwerking door enkel de logica te dupliceren die de algoritmen uitvoert. Zo kunnen *'pipelines'* worden gevormd, zoals op figuur 2.4. In 1 klokcyclus kunnen meerdere bewerkingen gedaan worden op dezelfde data. Een GPU moet dit binnen meerdere klokcycli doen, maar kan wel dezelfde bewerkingen doen op verschillende data in parallel. Het is belangrijk om dit verschil in acht te houden, aangezien de gebruikte Intel Graphics 530 een GPU is.

Loops Optimalisaties van een *'loop'*, waarbij een *'random access memory'* (RAM)-geheugenplaats wordt aangesproken bij elke iteratie, worden bereikt door opeenvolgende geheugenplaatsen te lezen in opeenvolgende iteraties. De compiler kan dan optimalisaties uitvoeren op het gebied van geheugen uitwisseling. Verder kunnen *'loops' 'unrolled'* worden. De code in de loop wordt niet 1 maal doorlopen per *'loop'*, maar een gespecificeerd aantal keren. Het aantal iteraties van een *'loop'* verminderd dan. Zodoende hoeft minder vaak bepaald te worden of de *'loop'* moet eindigen en hoeft de code minder vaak terug naar het begin van de *'loop'* te springen. Beide verbeteringen brengen tijdwinst met zich mee.

Buffers Buffers zijn geheugenplekken die de GPU gebruikt en opgeslagen zijn in het RAM. Een optimalisatie rond buffers in loops is reeds besproken in het deel over *'loops'*. Een verdere



Figuur 2.4: Voorbeeld van een pipeline ter optimalisatie van de snelheid van een loop [6].

optimalisatie is het beperken van de versatiliteit van een buffer. Zo zijn geheugenplekken waaruit enkel kan worden gelezen, of waar enkel in geschreven kan worden, sneller dan een buffer die beide operaties ondersteunt. Hoe meer restricties geplaatst zijn op een buffer, hoe efficiënter de geheugenplek aangesproken wordt.

Algemeen *'Integer'* delingen zijn langzaam, tenzij de noemer een macht is van 2. Bewerkingen zijn langzaam met de *'float'* datastructuur dan met de *'integer'* datastructuren.

Het opgeven van de grootte van een werkgroep, indien dit een bekende en niet veranderende waarde is, maakt het mogelijk voor de offline compiler om agressieve optimalisaties te verrichten. Er is een verminderde optimalisatie te bereiken indien enkel de maximale grootte van de werkgroep wordt opgegeven.

Het gebruikmaken van de kleinst mogelijke datatypen zorgt ervoor dat minder resources gebruikt worden van de FPGA, alsook de GPU.

2.3 Conclusie

Een eventuele manier om vlakken op te bouwen is het *'quickhull'*-algoritme. Hierdoor wordt de werkdruk al mogelijk verlaagd voor iedere creatie van een volume. Echter worden 2 eigen methoden gecreëerd voor het opbouwen van vlakken. Dit omwille van de ruwe benadering van de door de oprator bedoelde vorm die het *'quickhull'*-algoritme als resultaat geeft.

De transformatie tussen verschillende assenstelsels, en zelfs tussen ruimtes, wordt gerealiseerd op basis van de methode beschreven in [3]. Deze methode creëert de mogelijkheid om de detecties van een zekere nauwkeurigheid te voorzien. Verder zal deze methode steeds geldig zijn bij uitbreidingen op het bestaand model.

Een voorbeeld van een mogelijke uitbreiding is het introduceren van de *'octomap'*. De *'octomap'* kan dienen om het gehele detectie-algoritme te versnellen. Een element dat verder in acht wordt genomen is de compatibiliteit met het OpenCL-platform. Binnen dit platform zijn er tal van mogelijkheden voor het optimaliseren van het detectiealgoritme. Zo werd het voorbeeld aangehaald van het uitrollen van een *'loop'*, om het geheel te versnellen, een optie dat ook in [7] aan bod komt met goede resultaten.

Hoofdstuk 3

Gebruikte hardware

3.1 Kinect sensor

De Kinect sensor is ontwikkeld door Microsoft als uitbreiding van de Xbox 360 console en uitgebracht in 2010. Deze sensor is in staat om verschillende elementen uit de omgeving waar te nemen om, op basis van gedetecteerde bewegingen en audio commando's, een videospel te bedienen zonder fysieke controller.

Al snel blijkt dat de Kinect tot veel meer in staat is dan initieel gedacht. Toepassingen zoals gezichtsherkenning, *'tracking'* van lichaam poses en immersieve methodes om tele-meetings te houden, zijn maar enkele voorbeelden volgens [8]. Onderzoek is gedaan naar methodes om de inzetbaarheid van Kinect te bevorderen binnen verschillende velden. Uit dit onderzoek blijkt dat, in termen van prijs-kwaliteit verhouding, de Kinect voldoet als een alternatief voor bestaande computervisie producten. De competitie bestaat uit zowel producenten van stereo- als RGB-D - camera's, die gebruikt kunnen worden voor gelijkaardige doeleinden.

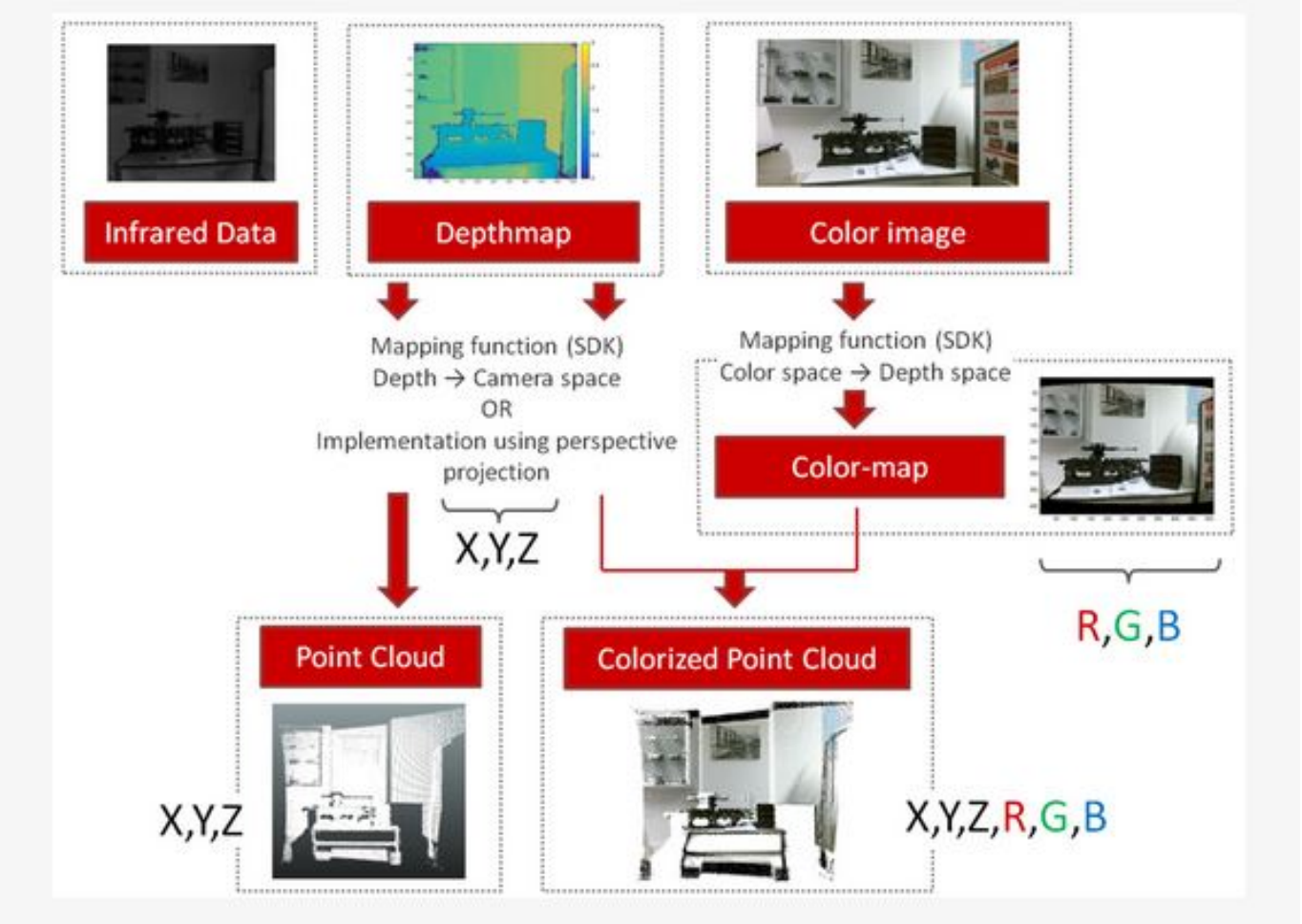
De Kinect voorziet een *'output'* van dieptebeelden en RGB-beelden, zoals in figuur 3.1, die door verschillende algoritmes gebruikt kunnen worden om informatie te verzamelen. Enkele van deze algoritmes werden voorzien door de Microsoft SDK, een ontwikkelingskit ontworpen door Microsoft.

Binnen deze toepassing werd door Arkite echter gekozen voor de Freenect-bibliotheek, een *'open source'* variant die eveneens tal van mogelijkheden biedt voor het ontwerpen van software die compatibel is met de Kinect op het Windows *'operating system'*.

3.1.1 Opbouw van de Kinect sensor

De Kinect is in essentie een verzameling van sensoren zoals afgebeeld op figuur 3.2. De RGB-camera neemt beelden waar op basis van hun kleurenwaarden binnen het spectrum van zichtbaar licht. De dieptecamera bestaat uit 2 delen, de zender en de ontvanger, die een beeld vormen op basis van de afstand van een gegeven punt tot de camera. Deze camera's worden optimaal gericht door gebruik te maken van de gemotoriseerde tilt die ingebouwd is.

Verder bezit de Kinect een stel microfonen die omgevingsgeluiden detecteren waaruit commando's gefilterd kunnen worden.



Figuur 3.1: KinectV2 *'output'* [9].

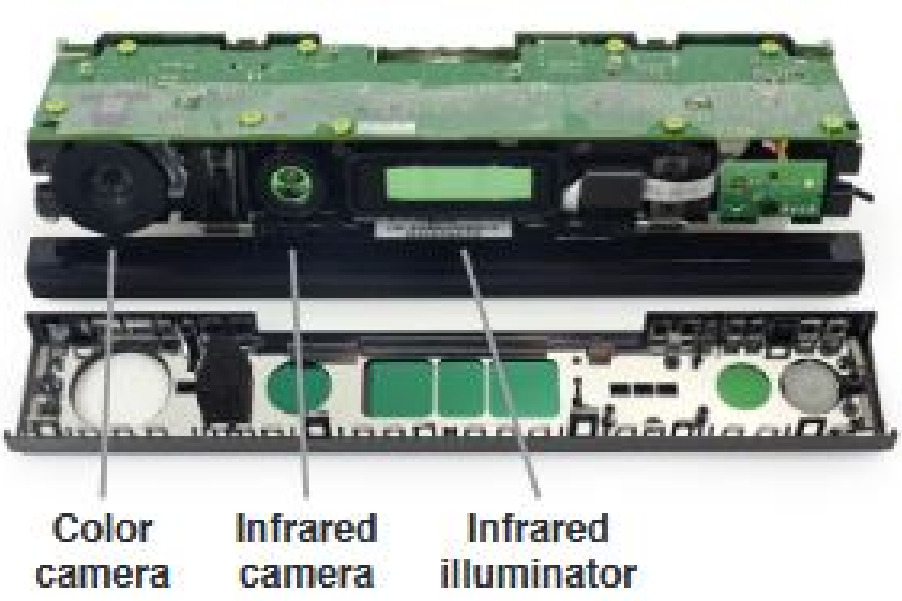
In het systeem van Arkite wordt specifiek gekozen voor de Kinect V2, een verbeterde versie van de Kinect uitgebracht in 2014. Deze recentere versie verschilt voornamelijk in de manier waarop en met welke resolutie beelden waargenomen worden. De toepassingen waarvoor de camera's gebruikt kunnen worden blijven gelijk, met als verschil de verhoogde nauwkeurigheid waarover de Kinect V2 beschikt volgens [11].

3.1.2 Werking van het RGB-D principe

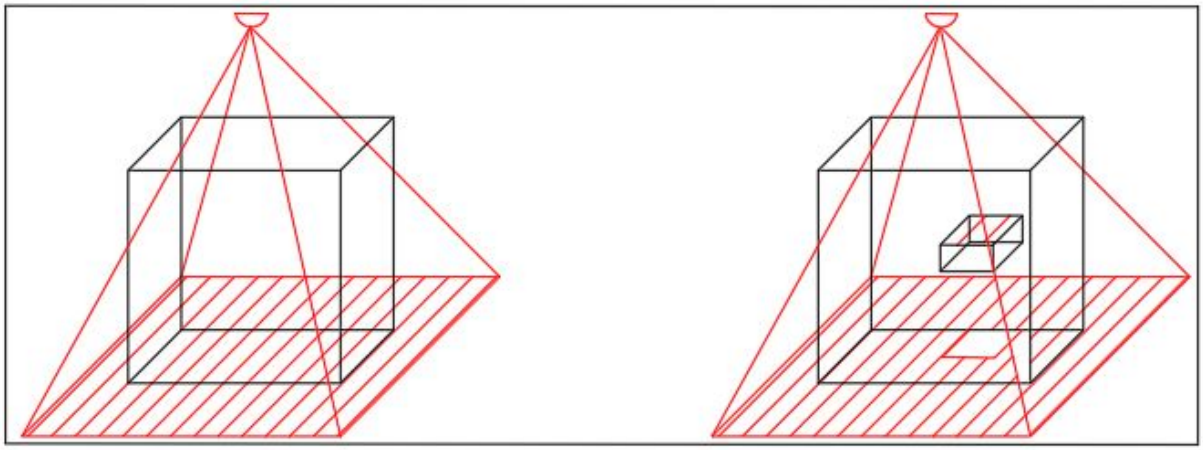
Het RGB-D principe combineert de beelden van de dieptecamera en de beelden van de RGB-camera om, in combinatie met gepaste belichting, een omgeving in drie dimensies voor te stellen. Dit is de meest eenvoudige toepassing waarvoor een RGB-D camera gebruikt wordt. Zoals reeds vermeld, kan dieptedata, verkregen uit de dieptecamera, gebruikt worden voor tal van meer ingewikkelde toepassingen.

De HIM gebruikt de dieptecamera om dieptewaarden te genereren van de werkomgeving. Zo zal de diepte van een bepaald punt op het beeld wijzigen indien bijvoorbeeld een hand of gereedschap boven het origineel werkvlak wordt geplaatst, zoals te zien is op figuur 3.3. De rol die deze verandering speelt binnen de detectie van handeling, wordt verder uitgelicht in hoofdstukken 4 en 5.

Om dieptedata aan een RGB-beeld toe te voegen volgens het RGB-D-model, is het nodig om de relatie tussen de dieptesensor en de RGB-camera te kennen, met eventuele bijkomende correcties.



Figuur 3.2: Kinect V2 opbouw [10].



Figuur 3.3: Verandering van dieptewaarden door detectie van operator of object.

De relatie tussen de dieptesensor en de camera wordt vastgesteld door middel van een kalibratie. De kalibratie resulteert als het ware in een samenvoeging van pixels van het RGB-beeld met pixels van het dieptebeeld, zoals besproken in [12], aan de hand van een gepaste transformatie (3.1).

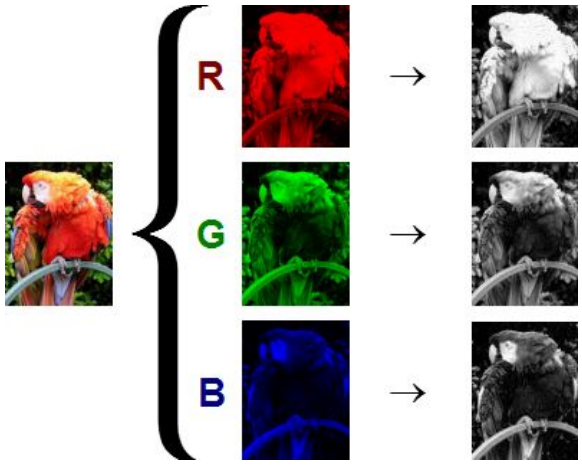
$$\begin{bmatrix} x_{rgb} \\ y_{rgb} \\ 1 \end{bmatrix} = T * \begin{bmatrix} x_{ir} \\ y_{ir} \\ 1 \end{bmatrix} = \begin{bmatrix} a11 & a12 & a13 \\ a21 & a22 & a23 \\ a31 & a32 & 1 \end{bmatrix} * \begin{bmatrix} x_{ir} \\ y_{ir} \\ 1 \end{bmatrix} \tag{3.1}$$

RGB-D is een alternatief voor de opstelling volgens de stereo-camera. Bij stereo-camera zullen 2 camera's vanuit verschillende punten een beeld waarnemen. Door de relatie tussen deze camera's vast te leggen, kan dieptedata verkregen worden. Dit kan voor bepaalde toepassingen rekenintensiever zijn, aangezien 2 verschillende beelden verwerkt dienen te worden vooraleer deze

samengevoegd kunnen worden.

3.1.3 RGB-beelden verkregen door de Kinect sensor

De RGB-camera neemt een kleurenbeeld waar volgens de 3 hoofdkleuren, hier kanalen genoemd, namelijk: rood, groen en blauw. Ieder kanaal neemt apart een beeld waar op basis van de invallende elektromagnetische golflengte. Deze 3 beelden worden samengevoegd om uiteindelijk het kleurenbeeld te creëren, zoals op figuur 3.4 te zien is.



Figuur 3.4: Kanalen van het RGB-beeld.
`https://commons.wikimedia.org/wiki/File:RGB_channels_separation.png`

Dit proces wordt herhaald met een snelheid van 30 beelden per seconde, bij een resolutie van 1920x1080 pixels.

3.1.4 Dieptebeelden verkregen door de Kinect sensor

Het dieptebeeld wordt gevormd door gebruik te maken van een infrarood verlichtingselement, wiens lichtstralen worden weerkaatst door objecten en opgevangen door een infrarood-camera. De tijd tussen het uitzenden en opvangen wordt voor iedere pixel opgemeten en dient bijgevolg als variabele bij het bepalen van de afstand van een punt tot de sensor. De lichtstralen zijn geheel onzichtbaar voor de RGB-camera, waardoor er geen storing optreedt. Dit principe heet *'Time-of-Flight* (ToF), en kent vele toepassingen binnen het veld van computervisie.

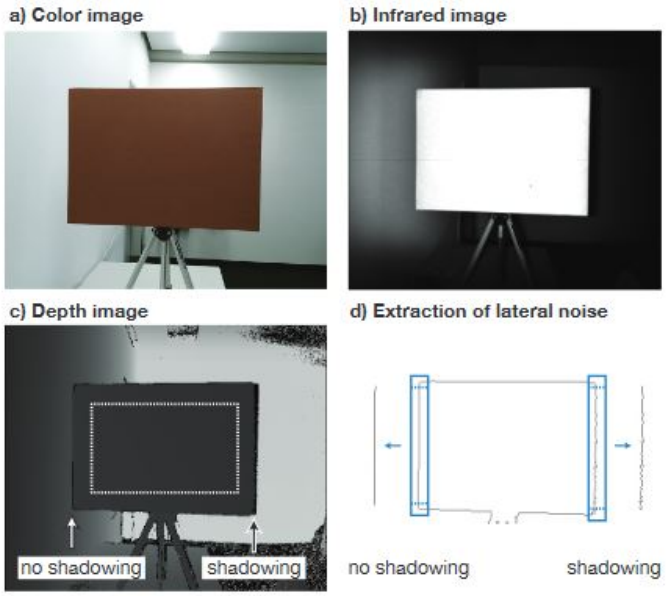
De combinatie van de infrarood-camera en de infrarode belichting vormt de dieptesensor. Concreet wordt aan iedere pixel van het beeld een (grijs)waarde toegekend die de afstand van dat punt ten opzichte van de camera voorstelt in millimeter. De kinect V2 biedt eveneens de mogelijkheid om enkel het infrarood beeld weer te geven, maar daar is in deze toepassing geen gebruikt van gemaakt.

De resolutie van het dieptebeeld van 512x424 pixels is aanzienlijk lager dan de resolutie van het kleurenbeeld. Toch blijkt deze resolutie hoger te liggen dan de meest gangbare concurrenten volgens [10]. Bijkomend biedt de kinect V2 een bredere zichthoek, waardoor een groter oppervlak gemonitord kan worden.

Op figuur 3.5 is een voorbeeld te zien van de verhouding van de verschillende beelden tot elkaar. Onderzoek, uitgevoerd naar de ruisgevoeligheid van de dieptesensor, wijst naar belangrijke

elementen waarmee rekening gehouden moet worden. Het voornaamste element is externe belichting, zoals de verlichting in een werkomgeving of zelfs direct zonlicht. Stralen van een externe lichtbron kunnen een infrarood componenten bevatten die de correcte werking van de dieptesensor verhinderen. Eveneens de afstand en de oriëntatie ten opzichte van een voorwerp spelen een rol in het verkrijgen van bruikbare dieptedata.

In deze toepassing wordt er geen rekening gehouden met direct zonlicht aangezien de beschouwde scenario's binnen gelegen zijn. Er wordt verwezen naar [10] (hoofdstuk 4) voor precieze data over de factoren die ruis in het dieptebeeld beïnvloeden binnen deze setting.



Figuur 3.5: Dieptebeeld en infrarood beeld met ruis [10].

3.1.5 Toepassing van de Kinect sensor binnen de masterproef

Er is gebruik gemaakt van de kinect V2 om een korte fragment te filmen, gebruikmakend van de dieptecamera. Dit fragment wordt frame per frame in een bestand opgeslagen volgens een data-indeling ontworpen door Arkite. De fragmenten kunnen achteraf gelezen worden door gebruik te maken van een *'reader'*, wederom ontwikkeld door Arkite. Dit resulteert in de mogelijkheid om gemaakte systemen te testen zonder de nood aan een fysieke HIM, maar slechts de werkelijke data van het fragment.

Dit fragment is door Arkite voorzien om het testen van het ontwikkelde detectie systeem mogelijk te maken zonder een fysieke HIM ter beschikking te hebben. Er wordt van uit gegaan dat, indien de methode besproken in deze thesis betrouwbare resultaten levert met de videofragmenten, eveneens betrouwbare resultaten geleverd wordt door de HIM.

De beelden waargenomen door de RGB-camera worden buiten beschouwing gelaten aangezien ze geen meerwaarde toevoegen aan dit onderzoek. Deze worden echter wel in de toepassing van Arkite gebruikt voor andere doeleinden dan detectie.

3.2 computatie hardware

Omdat de HIM niet gebruikt kan worden voor de ontwikkeling en het testen van de software, wordt een test-computer gebruikt. Het is belangrijk om de verschillen tussen computatie hardware in acht te nemen. De RAM specificaties voor de beide machines is 2x4GB, DDR4, 2133MHz. De CPU van de test-computer is een Intel I7-6700HQ, die van de HIM is een Intel I5 6500TE 6M. De performantie van CPU bewerkingen en calculaties zullen verschillend zijn tussen de HIM en de testcomputer, aangezien deze CPUs significant verschillend zijn van elkaar. De gebruikte GPU voor beide machines is de Intel Graphics 530. De tijd voor het afronden van het GPU algoritme zal insignificant verschillen van elkaar, aangezien dezelfde GPU gebruikt wordt en het detectie-algoritme voornamelijk aanspraak doet op de GPU. Beide machines gebruiken het Microsoft Windows 10 *'operating system'*.

Hoofdstuk 4

Opbouw van de verschillende systeem elementen

Het systeem ontworpen door Arkite maakt gebruik van volumes om zones aan te duiden waarin een relevante actie kan plaatsvinden. Deze volumes worden gecreëerd door, in eerste instantie, een punt op het beeld te selecteren, en vervolgens het aantal hoekpunten te kiezen. Op basis van deze hoekpunten kan een groot aantal vormen gerealiseerd worden, gaande van een simpele driehoek tot polygonen met 99 hoekpunten.

4.1 Definitie van de volumes in driedimensionale assenstelsels

4.1.1 Het vlak definiëren in driedimensionale assenstelsels

Een vlak kan gedefinieerd worden vanaf het moment dat 3 niet uitgelijnde hoekpunten gegeven zijn. De resolutie, nog een belangrijke parameter, speelt een grote rol in het vastleggen van de punten populatie tijdens de discretisatie. Dit is in essentie grotendeels analoog aan het systeem ontworpen door Arkite. De ruimte, waarin er in het origineel systeem gewerkt wordt, is geheel tweedimensionaal. De punten worden in beeldcoördinaten toegewezen in combinatie met dieptewaarden. Dit maakt van het origineel systeem echter geen zuiver driedimensionale toepassing.

Voor de definitie van de volumes zijn er 2 methodes uitgewerkt, die uiteindelijk hetzelfde doel dienen. Dit om aan de wensen te voldoen van Arkite. Met meerdere methoden kunnen namelijk onderlinge vergelijkingen plaatsvinden om de betere manier te vinden, hetzij op nauwkeurigheid als snelheid. Beide methodes vertrekken met dezelfde gegevens, namelijk een set van minimaal 3 punten. Beide methodes onderscheiden zich echter van het origineel systeem omwille van het zuiver driedimensionaal karakter van de punten.

De methodes worden respectievelijk de *radiale methode* en de *lineaire methode* genoemd, een verwijzing naar het patroon dat ze vormen na de discretisatie.

4.1.2 Vlak extruderen naar een volume volgens de radiale methode

De opgegeven hoekpunten worden in klokwijzerzin overlopen zoals in figuur 4.1 te zien is. De afstand tussen 2 punten wordt bepaald door de stelling van Pythagoras toe te passen (4.1).

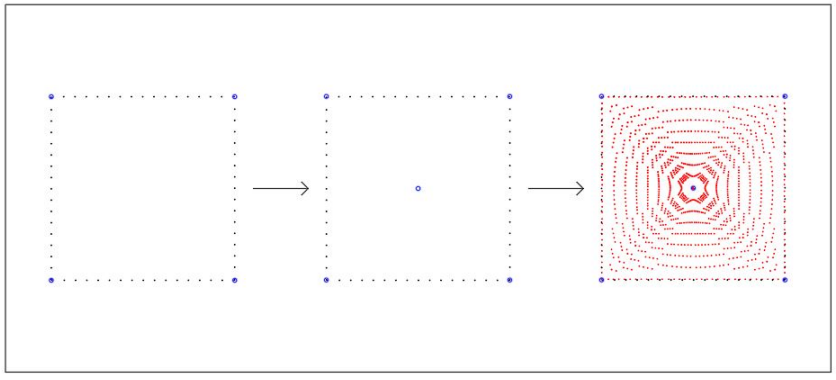
$$R = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} \tag{4.1}$$

Op basis van de opgegeven resolutie, die overigens in punten per millimeter uitgedrukt is, wordt een aantal punten op deze straal gelegd. Eens alle originele hoekpunten op deze manier verbonden zijn met hun buurpunten, is de omlijning van het vlak gevormd.

Van de originele hoekpunten wordt het gemiddelde genomen om zodoende het zwaartepunt van de figuur te vinden. Dit wordt gedaan door de som te nemen van zowel x- als y-coördinaten van ieder punt, en vervolgens te delen door het aantal hoekpunten (4.2).

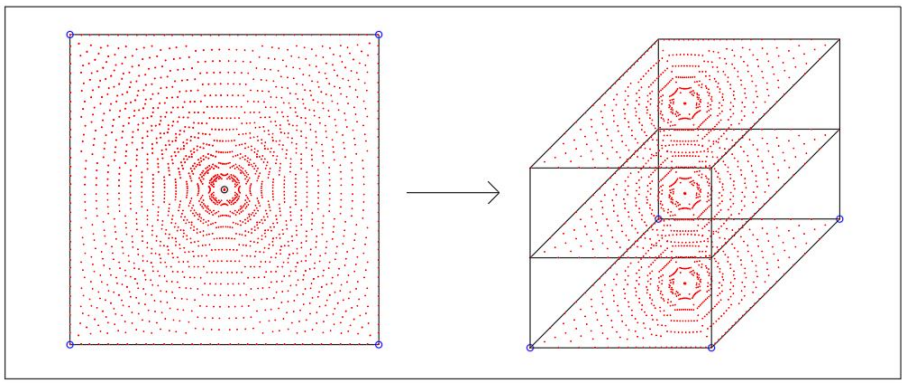
$$\begin{aligned} x_c &= \sum_{n=1}^{nrVertices} \frac{x_n}{nrVertices} \\ y_c &= \sum_{n=1}^{nrVertices} \frac{y_n}{nrVertices} \\ center &= P(x_c, y_c) \end{aligned} \tag{4.2}$$

Eens het zwaartepunt is gevonden, wordt een lijn getrokken naar ieder punt op de omtrek. De lengte van deze lijn wordt wederom met (4.1) berekend en ook op deze lijn worden punten gelegd. Vertrekkende van het zwaartepunt wordt op figuur 4.1 afgebeeld hoe het gediscretiseerde oppervlak gevormd wordt.



Figuur 4.1: Methode voor het opbouwen van het radiaal vlak.

In de laatste stap voor het vormen van het volume worden kopieën van het vlak gestapeld. Dit is aan de hand van een vast interval van 10 millimeter gerealiseerd. Er wordt telkens gestapeld totdat het volume de gewenste hoogte bezit, zoals afgebeeld op figuur 4.2.



Figuur 4.2: gediscretiseerd volume volgens de radiale methode.

4.1.3 Vlak extruderen naar een volume volgens de lineaire methode

Gelijkaardig aan de radiale methode, vertrekt deze methode van een aantal opgegeven hoekpunten. De richtingscoëfficiënt tussen opeenvolgende hoekpunten wordt berekend, waarbij het eerste punt de opeenvolging van het laatste punt is. Deze richtingscoëfficiënt wordt gebruikt om de lijn tussen 2 opeenvolgende punten te definiëren als een wiskundige formule. De richtingscoëfficiënten tussen x en y, genaamd xySlope, alsook die tussen x en z, genaamd xzSlope op (4.3), worden opgeslagen in een lijst, of *'array'*. In een aparte, gerelateerde *'array'* wordt het puntenpaar opgeslagen die betrekking heeft tot het richtingscoëfficiënten-paar. Ook worden de uiterste x en y posities opgeslagen van de hoekpunten.

$$\begin{aligned} xySlope &= \frac{y_{n+1} - y_n}{x_{n+1} - x_n} \\ xzSlope &= \frac{z_{n+1} - z_n}{x_{n+1} - x_n} \\ yzSlope &= \frac{z_{n+1} - z_n}{y_{n+1} - y_n} \end{aligned} \tag{4.3}$$

Aangezien een volume gediscretiseerd wordt door het opvullen ervan met punten, wordt de afstand tussen opeenvolgende volumepunten bepaald in millimeter. Dit wordt gedaan door de gemiddelde afstand tussen de randen van het volume te bepalen in x-, y-, en z-richting. De kleinste gemiddelde afstand wordt genomen en hierop wordt een gespecificeerd aantal punten gelegd. De onderlinge afstand van deze punten bepaald de afstand tussen alle opeenvolgende punten in het volume in x-, y- en z-richting en wordt 'staplengte' genoemd.

Om de puntenwolk van het volume te creëren, wordt voor elke x-waarde, tussen de uiterste punten, volgende code doorlopen:

1. Op basis van de x-waarden van opeenvolgende punten, wordt bepaald of de huidige x-waarde hiertussen valt. Zo wordt bepaald welke 'lijnen' betrekking hebben tot de huidige x-waarde. Enkel voor deze lijnen gaat dit proces verder.

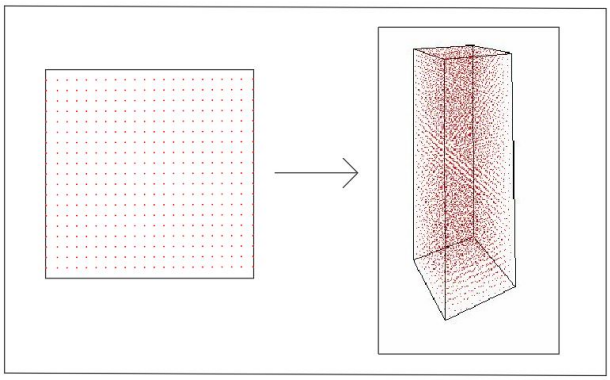
2. Voor deze lijnen zijn reeds de richtingscoëfficiënten berekend. Met (4.4), waarbij '0' het eerste punt voorstelt van huidige puntenpaar, wordt de y en z positie bepaald van het punt op de lijn horende bij de huidige x-waarde.

3. Tussen elk puntenpaar, die ontstaan is bij de huidige x-waarde, wordt de rico berekend tussen y en z, genaamd yzSlope op (4.3). Hiermee wordt de z waarde bepaald voor alle y punten tussen dit puntenpaar, genaamd zTussen in (4.4). Elk gecreëerd punt wordt enkele keren, afhankelijk van de ingestelde hoogte, gekopieerd. Vervolgens wordt de z waarde van elke kopie aangepast. Zo ontstaan verschillende lagen van het vlak die verschillen in hoogte, zoals te zien in figuur 4.3.

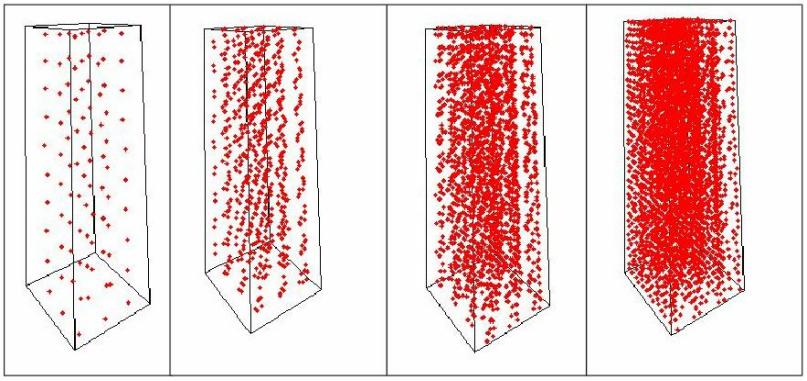
$$\begin{aligned} y &= xySlope * (x - x_0) + y_0 \\ z &= xzSlope * (x - x_0) + z_0 \\ zTussen &= yzSlope * (y - y_0) + z_0 \end{aligned} \tag{4.4}$$

De lengte tussen opeenvolgende x-,y- en z-waarden is gelijk aan de staplengte, die reeds berekend is.

De punten gecreëerd in stap 3 worden toegevoegd aan verschillende *'arrays'* in een bepaalde volgorde. Het aantal *'arrays'* is gelijk aan te variabele genaamd 'aantal resoluties', welke vrij te kiezen is voor elk volume. De *'arrays'* worden incrementeel genummerd beginnend vanaf 0. Voor elke richting (x,y,z) wordt een teller aangemaakt, geïnitialiseerd op 0, die geïncrementeerd wordt telkens als een punt is toegevoegd volgens respectievelijke richting. De tellers resetten zich indien hun waarden gelijk zijn aan het aantal resoluties. Vervolgens wordt bepaald in welke *'array'* het toe te voegen punt geplaatst wordt, afhankelijk van de huidige teller-waarden. De kleinste van de 3 waarden bepaald het nummer van de *'array'* waarin het geplaatst wordt. Uiteindelijk worden de *'arrays'* samengevoegd in aflopende volgorde. De grootte van elke *'array'* wordt bijgehouden. Het resultaat van dit is een finale *'array'*, of tabel, die opgebouwd is uit punten die gesorteerd staan in aflopende resoluties. Figuur 4.4 toont de punten behorende tot 1 'resolutie stap'. In hoofdstuk 5.3 wordt uitgelegd waarom dit gedaan is, aangezien deze stap enkel relevant is voor de optimalisatie van het systeem en niet de werking ervan.



Figuur 4.3: gediscretiseerde volume volgens de lineaire methode.



Figuur 4.4: gediscretiseerde volume volgens de lineaire methode opgedeeld in resoluties.

4.1.4 Toepassing van de gedefinieerde volumes in de praktijk

Deze volumes worden in de toepassing gebruikt om handelingen te detecteren. Om dit te doen worden alle punten, die te zien zijn op de figuren 4.2 en 4.3, in tabellen opgeslagen. Deze worden vergeleken met de dieptewaarden uit iedere pixel. Hoe dit precies gedaan wordt, staat verder uitgediept in hoofdstuk 5. In hoofdstuk 4.4.2 wordt het nut uitgelegd van de 'vulfactor'; een parameter die gekozen is door de operator per volume.

Belangrijke waarden en *'arrays'* die later gebruikt worden, worden aangemaakt en opgeslagen bij het volume:

1. Een *'array'* met alle punten van het volume.
2. Het aantal resolutie stappen.
3. Een *'array'* met het cumulatief aantal punten per resolutie stap.
4. Het aantal hoekpunten.
5. Een *'array'* met alle hoekpunten.
6. De staplengte.
7. De vulfactor.

4.2 Definitie van de verschillende assenstelsels

4.2.1 Definiëren van assenstelsels binnen het systeem

Een coördinaatstelsel wordt gebruikt om een ruimte zodanig in te delen dat ieder punt gekenmerkt wordt door 1, en slechts 1, coördinaat. Onder ruimte worden de 3 dimensie verstaan die gekenmerkt zijn door een assenstelsel.

Binnen deze toepassing komen assenstelsels aan bod in zowel de beeldruimte als de wereldruimte. De beeldruimte is niets anders dan het beeld dat gecapteerd is door de camera, waarin de pixels overeenkomen met de mogelijke coördinaten. Aan de andere hand spreken we van de wereldruimte als de werkelijke omgeving, waarin coördinaten in metrische maten uitgedrukt worden.

Een relatief assenstelsel is op haar beurt gekenmerkt door haar positie en oriëntatie ten opzichte van de oorsprong; een absoluut assenstelsel dat onveranderd blijft. Deze eigenschappen worden het beste weergegeven volgens een transformatiematrix.

In deze toepassing wordt een willekeurig punt gekozen waarin een assenstelsel geplaatst wordt. Dit assenstelsel kent geen rotatie en geen translatie en is gekend als het absolute assenstelsel. De transformatiematrix, zoals in (4.5) te zien is, is een eenheidsmatrix. Wiskundig wilt dit zeggen dat punten, die gedefinieerd zijn in dit assenstelsel, na transformatie niet zullen veranderen. Als het ware is dit een assenstelsel die is gedefinieerd als identiek aan de oorsprong.

$$T_1 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.5}$$

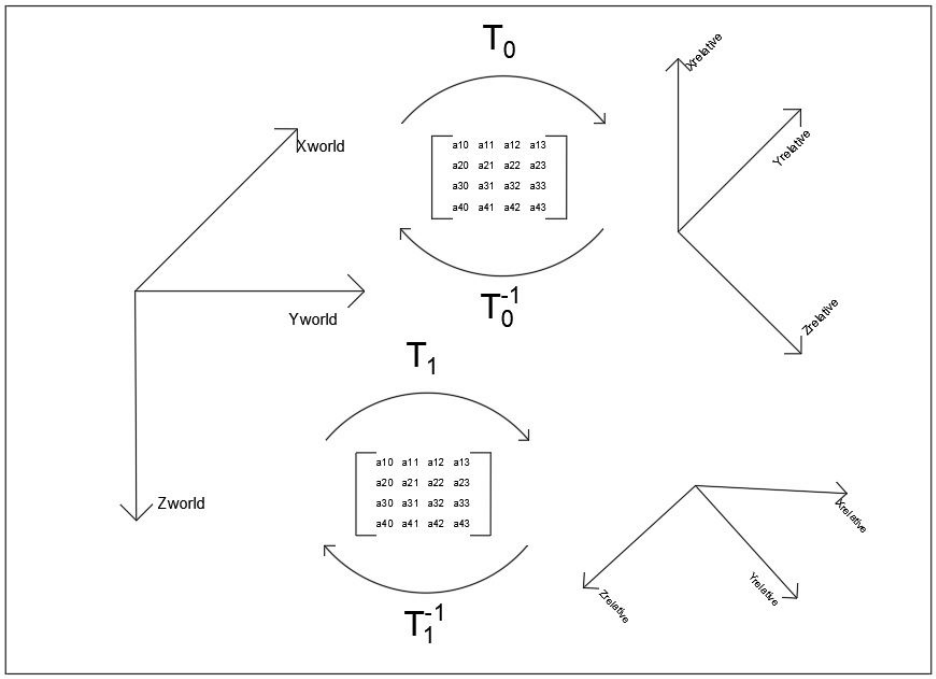
4.2.2 Definiëren van onderlinge relatie tussen assenstelsels

Een assenstelsel wordt relatief gedefinieerd door de verplaatsing en oriëntatie, ten opzichte van het absolute assenstelsel, in een transformatiematrix te steken. Deze matrix bestaat uit 4 verschillende matrices zoals beschreven in [3]. De matrices in kwestie, zoals afgebeeld in (4.6), zijn respectievelijk de rotatie rond de z-as, de rotatie rond de y-as, de rotatie rond de x-as en de translatie volgens alle assen van de oorsprong. De totale matrix stelt de wiskundige vertaling voor van een punt met relatie tot het relatieve assenstelsel naar datzelfde punt met relatie tot het absolute assenstelsel.

$$T_1 = \begin{bmatrix} cos\vartheta & -sin\vartheta & 0 & 0 \\ sin\vartheta & cos\vartheta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & cos\vartheta & -sin\vartheta & 0 \\ 0 & sin\vartheta & cos\vartheta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} cos\vartheta & 0 & sin\vartheta & 0 \\ 0 & 1 & 0 & 0 \\ -sin\vartheta & 0 & cos\vartheta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.6}$$

Punten die in het relatieve assenstelsel liggen, hebben ook een overeenkomstige coördinaat in het absolute assenstelsel. De transformatie van een punt is uniek in beide richtingen, zowel van absolute coördinaten naar relatieve coördinaten en andersom. Dit wordt berekend in (4.7), waarbij de punten voorgesteld zijn als een 1x4 matrix. Deze transformatie is visueel te zien in figuur 4.5.

$$T_1 * \begin{bmatrix} x_{rel} \\ y_{rel} \\ z_{rel} \\ 1 \end{bmatrix} <=> T_1^{-1} * \begin{bmatrix} x_{abs} \\ y_{abs} \\ z_{abs} \\ 1 \end{bmatrix} \tag{4.7}$$



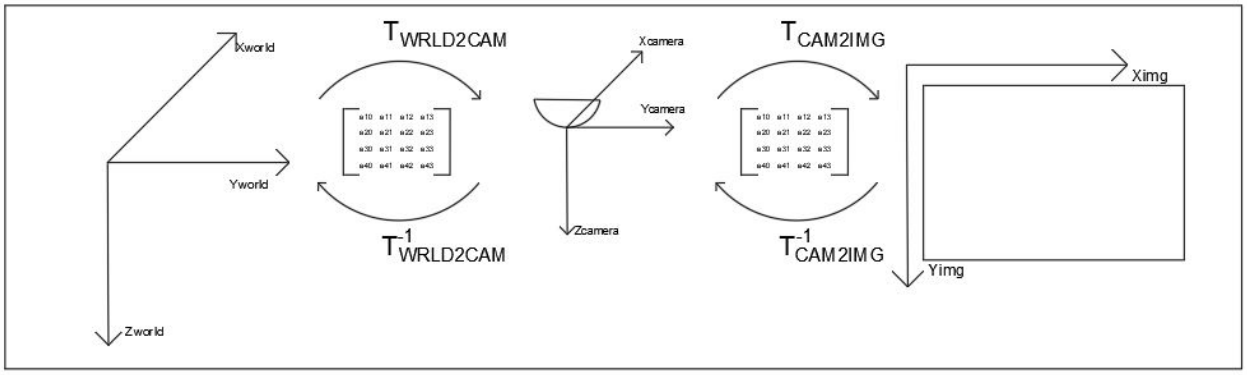
Figuur 4.5: Relatie tussen absolute en relatieve assenstelsels.

4.2.3 Definitie van assenstelsels na transformatie tussen ruimtes

Transformaties kunnen tussen assenstelsels gebeuren die behoren tot verschillende ruimtes. Een voorbeeld hiervan is de transformatie tussen de wereldruimte en de beeldruimte. Zoals beschreven in [3] moeten de intrinsieke en extrinsieke parameters van de transformatie in kaart gebracht worden. Deze parameters representeren respectievelijk de eigenschappen van de sensor en de opstelling van de sensor ten aanzien van de omgeving.

Deze eigenschappen worden benut ter transformatie van de wereldruimte naar de cameraruimte, een tussenruimte. De cameraruimte is gericht volgens het absolute assenstelsel. De eigenschappen van de camera zijn opgenomen in de intrinsieke parameters. Deze dienen om eventuele onnauwkeurigheden in het waargenomen beeld weg te werken. Verder zijn er extrinsieke parameters die de opstelling van de camera ten opzichte van het absolute assenstelsel beschrijven. In vergelijking (4.8) is de vermenigvuldiging getoond van de intrinsieke en extrinsieke eigenschappen. Het resultaat is een nieuwe 4x4-matrix, die de relatie tussen de wereldruimte en de cameraruimte legt.

De cameraruimte is strikt gezien een driedimensionale ruimte, net zoals de wereldruimte. Deze tussenruimte wordt gebruikt om de transformatie naar de beeldruimte mogelijk te maken, zoals te zien is op figuur 4.6. De transformatie is volledig eens, in de cameraruimte, de x-coördinaten en y-coördinaten van een punt gedeeld zijn door de z-coördinaat van hetzelfde punt (4.9). Zodoende worden de punten op een oppervlak geprojecteerd gelegen in de beeldruimte. Deze punten hebben enkel een x-coördinaat en y-coördinaat die relevant is voor de projectie. Deze punten worden beschouwd als tweedimensionaal. De z-coördinaat moet echter gekend blijven om de transformatie terug naar de wereldruimte mogelijk te maken.



Figuur 4.6: Relatie tussen wereldruimte en beeldruimte.

$$T_0 = \begin{bmatrix} a11 & a12 & a13 & a14 \\ a21 & a22 & a23 & a24 \\ a31 & a32 & a33 & a34 \\ a41 & a42 & a43 & a44 \end{bmatrix} * \begin{bmatrix} b11 & b12 & b13 & b14 \\ b21 & b22 & b23 & b24 \\ b31 & b32 & b33 & b34 \\ b41 & b42 & b43 & b44 \end{bmatrix} \tag{4.8}$$

$$\begin{aligned} P(x_{img}) &= \frac{P(x_{cam})}{P(z_{cam})} \\ P(y_{img}) &= \frac{P(y_{cam})}{P(z_{cam})} \end{aligned} \tag{4.9}$$

4.2.4 Toepassing van de gedefinieerde assenstelsels in de praktijk

In de praktijk zijn deze assenstelsels de spil van de toepassing. De implementatie van de assenstelsels creëert de mogelijkheid om intuïtief punten te definiëren in het systeem. De definitie kan gebeuren ten opzichte van van het absolute assenstelsel, zodat ieder punt overeenkomt met een werkelijke coördinaat in de wereldruimte. Echter kan een assenstelsel relatief gedefinieerd worden door het bijvoorbeeld te laten samenvallen met een belangrijk element in de omgeving. Dit is te zien aan de kleine assenstelsels afgebeeld op figuur 4.7. Punten in relatieve assenstelsel beschouwen de oorsprong van dit assenstelsel als het nulpunt. Deze punten worden, vóór het transformeren naar de beeldruimte, getransformeerd naar het absolute assenstelsel zoals beschreven in hoofdstuk 4.2.2.

Figuur 4.7: Werkomgeving voorzien van relatieve assenstelsels en een absoluut 3D-assenstelsel.

4.3 Relatie tussen assenstelsels, volumes en beelden

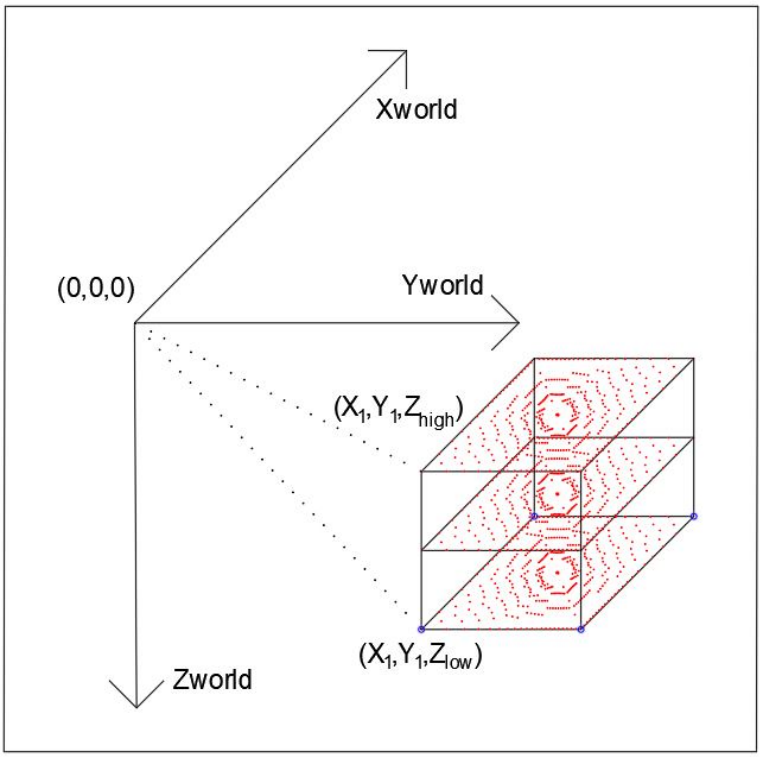
4.3.1 Relatie tussen assenstelsels en gedefinieerde volumes

De relatie tussen assenstelsels en volumes is van het *'parent-child'*-type, waarbij de assenstelsels de rol van *'parent'* en de volumes de rol van *'child'* dragen. Dit houdt in dat een assenstelsel verschillende volumes kan bezitten, die op hun beurt behoren tot hetzelfde assenstelsel. De volumes en assenstelsels zijn onlosmakelijk van elkaar te beschouwen. Een volume zonder assenstelsel is niet te definiëren en een assenstelsel waarin geen volumes geplaatst zijn, heeft geen functie te vervullen.

Een volume bestaat telkens uit een groot aantal gediscretiseerde punten. Deze punten, die tot de wereldruimte behoren, dienen ter detectie van handelingen. Deze handelingen zijn enkel meetbaar in de beeldruimte. Het leggen van de link tussen de volumes en de gemeten data is niet mogelijk zonder een transformatie tussen deze ruimtes, waarin gekozen is om de volumes van de wereldruimte naar de beeldruimte (met dieptewaarden) te transformeren. Hier komen de belangen van de assenstelsels aan het licht, want het zijn juist deze assenstelsels die als relatie dienen tussen de verschillende ruimtes. Door de relatie tussen ruimtes te definiëren, is in ene weg ook de relatie tussen de volumes en de meetwaarden gelegd.

Indien volumes in een relatief assenstelsel gedefinieerd zijn, kunnen deze een dynamisch karakter hebben. Door de transformatiematrix tussen het relatieve en absolute assenstelsel aan te passen, verschuiven alle punten, gedefinieerd in dit relatief assenstelsel, mee op. Voor de transformatie van punten van wereldruimte naar beeldruimte zijn 2 matrices van belang. Dit zijn de 'absolute' matrix met de intrinsieke en extrinsieke parameters, die de transformatie tussen het absolute assenstelsel en de beelruimte voorziet, en de 'relatieve' matrix , die de transformatie voorziet tussen het relatieve en absolute assenstelsel.

In figuur 4.8 is te zien hoe de coördinaten van een volume gedefinieerd zijn ten opizchte van het absolute assenstelsel.



Figuur 4.8: Relatie tussen assenstelsels en volumes.

4.3.2 Transformatie van volumes tussen verschillende ruimtes

De volumes, die in het absolute assenstelsel gedefinieerd zijn, worden in één weg vertaald naar de beeldruimte. Dit is een gevolg van het leggen van de relatie tussen beeldruimte en wereldruimte aan de hand van de intrinsieke en extrinsieke eigenschappen. De coördinaten van de volumes, die getransformeerd worden naar de beeldruimte, wijzigen naar een plaats die gedefinieerd is volgens twee dimensies. De wijziging van de coördinaten wordt in ieder punt van het volume doorgevoerd, maar niet voor ieder component van het punt. Zo zullen de x- en y-component wijzigen maar zal het z-component ongewijzigd blijven. De z-waarde, die overeenkomt met de diepte-waarde van de sensor, is reeds gedefinieerd in millimeter. Dit is inherent compatibel met de beeldruimte, welke dieptemetingen doet in millimeter.

4.3.3 Functie van de gedefinieerde volumes in de beeldruimte

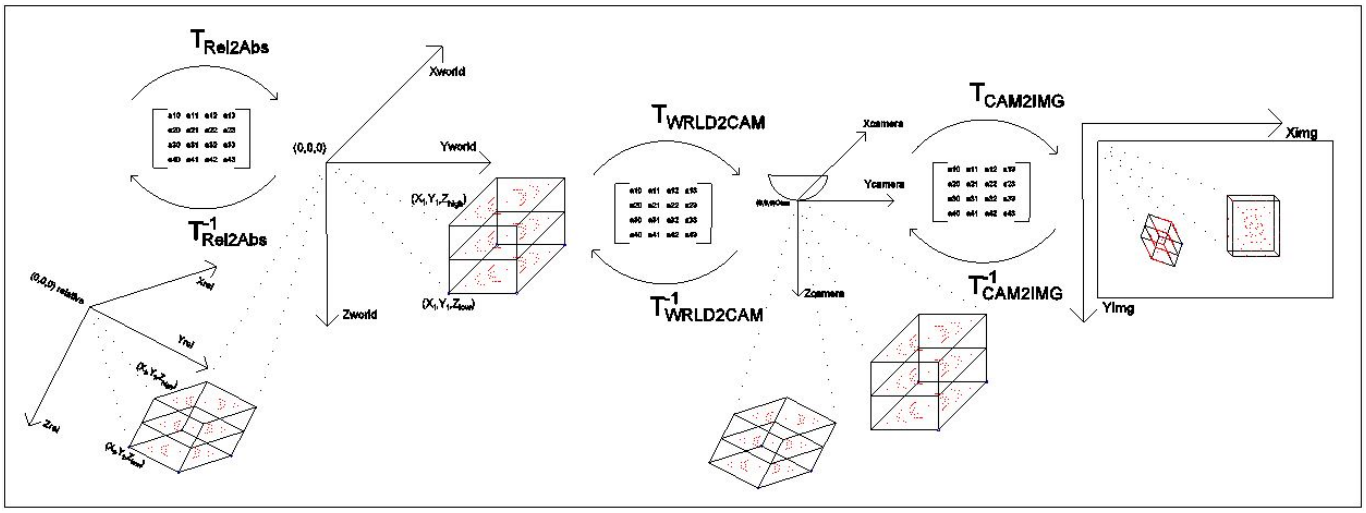
Eens het volume getransformeerd is naar de beeldruimte, worden de x- en y-componenten van ieder punt gezien als de representatie van een pixel op het beeld. De z-component heeft geen representatie in een strikt tweedimensionale omgeving, maar wordt als grijswaarde toegewezen aan pixels. In dit systeem is de z-component, die onveranderd blijft door de transformatie, zeer belangrijk om handelingen te detecteren aangezien deze de dieptewaarde voorstelt.

Hetzelfde wordt gesteld over het beeld dat waargenomen wordt door de ToF-sensor, waarbij de x- en y-componenten van ieder punt wederom een pixel op het beeld voorstellen. Omdat de grijswaarde fungeert als dieptewaarde, wordt het waargenomen beeld als driedimensionaal beschouwd.

De dieptedata van zowel de volumes als het dieptebeeld wordt gecombineerd om handelingen te detecteren. Zoals gesteld stellen zowel de punten in de volumes als de punten op het beeld pixels

voor met hun x- en y-componenten. De dieptewaarde van de volumepunten worden vergeleken met de dieptewaarde van de sensormeting op overeenkomstige pixel. Overeenkomsten tussen beide z-waarden wijst op de aanwezigheid van een lichaam op dit punt.

In figuur 4.9 is te zien hoe de transformaties plaatsvinden van 2 volumes tussen wereldruimte en beeldruimte, gebruikmakend van de cameraruimte als tussenruimte. Van de volumes is 1 relatief en 1 absoluut gedefinieerd.



Figuur 4.9: Relatie tussen assenstelsels, volumes en ruimtes.

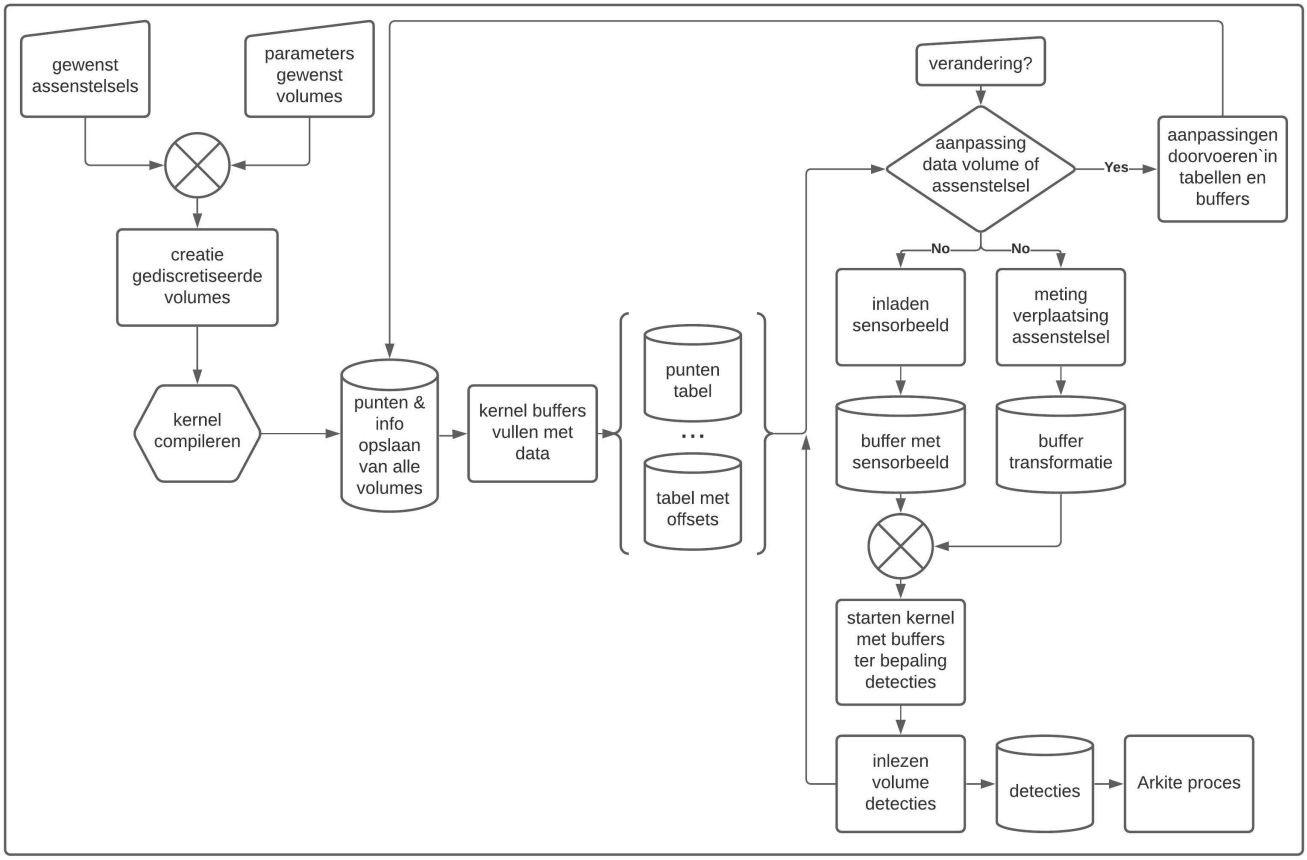
4.4 Toepassing van de verschillende elementen ter detectie van interactie met gedefinieerde volumes

Nu is bekend hoe de link wordt gelegd tussen de z-waarde van punten van volumes de overeenkomstige gemeten z-waarden van de sensor. De volgende stap implementeerd een algoritme om de vergelijkingen van deze dieptewaarden efficiënt volbrengt. Hierbij wordt aanspraak gedaan op de GPU met OpenCL.

Het OpenCL-platform in combinatie met de GPU wordt hierbij gezien als een *accelerator*; een component ter versnelling van het uitvoeren van bewerkingen. De detectie kan perfect functioneren op de CPU, maar voor dit systeem is het in snelheid enorm beperkt, omdat de CPU weinig geschikt is voor berekeningen parallel te verrichten. Dit komt door het beperkt aantal kernen die de CPU bezit. De CPU zou heel veel punten moeten vergelijken in serie, terwijl deze ook andere, ongerelateerde taken moet afhandelen. Zoals reeds vermeld, bezit de GPU een groot aantal kernen waar gebruik van wordt gemaakt door het platform om het detectieproces te versnellen.

4.4.1 Opbouw van het detectiemodel

Figuur 4.10 geeft het verloop weer van het creëren van de data en de real-time loop ter detectie van volumes.



Figuur 4.10: Flowchart van het detectiealgoritme.

4.4.2 Detectiebepaling en nauwkeurigheid

Zoals reeds gemeld, wordt een actie gedetecteerd als voldoende punten van het volume de aanwezigheid van een lichaam meten. Dit om ervoor te zorgen dat kleine lichamen, zoals afval of een vast deel van het werkvlak, geen valse detectie van acties genereren. Het minimum aantal detectiepunten wordt bepaald aan de hand van een vulgraad. Dit is een factor van het totaal aantal punten. Deze factor wordt 'vulfactor' genoemd.

Figuur 3.3 toont dat de sensor driedimensionale punten meet van een oppervlakte en niet een volume. Door deze eigenschap dient, voor met vulfactoren te kunnen werken, eerst het volume geprojecteerd te worden naar een oppervlak. Dit gebeurt tijdens de constructie van een volume, waarbij een oppervlakte wordt gecreëerd, aan de hand van de hoekpunten, die gestapeld wordt volgens de hoogte om een volume te verkrijgen. Hierdoor zijn het totaal aantal punten van het oppervlakte en het volume bekend. Door het aantal oppervlaktepunten te delen door het aantal volumepunten, wordt de factor bepaald om, van het totaal aantal punten van het volume, het totaal aantal punten van het oppervlakte te bepalen. Met een andere, zelfgekozen factor wordt bepaald hoeveel punten van dit oppervlakte een meting moeten geven voor een actie te detecteren. Door beide factoren met elkaar te vermenigvuldigen tot de 'vulfactor' en dit te vermenigvuldigen met het totaal aantal punten van het volume, wordt bepaald hoeveel punten gedetecteerd moeten zijn om detectie van een volume te bekomen. Deze benadering is voldoende binnen de context van de masterproef.

Dit systeem werkt perfect indien de z-assen, van het assenstelsel van het volume en de sensor,

gelijk liggen. Indien dit niet het geval is, wordt het oppervlakte verkeerd geprojecteerd en is de vulfactor fout. De vulfactor is dus een benadering voor het berekenen van de vulgraad, die gedefinieerd is als het aantal punten dat gedetecteerd wordt binnen het volume, gedeeld door het maximale aantal punten dat de sensor ziet binnen dit volume.

Om dit eenvoudig op te lossen kan Arkite gebruik maken van een vulfactor welke opgegeven wordt door de operator. De operator kan een waarde kiezen (bijvoorbeeld met een slider) en testen of deze factor voldoet. Voor dynamische assenstelsels kan 1 factor gekozen worden die geschikt is voor de meerdere mogelijke oriëntaties. Dit vergt wat meer tijd voor de operator.

Als conclusie kan gesteld worden dat, indien sterk asymmetrische volumes in verschillende oriëntaties afgevraagd worden en reeds veel punten gedetecteerd zijn als er geen actie plaatsvindt, dit systeem niet voldoet. In dit geval zouden meerdere vulgraden gebruikt kunnen worden die aangesproken worden afhankelijk van de oriëntatie van het volume. Indien zeer weinig punten detectie doorgeven als er geen actie plaatsvind, voldoet een zeer lage vulfactor altijd.

4.4.3 Toepassing van het detectiemodel in de praktijk

Volumes dienen voor de bepaling of instructies gevolgd zijn. De HIM volgt het productieproces stap voor stap en geeft informatie en instructies bij elke stap. Instructies zijn bijvoorbeeld "stap 1: neem object a", "stap 2: draai schroef b vast", enzovoort. Met de projector wordt aangegeven waar de relevante plek zich bevind binnen de werkplek door deze op te lichten. Ten alle tijden worden alle volumes afgevraagd op detectie, maar voor het bepalen of de actie opgevolgd is, is enkel het volume rond de relevante plek belangrijk. In het voorbeeld wordt de detectie van het volume rond object 'a' in het oog gehouden. Indien hier een detectie plaatsvind, wordt ervan uitgegaan dat object 'a' genomen is. De HIM weet nu dat de arbeider aan de volgende stap begint en bekijkt het volume rond 'schroef b'. Indien hier een detectie plaatsvind, weet de HIM dat de schroef aangedraaid wordt. Als de detectie wegvalt, gaat de HIM ervan uit dat de schroef vastgedraaid is en begint volgende stap.

Hoofdstuk 5

Detectie algoritme en optimalisatie strategieën

5.1 Het OpenCL-platform als accelerator

OpenCL stelt compatibele hardware, zoals de GPU, in staat om taken parallel uit te voeren op voorziene data. In deze toepassing wordt gebruik gemaakt van deze eigenschap om dezelfde taken uit te voeren op verschillende stukken data in parallel. De te verwerken data, die aan het platform aangeboden wordt, is enkel de data die van belang zijn voor het uitvoeren van het detectiealgoritme.

Het detectiealgoritme wordt verder ''kernel'' genoemd en bestaat enkel uit een set instructies die aan de GPU aangeboden wordt ter uitvoering.

5.1.1 Opbouw van het OpenCL-platform

OpenCL voorziet een gemeenschappelijk taal, programmeerinterface en hardware abstractie die ontwikkelaars toelaat om toepassingen te versnellen door dataparallel (of *'taskparallel'*) te werk te gaan. Dit allemaal in een heterogene, *'cross-platform'*, omgeving, bestaande uit een *'host'*-CPU en de daarmee verbonden *'OpenCL-devices'* [4]. Van deze devices wordt, in deze thesis, de GPU van de HIM gekozen. Heterogeniteit is bereikt door het ondersteunen van verschillende soorten hardware, die aan de OpenCL standaard voldoet, geproduceerd door verschillende partijen. Er kan verzekerd worden dat een *'kernel'*, geschreven voor het ene device, zal werken voor het andere device. Er kan echter geen garantie gegeven worden over de performantie van de *'kernel'* op andere hardware.

De *devices* kunnen gebruik maken van hun eigen geheugen, indien beschikbaar, of een deel van het RAM van de *host* gebruiken. Het geheugen, beschikbaar voor de GPU, wordt volgens 4 niveaus ingedeeld op basis van de toegankelijkheid. Dit is gevisualiseerd op figuur 5.1.

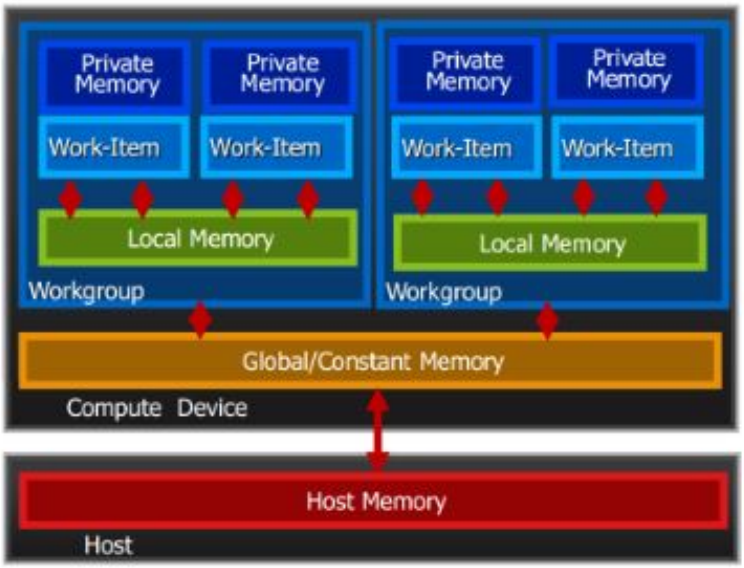
Het globaal geheugen is toegankelijk door alle werkgroepen, waarbij elke parallelle dataverwerking wordt gedaan door 1 werkgroep. Dit geheugen is de traagste van de 4 types, maar het meest toegankelijke. Het te lezen van en schrijven naar dit geheugen gaat gepaard met een grote overhead.

Het lokaal geheugen is bestemd voor gebruik van 1 werkgroep. Dit geheugentype kan wederom

geschreven en gelezen worden, maar met een kleinere overhead. Als binnen de werkgroep verschillende verwerkingseenheden gedefinieerd zijn, kunnen deze eveneens beroep doen op dit geheugen. Elke werkgroep kan namelijk verder opgedeeld zijn in stukken parallelle dataverwerking.

De verwerkingseenheden kunnen evengoed een eigen geheugenplek toegewezen krijgen. Dit geheugen heet het privaat geheugen en enkel het verwerkingselement kan aanspraak doen hierop. Door deze afscherming te voorzien kan data beschermd worden door wijzigingen van buitenaf.

Het laatste geheugentype is het constante geheugen. Dit geheugen kan enkel gelezen worden, waardoor de toegang veel sneller is aangezien de data in een *cache* opgeslagen wordt.



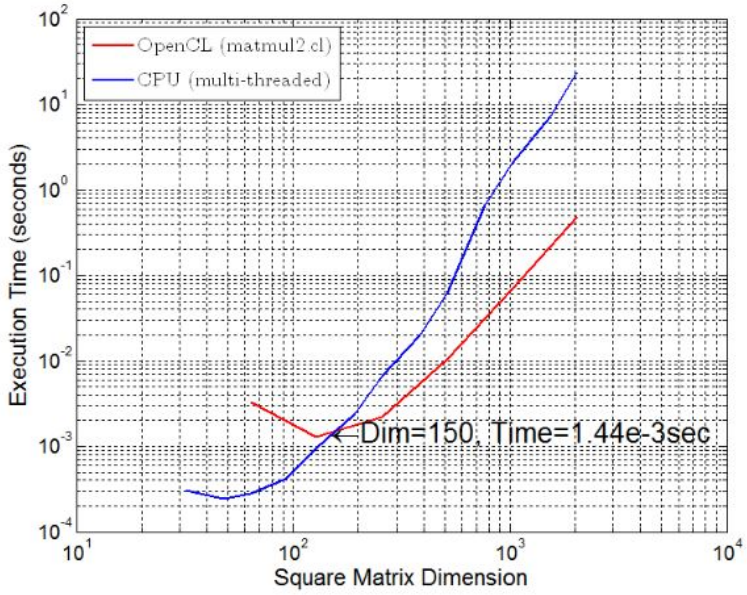
Figuur 5.1: OpenCL geheugenmodel [13].

5.1.2 De architectuur van de GPU

Het gebruikte type GPU heeft geen eigen werkgeheugen en gebruikt daarom de RAM van de *host* (de CPU). Het *device* (de GPU) kan maximaal de helft van het werkgeheugen van de *host* innemen om de nodige berekeningen uit te voeren.

Verder beschikt de GPU over 192 kernen die, voor dit systeem, allen als een aparte verwerkingseenheid beschouwd worden. Zoals eerder gesteld, zal deze GPU identieke bewerkingen uitvoeren op verschillende data om het detectie-algoritme tot stand te brengen.

De performantie van een GPU in vergelijking met CPU hangt sterkt af van de hoeveelheid data die verwerkt wordt. Over het algemeen wordt gesteld dat een CPU een hogere kloksnelheid bezit dan een vergelijkbare GPU van dezelfde generatie. Deze kloksnelheid stelt de CPU in staat om veel sneller een set instructies af te werken. De CPU is echter beperkt in de hoeveelheid data die parallel verwerkt kan worden. De conclusie uit [13] stelt dat het gebruik van OpenCL voor de combinatie van CPU en GPU, enkel gerechtvaardigd is indien de overhead relatief klein is. In figuur 5.2 wordt aangetoond dat het enkel wijs is om de GPU berekeningen te laten uitvoeren bij een dataset die voldoende groot is. Zodoende zal de overhead een steeds kleinere rol spelen in de performantie van het systeem.



Figuur 5.2: OpenCL: vergelijking CPU en GPU, performantie [13].

5.1.3 Gebruik van OpenCL ter acceleratie

De implementatie van OpenCL begint met het installeren van de OpenCL *'library'* binnen het project naar wens. Deze installatie wordt hier niet behandeld.

De bibliotheek dient op gepaste wijzen gelinkt te worden aan het project, zodat de *'compiler'* weet waar de nodige bestanden gelegen zijn om het programma foutloos te compileren. Eens de bibliotheek in het project geïmporteerd is, kan de *'set-up'* van start gaan.

Om het geheel leesbaar te houden, wordt alles wat betrekking heeft tot OpenCL in een aparte klasse geschreven binnen C++. Zodoende wordt in het programma een object aangemaakt waarop beroep gedaan wordt om de nodige OpenCL gerelateerde methodes op te roepen. Het object bestaat uit een *'header'* en een *'source'* die beiden in de bijlage toegevoegd zijn.

5.1.4 Opzetten van de nodige datastructuren

De *'set-up'* van OpenCL gebeurd in verschillende stappen, waarin de hardware en het platform gespecificeerd worden. Het is met bepaalde hardware mogelijk om bijvoorbeeld met het CUDA platform van Nvidia te werken. Er wordt voor OpenCL gekozen omwille van de onderlinge compatibiliteit met de gebruikte GPU en elke GPU die voldoet aan de OpenCL standaard. Hierdoor is Arkite niet afhankelijk van 1 specifieke GPU bouwer.

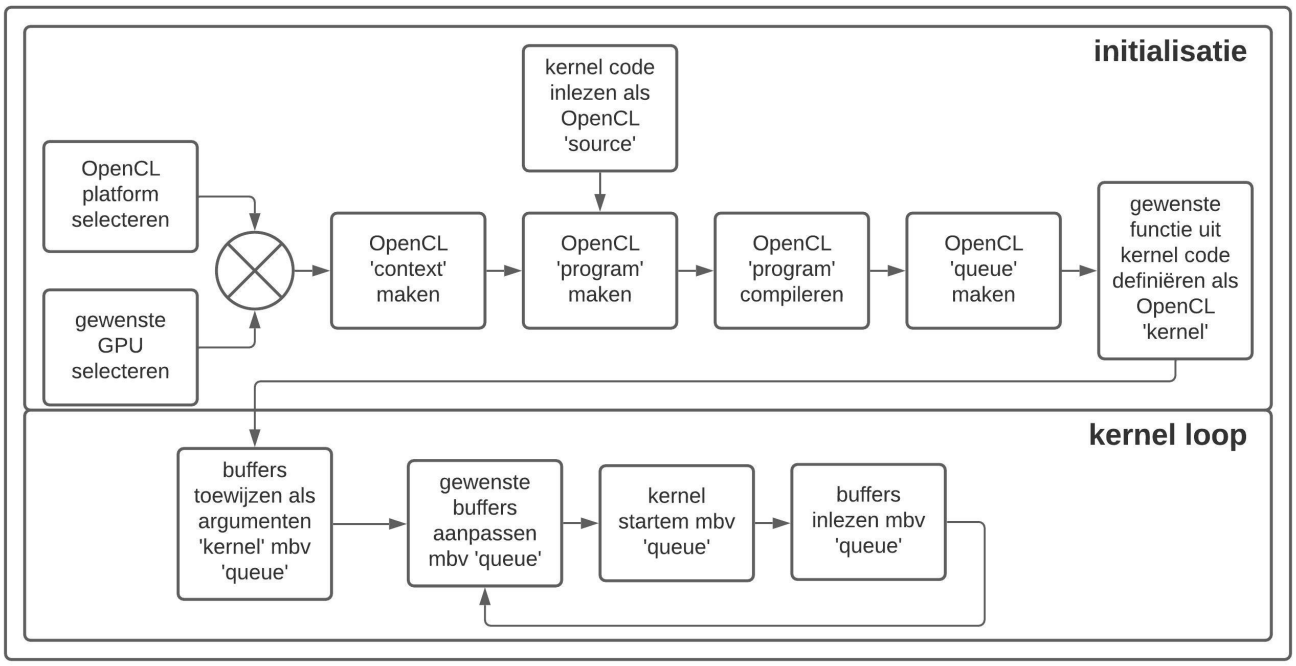
In eerste instantie worden alle mogelijke platformen ingeladen en wordt OpenCL geselecteerd. In de volgende stap worden alle *'devices'* ingeladen en wordt de gewenste GPU gekozen (Intel Graphics 530). Bij het inladen wordt een lijst gevormd met de mogelijke devices die bestaat uit: CPU, interne GPU en eventueel externe devices indien aanwezig. De platform en het device worden gelinkt door een 'context' aan te maken, dit is een object eigen aan OpenCL.

Vervolgens wordt door middel van een 'filestream', een object in C++ voor het lezen van en schrijven naar bestanden, de ''kernel''-code ingelezen. De ''kernel'' is nu eenmaal in een apart bestand gedefinieerd om leesbaarheid te waarborgen. Dit bestand wordt regel per regel gelezen door de 'filestream' en in een 'source' ingeladen, wederom een OpenCL object.

De context en de source worden gelinkt en het programma wordt gecompileerd. Door deze compilatie wordt het programma omgezet in machine code die uitgevoerd wordt door de GPU.

In de laatste stap wordt een 'queue' aangemaakt, een wachtrij voor commando's, op basis van dezelfde platform en device.

Met al deze elementen in plaats wordt de ''kernel'' gecreëerd die uitgevoerd wordt wanneer opgeroepen. Het proces hier beschreven staat samengevat in de *'flowchart'* op figuur 5.3.



Figuur 5.3: Flowchart van de OpenCL *'set-up'*.

5.1.5 Data toewijzen aan buffers ter calculatie

De data waarmee de *'kernel'* werkt, wordt in de vorm van buffers aangeboden. De buffers hebben verschillende parameters die hun werking bepaald.

De eerste parameter heeft betrekking tot de context. De buffer moet namelijk weten voor welke device de data vrijgesteld wordt. Vervolgens worden een aantal *'flags'* vastgezet die de vrijheden van zowel de host als de device bepalen met betrekking tot de geheugenplek van de buffer. Zo kan de rol van de host bestaan uit enkel lezen of schrijven, of beide en idem voor het device. Deze flags zijn een bepalende factor voor de snelheid waarmee geïnteracteerd wordt met de data in de buffer. In de laatste stap wordt de grootte van de te alloceren geheugen bepaald en de specifieke data die eventueel in dat geheugen geschreven wordt. Eens de buffer volledig is, wordt deze toegekend aan een argument van de *'kernel'*.

Indien de data van een buffer niet meteen beschikbaar is, wordt deze op een later tijdstip in de buffer geschreven. Het is belangrijk dat de buffer al geïnitialiseerd is en dat het nodige geheugen gealloceerd is. Zodoende kan op een later tijdstip data in de buffer geschreven worden. Dezelfde methode werkt ook om data uit een buffer te lezen. Beide gevallen zijn onder voorbehoud dat de buffer de *'host'* de nodige toelatingen verschaft bij het definiëren van de buffer. Een voorbeeld hiervan is in de bijlage terug te vinden in bijlage A.1.3.

5.1.6 Opbouw van een kernel

De *'kernel'* bezit de instructies die uitgevoerd worden door de GPU geschreven in een specifieke taal welke sterk gerelateerd is aan C.

De *'kernel'* wordt stap voor stap doorlopen, zoals het geval is bij de CPU, met als grote uitzondering dat de *'kernel'* gelijktijdig door meerdere werkgroepen uitgevoerd wordt.

Het grote nadeel aan de *'kernel'* te laten uitvoeren door de GPU is dat de *'compiler'* van de GPU niet zo geoptimaliseerd is als de compilers voor de CPU. Zo zal de CPU code typisch efficiënter gecompileerd wordt bij het omzetten naar machinetaal. De OpenCL compiler van Intel introduceert zelf optimalisaties, maar verwacht dat veel van deze optimalisaties opgeroepen worden tijdens het compilen van de code aan de hand van instructies gegeven door de programmeur. Dit vergt de nodige aandacht omtrent het opstellen van de *'kernel'*, zoals beschreven in [6].

5.2 Kernel van het detectie-algoritme

De werking en opbouw van het detectiealgoritme wordt hier besproken. Dit is het finale, geoptimaliseerde systeem. Het optimalisatieproces, beginnend met de ongeoptimaliseerde *'kernel'*, wordt verder in hoofdstuk 5.3.1 uitgelegd.

5.2.1 Buffers bepalen

Tijdens het creëren van de volumes zijn verschillende tabellen en waarden opgeslagen die relevant zijn voor het detectiealgoritme. De tabellen van alle volumes worden achter elkaar in globale tabellen gezet. Hier volgt een opsomming van alle buffers van de GPU, welke bevoegdheden deze hebben en hoe bepaald wordt welke informatie behoort tot welk volume, aangezien de GPU enkel werkt met basis datatypen (boolean, integer, short, float, ...):

- table: De puntentabellen van alle volumes worden achter elkaar geplaatst in deze buffer. Hierbij bestaat elk punt uit 3 shorts; 1 per dimensie. De GPU mag hier enkel uit lezen en de CPU heeft hiertoe geen toegang.
- tableOffset: Tijdens creëren van 'table' wordt telkens het aantal hoekpunten en het totaal aantal punten per volume toegevoegd aan deze buffer op cumulatieve wijze, beginnend vanaf 0. Zo behoren elke 3 opeenvolgende waarden tot eenzelfde volume en zijn respectievelijk het beginpunt, het eindpunt van de hoekpunten en het eindpunt van dat volume in 'table'. De GPU mag hier enkel uit lezen en de CPU heeft hiertoe geen toegang.
- detectionSteps: Per volume was een aantal resolutie stappen gekozen en het aantal punten per resolutiestap werd opgeslagen. Op cumulatieve wijze per volume worden deze aantallen toegevoegd aan deze buffer. Praktisch gezien worden de eindpunten van elke resolutiestap gegeven, relatief ten opzichte van het volume. De GPU mag hier enkel uit lezen en de CPU heeft hiertoe geen toegang.
- amountDetectionSteps: Omdat in 'detectionSteps' het aantal resolutiestappen variabel is per volume, dient gekend te zijn hoeveel resolutiestappen elk volume heeft. Deze buffer houd dat cumulatief bij beginnend vanaf 0. Zodoende wordt de waarde horende bij het vorige volume afgetrokken van de waarde horende bij het huidige volume om het aantal detectiestappen te bekomen. De reden waarom het aantal resolutiestappen per volume hier

niet direct in opgeslagen is, is omdat dit getal+1 nu direct overeenkomt met de positie van de overeenkomstige resolutiestap in 'detectionSteps'. De GPU mag hier enkel uit lezen en de CPU heeft hiertoe geen toegang.

- drawBuffer: Hierin worden de tweedimensionale hoekpunten, of *'vertices'*, opgeslagen van het volume getransformeerd naar de beeldruimte als 2 shorts. Dit vormt belangrijke informatie voor het visualiseren van de volumes, wat bereikt wordt als alle hoekpunten geconnecteerd worden met een lijn op een bepaalde volgorde. De GPU doet de transformatie van deze punten aangezien de volumes dynamisch kunnen zijn en schrijft het resultaat in deze buffer ter gebruik voor de CPU. De GPU mag hier enkel toe schrijven en de CPU mag hier enkel uit lezen.
- offsetVertices: Het aantal hoekpunten per volume wordt cumulatief bijgehouden in deze tabel om te bepalen welke hoekpunten in 'drawbuffer' horen bij welk volume. De GPU mag hier enkel uit lezen en de CPU heeft hiertoe geen toegang.
- transformationDyn: Elk assenstelsel heeft een transformatiematrix voor de transformatie van relatief naar absoluut assenstelsel. Indien een assenstelsel verplaatst wordt, veranderen de waarden in de matrix en verplaatst elk punt zich. De GPU dient de nieuwe overeenkomstige posities te bepalen van deze punten op de beeldruimte. Deze tabel heeft de eventueel veranderende matrix van elk assenstelsel per assenstelsel. In deze buffer vormen elke 16 opeenvolgende float waarden vormen een assenstelsel. De bufferwaarden kunnen worden aangepast in real-time. De GPU mag hier enkel uit lezen en de CPU mag hier enkel toe schrijven.
- IDax: Deze buffer onthoud, per volume, de ID van het assenstelsel behorende tot dat volume. Zo wordt het volume gelinkt met de bijbehorende transformatiematrix uit 'transformationDyn'. De GPU mag hier enkel uit lezen en de CPU heeft hiertoe geen toegang.
- matrixWorldToImage. De matrix die de transformatie bezit van het absolute assenstelsel in de wereldruimte naar de beeldruimte als 16 float waarden. De GPU mag hier enkel uit lezen en de CPU heeft hiertoe geen toegang.
- fillFactor. De vulfactor behorende tot elk volume. De GPU mag hier enkel uit lezen en de CPU heeft hiertoe geen toegang.
- heightStep. De afstand tussen opeenvolgende punten van elk volume. De GPU mag hier enkel uit lezen en de CPU heeft hiertoe geen toegang.
- heightWidth. De resolutie van het sensorbeeld. De GPU mag hier enkel uit lezen en de CPU heeft hiertoe geen toegang.
- frame. Het sensorbeeld van pixels met bijbehorende grijswaarde als maat voor de diepte. Elk punt bestaat uit 3 shorts. Bij elk gecapteerd sensorbeeld wordt deze buffer aangepast. De GPU mag hier enkel uit lezen en de CPU mag hiet enkel toe schrijven.
- detections. De tabel die per volume aangeeft of er detectie is of niet. De GPU mag hier enkel toe schrijven en de CPU mag hier enkel uit lezen.

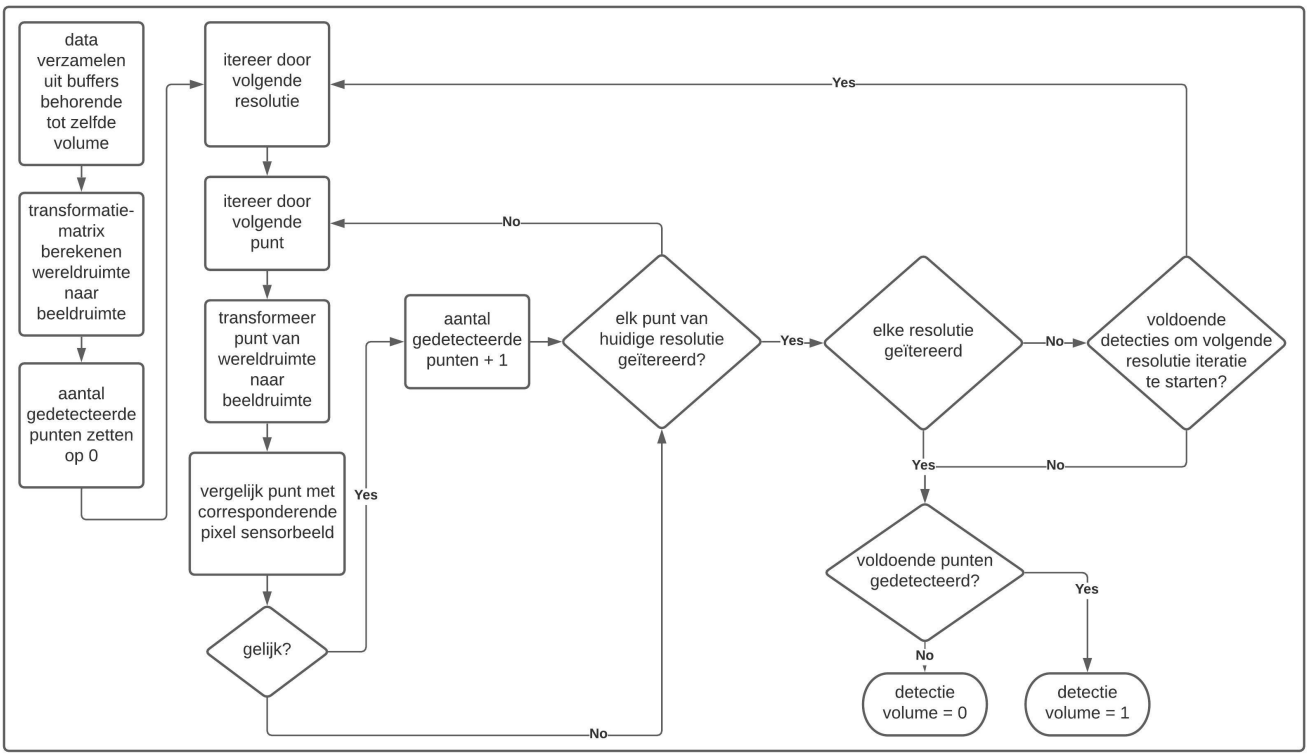
De buffers waarbij niets vermeld is over welke buffer waarde bij welk volume hoort, zijn dusdanig ingedeeld dat opeenvolgende waarden behoren tot opeenvolgende volumes. De buffers vormen de argumenten van de functie 'detection' in de kernel.

```
void kernel detect
(
        constant short* restrict frame, //0
        constant short* restrict table,//1
        global char* detections,//2
        constant int* restrict tableOffset,//3
        constant short* restrict heightWidth,//4
        constant float* restrict fillFactor,//5
        constant float* restrict transformationDyn,//6
        constant int* restrict IDax, //7
        global short* drawBuffer, //8
        constant short* restrict offsetVertices, //9
        constant int* restrict detectionSteps, //10
        constant int* restrict amountDetectionSteps, //11
        constant float* restrict matrixWorldToImage, //12
        constant short* restrict heightStep //13
)
```

5.2.2 Detectie-algoritme

Elk volume wordt toegewezen aan 1 werkgroep met 1 verwerkingseenheid, waardoor elke werkgroep het algoritme uitvoert. In hoofdstuk 5.2.1 is reeds besproken hoe waarden van de buffers toegewezen worden aan het bijbehorend volume. Concreet wordt dit gedaan aan de hand van het 'global_id', een unieke, incrementele ID die elke werkgroep heeft, beginnend vanaf 0. In dit geval krijgt elk volume dus een eigen 'global_id'.

De kernel, die verder uitgelicht is in hieronder, is samengevat in een flowchart op figuur 5.4. Hierbij is de flow van 1 werkgroep te zien.



Figuur 5.4: Flowchart van de OpenCL kernel.

Als eerste stap in het algoritme wordt bepaald welke 'global_id' (globalid) hoort bij de de huidige werkgroep, of het huidige volume. Ook wordt de afstand tussen opeenvolgende punten (step) van het huidige volume ingeladen, alsook de breedte (width) en hoogte (height) van het frame in pixels.

```
int globalid = get_global_id(0);
short step = heightStep[globalid];
short height = heightWidth[0];
short width = heightWidth[1]
```

Vervolgens wordt de index van het eerste punt (start), laatste punt van de hoekpunten (corner) en laatste punt (end) van het huidige volume in 'table' ingeladen. Hieruit wordt berekend hoeveel hoekpunten (amountCornerPoints) en punten (totalPoints) het volume bevat.

```
//iterators for first point of table with points belonging to current
    polygon, as well as last point of corner and last point of table.
int end = tableOffset[globalid * 3 + 2] * 3;
int corner = tableOffset[globalid * 3 + 1] * 3;
int start = tableOffset[globalid * 3] * 3;
int totalPoints = end - start;
int amountCornerPoints = (corner - start) / 3;
```

De index van de eerste resolutiestap (amountStepsStart) in 'detectionSteps' en het aantal resolutiestappen (amountDetectionSteps) wordt bepaald. Een kopie van 'amountDetectionSteps' wordt gezet in 'j'. De huidige resolutiestap ('currentStep') wordt bepaald, welke nu de eerste stap is. Het startpunt van de huidige resolutiestap (startStep) wordt bepaald, welke nu gelijk

is aan 'start'. De afstand tussen opeenvolgende punten is afhankelijk van 'step' en het aantal resoluties, dus wordt dit berekend als 'currentHeightStep'.

```
//resolution steps.
int      amountStepsStart = amountDetectionSteps[globalid];
int amountSteps = amountDetectionSteps[globalid + 1] -
   amountDetectionSteps[globalid];
//point offsets, from box start, of each resolution.
int j = amountStepsStart;
int currentStep = detectionSteps[j] * 3 + corner;
int startStep = start;
int currentHeightStep = step * amountSteps;
```

Daarna worden de matrices 'transformationDyn' en 'matrixWorldToImage', van het bijhorende assenstelsel, matrixgemultipliceerd om de totale transformatiematrix (transformation) te bekomen van de 3D-punten naar de bijhorende pixels in het sensorbeeld. Elke matrix heeft 16 waarden, dus kijkt elk volume naar de 16 waarden horend bij hun 'global_id'.

```
//refer to the axis which acts as the volume's parent.
int IDaxs = IDax[globalid];

//get transformationmatrix values by transforming dynamic world
   points to image points.
float transformation[16];
#pragma unroll 4
for (int i = 0; i < 4; i++)
{
        #pragma unroll 4
        for (int j = 0; j < 4; j++)
        {
                //matrixmultiply dynamic transformation (combination
                   of relative coordinate system's matrix and it's
                   transformation) and world to image matrix.
                transformation[4*i+j] = matrixWorldToImage[(IDaxs*16)
                    + i*4] * transformationDyn[(IDaxs*16) + j] +
                        matrixWorldToImage[(IDaxs*16) + i*4 + 1] *
                           transformationDyn[(IDaxs*16) + j + 4] +
                        matrixWorldToImage[(IDaxs*16) + i*4 + 2] *
                           transformationDyn[(IDaxs*16) + j + 8] +
                        matrixWorldToImage[(IDaxs*16) + i*4 + 3] *
                           transformationDyn[(IDaxs*16) + j + 12];
        }
}
```

Vervolgens wordt het minimum aantal te detecteren punten berekend om te stellen dat een actie plaatsvindt in het volume. De factor van het aantal te detecteren punten van de huidige resolutie ('factor'), om naar de volgende resolutie te mogen, wordt berekend.

```
//fillFactor detection.
```

```
float factor = fillFactor[globalid];
//determine minimum points to detect with fillFactor and point amount
    .
int pointsToDetect = totalPoints * factor;
//determine fillFactor for resolutions on basis of ratio points in
    current resolution and total points -> make it smaller to not get
    false negatives due to the
//higher resolution's lower accuracy.
float resolutionFactor = .25;
factor = (((float)(currentStep - start) / (float)totalPoints)) *
    resolutionFactor;
```

De index van de hoekpunten van dit volume in 'drawBuffer', wordt ingeladen. Het aantal gedetecteerde punten (detection) wordt gezet op 0, alsook 'detections'.

```
int offsetVertex = offsetVertices[globalid];

//reset detections.
detections[globalid] = 0;
int detection = 0;
```

Dan wordt een *'loop'* gestart die itereert totdat het aantal resolutiestappen bereikt zijn, waarin een *'loop'* start die itereert totdat het eindpunt van de huidige resolutiestap bereikt is. Zodoende kan elk punt overlopen worden. Bij elke iteratie wordt het een volgende punt uit 'table' genomen en matrix vermenigvuldigd met de berekende totale transformatiematrix. Hierbij worden de berekende x- en y-waarde gedeeld door de z waarde, waardoor de pixel, behorende bij het punt, bepaald is. Vervolgens wordt de overeenkomstige pixel in 'frame' ingeladen. De z-waarden van het punt en de pixel worden vergeleken. Aangezien dit hoogteverschil binnen 'currentStepHeight' mag zijn, worden beide hoogten gedeeld door 'currentHeightStep'. Indien beide z-waarden minder dan 1 'currentHeightStep' en niet minder dan 0 van elkaar af liggen, krijgen deze dezelfde z-waarde. Dit omdat integer delingen getallen afronden naar beneden. Indien beide waarden hetzelfde zijn, en er dus detectie is, wordt 'detection' geïncrementeerd. Wanneer de binnenste *'loop'* afgerond is, wordt berekend of voldoende punten gedetecteerd zijn om naar de volgende resolutiestap te gaan met 'factor' en het totaal aantal te detecteren punten. Indien wel, worden de resolutie gerelateerde waarden berkend voor de volgende resolutiestap en wordt de volgende resolutiestap geïtereerd. Indien niet, wordt de *'loop'* onderbroken.

```
int offsetVertex = offsetVertices[globalid];
//loop through resolutions.
//#pragma unroll 1 <— unroll does not help here; amountSteps could
    be 1.
for (int h = 0; h < amountSteps; h++)
{
        //loop through all points belonging to current resolution.
        #pragma unroll 3
        for (int i = startStep; i < currentStep; i += 3)
        {
```

```
                //start by rotating & translating the points in world
                    space to image space using transformationmatrix,
                   each point has a 2D-point and a depth value.
                int z = (int)(transformation[8] * (float)table[i + 2]
                    + transformation[9] * (float)table[i + 1] +
                        transformation[10] * (float)table[i] +
                            transformation[11]);

                int x = (int)((transformation[0] * (float)table[i +
                   2] + transformation[1] * (float)table[i + 1] +
                        transformation[2] * (float)table[i] +
                            transformation[3]) / z);

                int y = (int)((transformation[4] * (float)table[i +
                   2] + transformation[5] * (float)table[i + 1] +
                        transformation[6] * (float)table[i] +
                            transformation[7]) / z);

                //point detection if difference in depthvalue of
                   corresponding points in point table & sensor frame
                    is within 1 step distance.
                if (frame[y * width + x] / currentHeightStep == z /
                   currentHeightStep)
                                detection+=1;
        }
        //stop if amount detections too low.
        if (detection < pointsToDetect * factor)
                break;

        //go to next step.
        j+=1;
        startStep = currentStep;
        currentStep = detectionSteps[j] * 3 + corner;
        factor = (((float)(currentStep - start) / (float)totalPoints)
           ) * resolutionFactor;
        currentHeightStep -= step;
}
```

Na afronding van de *'loops'* start een nieuwe *'loop'* die door alle hoekpunten itereert, dan transformeert en vervolgens de resulterende x- en y-waarde schrijft naar 'drawBuffer'. Het tekenalgoritme verwacht 2 waarden die fungeren als eind-criteria voor het tekenen van 1 volume, dus worden deze ook toegevoegd.

```
//adjust drawbuffer for all cornerpoints.
j=0;
#pragma unroll 3
for (int i = start; i < corner; i += 3)
```

```
{
        //update drawbuffer to new x,y image values as calculated
           with transformationmatrix.
        int z = (int)(transformation[8] * (float)table[i + 2] +
           transformation[9] * (float)table[i + 1] +
                transformation[10] * (float)table[i] + transformation
                   [11]);

        int x = (int)((transformation[0] * (float)table[i + 2] +
           transformation[1] * (float)table[i + 1] +
                transformation[2] * (float)table[i] + transformation
                   [3]) / z);

        int y = (int)((transformation[4] * (float)table[i + 2] +
           transformation[5] * (float)table[i + 1] +
                transformation[6] * (float)table[i] + transformation
                   [7]) / z);

        drawBuffer[offsetVertex + j] = (short)x;
        drawBuffer[offsetVertex + j + 1] = (short)y;
        j+=2;
}
//add endcriteria for drawing function.
amountCornerPoints *= 2;
drawBuffer[offsetVertex + amountCornerPoints+1] = 32000;
drawBuffer[offsetVertex + amountCornerPoints] = 32000;
```

Ten slotte wordt het aantal te detecteren punten vergeleken met het aantal gedetecteerde punten, om, bij onvoldoende detecties, een 0 of anders een 1 te schrijven naar 'detections'.

```
//check if enough points detected for detection of total polygon.
if (detection > pointsToDetect)
        detections[globalid] = 1;
```

5.3 Optimalisatie van het systeem

Om tot het huidige systeem te komen, zijn enkele verschillende kernels geschreven en verschillende optimalisaties toegepast.

5.3.1 Evolutie van kernels

De kernels, die hieronder besproken worden, bezitten een optimalisatie nummer in de titel. Dit nummer komt overeen met het nummer in de tabellen met testwaarden in hoofstuk 7. De *'kernels'* zijn hier getest op snelheid. Hierbij is optimalisatie 0 de referentie. Andere optimalisaties vertrekken van deze referentie en veranderen slechts 1 aspect van de optimalisatie. In het test stadium worden compatibele optimalisaties gecombineerd.

Punt-parallel kernel

De eerste *'kernel'* is opgezet zonder enige kennis van GPU programmatie. Hierbij is verkozen om elk punt toe te wijzen aan een werkgroep. Er wordt gecontroleerd of het punt detectie doorgeeft. Per detectie dient een globale teller per volume geïncrementeerd te worden. Aangezien meerdere punten van hetzelfde volume tegelijkertijd vergeleken worden, wordt die globale teller tegelijkertijd geïncrementeerd door meerdere werkgroepen. Elke werkgroep leest hetzelfde getal in, incrementeerd dit getal, en schrijft het terug naar de globale teller. De teller is dus slechts geïncrementeerd met 1, in plaats van het werkelijke aantal detecties. Om dit op te lossen moet de GPU de werkgroepen synchroniseren, waardoor ze één voor één de teller incrementeren. Alle werkgroepen worden gesynchroniseerd, waardoor het parallelle karakter sterk afneemt en eerder richting een serieel karakter gaat. Deze *'kernel'* is zo traag dat hier geen verdere aandacht aan gegeven is.

Volume-parallel kernel optimalisatie 0

Het synchroniseren van *'kernels'* was het grootste probleem in de voorgaande *'kernel'*. Hierom is een *'kernel'* ontworpen, analoog aan de finale *'kernel'*, die geen synchronisatie behoeft. Echter heeft deze *'kernel'* geen weet van resoluties en overloopt alle punten altijd. Ook de ondersteuning van dynamische assenstelsels is anders. Ten eerste zijn alle punten, gegeven aan deze *'kernel'*, reeds in beeldruimte gezet door de CPU. Om verplaatsingen van assenstelsels te voorzien, dienen de transformatiewaarden in de transformatiematrix gerelateerd te zijn aan de tweedimensionale beeldruimte, waardoor verplaatsingen volgens de derde dimensie, de z-as, onmogelijk zijn. De transformatie van punten gebeurt ook niet voor elk frame, maar enkel indien een verplaatsing gedetecteerd is. Dit is bereikt door de toevoeging van een extra buffer 'tableMod', die de kopie is van de punten tabel. Met de combinatie van een if-functie en een boolean buffer, die per assenstelsel aangeeft of het assenstelsel verplaatst is en bepaald is door de CPU, wordt bepaald of het volume getransformeerd moet worden of niet. Voor elke iteratie in de *'loop'* wordt de if-functie aangesproken die aangeeft of het punt getransformeerd wordt of niet. Bij een veranderd assenstelsel wordt het huidige punt uit 'table' getransformeerd en opgeslagen in 'tableMod'. Zodoende zijn de originele punten behouden in de ene tabel, waar alle transformaties tot gerelateerd zijn, en de nieuwe punten zijn opgeslagen in de andere tabel. Vervolgens wordt, analoog aan de finale *'kernel'*, het punt op detectie gecontroleerd. Bij een onveranderd assenstelsel wordt het huidige punt uit 'tableMod' gecontroleerd op detectie. Verder werkt deze *'kernel'* al met restricties op buffers (GPU/CPU enkel schrijven, enkel lezen, lezen en schrijven, geen toegang). De optimalisatie, die bekomen wordt door opeenvolgende geheugenplekken aan te spreken in *loops*, is ook toegepast, aangezien opeenvolgende punten uit de tabel opgeroepen worden tijdens elke iteratie.

Volume-parallel kernel optimalisatie 1: kernelgrootte

Zoals besproken in de bronnenstudie, kan de OpenCL *'precompiler'* optimalisaties verrichten indien het aantal verwerkingseenheden, ofwel de grootte van een werkgroep, bekend is. De werkgroep kan 3 dimensies hebben, maar dit systeem gebruikt slechts 1. Elke werkgroep heeft slechts 1 verwerkingseenheid in dit systeem. De instructie om de grootte van 1 vast te stellen, is *__attribute__((reqd_work_group_size(1,1,1)))*.

Volume-parallel kernel optimalisatie 2: verdere buffer restricties

Verdere buffer restricties worden opgelegd aan de OpenCL *'precompiler'*, zodat deze buffers sneller aangesproken worden. De buffers waar naartoe geschreven wordt, zoals 'tableMod', 'drawBuffer' en 'detections', worden niet veranderd. Naar de andere buffers wordt niet geschreven door de GPU en zijn reeds als enkel lezen bestempeld. De enkel lezen restrictie wordt opgelegd door de CPU nadat de *'precompiler'* de kernelcode gecompileerd heeft. De *'precompiler'* heeft daarom geen weet van deze restricties en kan geen gerelateerde optimalisaties toepassen. Door enkel lezen geheugenplekken in de *'kernel'* zelf te definiëren als *constant* en *restricted*, kan de *'precompiler'* wel gerelateerde optimalisaties toepassen.

Volume-parallel kernel optimalisatie 3: altijd dynamisch

Enkel de code voor veranderde assenstelsels blijft bestaan, waarbij de 'tableMod' verwijderd is. Nu dient de transformatie van elk punt te gebeuren altijd, maar is er geen if-functie meer per iteratie. Het hebben van code onder een if-functie is inefficiënt. De *'compiler'* moet namelijk code genereren achter elke if-functie. Indien die code niet doorlopen wordt, moet all die code verwijderd worden, wat tijd inneemt. Ook het lezen van de schrijf en lees ondersteunende 'tableMod' buffer, die niet bestempeld kan worden als enkel lezen, *constant* en *restricted*, is weggewerkt. het lezen van de 'table' buffer is sneller, aangezien deze enkel lezen ondersteunt.

Volume-parallel kernel optimalisatie 4: unroll

In de bronnenstudie is reeds het voordeel van het uitrollen van *'loops'* aangehaald. De hoofd- en 'drawBuffer' *'loops'* worden 'a' aantal keer uitgerold door, boven elke *'for-loop'*, het commando *#pragma unroll 'a'* te plaatsen.

Volume-parallel kernel optimalisatie 5: dynamische en statische loop

In plaats van 1 transformatie gerelateerde if-functie per iteratie te doen, wordt er enkel 1 gedaan in totaal. Een dynamische *'loop'*, die uitgevoerd wordt als het assenstelsel bewogen is, is analoog aan de originele *'kernel'*, maar met een verwijderde if-functie. De code, die uitgevoerd zou worden indien de if-functie geen verplaatsing aangeeft, is geplaatst in een andere *'loop'*, gekend als de statische *'loop'*, die aangesproken wordt als het assenstelsel niet bewogen is. Het resultaat van het plaatsen van een grote hoeveelheid code achter een if-functie is een vertraging van het systeem. Echter wordt deze grote if-functie slechts 1 keer uitgevoerd in tegenstelling tot het voorgaande systeem, waarbij een kleinere if-functie opgeroepen werd per iteratie.

Volume-parallel kernel optimalisatie 6: resolutiestappen

Hier worden de resolutiestappen geïntroduceerd zoals eerder besproken. Indien volumes een lage vulgraad hebben in het huidige frame, wordt niet het gehele volume afgevraagd. Echter zijn hiervoor 2 versies ontwikkeld. De geïmplementeerde optimalisatie is 6v1 (versie 1). Een voorgaande versie 6v0 (versie 0) werkt anders. In plaats van 2 *'loops'* is er slechts 1. In die loop staat een extra if-functie die afvraagt of het laatste punt van de huidige resolutiestap bereikt is bij elke iteratie. Indien dit punt bereikt is, wordt de vergelijking gedaan of voldoende punten bereikt zijn om naar de volgende resolutiestap te gaan. Indien niet wordt uit de *'loop'*, die nu alle punten doorloopt, gestapt.

5.3.2 Conclusie

Verschillende optimalisatiemethoden en combinaties hiervan hebben verschillende invloeden op performantie. Verschillende testen op performantie staan beschreven in hoofdstuk 7. Elke mogelijke combinatie van optimalisatiemethoden wordt apart getest om een volledig beeld te krijgen van het effect van alle verschillende methoden.

Optimalisaties 3 en 5 zijn incompatibel met elkaar, aangezien 5 de dynamische en statische *'loops'* opsplit en 3 de het statische gedeelte verwijderd. Optimalisatie 6 wordt in eerste instantie genegeerd. Uit de vorige 5 optimalisaties wordt die met het beste resultaat gekozen. Hierop wordt optimalisatie 6 geïmplementeerd. Omdat de performantie van deze optimalisatie afhankelijk is van de ratio tussen het aantal volumes die wel en niet gevuld zijn met lichamen, zijn hier meerdere aparte tests op gedaan met veranderend ratio.

Bij de uitleg van het detectie algoritme is de *'kernel'* met de beste optimalisaties uitgelegd. Dit is de combinatie van optimalisaties 1, 2, 3, 4, 6.

Hoofdstuk 6

Gebruikersaanpassingen in real-time

De gedefinieerde volumes en assenstelsels zijn gecreëerd voordat het detectie-algoritme gestart is. Echter moet de operator van de HIM volumes en assenstelsels kunnen aanpassen, verwijderen en bijvoegen. Omdat gebruik gemaakt wordt van dynamische assenstelsels is het aanpassen van de matrix van bestaande assenstelsels reeds bereikt, maar verdere aanpassingen worden in dit hoofdstuk besproken. Belangrijk bij *'real-time'* aanpassingen is de snelheid waarop de aanpassingen aangebracht worden en de tijd dat het detectie-algoritme onderbroken is om de veranderingen toe te passen. Er zijn geen objectieve tijden gegeven als doelstelling. In hoofdstuk 7 worden enkele metingen op tijd gedaan gerelateerd aan het doorvoeren van aanpassingen.

6.1 Threads

Vooraleer gesproken wordt over het aanbrengen van aanpassingen, wordt de 'thread' toegelicht. Threads staan de CPU toe om meerdere instructies tegelijkertijd uit te voeren.

6.1.1 Mainthread en subthread

Het deitectiealgorthme mag niet onderbroken worden terwijl aanpassingen aangebracht worden. Het detectie-algoritme op de CPU bestaat uit het schrijven van nieuwe buffers naar de *'kernel'* (buffer met dynamische assenstelsels, buffer met het sensorbeeld, ...), het starten van de *'kernel'* en het inlezen van de relevante buffers van de *'kernel'* (buffer met hoekpunten, buffer met detectiewaarden, ...) per frame. Hieronder valt ook de code die Arkite uit moet voeren voor elke frame. Dit wordt in een aparte thread, genaamd 'mainthread', gedaan. Bij het aanpassen wordt een andere thread, genaamd 'subthread' aangemaakt. Deze start de tool die Arkite gebruikt om aanpassingen te maken aan een volume of aan assenstelsels. Deze tool valt buiten het bestek van de masterproef en consequent wordt ervan uitgegaan dat de hoekpunten, hoogte, aantal resolutiestappen, vulfactor en het ID van het volume gegeven wordt na het aanpassen van een volume. Bij het aanpassen van een assenstelsel wordt verwacht dat de waarden voor de transformatiematrix en het ID van het assenstelsel bekend zijn.

Beide threads zijn losgekoppeld en kunnen dus onafhankelijk van elkaar code uitvoeren. Toch moet er onderlinge communicatie plaatsvinden wat in het volgende deel besproken wordt.

6.1.2 Mutex

De buffers, waarvan de GPU gebruik maakt, krijgen andere waarden afhankelijk van de aanpassingen. Indien deze buffers aangepast worden terwijl het detectie-algoritme overlopen wordt, ontstaat een kritische fout. Hierom is het belangrijk om een *mutex* te introduceren. De subthread en mainthread maken beide gebruik van eenzelfde mutex.

Beide threads hebben de bekwaamheid om te vragen aan de mutex of de thread door mag gaan met haar code, of moet wachten. Deze eigenschap wordt gebruikt om ervoor te zorgen dat het aanpassen van de buffers en het detectie-algoritme nooit tegelijkertijd gebeuren. De mainthread vraagt, juist voor het begin van het detectie-algoritme, aan de mutex of het algoritme gestart mag worden. Indien wel, sluit de mainthread de mutex. Indien nu de subthread aanpassingen wil verrichten aan de buffers, vraagt het eerst aan de mutex of dit mag. Aangezien de mutex nu gesloten is door de mainthread, mag de subtread niet verder en wacht het. De mainthread voert het detectie-algoritme uit en opent vervolgens de mutex en stuurt een signaal naar de eventueel wachtende subtread. De subthread mag nu verder en sluit de mutex alvorens het de buffers aanpast. Nu moet de mainthread wachten op de subthread voor het begin van het detectie-algoritme. Eens de subthread de buffers heeft aangepast opent het de mutex en stuurt het een signaal naar de eventueel wachtende mainthread.

6.2 Real-time aanpassingen

Met de nodige informatie voor het real-time aanpassen beschikbaar, worden de aanpassingen aangebracht en de buffers geüpdatet.

Bij het creëren van volumes en assenstelsels en hun *'arrays'* is nog niet gesproken over de ID's. Elk volume en assenstelsel heeft een unieke ID, welke worden bijgehouden in een tabel. Het is de index in deze tabel dat de volumes en de assenstelsels, op basis van hun ID, linkt met de index(en) van hun waard(en) in alle andere tabellen. Verder is er nog een tabel die, per volume, het ID heeft opgeslagen van het bijbehorende assenstelsel. Deze is reeds besproken, aangezien dit de tabel is die de volumes linkt met hun assenstelsel.

Alle assenstelsels zijn geplaatst in een lijst en elk assenstelsel bezit een lijst met volumes. Elk assenstelsel en elk volume heeft een ID die ze zelf opgeslagen hebben. Het aan te passen volume of assenstelsel wordt, op basis van de opgegeven ID('s), gezocht in deze tabellen.

De subthread wordt aangemaakt om 1 blok code af te ronden. Daarna wordt de thread vernietigd. Alle mogelijke blokken code ter aanpassing staan in volgende subsecties beschreven.

6.2.1 Aanpassen van een volume

Het algoritme om volumes op te bouwen wordt opnieuw doorlopen en de lijsten en variabelen met belangrijke waarden worden opnieuw aangemaakt. Bij het aanpassen van volumes wordt ook het ID van het aan te passen volume meegegeven.

De index van het huidige volume, in de tabel met ID's, wordt bepaald en onthouden. Het verschil in aantal punten, aantal resoluiestappen en aantal hoekpunten tussen het oude en nieuwe volume wordt berekend. Met deze informatie worden de tabellen en buffers aangepast.

Volgende aanpassingen worden aangebracht wanneer de mutex dit toelaat. De waarden in de tabel met offsets, behorende bij het huidige volume, wordt geüpdatet. Alle volgende waarden in deze tabel worden verminderd met het verschil in aantal punten. Vervolgens worden alle punten, behorende tot het oude volume, verwijderd uit de punten tabel. De nieuwe punten lijst van het volume wordt, op dezelfde beginpositie, in deze tabel bijgevoegd. Daarna wordt het aantal resolutiestappen, van het volume, in de tabel met aantal resolutiestappen geüpdatet. Alle volgende waarden in deze tabel worden verminderd met het verschil in aantal resolutiestappen. Dan worden de oude aantallen punten per resolutiestap in de tabel met resolutiestappen verwijderd. De nieuwe resolutie punten lijst van het volume wordt, op dezelfde beginpositie, in deze tabel bijgevoegd. De waarden in de tabel met hoekpunten offsets, behorende bij het huidige volume, wordt geüpdatet. Alle volgende waarden in deze tabel worden verminderd met het verschil in aantal hoekpunten. Dan worden de oude hoekpunten in de tabel met hoekpunten verwijderd. De nieuwe hoekpunten lijst van het volume wordt, op dezelfde beginpositie, in deze tabel bijgevoegd. De staplengte wordt geüpdatet in de tabel met staplengtes en de vulfactor wordt geüpdatet in de tabel met vulfactoren. Ten slotte worden alle tabellen geplaatst in de buffers voor gebruik door de GPU. De subtread wordt beëindigd.

6.2.2 Verwijderen van een volume

De index van het huidige volume, in de tabel met ID's, wordt bepaald en onthouden. Het aantal punten, aantal resoluiestappen en aantal hoekpunten van het oude volume wordt berekend. Met deze informatie worden de tabellen en buffers aangepast.

Volgende aanpassingen worden aangebracht wanneer de mutex dit toelaat. De waarden in de tabel met offsets, behorende bij het huidige volume, wordt verwijderd. Alle volgende waarden in deze tabel worden verminderd met het aantal punten. Vervolgens worden alle punten, behorende tot het oude volume, verwijderd uit de punten tabel. Daarna wordt het aantal resolutiestappen, van het volume, in de tabel met aantallen resolutiestappen verwijderd. Alle volgende waarden in deze tabel worden verminderd met het aantal resolutiestappen. Dan worden de oude aantallen punten per resolutiestap in de tabel met resolutiestappen verwijderd. De waarden in de tabel met hoekpunten offsets, behorende bij het huidige volume, wordt verwijderd. Alle volgende waarden in deze tabel worden verminderd met aantal hoekpunten. Dan worden de oude hoekpunten in de tabel met hoekpunten verwijderd. De staplengte wordt verwijderd in de tabel met staplengtes en de vulfactor wordt verwijderd in de tabel met vulfactoren. Het volume wordt verwijderd uit het assenstelsel waartoe het hoorde. Het ID, van alle volumes met een hoger ID dan het verwijderde volume, wordt verlaagd met 1, zowel in de volumes zelf als in de tabel met volume ID's. Het ID van het verwijderde volume en bijbehorend assenstelsel wordt verwijderd uit de tabel met volume ID's en assenstelsel ID's respectievelijk. Dan wordt de tabel met detecties verkleind met 1. Ten slotte worden alle tabellen geplaatst in de buffers voor gebruik door de GPU. De subtread wordt beëindigd.

De ID nummers passen zich dynamisch aan, zodat deze altijd opeenvolgend zijn. Dit neemt extra tijd in beslag en heeft geen praktisch nut in dit systeem. Indien Arkite ook geen gebruik maakt van deze eigenschap, wordt het aanpassen van de ID nummers afgeraden. In de code staat beschreven hoe dit ontdaan wordt.

6.2.3 Toevoegen van een volume

Het algoritme om volumes op te bouwen wordt opnieuw doorlopen en de lijsten en variabelen met belangrijke waarden worden opnieuw aangemaakt. Bij het toevoegen van volumes wordt eventueel het ID meegegeven die het volume moet krijgen. Indien dit niet het geval is, wordt het hoogste bestaande ID gezocht in de tabel met volume ID's. Het ID van het nieuwe volume is deze ID+1. Het volume wordt toegevoegd aan het assenstelsel met het opgegeven assenstelsel ID.

Volgende aanpassingen worden aangebracht wanneer de mutex dit toelaat. De tabel met resolutiestappen wordt bijgevuld met de lijst met resolutiestappen van het huidige volume. De tabel met aantallen resolutiestappen wordt cumulatief bijgevuld met het aantal resolutiestappen van het huidige volume. De 3 offset waarden worden cumulatief bijgevuld in de offset tabel. De punten tabel wordt aangevuld met de punten lijst van het volume. De tabellen met volume ID's en assenstelsel ID's worden aangevuld met het volume ID en assenstelsel ID. De hoekpunten offset tabel wordt cumulatief aangevuld met de hoekpunten offset van het volume. De lijst van hoekpunten, van het volume, wordt toegevoegd aan de hoekpunten tabel. De staplengte wordt toegevoegd aan de tabel met afstanden en de vulfactor wordt toegevoegd aan de tabel met vulfactoren. De detectie tabel wordt vergroot met 1. Ten slotte worden alle tabellen geplaatst in de buffers voor gebruik door de GPU. De subtread wordt beëindigd.

6.2.4 Verwijderen van een assenstelsel

Volgende aanpassingen worden aangebracht wanneer de mutex dit toelaat. Voor elk volume, dat toegewezen is aan dit assenstelsel, wordt een algoritme opgeroepen analoog aan die besproken in hoofdstuk 6.2.2. De enige verschillen zijn dat het sluiten en openen van de mutex niet meer gebeuren voor het verwijderen van 1 volume, aangezien de mutex reeds gesloten is, en de buffers worden niet aangepast na het verwijderen van elk volume. Vervolgens worden de transformatiematrices, behorende tot het assenstelsel, verwijderd uit de tabel met dynamische en de tabel met statische transformatiematrices. In de tabel, die bijhoud bij welk assenstelsel een volume hoort, worden de hogere ID's verlaagd met 1. In de tabel, die de ID's bijhoud van alle assenstelsels, worden alle hogere ID's verlaagd met 1 en het ID verwijderd. In de lijst van assenstelsels wordt ook het ID, van elk assenstelsel met een hoger ID, verlaagd met 1. Het assenstelsel wordt verwijderd uit de tabel met assenstelsels. Ten slotte worden alle tabellen geplaatst in de buffers voor gebruik door de GPU. De subtread wordt beëindigd.

De ID nummers passen zich dynamisch aan, zodat deze altijd opeenvolgend zijn. Dit neemt extra tijd in beslag en heeft geen praktisch nut in dit systeem. Indien Arkite ook geen gebruik maakt van deze eigenschap, wordt het aanpassen van de ID nummers afgeraden. In de code staat beschreven hoe dit ontdaan wordt.

6.2.5 Toevoegen van een assenstelsel

Bij het toevoegen van assenstelsels wordt eventueel het ID meegegeven dat het assenstelsel moet krijgen. Indien dit niet het geval is, wordt het hoogste bestaande ID gezocht in de tabel met assenstelsel ID's. Het ID van het nieuwe assenstelsel is deze ID+1.

Volgende aanpassingen worden aangebracht wanneer de mutex dit toelaat. Het assenstelsel wordt toegevoegd aan de tabel met assenstelsels. Het assenstelsel maakt zelf de statische matrix aan

welke de rotatie en translatie waarden bezit die gegeven zijn als parameters. De tabel met dynamische en statische matrices wordt aangevuld met de matrix van dit assenstelsel. Het ID wordt toegevoegd aan de tabel met assenstelsel ID's. Ten slotte worden alle tabellen geplaatst in de buffers voor gebruik door de GPU. De subtread wordt beëindigd.

Geen relevante GPU informatie wordt aangepast, dus mag het updaten van de buffers achterwegen worden gelaten. Dit is gemeld in de code als commentaar.

6.2.6 Optimalisatie

De functie, voor het veranderen van volumes, was initieel anders geïmplementeerd. In plaats van de bestaande tabellen aan te passen, zoals hierboven besproken, werden alle tabellen leeggehaald en opnieuw opgebouwd. Dit aangezien, bij het opbouwen van lijsten, waarden achter elkaar in de lijst gezet worden waardoor geen nood is aan het bepalen van de locaties van waarden behorende bij het aan te passen volume, het verwijderen van deze waarden en het invoegen van nieuwe waarden op dezelfde locatie. Telkens waarden verwijderd worden de tabel, worden geheugen kopiën gemaakt van de tabel om de lege plek op te vullen. Bij het plaatsen van waarden in de tabel, worden geheugen kopiën gemaakt van de tabel om plek te creëren voor de nieuwe waarden.

Het was niet duidelijk of het creëren van nieuwe tabellen sneller is dan de bestaande lijsten aanpassen. Beide methoden zijn geïmplementeerd en getest. Het resultaat is te zien in hoofdstuk 7.2. Het aanpassen van bestaande tabellen is verkozen aangezien dit sneller is.

Hoofdstuk 7

Testresultaten

Om het resultaat te tonen is een test omgeving gecreëerd die gebruik maakt van alle aspecten die tot nu toe behandeld zijn.

7.1 Tests op kernelsnelheid

7.1.1 Werking van testsysteem

Zoals eerder besproken heeft Arkite een videofragment van de 3D-sensor vrijgesteld ter gebruik. Dit fragment bestaat uit 107 verschillende beelden. Tijdens elk beeld wordt het detectie-algoritme overlopen op 500 volumes. Hierbij is slechts 1 assenstelsel gekozen. De totale tijd van het aanpassen van de matrices van assenstelsels op CPU, inladen van deze matrix en het sensorbeeld in de buffers, starten en afronden van het detectie-algoritme op GPU en het inladen van de hoekpunten en detecties vanaf de buffers wordt gemeten. In de werkelijkheid hoeft niet voor elk frame de hoekpunten ingeladen te worden. Aangezien dit slechts relevant is indien het volume getekend wordt. Het aanpassen van de matrix van het dynamische assenstelsel gebeurt in de werkelijkheid enkel indien het assenstelsel beweegt en zal daarom niet elk frame gebeuren. Echter kunnen tot en met 10 assenstelsel gedefinieerd zijn, die allemaal veranderende matrices kunnen hebben. Dit systeem meet dus niet de tijd van de absolute *'worst case'* scenario, maar geeft wel de tijd van een *'bad case'*, die in de realiteit minder vaak voorkomt.

Voor elk van de 107 beelden wordt de tijd berekend en onthouden. Na het doorlopen van het fragment wordt de gemiddelde tijd berekend en onthouden. Deze tijd wordt 10 maal overlopen door 10 maal hetzelfde fragment af te vragen op detectie. Vervolgens wordt de gemiddelde tijd van de 10 gemiddelde tijden berekend. Dit is de tijd waarop de evaluatie gedaan wordt.

7.1.2 Evaluatie van de tussentijdse resultaten

De uitgevoerde optimalisaties hebben, zoals eerder besproken, een nummer gekregen, met optimalisatie 0 als referentie. In kolom 'optimalisatie', in tabel 7.1, wordt aangegeven welke optimalisatie en combinatie van optimalisaties getest is. De kolom 'unroll' toont hoeveel maal de loops van optimalisatie 4 uitgerold zijn. Er is gekozen voor 2 of 3 *'unrolls'* te testen, aangezien hogere waarden geen significant verschil tonen ten opzichte van een *'unroll'* van 2. In kolom 'tijd gem' is de gemiddelde tijd weergeven die hierboven besproken is. Kolom 'factor' toont de factor tussen

de tijd van de optimalisatie en de tijd in optimalisatie 0. Deze factor toont de tijdwinst van elke optimalisatie. Er zijn 3 paren met gemiddelde tijd en factor; 1 per totaal aantal punten welke gegeven is boven elk paar.

In tabel 7.1 is te zien dat de combinatie van optimalisaties 1, 2, 3 en 4 zorgt voor de beste resultaten bij alle geteste aantallen punten.

Optimalisaties 3 en 4 apart geven een significante, maar tamme vermindering in tijd. Een grote tijdwinst wordt bereikt indien deze optimalisaties samengevoegd zijn. Dit is omdat de *'unroll'* (optimalisatie 4) de code in de *'loop'* 2 of 3 maal kopieert en achter elkaar plakt en het verwijderen van de if-functie (optimalisatie 3) de *'loop'* korter maakt. Elke verkleining, van de *'loop'* code, verkleint 2 of 3 maal zoveel in de uitgerolde code. De *'unroll'* van 3 geeft de beste resultaten. Optimalisaties 1 en 2 veranderen niets aan de geïmplementeerde code, omdat ze enkel restricties opleggen. Daarom is het onmogelijk dat deze, in dit algoritme, een vertraging in tijd veroorzaken.

7.1.3 Evaluatie van de resultaten

Optimalisatie 1+2+3+4 wordt uitgebreid met optimalisatie 6 om de totale optimalisatie van de kernel te verkrijgen. Zoals eerder gemeld zijn er 2 versies van optimalisatie 6. Versie 0 (6v0) heeft een extra if-functie in de *'loop'*, versie 1 (6v1) heeft een extra *'loop'*.

Voorheen is gemeld dat de tabel met punten in sensorbeeld is gegeven en dat dit het onmogelijk maakte om verplaatsingen in de hoogte te verrichten. Aangezien optimalisatie 3 gekozen is, welke altijd uitgaat van een dynamisch assenstelsel en consequent altijd een matrixtransformatie doet, is dit eenvoudig opgelost. De buffer met de matrix, die de transformatie van het assenstelsel in sensorbeeld voorstelde, is nu de dynamische matrix van het relatieve assenstelsel en de transformatiematrix, die wereld naar sensorbeeld omzet, is toegevoegd in een extra buffer. In het begin van de kernel worden deze 2 matrices met elkaar matrixvermenigvuldigd. Daarna wordt deze matrix gebruikt om elk punt om te zetten van wereld naar sensor coördinaten. De extra toegevoegde matrixvermenigvuldiging geeft een onmeetbaar kleine vertraging in het programma, vooral wanneer gerealiseerd wordt dat (tien)duizenden punten per volume matrixvermenigvuldigd worden. Het toevoegen van 1 extra vermenigvuldiging heeft enkel een theoretische impact.

Tabel 7.2 geeft de resultaten van beide versies van optimalisatie 6. Kolom 'factor detecties' toont de factor van het totaal aantal volumes die detectie geven en consequent volledig worden doorlopen. Voor alle andere volumes wordt slechts 1 van de resolutiestappen doorlopen. Er is voor 4 resolutiestappen gekozen. Kolom 'std dev' toont de standaarddeviatie op de 10 gemiddelde gemeten tijden. 'factor verb' is hetzelfde als factor, maar optimalisatie 1+2+3+4 is de referentie. Zodoende kan de tijdwinst van enkel optimalisatie 6 bepaald worden.

Op tabel 7.2 is te zien dat optimalisatie 6v1 betere resultaten geeft dan 6v0. De volgende vergelijkingen focussen zich consequent op optimalisaties 1+2+3+4+6v1, nu genaamd 'de optimalisatie', en 1+2+3+4 nu genaamd 'de referentie'. In het *'worst case'* scenario, bij factor detecties gelijk aan 1, was verwacht dat de tijd langer duurde bij de optimalisatie dan de referentie, aangezien een extra vergelijking bijgevoegd is voor elke resolutiestap. In de resultaten is echter geen significante verandering in tijd te zien, aangezien het verschil in gemiddelde tijden kleiner is dan de standaarddeviatie. De toevoeging van de optimalisatie heeft, bij deze metingen, geen nadelen. Echter geeft elke lagere 'factor detecties' een positieve vermindering van tijd. De tijdwinst is vooral zichtbaar als geen volume detectie geeft. Aangezien geen concrete tests gedaan zijn met

aantal punten totaal		1018000		4918000		9832000	
optimalisatie	unroll	tijd gem	factor	tijd gem	factor	tijd gem	factor
0		8698	1,0000	30001	1,0000	53303	1,0000
1		8345	1,0423	28052	1,0695	49580	1,0751
1+2		8275	1,0511	28274	1,0611	49772	1,0709
1+2+3		8067	1,0782	27077	1,1080	47259	1,1279
1+2+3+4	2	6482	1,3419	19382	1,5479	37294	1,4293
	3	6125	1,4201	18784	1,5972	33998	1,5678
1+2+4	2	7126	1,2206	21931	1,3680	39278	1,3571
	3	7692	1,1308	25203	1,1904	44165	1,2069
1+2+4+5	2	6841	1,2715	21610	1,3883	38755	1,3754
	3	7503	1,1593	24437	1,2277	42703	1,2482
1+2+5		7690	1,1311	24718	1,2137	47466	1,1230
1+3		7945	1,0948	25307	1,1855	46834	1,1381
1+3+4	2	6331	1,3739	19164	1,5655	35944	1,4829
	3	6385	1,3623	18615	1,6117	34285	1,5547
1+4	2	7888	1,1027	27165	1,1044	47038	1,1332
	3	7778	1,1183	26184	1,1458	46676	1,1420
1+4+5	2	7885	1,1031	26218	1,1443	46087	1,1566
	3	7811	1,1136	26562	1,1295	45693	1,1665
1+5		7632	1,1397	25095	1,1955	47114	1,1314
2		8486	1,0250	30164	0,9946	53187	1,0022
2+3		8096	1,0744	25826	1,1617	48851	1,0911
2+4	2	8365	1,0398	28354	1,0581	50346	1,0587
	3	8212	1,0592	28053	1,0694	49595	1,0748
2+4+5	2	8038	1,0821	27101	1,1070	49167	1,0841
	3	8071	1,0777	27453	1,0928	49607	1,0745
2+5		8402	1,0352	28206	1,0636	49946	1,0672
3		7938	1,0957	26180	1,1460	49192	1,0836
3+4	2	7158	1,2151	22522	1,3321	42540	1,2530
	3	7035	1,2364	22801	1,3158	43083	1,2372
4	2	8384	1,0375	29165	1,0287	51659	1,0318
	3	8538	1,0187	29637	1,0123	52483	1,0156
4+5	2	8441	1,0304	28629	1,0479	51930	1,0264
	3	8339	1,0431	28562	1,0504	51228	1,0405
5		8062	1,0789	25884	1,1591	48951	1,0889

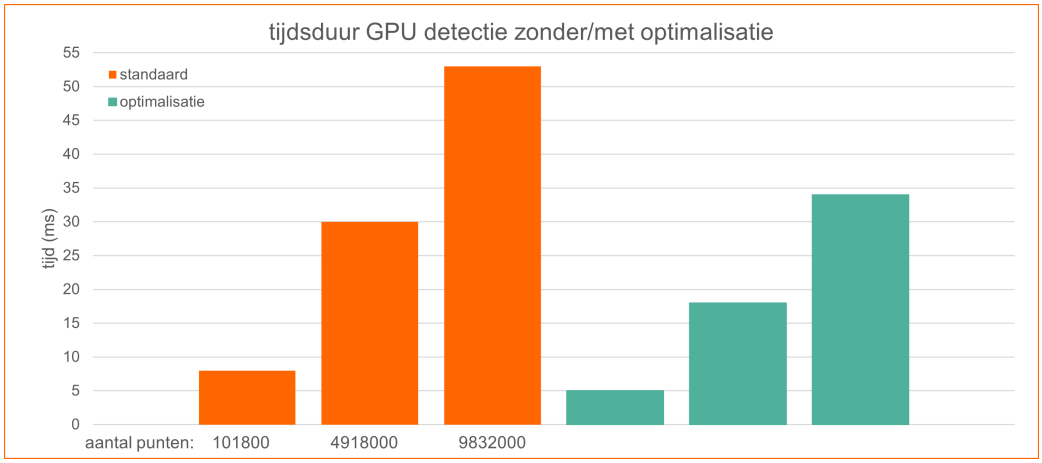
Tabel 7.1: Effect van verschillende optimalisaties op tijdduur detectie-algoritme in microseconde.

optimalisatie	factor detecties	tijd gem	std dev	factor verb	factor
aantal punten totaal		1018000			
1+2+3+4		5539	91	1,0000	1,5703
1+2+3+4+6v0	1	6498	237	0,8524	1,3386
	3/4	6288	74	0,8809	1,3833
	2/3	5157	90	1,0741	1,6866
	1/2	5054	67	1,0960	1,7210
	1/3	4213	95	1,3147	2,0646
	1/4	3973	136	1,3942	2,1893
	0	2640	41	2,0981	3,2947
1+2+3+4+6v1	1	5474	72	1,0119	1,5890
	3/4	5353	95	1,0347	1,6249
	2/3	4689	72	1,1813	1,8550
	1/2	4646	174	1,1922	1,8721
	1/3	3879	45	1,4279	2,2423
	1/4	3641	49	1,5213	2,3889
	0	2694	137	2,0561	3,2287
aantal punten totaal		4918000			
1+2+3+4		18451	92	1,0000	1,6260
1+2+3+4+6v0	1	22386	153	0,8242	1,3402
	3/4	21965	47	0,8400	1,3659
	2/3	16200	122	1,1390	1,8519
	1/2	15720	147	1,1737	1,9085
	1/3	10051	42	1,8357	2,9849
	1/4	9312	54	1,9814	3,2218
	0	3046	51	6,0575	9,8493
1+2+3+4+6v1	1	18432	91	1,0010	1,6277
	3/4	17919	91	1,0297	1,6743
	2/3	13166	91	1,4014	2,2787
	1/2	12796	67	1,4419	2,3446
	1/3	8296	72	2,2241	3,6163
	1/4	7680	107	2,4025	3,9064
	0	2667	41	6,9183	11,2490
aantal punten totaal		9832000			
1+2+3+4		34150	150	1,0000	1,5608
1+2+3+4+6v0	1	42319	184	0,8070	1,2596
	3/4	41568	87	0,8215	1,2823
	2/3	29791	114	1,1463	1,7892
	1/2	28816	36	1,1851	1,8498
	1/3	17937	182	1,9039	2,9717
	1/4	16198	72	2,1083	3,2907
	0	3167	130	10,7831	16,8308
1+2+3+4+6v1	1	34297	65	0,9957	1,5542
	3/4	32931	187	1,0370	1,6186
	2/3	24202	56	1,4110	2,2024
	1/2	23348	178	1,4627	2,2830
	1/3	14274	131	2,3925	3,7343
	1/4	13328	163	2,5623	3,9993
	0	2995	116	11,4023	17,7973

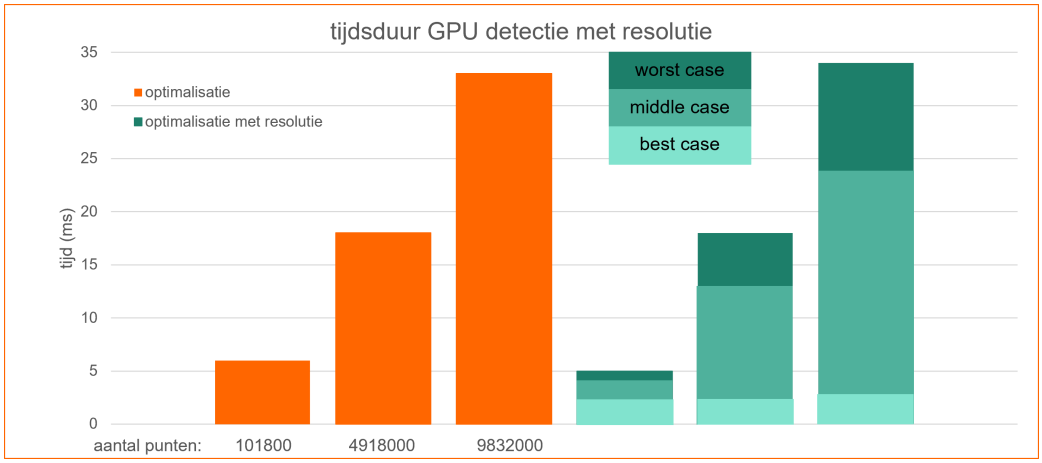
Tabel 7.2: Effect van optimalisaties op tijdduur detectie-algoritme in microseconde.

dit systeem, is het onduidelijk of 4 resolutiestappen betrouwbare resultaten leveren. Indien dit aantal minder is, verkleint de potentiële tijdswinst, maar bij elk aantal groter dan 1 wordt er winst verkregen.

Figuur 7.1 toont de tijdsduur van de kernel met optimalisatie 1+2+3+4 ten opzichte van optimalisatie 0 visueel. Figuur 7.2 toont de tijdsduur van de kernel met optimalisatie 1+2+3+4+6v1 ten opzichte van optimalisatie 1+2+3+4 visueel. Hierbij is optimalisatie 1+2+3+4+6v1 opgesplitst in *'worst-case'*, met factor detecties gelijk aan 1, *'middle-case'*, met factor detecties gelijk aan 1/2 en *'best-case'*, met factor detecties gelijk aan 0. Beide zijn een visualisatie van tabellen 7.1 en 7.2 respectievelijk.



Figuur 7.1: Visualisatie tijdduur van kernel met optimalisatie 0 als 'standaard' en 1+2+3+4 als 'optimalisatie'.



Figuur 7.2: Visualisatie tijdduur van kernel met optimalisatie 1+2+3+4 als 'optimalisatie' en 1+2+3+4+6v1 als 'optimalisatie met resolutie'.

7.2 Test op snelheid volumes aanpassen

In hoofstuk 6.2 zijn 2 methoden ontwikkeld om *'real-time'* aanpassingen aan te brengen. In tabel 7.3 zijn de resultaten gegeven van beide methoden. Hierbij is methode 0 diegene waarbij alle tabellen opnieuw opgebouwd worden en methode 1 diegene waarbij enkel de veranderende waarde van de tabellen aangepast worden. De gemeten tijd geeft aan hoelang het duurt om de tabellen te veranderen en niet om een nieuw volume te creëren. Het is deze tijd die bepaald hoelang

methode	tijd	std dev	factor
0	3684583	8479	1,0000
1	601805	2091	6,1226

Tabel 7.3: Tijdduur aanpassen tabellen in microseconde bij het aanpassen van 1 volume.

conversie	tijd	std dev	factor
aantal punten	101800		
ja	39482078	1594592	1,0000
nee	2701994	40857	14,6122
aantal punten	4918000		
ja	186929859	1414301	1,0000
nee	12438042	95592	15,0289
aantal punten	9832000		
ja	373420534	5917559	1,0000
nee	24038145	258380	15,5345

Tabel 7.4: Tijdduur creëren van 500 volumes met en zonder wereld naar beeld conversie in microseconde.

het detectie-algoritme moet wachten. Omdat, bij het aanpassen van cumulatieve tabellen, alle waarden achter de aan te passen waarde(n) veranderd worden, is het aanpassen van het eerste volume het *'worst case'* scenario. De test is gedaan op dit scenario bij een totaal van 9832000 punten verdeeld over 500 volumes. Kolom 'factor' toont wederom de tijdwinst factor tussen de methodes en methode 0.

De getoonde tijd is het gemiddelde van 10 tests. De standaarddeviatie van deze tests is ook gegeven.

Deze test is enkel uitgevoerd om te bepalen welke methode sneller is en niet om de wekelijke duur van aanpassingen te meten. Dit aangezien de testcomputer en de HIM andere CPU's gebruiken en deze code puur op CPU gebeurt.

7.3 Test op snelheid volumes creëren

Zoals vermeld is de overstap gemaakt van punten opslaan in sensorcoördinaten naar wereldcoördinaten. In hoofdstuk 4.1 is getoond dat de volumes in de wereldruimte gecreëerd worden. In de eerdere kernel versies deed de CPU de conversie van elk punt van het volume van wereld- naar beeldcoördinaten. Aangezien dit niet meer gebeurt, is achterhaald hoeveel tijdwinst dit met zich meebrengt. Dit is vooral relevant indien *'real-time'* aanpassingen/creaties van volumes verricht worden. De 'lineaire' methode voor het creëren van volumes is hier getest.

Tabel 7.4 toont de tijd voor het opbouwen van 500 volumes met het gespecificeerd totaal aantal punten. Ook hier zijn 10 tests gedaan waarvan de gemiddelde tijd en de standaarddeviatie berekend zijn. Kolom 'factor' toon de tijdwinst factor ten opzichte van de tijd om volumes op te bouwen met wereld naar beeld conversie.

Deze test is enkel uitgevoerd om een ruw beeld te krijgen van de duur om volumes op te bouwen. Dit aangezien de testcomputer en de HIM andere CPU's gebruiken en deze code puur op CPU gebeurt. Het is duidelijk zichtbaar dat het overgrote gedeelte van de tijd, voor de creatie van

een volume, gespendeerd is aan het transformeren van elk punt. Een substantiële versnelling is bereikt door de conversie achterwege te laten.

7.4 Keuze van volume opbouwmethode

7.4.1 Vergelijking van volumes gecreëerd met de verschillende methoden

Van de 2 ontwikkende methoden om volumes op te bouwen, is getest welke methode het beste werkt met het ontwikkeld systeem.

Puntenopvulling

De 'radiale' methode vult een oppervlakte in een circulair patroon, met het zwaartepunt als middelpunt. Hierdoor ontstaat een dichtere bevolking van punten in de buurt van het zwaartepunt en een karigere bevolking verder van het zwaartepunt af. Dit wil zeggen dat de gevoeligheid van de detectie afhankelijk is van waar in het figuur een lichaam zich bevind. Om dit op te lossen zou de vulfactor afhankelijk moeten zijn van de positie van het gedetecteerde punt in het volume. Dit moet ingebracht worden in de GPU en neemt extra tijd in beslag. Het ontwikkelen van zo een systeem is omslachtig en niet eenvoudig. Aangezien de afstand tussen opeenvolgende punten op de rand een vaste waarde is, wordt de punten populatie bij grote oppervlakten zeer groot, vooral in het midden van het figuur. Dit vertraagd het detectiesysteem. De 'lineaire' methode behoud altijd dezelfde afstand tussen opeenvolgende punten.

De 'lineaire' methode heeft daarom het voordeel op dit vlak.

Gevoeligheid aan oriëntatie

Het verschil in hoogte tussen opeenvolgende punten moet gekend zijn door de GPU. Bij de 'lineaire' methode kan de afstand tussen opeenvolgende punten in alle richtingen gelijk worden gezet, waardoor de oriëntatie van het volume een beperkte invloed heeft op de hoogte tussen opeenvolgende punten. Bij de 'radiale' methode is de hoogte-afstand altijd gelijk volgens de z-as, maar de afstanden tussen punten in de x- en y-richting is variabel en zeer slecht voorspelbaar.

De 'lineaire' methode heeft daarom het voordeel op dit vlak.

Optimalisatiemogelijkheid

Optimalisatie 6 (resolutiestappen) is in te brengen in de 'radiale' methode indien elke 'puntenring' van het circulair opvulpatroon gezien wordt als 1 resolutie. Zo een 'puntenring' neemt de vorm van het oppervlakte aan, aangezien hetzelfde aantal punten wordt geplaatst op elke radiaal gaande van een punt op de rand naar het zwaartepunt. De 'lineaire' methode heeft reeds een optimalisatiemethode voor optimalisatie 6.

Het is slecht te achterhalen welke methode betere is op dit vlak zonder grondige testen te doen gebruikmakend van een HIM.

Bereik van mogelijke volume vormen

Beide methoden worden recht geëxtrudeert volgens de z-as. De vormgeving wordt dus volledig bepaald aan de hand van het opgegeven oppervlakte door de operator.

Bij de 'radiale' methode wordt uitgegaan van een vlak die volledig ligt op dezelfde z-waarde.

Bij het *'Quickhull'*-algoritme zijn de verschillende vormen, die gecreëerd kunnen worden, verder beperkt doordat het aantal hoekpunten wordt geminimaliseerd. Een inkeping in een vierkant, bijvoorbeeld, wordt genegeerd en een solide vierkant wordt verkregen. Verder moet het zwaartepunt van het gewenste oppervlakte binnen dat oppervlakte liggen. Dit is een verdere limiterende factor op het gebied van mogelijke volume vormen. Hierdoor wordt het *'Quickhull'*-algoritme niet gebruikt.

De 'lineaire' methode maakt het mogelijk om hoekpunten te kiezen die op verschillende z-waarden liggen. Verder wordt een figuur gecreëerd die alle hoekpunten connecteert, waardoor elke oppervlaktevorm ondersteund is. Het maximum aantal hoekpunten waarmee dit systeem werkt is theoretisch oneindig. Echter resulteren meerdere hoekpunten in een langere opbouwtijd.

De 'lineaire' methode heeft daarom het voordeel op dit vlak.

Snelheid

Zoals voorheen gemeld gebeurt de opbouw van volumes volledig op CPU. De CPU van de testcomputer en de HIM is anders, dus kan niet met zekerheid bepaald worden welke methode sneller is. Toch wordt verwacht dat de 'radiale' methode sneller is dan de 'lineaire', aangezien dit het geval is bij de testcomputer. Dit is ook te verwachten aangezien de 'lineaire' methode significant meer overhead heeft dan de 'radiale'.

De 'radiale' methode heeft daarom waarschijnlijk het voordeel op dit vlak, maar dit kan niet met zekerheid gesteld worden.

7.4.2 Conclusie

De meer robuuste 'lineaire' methode wordt gebruikt die dient voor alle mogelijke oppervlaktevormen. Bij de meeste voorgaande vergelijkingen wordt deze als beste geacht. De waarschijnlijke lagere snelheid voor het opbouwen van volumes is niet zeer belangrijk, aangezien het opbouwen van volumes niet vaak gebeurt.

7.5 Toekomstig onderzoek en mogelijke verbeteringen

In dit hoofdstuk wordt besproken welke aspecten van de code eventueel te verbeteren zijn. Omdat er geen fysieke HIM aanwezig is voor het grondig testen van het detectie-algoritme in realistische omgevingen, wordt gemeld welke parameters en stukken code best getest worden.

Bepaling van afstand opeenvolgende punten

Bij de uitleg van het opbouwen van volumes met de lineaire methode (4.1.3) is reeds gemeld dat de afstand tussen 2 opeenvolgende punten bepaald wordt aan de hand van de gemiddelde lengte

tussen de zijden van het volume. Op dit gemiddelde worden 10 punten gelegd. De onderlinge afstand van deze punten wordt de onderlinge afstand tussen alle punten.

Het aantal punten dat op de gemiddelde afstand gelegd wordt, is deels willekeurig gekozen. Dit getal werkt voor het detecteren van handelingen in de testomgeving. Arkite kan testen welke laagste waarde nog betrouwbare resultaten geeft. Het is dit getal dat best aangehouden wordt, aangezien een grotere afstand tussen punten resulteert in minder punten en een lagere tijd voor het doorlopen van het detectie-algoritme.

Verder is onduidelijk of de gemiddelde afstand een goede maat is voor het bepalen van de onderlinge afstand. Gemiddelden worden sterk beïnvloed door uitschieters. Dit kan nuttig zijn indien een volume met een smal gedeelte gemaakt wordt, aangezien dit gedeelte beter gevuld wordt. Echter wordt een volume met een breed gedeelte kariger gevuld, wat een negatieve impact heeft op gevoeligheid. Arkite kan testen of de mediaan van de afstanden een beter resultaat geeft.

De afstand van opeenvolgende punten kan theoretisch tussen en inclusief 0 en 2147483647 vallen. Arkite kan best een minimum en maximum afstand introduceren. Deze waarden worden best getest op het kleinste en grootste volume dat nog praktisch nut heeft.

Indien een volume gecreëerd wordt met een grote oppervlakte, maar een lage hoogte, wordt de afstand tussen opeenvolgende punten klein. Consequent heeft dit volume zeer veel punten, wat de snelheid negatief beïnvloed. In de code voor het opbouwen van volumes is het eenvoudig om een andere afstand tussen opeenvolgende punten te definiëren afhankelijk van de richting (x,y,z). Hiervan is nog geen gebruik gemaakt, aangezien de kernel gebruikt maakt van de afstand tussen opeenvolgende punten om te bepalen of de gemeten hoogte tussen 2 opeenvolgende punten in het volume valt. Als deze afstand afhankelijk is van de oriëntatie van het figuur, zijn de eigenschappen van het detectie-algoritme variabel indien het volume dynamisch kan zijn. Het is niet bekend of dit een significant negatieve invloed op het detectie-algoritme heeft. Indien het volume nooit van oriëntatie veranderd zijn er geen nadelen. Arkite kan de onderlinge punten afstand in alle richtingen (x,y,z) onafhankelijk maken. De gemiddelde afstand tussen zijden, wat de onderlinge punten afstand bepaald, is reeds berekend in alle richtingen.

Oriëntatieafhankelijkheid bij de vulgraad van een volume

In hoofdstuk 4.4.2 is reeds gesproken over de problemen die ontstaan met de vulgraad van volumes indien de z-as van het volume en de z-as van het camerabeeld sterk verschillen van elkaar.

7.5.1 Real-time aanpassingen van volumes en asssenstelsels

In hoofdstuk 6 is gesproken over de ID's van de assenstelsels en volumes. Elk assenstelsel en elk volume heeft een uniek ID. Deze ID's worden in tabellen gezet. De tabel met de ID's van de volumes geeft de positie aan van de bijbehorende waarde in andere tabellen aan de hand van de index van het ID. De lijst met assenstelsel ID's bepaald, voor elk volume, tot welk assenstelsel het behoord op basis van het ID van het assenstelsel.

Het enige relevantie is dat de ID's juist gelinkt zijn met elkaar, niet welke waarde deze ID's hebben. Toch zorgt het huidige systeem dat er altijd opeenvolgende ID's zijn, beginnend vanaf nul. Er wordt tijd verloren aan het aanpassen van ID's indien een volume of assenstelsel verwijderd wordt. Indien dit niet gedaan wordt kan een theoretisch probleem optreden, indien volumes

verwijderd en vervolgens bijgevoegd worden. Het ID nummer blijft dan namelijk oplopen totdat de *'integer overflow'* optreed. Daarna kunnen er 2 identieke ID's zijn, waardoor het systeem niet meer werkt. Echter is deze limiet praktisch onhaalbaar, aangezien het ID kan oplopen tot en met 2147483647. Het aanpassen van ID's kan dus achterwegen worden gelaten. Het verwijderen van volumes en assenstelsels in *'real-time'* versnelt dan.

Indien Arkite ervoor kiest om de altijd opeenvolgende ID's te behouden, kan best de tabel met volume ID's verwijderd worden. De opeenvolgende ID's zorgt er namelijk voor dat het ID nummer en de index dezelfde waarde hebben. Normaliter wordt deze tabel overlopen om te bepalen welke detectie behoort tot welk volume. Dit gebeurt dus voor elk frame. Nu is de index altijd hetzelfde als het ID, dus is deze tabel nutteloos. Het verwijderen, aanpassen en toevoegen van volumes en het verwijderen van assenstelsels in *'real-time'* wordt ook versneld, aangezien 1 tabel minder aangepast wordt.

7.5.2 Kernel

De huidige kernel heeft code die de hoekpunten in een aparte buffer zet in beeldcoördiaten ter gebruik voor het tekenen. Indien het detecialgoritme in de praktijk gebruikt wordt, is het tekenen van volumes minder relevant. Om de tijd van het transformeren en opslaan van hoekpunten te besparen wanneer het onnodig is, kan een extra kernel gemaakt worden die dit achterwegen laat. Dit wordt bereikt door een kopie van de huidige kernel functie maken, waarbij de 'offsetVertices' en 'drawBuffer' wordt weggelaten alsook de code die de tekenbuffer invult, en deze functie een andere naam te geven. Indien beide functies in hetzelfde bestand zijn, worden beide gecompileerd. Echter moet nu, bij het opzetten van de GPU code, een extra kernel object aangemaakt worden met de naam van de nieuwe functie. De buffers worden vervolgens ingevuld, volledig analoog aan hoe dit gedaan is bij de bestaande kernel min de 2 voorgenoemde buffers. Het starten van de nieuwe kernel en het inlezen van de buffers wordt ook bereikt volledig analoog aan de bestaande kernel, met als uitzondering dat niet gelezen wordt uit de buffer met hoekpunten. Indien dit geïmplementeerd is, kunnen beide kernels aangesproken worden. Arkite beslist zelf wanneer welke kernel wordt aangesproken. Beide kernels krijgen best dezelfde geheugenplek toegewezen voor dezelfde data om te voorkomen dat de data wordt gekopieerd bij het creëren van nieuwe buffers. De huidige methode voor het creëren van buffers verwijst deze buffers naar bestaande geheugenplekken en wordt het geheugen niet gekopieerd.

Het verschil in de gemiddelde tijd van het algoritme tussen dynamische en statische assenstelsels is 10 maal getest op 107 frames met 8309500 punten verdeeld over 500 volumes. De ruwe gemiddelde testresultaten zijn 28500 microseconde voor de kernel met de transformatie van elk punt en 19000 microseconde voor een kernel zonder transformaties. Arkite kan ervoor kiezen om een extra kernel toe te voegen die geen transformaties doet. Deze wordt opgeroepen indien geen transformaties plaatsvinden. Het is dan wel belangrijk om een extra buffer te voorzien waarin de punten in sensorcoördinaten gedefinieerd zijn. Dit wordt bereikt door eerste de dynamische kernel op te roepen en de getransformeerde waarden in een extra buffer te zetten. Deze buffer wordt, wanneer de statische kernel aangesproken is, gekopieerd naar de statische kernel door de CPU. Indien er geen verdere veranderingen van assenstelsels zijn, wordt de statische kernel opgeroepen. Echter zorgt dit systeem voor grote vertragingen als 1 assenstelsel elk frame veranderd. De dynamische kernel wordt dan opgeroepen voor alle volumes. Elk punt wordt gekopieerd in een lees en schrijf buffer. Dit veroorzaakt een vertraging. Indien de statische buffer vervolgens opgeroepen wordt,

wordt de buffer met getransformeerde punten gekopieerd door de CPU ter gebruik van de andere kernel, wat extra tijd kost. Dit systeem heeft voordelen bij een systeem waarbij 1 transformatie plaatsvind per significant aantal beelden, maar heeft grote nadelen indien verplaatsingen vaak voorkomen, zoals bijvoorbeeld bij het volgen van een transportband.

7.6 Conclusie

De doelstelling voor het afvragen van 500 volumes op detectie is bereikt, mits het aantal af te vragen punten niet boven een bepaald aantal komt. Om dit aantal te maximaliseren zijn succesvolle optimalisaties geïmplementeerd. Tot en met 10 verschillende assenstelsels moeten gedefinieerd kunnen worden. Dit systeem kan veel meer verschillende assenstelsels implementeren zonder enige impact op de tijdsduur van het detectie algoritme. Volumes worden opgebouwd aan de hand van een aantal punten die hoger mag zijn dan het maximum van 99, welke gehanteerd wordt door Arkite. De optionele doelstelling voor het implementeren van dynamische volumes is eveneens bereikt. Echter wordt, bij volumes die van oriëntatie veranderen, de keuze van 1 specifieke parameter gelegd bij de operator van de HIM. Arkite wil dit voorkomen, maar dit valt buiten de scope van de originele doelstellingen. Verder zijn er 2 methoden ontwikkeld voor het definiëren van volumes, waarbij de beste methode geïdentificeerd en gekozen is. Ook dit was naar de wensen van Arkite. Vele zwakten en eventuele verbeterpunten zijn geïdentificeerd en gedocumenteerd met suggesties voor eventuele oplossingen.

Het eindresultaat voldoet aan de verwachtingen van Arkite en er wordt gesproken over een succesvolle masterproef.

Literatuurlijst

[1] E. Mucke, "Computing prescriptions: Quickhull: Computing convex hulls quickly," *Computing in Science & Engineering*, vol. 11, no. 5, pp. 54–57, 2009.

[2] C. A. Hoare, "Quicksort," *The Computer Journal*, vol. 5, no. 1, pp. 10–16, 1962.

[3] C. Ricolfe-Viala and A.-J. Sánchez-Salmerón, "Robust metric calibration of non-linear camera lens distortion," *Pattern Recognition*, vol. 43, no. 4, pp. 1688–1699, 2010.

[4] J. E. Stone, D. Gohara, and G. Shi, "Opencl: A parallel programming standard for heterogeneous computing systems," *Computing in science & engineering*, vol. 12, no. 3, p. 66, 2010.

[5] A. Hornung, K. M. Wurm, M. Bennewitz, C. Stachniss, and W. Burgard, "Octomap: An efficient probabilistic 3d mapping framework based on octrees," *Autonomous robots*, vol. 34, no. 3, pp. 189–206, 2013.

[6] I. F. Intel, "Sdk for opencl pro edition best practices guide, v20.4," 2020.

[7] B. Volkaerts, "Onderzoek naar de hardware implementatie van beeldverwerkingsfuncties uit opencv," 2013.

[8] Z. Zhang, "Microsoft kinect sensor and its effect," *IEEE MultiMedia*, vol. 19, no. 2, pp. 4–10, 2012.

[9] E. Lachat, H. Macher, T. Landes, and P. Grussenmeyer, "Assessment and calibration of a rgb-d camera (kinect v2 sensor) towards a potential use for close-range 3d modeling," *Remote Sensing*, vol. 7, p. 13070–13097, Oct 2015.

[10] P. Fankhauser, M. Bloesch, D. Rodriguez, R. Kaestner, M. Hutter, and R. Siegwart, "Kinect v2 for mobile robot navigation: Evaluation and modeling," in *2015 International Conference on Advanced Robotics (ICAR)*, pp. 388–394, 2015.

[11] O. Wasenmüller and D. Stricker, "Comparison of kinect v1 and v2 depth images in terms of accuracy and precision," in *Asian Conference on Computer Vision*, pp. 34–45, Springer, 2016.

[12] C. Kim, S. Yun, S.-W. Jung, and C. S. Won, "Color and depth image correspondence for kinect v2," in *Advanced multimedia and ubiquitous engineering*, pp. 111–116, Springer, 2015.

[13] J. Tompson and K. Schlachter, "An introduction to the opencl programming model," *Person Education*, vol. 49, p. 31, 2012.

[14] L. Zhang, L. Wei, P. Shen, W. Wei, G. Zhu, and J. Song, "Semantic slam based on object detection and improved octomap," *IEEE Access*, vol. 6, pp. 75545–75559, 2018.

[15] J. Han, L. Shao, D. Xu, and J. Shotton, "Enhanced computer vision with microsoft kinect sensor: A review," *IEEE Transactions on Cybernetics*, vol. 43, no. 5, pp. 1318–1334, 2013.

[16] T. Brandes, A. Arnold, T. Soddemann, and D. Reith, "Cpu vs. gpu-performance comparison for the gram-schmidt algorithm," *The European Physical Journal Special Topics*, vol. 210, no. 1, pp. 73–88, 2012.

Bijlage A

Appendix

Aangezien de kernel het voornaamste deel uitmaakt van deze masterproef, wordt in de appendix de code getoond die direct betrekking heeft tot de kernel. De *'comments'* lichten toe wat de code doet en staat ook in de werkelijke code.

De namen van de buffers aan de CPU zijde zijn gelijkaardig aan de reeds besproken namen van de buffers aan de GPU zijde. Volumes worden, in de *'comments'* van de code, 'polygons' genoemd.

A.1 OpenCL: Accelerator source

A.1.1 Set-up door middel van constructor

```
Accelerator::Accelerator()
{
        //select OpenCL and desired GPU and build kernel program
           using the appropriate index as shown in code. this index
           varies for different systems

        //select platform (OpenCL).
        std::vector<cl::Platform> all_platforms;
        cl::Platform::get(&all_platforms);
        if (all_platforms.size() == 0)
        {
                setupInfo[0] = "no platforms found, check OpenCL
                   installation";
                exit(1);
        }
        else
                setupInfo[0] = "platforms found";

        cl::Platform default_platform = all_platforms[1]; //OpenCL
           platform is at index=1 for currrent system
        setupInfo[1] = default_platform.getInfo<CL_PLATFORM_NAME>();
```

```
        //select device (GPU).
        std::vector<cl::Device> all_devices;
        default_platform.getDevices(CL_DEVICE_TYPE_ALL, &all_devices)
            ;
        if (all_devices.size() == 0)
        {
                setupInfo[2] = "no Devices found";
                exit(1);
        }
        else
                setupInfo[2] = "Devices found";

        cl::Device default_device = all_devices[0]; //Intel Graphics
            530 is at index=0 for current system

        //create context with platform & device.
        _context = new cl::Context({ default_device });

        //put kernel in program source as a string from the kernel .
            cl file and build kernel program.
        std::ifstream kernel("kernel.cl");
        std::string kernelCode(std::istreambuf_iterator<char>(kernel)
            , (std::istreambuf_iterator<char>()));
        cl::Program::Sources sources(1, std::make_pair(kernelCode.
            c_str(), kernelCode.length() + 1));
        cl::Program program(*_context, sources);
        if (program.build({ default_device }) != CL_SUCCESS)
        {
                setupInfo[3] = "error building: " + program.
                    getBuildInfo<CL_PROGRAM_BUILD_LOG>(default_device)
                    ;
                exit(1);
        }
        else
                setupInfo[3] = "succes";

        //link queue with device and context.
        _queue = new cl::CommandQueue(*_context, default_device);

        //specify name of desired object within the kernel code.
        _kernelDetect = new cl::Kernel(program, "detect");
}
```

A.1.2 Geheugenvrijgave door middel van destructor

```
void Accelerator::clearBuffers()
```

```
{
        delete _bufferDetections;
        delete _bufferTableWorldPoints;
        delete _bufferTableOffset;
        delete _bufferHeightWidth;
        delete _bufferFillFactor;
        delete _bufferFrame;
        delete _bufferTransformationDynamic;
        delete _bufferMatrixWorldToImage;
        delete _bufferDetectionSteps;
        delete _bufferAmountDetectionSteps;
        delete _bufferStepDistance;
}
Accelerator::~Accelerator()
{
        delete _kernelDetect;
        delete _queue;
        delete _context;
        clearBuffers();
}
```

A.1.3 Buffers bepalen

```
void Accelerator::setBufferDetections(std::vector<std::tuple<short,
   short, short>>& tableWorldPoints,
        std::vector<int>& tableOffset, std::vector<float>& fillFactor
           , std::vector<int>& ID, int nrAxis, int nrVertex,
        std::vector<short>& offsetVertices, std::vector<int>&
           amountDetectionSteps, std::vector<int>& detectionSteps,
        std::vector<float>& tableMatrixWorldToImage, std::vector<
           short> tableStepDistance)
{
        clearBuffers();
        //amount of workgroups = amount of boxes, as is amount of
           fillFactors.
        _workGroups = fillFactor.size();

        //IMPORTANT: buffers for points represent total amount of
           points times 3, because every point has 3 short values(x,
           y, z)!

        //assign buffer for sensor values.
        _bufferFrame = new cl::Buffer(*_context, CL_MEM_READ_ONLY |
           CL_MEM_HOST_WRITE_ONLY, sizeof(unsigned short) * WIDTH *
           HEIGHT);
        (*_kernelDetect).setArg(0, *_bufferFrame);
```

```
//assign buffer for table with points.
_bufferTableWorldPoints = new cl::Buffer(*_context,
   CL_MEM_READ_ONLY | CL_MEM_HOST_NO_ACCESS |
   CL_MEM_USE_HOST_PTR,
         sizeof(short) * tableWorldPoints.size() * 3,
            tableWorldPoints.data());
(*_kernelDetect).setArg(1, *_bufferTableWorldPoints);

//assign buffer to indicate detection per polygon.
_bufferDetections = new cl::Buffer(*_context,
   CL_MEM_WRITE_ONLY | CL_MEM_HOST_READ_ONLY, sizeof(char) *
   _workGroups, nullptr);
(*_kernelDetect).setArg(2, *_bufferDetections);

//assign buffer for table offset (contains offsets of veteces
    per polygon and offsets of polygons).
_bufferTableOffset = new cl::Buffer(*_context,
   CL_MEM_READ_ONLY | CL_MEM_HOST_NO_ACCESS |
   CL_MEM_USE_HOST_PTR, sizeof(int) * tableOffset.size(),
   tableOffset.data());
(*_kernelDetect).setArg(3, *_bufferTableOffset);

//assign buffer for frame's height, width.
short heightWidth[2] = { HEIGHT, WIDTH };
_bufferHeightWidth = new cl::Buffer(*_context,
   CL_MEM_READ_ONLY | CL_MEM_HOST_NO_ACCESS |
   CL_MEM_USE_HOST_PTR, sizeof(short) * 2, heightWidth);
(*_kernelDetect).setArg(4, *_bufferHeightWidth);

//assign buffer for fillFactors.
_bufferFillFactor = new cl::Buffer(*_context,
   CL_MEM_READ_ONLY | CL_MEM_HOST_NO_ACCESS |
   CL_MEM_USE_HOST_PTR, sizeof(float) * fillFactor.size(),
   fillFactor.data());
(*_kernelDetect).setArg(5, *_bufferFillFactor);

//assign buffer for transformationmatrix relative to absolute
    axis, per axis.
_bufferTransformationDynamic = new cl::Buffer(*_context,
   CL_MEM_READ_ONLY | CL_MEM_HOST_WRITE_ONLY, sizeof(float) *
    16 * nrAxis, nullptr);
(*_kernelDetect).setArg(6, *_bufferTransformationDynamic);

//assign buffer for axisID's per polygon.
_bufferID = new cl::Buffer(*_context, CL_MEM_READ_ONLY |
```

```
    CL_MEM_HOST_NO_ACCESS | CL_MEM_USE_HOST_PTR, sizeof(int) *
     ID.size(), ID.data());
(*_kernelDetect).setArg(7, *_bufferID);

//assign buffer for drawing dynamic volumes. drawing is
    achieved by connecting consecutive points. this buffer
    contains corner points.
_bufferDraw = new cl::Buffer(*_context, CL_MEM_WRITE_ONLY |
    CL_MEM_HOST_READ_ONLY, sizeof(short) * nrVertex, nullptr);
(*_kernelDetect).setArg(8, *_bufferDraw);

//assign buffer for offset vertices. this is used for drawing
    .
_bufferOffsetVertices = new cl::Buffer(*_context,
    CL_MEM_READ_ONLY | CL_MEM_HOST_NO_ACCESS |
    CL_MEM_USE_HOST_PTR, sizeof(short) * ID.size(),
    offsetVertices.data());
(*_kernelDetect).setArg(9, *_bufferOffsetVertices);

//assign buffer for detection steps or resolutions. this is
    the offset to a lower resolution.
_bufferDetectionSteps = new cl::Buffer(*_context,
    CL_MEM_READ_ONLY | CL_MEM_HOST_NO_ACCESS |
          CL_MEM_USE_HOST_PTR, sizeof(int) * detectionSteps.
              size(), detectionSteps.data());
(*_kernelDetect).setArg(10, *_bufferDetectionSteps);

//assign buffer for amount steps or resolutions.
_bufferAmountDetectionSteps = new cl::Buffer(*_context,
    CL_MEM_READ_ONLY | CL_MEM_HOST_NO_ACCESS |
          CL_MEM_USE_HOST_PTR, sizeof(int) *
              amountDetectionSteps.size(), amountDetectionSteps.
              data());
(*_kernelDetect).setArg(11, *_bufferAmountDetectionSteps);

//assign buffer for matrix world-image, per axis.
_bufferMatrixWorldToImage = new cl::Buffer(*_context,
    CL_MEM_READ_ONLY | CL_MEM_HOST_NO_ACCESS |
          CL_MEM_USE_HOST_PTR, sizeof(float) *
              tableMatrixWorldToImage.size() * 16,
              tableMatrixWorldToImage.data());
(*_kernelDetect).setArg(12, *_bufferMatrixWorldToImage);

//assign buffer for distance between 2 consecutive points.
_bufferStepDistance = new cl::Buffer(*_context,
    CL_MEM_READ_ONLY | CL_MEM_HOST_NO_ACCESS |
```

```
                CL_MEM_USE_HOST_PTR, sizeof(short) *
                    tableStepDistance.size(), tableStepDistance.data()
                    );
        (*_kernelDetect).setArg(13, *_bufferStepDistance);
}

\subsection{Buffers schrijven/lezen naar/van GPU}
\begin{lstlisting}[language = c++,breaklines=true]
void Accelerator::setTransformationMatrix(std::vector<float>&
   transformationValues)
{
        (*_queue).enqueueWriteBuffer(*_bufferTransformationDynamic,
           CL_FALSE, 0, sizeof(float) * transformationValues.size(),
           transformationValues.data());
}

void Accelerator::setFrame(cv::Mat& depth)
{
        (*_queue).enqueueWriteBuffer(*_bufferFrame, CL_FALSE, 0,
           sizeof(unsigned short) * HEIGHT * WIDTH, depth.data);
}

void Accelerator::getDetection(std::vector<char>& detections)
{
        (*_queue).enqueueReadBuffer(*_bufferDetections, CL_TRUE, 0,
           sizeof(char) * _workGroups, detections.data());
}

void Accelerator::getVertices(std::vector<short>& vertices)
{
        (*_queue).enqueueReadBuffer(*_bufferDraw, CL_TRUE, 0, sizeof(
           short) * vertices.size(), vertices.data());
}
```

A.1.4 Kernel starten

```
void Accelerator::startDetection()
{
        (*_queue).enqueueNDRangeKernel(*_kernelDetect, cl::NullRange,
            cl::NDRange(_workGroups), cl::NDRange(1));

}
```

A.2 OpenCL: Accelerator header

```
#ifndef ACCELERATOR_H
#define ACCELERATOR_H

#include"vector"
#include"iostream"
#include"string"
#include"fstream"

#include"opencv2/core.hpp"

#include"CL/cl.hpp"

class Accelerator
{
public:
        Accelerator();
        ~Accelerator();
        void setBufferDetections(std::vector<std::tuple<short, short,
            short>>& tableWorldPoints, std::vector<int>& tableOffset,
                std::vector<float>& fillFactor, std::vector<int>& ID,
                    int nrAxis, int nrVertex, std::vector<short>&
                    offsetVertices,
                std::vector<int>& amountDetectionSteps, std::vector<
                    int>& detectionSteps, std::vector<float>&
                    tableMatrixWorldToImage,
                std::vector<short> heightStep);
        void setTransformationMatrix(std::vector<float>&
            transformationValues);
        void setFrame(cv::Mat& depth);
        void getDetection(std::vector<char>& detections);
        void getVertices(std::vector<short>& vertices);
        void startDetection();
        void clearBuffers();
        std::string* setupInfo = new std::string[4];
private:

        cl::Kernel* _kernelDetect;
        cl::CommandQueue* _queue;
        cl::Context* _context;
        cl::Buffer* _bufferDetections;
        cl::Buffer* _bufferTableWorldPoints;
        cl::Buffer* _bufferTableOffset;
        cl::Buffer* _bufferDraw;
        cl::Buffer* _bufferHeightWidth;
        cl::Buffer* _bufferFillFactor;
        cl::Buffer* _bufferFrame;
```

```
        cl::Buffer* _bufferTransformationDynamic;
        cl::Buffer* _bufferMatrixWorldToImage;
        cl::Buffer* _bufferID;
        cl::Buffer* _bufferOffsetVertices;
        cl::Buffer* _bufferAmountDetectionSteps;
        cl::Buffer* _bufferDetectionSteps;
        cl::Buffer* _bufferStepDistance;

        short _workGroups;
};
#endif
```

A.3 OpenCL: kernel

```
__attribute__((reqd_work_group_size(1,1,1)))
void kernel detect
(
        constant short* restrict frame, //0
        constant short* restrict table,//1
        global char* detections,//2
        constant int* restrict tableOffset,//3
        constant short* restrict heightWidth,//4
        constant float* restrict fillFactor,//5
        constant float* restrict transformationDyn,//6
        constant int* restrict IDax, //7
        global short* drawBuffer, //8
        constant short* restrict offsetVertices, //9
        constant int* restrict detectionSteps, //10
        constant int* restrict amountDetectionSteps, //11
        constant float* restrict matrixWorldToImage, //12
        constant short* restrict heightStep //13
)
{
        int globalid = get_global_id(0);
        short step = heightStep[globalid];
        short height = heightWidth[0];
        short width = heightWidth[1];

        //iterators for first point of table with points belonging to
            current polygon, as well as last point of corner and last
            point of table.
        int end = tableOffset[globalid * 3 + 2] * 3;
        int corner = tableOffset[globalid * 3 + 1] * 3;
        int start = tableOffset[globalid * 3] * 3;
        int totalPoints = end - start;
```

```
int amountCornerPoints = (corner - start) / 3;

//resolution steps.
int     amountStepsStart = amountDetectionSteps[globalid];
int amountSteps = amountDetectionSteps[globalid + 1] -
   amountDetectionSteps[globalid];
//point offsets, from box start, of each resolution.
int j = amountStepsStart;
int currentStep = detectionSteps[j] * 3 + corner;
int startStep = start;
int currentHeightStep = step * amountSteps;

//refer to the axis which acts as the volume's parent.
int IDaxs = IDax[globalid];

//get transformationmatrix values by transforming dynamic
   world points to image points.
float transformation[16];
#pragma unroll 4
for (int i = 0; i < 4; i++)
{
        #pragma unroll 4
        for (int j = 0; j < 4; j++)
        {
                //matrixmultiply dynamic transformation (
                   combination of relative coordinate system'
                   s matrix and it's transformation) and
                   world to image matrix.
                transformation[4*i+j] = matrixWorldToImage[(
                   IDaxs*16) + i*4] * transformationDyn[(
                   IDaxs*16) + j] +
                        matrixWorldToImage[(IDaxs*16) + i*4 +
                            1] * transformationDyn[(IDaxs*16)
                            + j + 4] +
                        matrixWorldToImage[(IDaxs*16) + i*4 +
                            2] * transformationDyn[(IDaxs*16)
                            + j + 8] +
                        matrixWorldToImage[(IDaxs*16) + i*4 +
                            3] * transformationDyn[(IDaxs*16)
                            + j + 12];
        }
}

//fillFactor detection.
float factor = fillFactor[globalid];
//determine minimum points to detect with fillFactor and
```

```
    point amount.
int pointsToDetect = totalPoints * factor;
//determine fillFactor for resolutions on basis of ratio
    points in current resolution and total points -> make it
    smaller to not get false negatives due to the
//higher resolution's lower accuracy.
float resolutionFactor = .25;
factor = (((float)(currentStep - start) / (float)totalPoints)
    ) * resolutionFactor;

int offsetVertex = offsetVertices[globalid];

//reset detections.
detections[globalid] = 0;
int detection = 0;

//*****************SETUP*******************//
//loop through resolutions.
//#pragma unroll 1 <-- unroll does not help here; amountSteps
    could be 1.
for (int h = 0; h < amountSteps; h++)
{
        //loop through all points belonging to current
            resolution.
        #pragma unroll 3
        for (int i = startStep; i < currentStep; i += 3)
        {
                //start by rotating & translating the points
                    in world space to image space using
                    transformationmatrix, each point has a 2D-
                    point and a depth value.
                int z = (int)(transformation[8] * (float)
                    table[i + 2] + transformation[9] * (float)
                    table[i + 1] +
                        transformation[10] * (float)table[i]
                            + transformation[11]);

                int x = (int)((transformation[0] * (float)
                    table[i + 2] + transformation[1] * (float)
                    table[i + 1] +
                        transformation[2] * (float)table[i] +
                            transformation[3]) / z);

                int y = (int)((transformation[4] * (float)
                    table[i + 2] + transformation[5] * (float)
                    table[i + 1] +
```

```
                        transformation[6] * (float)table[i] +
                            transformation[7]) / z);

                //point detection if difference in depthvalue
                    of corresponding points in point table &
                   sensor frame is within 1 step distance.
                if (frame[y * width + x] / currentHeightStep
                   == z / currentHeightStep)
                        detection+=1;
        }
        //stop if amount detections too low.
        if (detection < pointsToDetect * factor)
                break;

        //go to next step.
        j+=1;
        startStep = currentStep;
        currentStep = detectionSteps[j] * 3 + corner;
        factor = (((float)(currentStep - start) / (float)
           totalPoints)) * resolutionFactor;
        currentHeightStep -= step;
}
//adjust drawbuffer for all cornerpoints.
j=0;
#pragma unroll 3
for (int i = start; i < corner; i += 3)
{
        //update drawbuffer to new x,y image values as
           calculated with transformationmatrix.
        int z = (int)(transformation[8] * (float)table[i + 2]
            + transformation[9] * (float)table[i + 1] +
                transformation[10] * (float)table[i] +
                   transformation[11]);

        int x = (int)((transformation[0] * (float)table[i +
           2] + transformation[1] * (float)table[i + 1] +
                transformation[2] * (float)table[i] +
                   transformation[3]) / z);

        int y = (int)((transformation[4] * (float)table[i +
           2] + transformation[5] * (float)table[i + 1] +
                transformation[6] * (float)table[i] +
                   transformation[7]) / z);

        drawBuffer[offsetVertex + j] = (short)x;
        drawBuffer[offsetVertex + j + 1] = (short)y;
```

```
                j+=2;
        }
        //add endcriteria for drawing function.
        amountCornerPoints *= 2;
        drawBuffer[offsetVertex + amountCornerPoints+1] = 32000;
        drawBuffer[offsetVertex + amountCornerPoints] = 32000;

        //check if enough points detected for detection of total
           polygon.
        if (detection > pointsToDetect)
                detections[globalid] = 1;
}
```