A-Multi-Camera System for Interactive Videogames

Cuypers Tom

May 23, 2007

Word of Thanks

First of all I would like to thank my promotor Prof. dr. Philippe Bekaert who made it possible to me to make this thesis. Also a lot of graditude goes to Yannick Francken and Cedric Vanaken who guided me during this work. I could always count on them to get answers to problems regarding this thesis. Even though I spent the first five months of this this year in Spain as an Erasmus student, they were still able to help me. I also would like to thank Chris Hermans, Maarten Dumont and Bert Dedecker for sharing their time and information with me. This thesis would not have been the same without them.
At the *Universitat de les Illes Balears*, which was my Erasmus university in Mallorca, I received a lot of help too. I especially like to thank my tutor dr. Francisco J. Perales. He made it possible to me to do research by providing information and hardware. I would also like to thank Antoni Jaume Capo and Dr. Jose Maria Buades for their guidance during my time there.
Eventually I would like to express my thanks to my parents and brothers for supporting me and helping me revising this work.

Contents

List of Figures

List of Tables

Abstract

Modern days, the traditional input devices for computer games are keyboard, mouse and joystick. But there also exists other hardware to interact with games, for example a microphone or a camera. Most of them are still in research state, but they are finding their way to the commercial world. The *Eye Toy* of Sony Playstation 2 is a good example for this. The purpose of this thesis is to describe such an alternative input for games by using two or more cameras. Because of this, the player must be able to interact intuitively in 3D computer games without physical contact with any gaming device so that everyone can play those games.
Using multiple cameras for computer games has to meet with two properties, it has to be cheap and it has to be interactive. In the past there has been a lot of research for creating cheap and efficient hardware for this purpose, but modern cameras are often cheap enough. So the first property is not a problem anymore because for example web cams are very low cost. The second demand has been researched in the computer vision and image processing area a lot and there are many ways to do this, each one has his advantages and disadvantages. It is a large research domain and there are no specific formulations of how computer vision problems should be solved. This work will present a collection of techniques and algorithms capable of extracting useful information from captured images in real time and how this information can be used for interaction in computer games. This can all be done with a standard single processor computer and no specialized hardware.

Nederlandse samenvatting: Een multi-camera systeem voor interactieve videospellen

Introductie

Het doel van deze thesis is om computerspellen aan te sturen met behulp van twee of meerdere camera's. De bedoeling is uiteraard dat deze interactie real time gebeurt opdat een interactief spel bekomen kan worden. De wijze waarop dit kan gebeuren wordt gezocht in het domein computervisie welke zich bezig houdt met het halen van informatie uit een of meerdere beelden [1].
Gelijkaardige verwezenlijkingen kunnen gevonden worden in het vakgebied *mens computer interactie* en *virtuele realiteit* waar een persoon interactief met een computer kan interageren door middel van bewegingen en gebaren [2, 3]. De muis kan als voorbeeld vervangen worden door een camera waardoor de gebruiker objecten kan selecteren en verplaatsen door er gewoon naar te wijzen. Er zijn ook reeds interactieve computerspellen die aangestuurd worden door middel van een webcam op de commerciële markt verschenen. De meest bekende tegenwoordig is de Eyetoy van Playstation 2 [4], maar ook in het verleden zijn er reeds succesvolle spellen uitgekomen. Door middel van de Artifical Retine Chip van Mitsubishi Electronics [2, 5] presenteerde Sega Saturn drie spellen: Nights, Magic Carpet en Decathlete.
Maar in tegenstelling tot dit werk maken alle vorige toepassingen gebruik van slechts 1 camera. Hierdoor is enkel twee dimensionale informatie beschikbaar. Wanneer meerdere beelden gebruikt worden als invoer kan ook de diepte informatie in rekening gebracht worden voor het aansturen van een spel.
Deze samenvatting start met een beschrijving van enkele veel onderzochte technieken uit de computervisie. Deze zullen eerst zeer beknopt beschreven worden. Vervolgens komt er een overzicht van de algoritmen die geïmplementeerd zijn geweest voor de computerspellen, gevolgd door de resultaten. Tot slot wordt er een besluit en toekomstig werk gegeven.

Gerelateerd werk

Deze sectie beschrijft enkele veel gebruikte computervisie technieken die bovendien ook gebruikt kunnen worden voor interactieve spellen. Ze worden onderverdeeld in *diepte schatting*, *visual hull constructie* en *object tracking*. Vooraleer er met meerdere camera's gewerkt kan worden, moet de relatie ertussen bekend zijn. Dit wordt ook wel camera kalibratie genoemd. Dit concept wordt als eerst aangehaald voordat wordt in gegaan op de computervisie methoden.

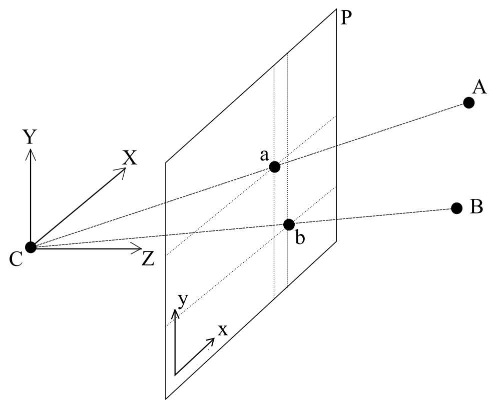

Figure 1: Een camera met centrum C en projectievlak P. Het drie dimensionaal assenstelsel (X, Y, Z) bevindt zich in de oorsprong van de camera. Het assenstelsel (x, y) is het twee dimensionaal assenstelsel van de foto op het vlak P. Bij het maken van een foto worden alle punten uit de drie dimensionale ruimte, A en B, geprojecteerd op het projectievlak, respectievelijk op de punten a en b.

Camera kalibratie

Voor het verder verloop van dit werk is het natuurlijk belangrijk om te weten hoe een camera werkt en hoe het gebruikt kan worden. Een camera bestaat uit een projectiecentrum en een projectievlak. Wanneer een foto genomen wordt, wordt de gefotografeerde scène geprojecteerd op dit projectievlak en deze projectie is de foto welke als resultaat bekomen wordt, zie foto 1. Dit proces kan wiskundig beschreven worden door een 4×3 projectiematrix welke een punt uit de drie dimensionale ruimte omzet naar een twee dimensionaal punt op het vlak. Een veel gebruikte matrix hiervoor is de interne kalibratiematrix welke de oorsprong van het drie dimensionale assenstelsel plaatst in het projectiecentrum van de camera. Camera's kunnen dan als volgt gekalibreerd worden door ervoor te zorgen dat het assenstelsel van de scène hetzelfde gekozen wordt voor elk van hen.
Als er met slechts twee camera's gewerkt wordt kan de relatie tussen de beelden ook beschreven worden met de fundamentele matrix. Wanneer er een punt op het eerste beeld bekend is kan de overeenkomstige rechte op het tweede beeld berekend worden door middel van deze matrix. Deze relateert dus een punt van één beeld met een rechte op het andere beeld, dit omdat het onmogelijk is exact te bepalen wat de projectie op het andere beeld is. Dit komt omdat de diepte van het eerste punt niet beschikbaar is [1].

Diepte schatting

De naam diepte schatting vertelt al ongeveer wat deze techniek inhoud. Met behulp van ten minste twee camera's wordt de afstand van een object tot een bepaalde camera ingeschat. Het principe hierachter kan vergeleken worden met het menselijk oog. Een mens ziet twee keer dezelfde scène, maar telkens vanuit een andere positie, linkeroog en rechteroog. Hierdoor is hij in staat de diepte van een voorwerp te bepalen. Deze diepte informatie kan gehaald worden

uit de verschillen van de twee beelden. Een object dat zich dichter bij de waarnemer bevindt, heeft de eigenschap meer verplaatst te zijn tussen de twee beelden dan een object verder weg. Deze eigenschap wordt ook in de computervisie uitgebuit om een afstand te bepalen. Een scène wordt gefilmd door twee dicht bij elkaar geplaatste camera's. Voor iedere pixel in het eerste beeld kan zijn correspondeerde pixel gezocht worden in het tweede beeld. De locaties waar gezocht moeten worden naar het overeenkomstige punt kunnen bepaald worden door middel van de kalibratie data. Ter vereenvoudiging wordt vaak met gerectificeerde beelden gewerkt, welke de eigenschap hebben dat de projectie van een punt op de twee beelden op dezelfde hoogte ligt. Daarom kan een overeenkomstige pixel gezocht worden op een horizontale rechte.
Een veel gebruikte voorstelling van de diepte informatie is een dieptemap. Dit is een twee dimensionale matrix met dezelfde resolutie als de invoer beelden waarvan iedere waarde gelijk is aan de diepte met de overeenkomstige pixel uit het referentiebeeld. Het berekenen van deze dieptemap kan op verschillende manieren gedaan worden, maar deze algoritmen bestaan vaak uit dezelfde vier stappen [6]. Deze stappen zijn *overeenkomst berekenen*, *overeenkomsten aggregeren*, *berekenen van verplaatsing* en *verbetering aanbrengen.*

Overeenkomst berekenen

Eerst moet de overeenkomst tussen twee pixels berekend kunnen worden. Dit wordt vaak gedaan door de absolute waarde of het kwadraat van hun verschil te nemen. Hoe kleiner deze waarde is hoe groter de overeenkomst tussen de twee pixels.

Overeenkomsten aggregeren

De overeenkomst tussen twee pixels alleen geeft onvoldoende informatie, daarom wordt de overeenkomst vaak over een venster rond de pixel berekend. Een groter venster geeft beter aan hoe goed twee pixels op elkaar gelijken. Veelgebruikte methoden hier zijn de som van het absolute verschil en de som van het kwadraat van de verschillen. Hierbij worden de overeenkomsten van de pixels over een volledig venster met elkaar opgeteld.

Berekenen van verplaatsing

Voor iedere pixel in het eerste beeld wordt de overkomst bepaald met alle mogelijk corresponderende pixels uit het andere beeld. De keuze van overeenkomstige pixel kan dan op twee verschillende manieren gebeuren: lokaal of globaal. Wanneer lokaal gewerkt wordt, wordt de corresponderende pixel deze met het kleinste verschil. In een globaal algoritme daarentegen streeft men naar een globaal minimum. De verplaatsing tussen een pixel en zijn overeenkomstige wordt in de dieptemap bijgehouden.

Verbetering aanbrengen

In de laatste stap worden soms nog enkele verbeteringen uitgevoerd om het resultaat te optimaliseren. Deze proberen ruis te vermijden of foute oplossingen te elimineren.

Visual hull constructie

Deze techniek houdt in dat er een drie dimensionale reconstructie wordt gemaakt van een object of persoon aan de hand van enkele projecties en wordt vaak ook shape-from-silhouette

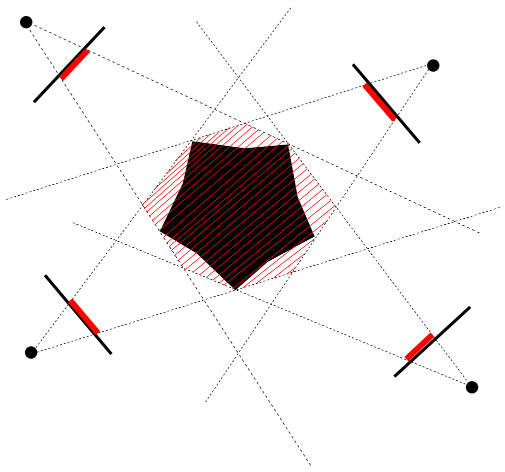

Figure 2: Het object wordt gefotografeerd met 4 verschillende cameras. Deze projecties worden terug geprojecteerd naar de ruimte in de vorm van een "kegel". De doorsnede van deze kegels, hier rood gearceerd, vormt de visual hull van het object.

genoemd. Een silhouette is een binair beeld welke alle pixels bevat van de projectie van het object en waarvan dus de achtergrond verwijderd is. Dit silhouette kan terug geprojecteerd worden naar de ruimte in de vorm van een "kegel" met het middelpunt van de camera als top. Deze kegel is dan de verzameling van alle drie dimensionale punten die geprojecteerd worden ergens op een voorgrondpixel. In het geval dat er meerdere beelden beschikbaar zijn wordt er een doorsnede genomen van deze kegels. Het resultaat van deze bewerking wordt ook wel een visual hull genoemd. Dit model is de verzameling van alle punten in de ruimte die op elke gegeven silhouette geprojecteerd worden. Deze reconstructie is daarom ook de best mogelijk oplossing startende van silhouetten [7]. Een twee dimensionaal voorbeeld wordt getoond op figuur 2.
De wijze waarop deze berekend kan worden, hangt grotendeels af van de manier hoe deze visual hull gaat voorgesteld worden. De meest voor de hand liggende representatie noemt constructive solid geometry (CGS). Deze beschrijft een drie dimensionaal model door middel van primitieven en operaties ertussen. In dit geval kunnen de primitieven de terug geprojecteerde kegels zijn en de operatie ertussen is de doorsnede. Een andere methode gaat de ruimte voorstellen door blokken, welke ook wel voxels worden genoemd. De term voxel is een samensmelting van de woorden *volumetric* en *pixel* en kan dus beschouwd worden als een drie dimensionale pixel. Deze blokken kunnen allemaal dezelfde grootte hebben, maar dit is niet noodzakelijk. Een voorbeeld is een octree [8] waarbij de ruimte bestaat uit een grote voxel, maar deze wordt steeds in acht opgesplitst wanneer het niet volledig binnen of volledig buiten de visual hull ligt. Een laatste voorbeeld van representatie is het definiëren van de randen. Dit kan gedaan worden door het model te benaderen met vlakken of hogere graads curves [9, 10].
Nadat de visual hull berekend is kan het gebruikt worden voor collision detection of om een drie dimensionaal model van de speler in het spel te plaatsen.

Object tracking

De laatste computervisie techniek die beschreven wordt in deze thesis heet object tracking en heeft als doel objecten te gaan lokaliseren en te volgen in een scène. Hiervoor zijn in de laatste 50 jaar veel algoritmen voor ontwikkeld waarvan in het bijzonder de Kalman filter [11] en de particle filter [12]. Deze twee zijn algoritmen voor algemene doeleinden en kunnen daarom gebruikt worden voor verschillende soorten objecten. Aan de andere kant bestaan er ook algoritmen speciaal bedoeld voor één soort tracking zoals het gezicht en de handen van personen [13, 14].

Kalman filter

Een Kalman filter is een efficiënte recursieve filter welke gebruikt kan worden om een toestand van een lineair systeem te schatten wanneer de metingen maar beperkt zijn en ruis bevatten. Deze toestand kan bijvoorbeeld de locatie en snelheid van het object zijn die gevolgd moet worden. Deze toestand kan door deze filter benaderd worden door een reeks geobserveerde posities, welke ook fouten kunnen bevatten.
Een uitbreiding op dit algoritme is de extended Kalman filter welke er niet vanuit gaat dat het systeem lineair is. Deze filter kan bijvoorbeeld gebruikt worden om een persoon te volgen in een scène. Hierbij is het mogelijk een benadering van de speler zijn locatie te vinden zonder de volledige scène te moeten doorzoeken [11, 15, 16, 17].

Particle filter

Een particle filter heeft hetzelfde doel als de Kalman filter maar gaat dit op een andere manier aanpakken. Hierbij wordt een dichtheidsfunctie opgesteld van de mogelijke waarden van de toestand, welke benaderd wordt door middel van een Monte Carlo simulatie. Deze simulatie creëert particles welke een bepaalde toestand beschrijven, zoals de locatie van het object. Bovendien bevat iedere particle ook een gewicht welke de waarschijnlijkheid van de toestand voorstelt. Bij het uitvoeren van deze filter zal de particle dichtheid rond de echte toestand van het object het grootste zijn. Een intuitieve variant op de particle filter heet het condensation algoritme en wordt later nog beschreven [12, 18].

Kinematica

Na het lokaliseren van bepaalde lichaamsdelen is het mogelijk om het volledig menselijk lichaam in de vorm van een skelet voor te stellen. Dit skelet is een boomstructuur van beenderen die verbonden zijn met gewrichten. Een *pose* is een mogelijk opstelling van dit skelet door alle hoeken van alle gewrichten te bepalen, wat ook wel *voorwaartse kinematica* wordt genoemd.
In andere methode wordt de gewenste locatie van de gewichten op het einde van de boom, zoals bijvoorbeeld de handen, bepaald en de hoeken van de tussenliggende gewichten berekend zodat deze doel locatie bereikt wordt. Dit wordt meestal op een incrementele manier gedaan zodat er een vlotte beweging gegenereerd wordt. Deze laatste techniek wordt ook wel *inverse kinematica* genoemd [19, 20].

Implementatie

De hierboven beschreven technieken kunnen gebruikt worden voor het aansturen van computer spellen. In dit gedeelte worden daarom enkele algoritmes besproken die dienen voor het berekenen van een dieptemap, visual hull of locatie van objecten.

Diepte schatting

Een eerste algoritme wordt gebruikt om een dieptemap te berekenen van een bepaalde scène [21]. De invoerbeelden zijn gerectificeerd zodat een corresponderende pixel op dezelfde hoogte gezocht moet worden in beide invoerbeelden. Om de overeenkomst te bepalen wordt de som van het absoluut verschil over een venster berekend. De verplaatsing waarvoor het verschil tussen de pixels het kleinste is wordt gekozen.
Om dit algoritme te optimaliseren worden berekeningen die reeds gebeurd zijn niet meer opnieuw uitgevoerd. Dit is vooral van toepassing bij het berekenen van de som van het absoluut verschil van een venster. Nadat deze waarde over het venster berekend is, wordt het venster één stap naar rechts opgeschoven. De nieuwe overeenkomst kan dan berekend worden door de linkse kolom van het venster van de vorige uitkomst af te trekken en enkel de rechtse kolom erbij op te tellen. Hierdoor worden alle waarden ertussen behouden en moeten ze niet meer herberekend worden. Een andere optimalisatie houdt in dat de volgorde van de for-lussen goed worden geschikt. Deze lussen dienen om door de x en y coordinaten te lopen en door de diepte. Door deze volgorde aan te passen wordt het cache geheugen meer optimaal gebruikt resulterend in een enorme versnelling. Een laatste optimalisatie zorgt niet voor een versnelling, maar voor een verbetering van de dieptemap. Door een mediaanfilter over de dieptemap uit te voeren wordt de ruis gereduceerd.

Visual hull

Een eerste punt welke bij al deze algoritmen aan bod komt is het verwijderen van de achtergrond. Hiervoor is het noodzakelijk om de scène te kennen. Dynamische achtergronden [22] kunnen dit proces bemoeilijken en daarom is er meestal voor gekozen om een groen doek achter de speler op te hangen. Elke pixel waarbij de groene component groter is dan de rode en de blauwe wordt als achtergrond beschouwd, de rest als voorgrond.
Wanneer de visual hull wordt voorgesteld door kubus vormige voxels bestaat er een eenvoudige manier om deze te construeren. Het centrum van elke voxel wordt geprojecteerd op elk projectievlak en wanneer deze binnen elke silhouette ligt, wordt de voxel aan de visual hull toegevoegd. Een octree voorstelling kan ook real time berekend worden zonder eerst de bovenstaande visual hull te berekenen. Dit gebeurt door in een eerste stap voor elke pixel te berekenen wat zijn afstand tot de rand van de silhouette is. Wanneer er dan een beslissing moet genomen worden of een voxel in acht opgesplitst moet worden of niet kan het centrum geprojecteerd worden. Door de afstand die de pixel bevat te vergelijken met de doorsnede van de voxel kan er snel nagegaan worden of deze volledig binnen of volledig buiten is. Wanneer geen van beiden vastgesteld kan worden zal er verder opgesplitst moeten worden [23].
De visual hull kan tegenwoordig ook op de grafische kaart geprogrammeerd worden. Een voorbeeld hiervan is waar de ruimte wordt voorgesteld door een serie parallel lopende vlakken. Op elk vlak worden de verschillende invoerbeelden geprojecteerd. Pixels die deel uitmaken van de voorgrond krijgen een alfa waarde van 1, de achtergrond een waarde van 0. In een

fragmentshader wordt de doorsnede van deze verschillende projecties berekend en de kleur van ieder fragment wordt op een gewogen manier samengesteld vanuit de referentiebeelden. Deze gewichten zijn bepaald door de oriëntatie tussen de virtuele camera en de reële camera's [24].

Object tracking

Het eerste algoritme voor object tracking heet het condensation algoritme en is een variant op de particle filter. Het wordt toegepast om een object te vinden en te volgen op basis van de kleur. Als voorbeeld is een rode pingpong bal gekozen. In de voorbereidende stap wordt een verzameling particles aangemaakt die elk eenzelfde gewicht krijgen. Iedere keer wanneer er nieuwe invoer is, worden drie stappen uitgevoerd om de locatie van de bal te bepalen [25, 26].

Selectie stap

In de eerste stap wordt een nieuwe verzameling particles aangemaakt op basis van de vorige verzameling en hun gewichten. Hierbij worden de oude particles met een hoog gewicht vaker herkozen terwijl deze met een gewicht 0 niet meer opnieuw zullen voorkomen. Dit gebeurt met behulp van het cumulatieve gewicht van de particles.

Voorspellende stap

Om elke particle opnieuw een unieke locatie te geven wordt er bij iedere x, y en z coördinaat een gaussiaanse random waarde opgeteld. Dit zorgt voor een verstrooiing. Het resultaat na deze twee stappen is dat er zich rond de positie van het te volgen object veel meer particles bevinden dan elders in de ruimte.

Meet stap

Tot slot krijgt iedere particle een nieuw gewicht toegekend. Dit gebeurt door de locatie te projecteren op de invoerbeelden en de overkomst met het gezochte te gaan berekenen. Hoe groter deze overkomst hoe hoger het gewicht.

Een tweede algoritme dat nuttig kan zijn voor het aansturen van interactieve computerspellen dient voor het lokaliseren van de handen en het gezicht van de speler. Ook deze is opgesplitst in drie stappen [14].
Dit start in de eerste stap met het detecteren van pixels die gelijken op de huidskleur van de speler. Dit kan gedaan worden op basis van enkele voorbeeld pixels van de persoon. Aan elkaar grenzende pixels met huidskleur kunnen gegroepeerd worden tot een zogenaamde blob.
Een volgende stap associeert deze blobs met de reeds gekende locaties van handen en gezicht uit de vorige frame door te kijken hoeveel pixels ze gemeen hebben. De nieuwe posities worden geüpdate door de locatie van de overeenkomstige blobs te gebruiken. Eens de locaties bekend zijn op de twee invoerbeelden kan de drie dimensionale positie berekend worden. Dit gebeurt door het terug projecteren van de punten naar een rechte in de ruimte en hun doorsnede te berekenen, ook wel bekend as triangulatie.
In een volgende stap kunnen de gevonden locaties gebruikt worden voor het aansturen van een skelet door middel van inverse kinematica. In dit werk is dit gebeurd met behulp van de getransponeerde jacobiaan methode [19].

Resultaten

De beschreven algoritmen hebben uiteenlopende eigenschappen waardoor ze elk hun voor- en nadelen kennen. De keuze welke gebruikt moeten worden in een spel hangt daarom af van verschillende factoren. Deze zijn hieronder opgesomd samen met hun sterkten en zwakten.

Camera opstelling

Twee soorten opstellingen kunnen onderscheiden worden voor camera's. Deze worden *small baseline* en *wide baseline* genoemd. Hierbij staan de camera's respectievelijk dicht bij elkaar of zijn ze ver van elkaar verwijderd. Voor een diepteschatting is het beter dat ze dicht bij elkaar staan, maar voor een visual hull reconstructie is het net omgekeerd. Deze opstelling zal dus in grote mate de keuze van algoritme beïnvloeden. Voor een commerciëel product heeft een small baseline opstelling nog een extra voordeel. De camera's kunnen in een opstelling aan elkaar vast gemaakt worden waardoor ze niet gekalibreerd moeten worden voordat een spel gaat beginnen.

Resolutie van de invoerbeelden

De complexiteit van sommige algoritmen, waaronder constructie van een octree [23] visual hull en het volgen van handen en gezicht [14], hangt af van de resolutie van de invoerbeelden. Dit is vooral duidelijk voor het berekenen van een dieptemap. Als er een berekening voor elke pixel moet uitgevoerd worden is het natuurlijk veel voordeliger dat de resolutie laag is. Algoritmen zoals het berekenen van de visual hull bestaande uit kubus vormige voxels en de particle filter kennen geen vertraging wanneer de invoerbeelden een hogere resolutie hebben.

Aantal camera's

De berekening van een octree visual hull heeft een eerste stap waarin voor elk beeld een transformatie plaatsvindt. Hierdoor resulteren meerdere beelden in een enorme vertraging. Ditzelfde geldt voor het het berekenen van een dieptemap en het volgen van handen en gezicht. Langs de andere kant zorgen meerdere camera's ook voor een beter resultaat. Een dieptemap kan verbeterd worden omdat er meer referenties zijn en een visual hull kan nauwkeuriger berekend worden omdat de doorsnede compacter wordt. Ook object tracking kent een voordeel dat occlusies kunnen aanpakt worden. Zolang het object zichtbaar is in ten minste twee beelden kan zijn locatie gevonden worden.

Gewenste framerate

Omdat het zeer belangrijk is voor een interactief computer spel dat er een bepaalde framerate wordt gehaald moet de nauwkeurigheid van het algoritme aangepast kunnen worden. Door nauwkeurigheid op te offeren kan het algoritme sneller uitgevoerd worden resulterend in een hogere framerate. Het aantal voxels waaruit een visual hull bestaat beïnvloedt op deze manier de snelheid van het spel. Voor de particle filter als ander voorbeeld is deze parameter het aantal particles.

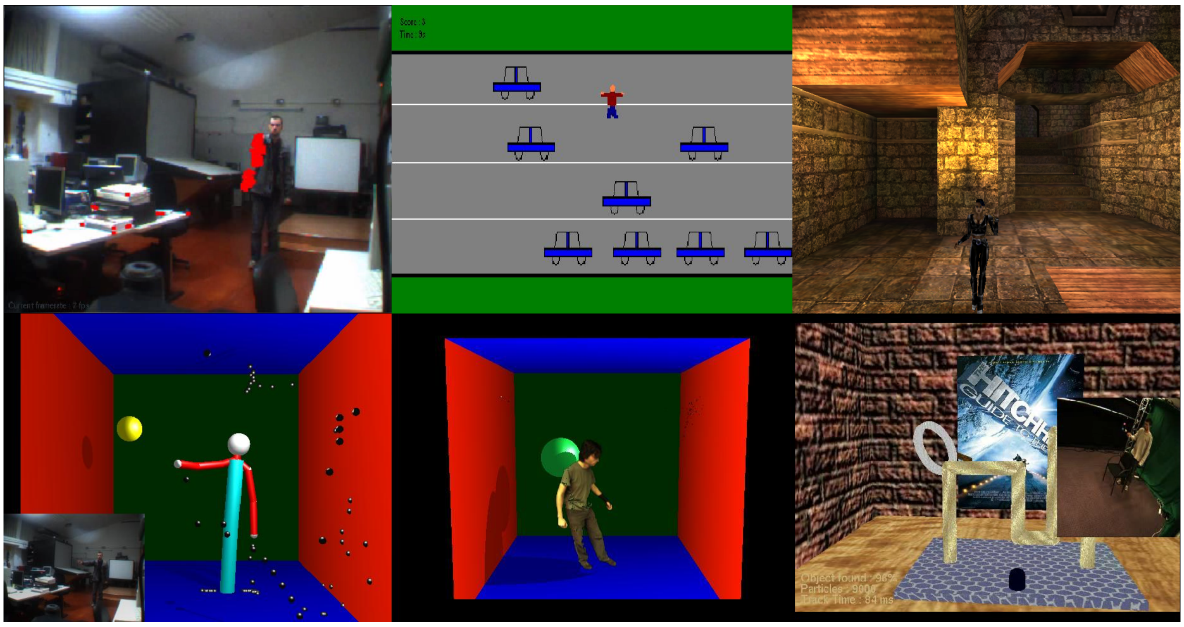

Figure 3: (a) Een 3d tekenspel waarbij de enkel bewegingen voor een bepaalde afstand geregistreerd worden (b) De locatie van de speler worden bepaald met behulp van een dieptemap en naar het spel Frogger gemapt (c) De speler wordt door middel van een visual hull gelocaliseerd en in de 3d engine irrlicht [28] geplaatst (d) Een skelet in het spel doet dezelfde bewegingen als de speler, de bedoeling is om de gele ballonnen af te schieten (e) De speler moet de objecten in de ruimte proberen allelei objecten te ontwijken (f) De speler moet een pad afleggen met een ringvormig object zonder de spiraal aan te raken.

Speltype

Tot slot bepaalt uiteraard het soort spel dat gemaakt wordt welke de achterliggende algoritmen zijn. In dit werk worden 6 verschillende spellen gepresenteerd, allen gebruikmakend van een ander invoer algoritme. Bovendien zijn er ook verschillende manieren van uitvoer gebruikt. Een eerste spel is een drie dimensionaal tekenspel waarbij bewegingen op een bepaalde afstand opgenomen worden. Frogger is het tweede spel waarbij de locatie van de speler berekend wordt met behulp van de dieptemap en naar twee dimensionaal spel gemapt wordt. De volgende twee spellen maken elk gebruik van een visual hull. Door naar het centrum van de massa te kijken van de visual hull kan een persoon in een drie dimentionaal spel geplaatst worden en kunnen enkele animaties toegevoegd worden. Door bijvoorbeeld te kijken naar de hoogte van dit punt kan er verondersteld worden of de speler rechtstaat, gebukt is of springt. Het andere spel gebruikt de visual hull voor collision detection en response tussen de speler en de virtuele objecten. Een vijfde spel gebruikt de particle filter om een object te tracken en zijn locatie en orientatie te mappen naar een virtuele wereld [27]. Als finaal spel worden de handen en gezicht van de speler gedetecteerd en gebruikt om een skelet in het spel aan te sturen met behulp van inverse kinematica. Deze worden getoond op figuur 3.

Conclusie

In dit werk werden verschillende spellen voorgesteld die aangestuurd worden door middel van meerdere camera's. Hiervoor is gebruikt gemaakt van diepteschatting, visual hull constructies en object tracking technieken om drie dimensionale informatie uit meerdere invoer beelden te verkrijgen. Ze werden gebruikt voor het localiseren van de speler, objecten of bepaalde gebeurtenissen zoals bewegingen om zo de verschillende interactieve spellen aan te sturen.
Voor spellen is nauwkeurigheid vaak minder belangrijk dan de interactie en daarom wordt de berekeningstijd vaak beperkt. De visual hull is bijvoorbeeld slechts een benadering van de speler en is bovendien weinig gedetaileerd berekend om toch interactief te kunnen blijven. Maar toch blijkt dat deze geschikt is voor een realistische interactie tussen speler en virtuele objecten. Ook inverse kinematica wordt slechts gebruikt voor het schatten van de houding van de speler omdat enkel de locaties van de handen en gezicht bekend zijn. Toch geeft deze methode een realistisch resultaat tegen een hoge snelheid.
De speler heeft geen fysiek contact met de computer en toch kan hij het spel op een intuïtieve manier aansturen omdat de avatar in het spel dezelfde bewegingen maakt als hem. Hierdoor onstaan ook enkele onrealistische situaties zoals een muur in het spel waar de speler niet door kan gaan terwijl er in werkelijkheid geen muur staat.
De extra diepte informatie die gewonnen kan worden door het gebruik van meerdere camera's zorgt voor een enorme verhoging van het aantal mogelijke spellen die kunnen gecreëerd worden.
De conclusie van dit werk is dat de oplossing voor het maken van een multi camera video spel te vinden is in het computervisie vakgebied. Hier zijn slechts enkele algoritmen gepresenteerd, maar er bestaan nog talrijke andere die voor hetzelfde doel kunnen gebruikt worden.

Toekomstig werk

Om meer nauwkeurige oplossingen te bekomen kan er ook gebruik gemaakt worden van de grafische kaart. Deze is reeds nuttig bevonden voor het berekenen van een dieptemap [29, 30] of collision detection [31, 32].
Een andere mooie uitbreiding is het detecteren van botsingen in "imagespace", zonder eerst een visual hull te berekenen [33].
Het schatten van de houding van de speler door middel van inverse kinematica heeft momenteel nog de beperking dat er verwacht wordt dat de speler gedurende het spel op dezelfde plaats blijft staan. Deze beperking zou bijvoorbeeld weggewerkt kunnen worden door verschillende technieken te combineren. Door bijvoorbeeld eerst de locatie van de speler te bepalen door middel van een visual hull en pas nadien de houding door object tracking en inverse kinematica.
De spellen in dit werk zijn beperkt gebleven tot één enkele speler per spel. Om dit uit te breiden naar meerdere spelers kunnen we twee scenarios onderscheiden. De meerdere spelers bevinden zich in dezelfde ruimte en worden met dezelfde camera's gefilmd. In dit geval moet de informatie van de verschillende spelers onderscheiden kunnen worden. De tweede situatie is waar de verschillende spelers zich in verschillende ruimten bevinden. De computers waarop de spellen draaien zijn dan in een netwerk op elkaar aangesloten. Hiervoor is het dus nodig het netwerkaspect te onderzoeken opdat de vertraging, door het versturen van de juiste informatie over het netwerk, minimaal is.

1
Introduction

The purpose of this work is to use two or more calibrated cameras to film the scene and the player. From those the necessary data will be extracted, depending on the kind of information required. Because a game has to be interactive, the framerate must be high and therefore it is not always necessary to obtain every detail of the player. Detecting his location for example can sometimes also be useful and more efficient. But when more information is extracted, more sophisticated games can be created. The purpose is to only extract the required information to reach the decent frame rate. The extraction of this kind of data is researched in the area of computer vision.
To achieve a natural interaction based on computer vision we would require visual competence near the level of a human being, which is still beyond the state of art [34]. Therefore the vision problem must be restricted to the information needed. For example if a person is walking around in a room. The only interest is his location while an understanding of the full activity of the person or the description of the room is not necessary. Some of those sub-problems can be solved with computer vision techniques.

1.1 Computer Vision

Making a computer see is something that started in the field of Artificial Intelligence in the sixties. The real studies started in the late 1970's, because before computers were not capable of processing large data sets like for example images. Current days this problem is still unsolved. The field, called computer vision, has emerged as a discipline in itself with strong connections to mathematics and computer science. One of the reasons why this problem is still being researched is the fact that perception in general and visual perception in particular are far more complex than was initially thought. Over the years, computer vision researchers have obtained some outstanding successes, both practical and theoretical [1].
One example of the practical accomplishments is the possibility of guiding vehicles such as cars on regular roads or on rough terrain using computer vision. This was demonstrated many years ago in Europe, USA and Japan. This requires a real time three dimensional

dynamic scene analysis. Today, many car manufacturers are slowly incorporating some of these functions in their products.
On the theoretical side, there are a lot of achievements in the area of *geometric computer vision*. This is the field that researches the way the appearance of objects changes when viewed from different viewpoints as a function of the objects' shape and the camera parameters, by using sophisticated mathematical techniques encompassing many areas of geometry.

1.2 Previous Work

There has been a lot of research about interaction with a computer using one or more cameras. The biggest research domains are Human Computer Interaction, Virtual Reality, but also computer games. The collection of applications is very large like using hand gestures instead of a remote control for your television as presented by Ohta et al [2]. This technique will record the actor and track his hands. When it locates the hand it will try to understand the actors pose and the movements he makes. Eventually, it will take the corresponding action.
A good example in the category virtual reality is interacting with a virtual human presented by Thalmann [3]. The actor is recorded and placed into a virtual reality world. In this world there are also virtual characters which will correspond to the actions of the actor. Another example is replacing the mouse in your computer with a camera. The actor can point with his finger to the screen to select an object. This was presented by Marc Erick Latoschik [35].
There has been a lot of inventions in the domain Interactive Computer Games as well. Because of the need for low cost and high speed Mitsubishi Electric invented the Artificial Retina Chip [36] (by analogy with biological retinas which also combine the functions of detection and processing) in 1996. This chip is 6.5 by 6.5 millimeters and has a built-in detector for 32×32 to 256×256 images. The detected image is just a matrix and the processing can therefore be expressed as a matrix equation [2, 5]. The used algorithms are called *images moments* and *orientation histograms* to extract information for interacting with the game. Three games where published for Sega Saturn using this chip : Nights, Magic Carpet and Decathlete (figure 1.1). Also the Nintento Gameboy Camera used the artificial retina chip in 1998 to allow the an image of the players face to be inserted into a simple game.
Eventually Sony created a commercial camera for the Playstation 2 and soon for the Playstation 3, called the *Eye Toy* [4]. Figure 1.2 shows an image of *Eye toy : kinetic combat* presented at E3 2006.
But all these previous described image based games only used one input camera and therefore only two dimensions, no depth information. When working with multiple cameras there is the possibility to use three dimensional information from the scene.

1.3 Overview

The remains of this thesis is divided into several chapters. The next chapter will give a small overview of some of the current computer vision techniques which can be used for interacting between real objects and virtual objects. These techniques are divided into *depth estimation*, *visual hull* and *object tracking*. The important topics for these techniques are the representation of the data, the calculation and how they can be used.

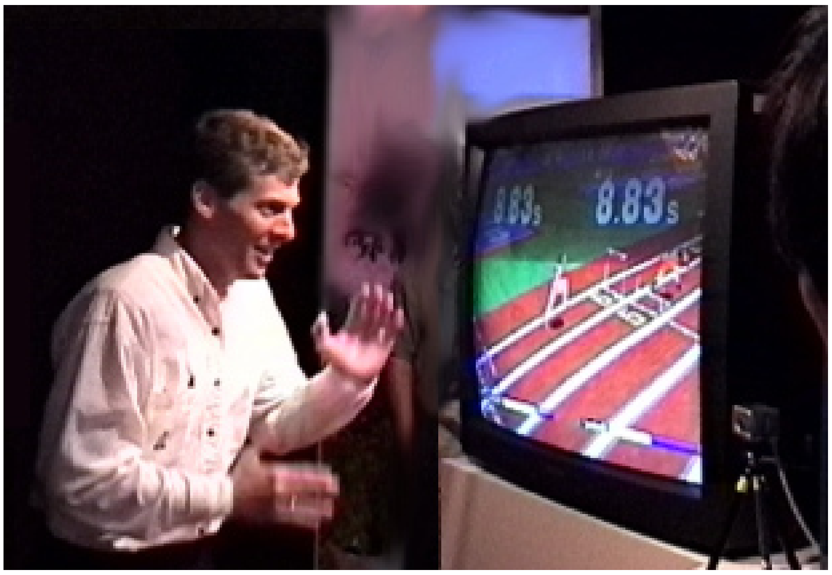

Figure 1.1: Decathlete is a game for Sega Saturn where the movements of the player are projected into the game

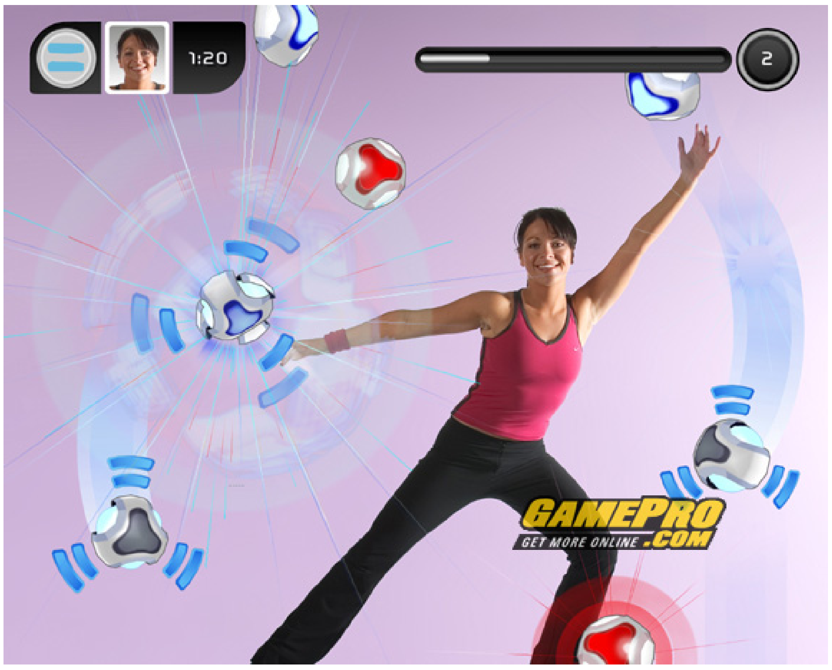

Figure 1.2: Eye toy : kinetic combat. A game for playstation 2 using 1 camera to map the player in a combat game.

After describing some algorithms to obtain this kind of information a more detailed description of the implementation is given. Chapter 3 describes the implementation of the algorithms used in this thesis.
These algorithms will be compared with each other in chapter 4. Each algorithm has its advantages and disadvantages. These can be defined based on some parameters like image resolution and number of cameras. Also six example games are presented in that chapter.
The conclusions are given in chapter 5.
The final chapter describes shortly the future work of this thesis.

2 Related Work

This chapter gives an overview of some modern computer vision techniques which can be used for obtaining useful information from a scene.
Because computer vision usually needs more then one camera it is necessary to have an understanding of the location and orientation of each of them. So the relationship between the cameras must be known. This relationship is called *camera calibration* and a short introduction is given in section 2.1.
The next section, section 2.2, describes the first common used technique for extracting three dimensional information. The technique is called *depth estimation* and as the name suggests it is used to estimate the depth of an object. In other words, the distance from a camera to an object can be estimated using two or more cameras. With this depth information a three dimensional reconstruction of the scene can be estimated.
Section 2.3 describes the *visual hull*. This is a three dimensional model of an object, created from two or more images. This technique is often also called shape-from-silhouette because it only requires the silhouettes of the object on several image planes. This reconstruction is not always equal to the real object, but it is the best approximation possible from silhouettes.
The last technique uses a different approach. Instead of trying to create a three dimensional model of the scene, it is used for locating and following objects or part of them. Therefore this technique is called *object tracking*. It can be used to only obtain necessary information and can give a better understanding of the scene. For example after reconstructing a person with visual hull techniques the location of the hands and face are still unknown. This is given in section 2.4.

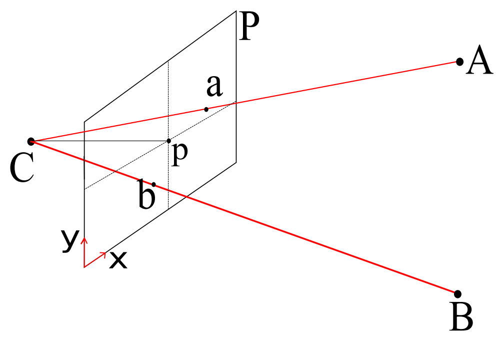

Figure 2.1: A camera is represented by a centre point (C) and an image plane (P). Three dimensional point A and B in space are projected on the image plane at respectively point a and b. The closest point on the imageplane to C is p and is called the principle point.

2.1 Calibration

Before starting this work, it sounds logic to explain what a camera is and how it works. In other words, the link between three dimensional entities and their images must be described. Another step towards computer vision is knowing the relationship between multiple cameras. These basic principles are explained in the next sections before going on to the computer vision techniques. The same notation as Hartley et al [1] is used here.

2.1.1 Description of a Camera

A camera in computer vision is generally represented as a centre point and an image plane. This is shown at figure 2.1. The image plane P is the actual image received from the camera. The recorded world is projected onto this plane with the centre point C as centre of projection. This is described for the three dimensional points A and B. They are projected on the image plane by taking the intersection between respectively the line CA and CB with the plane P, resulting in the points a and b.
The relationship between the three dimensional location and the projected location can be expressed as a matrix multiplication. The mapping from a point in a world coordinate frame to a point in image coordinates is given by:

$$x = PX \tag{2.1}$$

where P is a 4×3 matrix, because of the use of homogeneous coordinates. The concrete values of this matrix are explained later when calibrated cameras are introduced.
The point on the imageplane which is the closest to the centre of projection is called the *principle point* p. The line created from the centre of projection through this point is therefore called the *principle line*. At last, the distance C to p is called the focal length f.

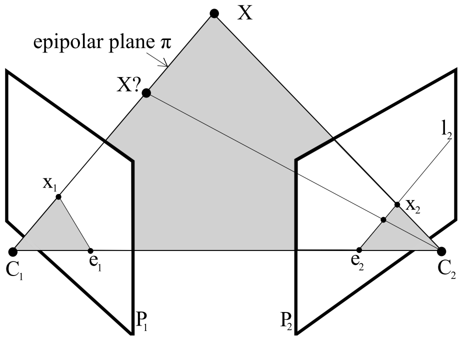

Figure 2.2: The two cameras with centre points C_1 and C_2 and image planes P_1 and P_2. The camera centres and the three dimensional space point X define the plane π. The projections x_1 and x_2 on the cameras are in the same plane. If only the projection x_1 is known, the exact location of X is restricted to a line in space, this line projected onto P_2 is l_2

2.1.2 Epipolar Geometry

The epipolar geometry is the intrinsic projective geometry between two views. It is independent of scene structure, and only depends on the cameras' internal parameters and relative pose.
This geometric relation between two views is essentially the geometry of the intersection of the image planes with the pencil of planes having the baseline as axis. The baseline is the line joining the camera centres. The geometric relation is usually motivated by considering the search for corresponding points in stereo matching.
When the projection of a three dimensional point is known on the first image, the corresponding projection onto the other image is constrained. This is shown in figure 2.2 where the projection x_1 of X is known. The real location of X is now constrained to a line started in C_1 and going through x_1. This line together with the centre of projection C_2 of the other image define a plane π. The possible locations of x_2 is the intersection of π and image plane P_2. This intersection is the line l_2.
The geometric entities involved in epipolar geometry are:

- The epipole of imageplane P_1 is the point of intersection of the line C_1C_2, joining the camera centres, with the plane P_1. This point is shown in figure 2.2 as e_1.
- An epipolar plane is a plane containing the baseline and going through a point in space. The plane π is for example the epipolar plane going through the point X.
- An epipolar line is the intersection of an epipolar plane with the image plane. All epipolar lines intersect at the epipole. l_2 is an epipolar line on image plane P_2.

The fundamental matrix F is the algebraic representation of epipolar geometry. It is a 3×3 matrix of rank 2. The relationship between the projections x_1 and x_2 is given as:

$$x_2^T F x_1 = 0 \tag{2.2}$$

2.1.3 Calibrated Cameras

To work with multiple cameras, a projection matrix for each of them is used. Each matrix describes the relationship between the Euclidean position in space and its image coordinates. To calculate this matrix the *camera calibration matrix* of each camera is required. A simplified representation of this matrix is described as follows

$$K = \begin{bmatrix} f & 0 & p_x \\ 0 & f & p_y \\ 0 & 0 & 1 \end{bmatrix} \tag{2.3}$$

and its parameters are

- f is the focal length as described before. This is the distance between the centre of projection and the principle point.
- (p_x, p_y) are the coordinates of the principle point.

The parameters contained in K are called the *internal camera parameters.*
This representation of the camera calibration matrix is correct when the image cordinates are Euclidean coordinates having equal scales in both axial directions. In reality this is not always the case. The next equation represents the calibration matrix if the number of pixels per unit distance in image coordinates are m_x and m_y in the x and y direction.

$$K = \begin{bmatrix} \alpha_x & s & x_0 \\ 0 & \alpha_y & y_0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.4}$$

In this equation $\alpha_x = fm_x$ and $\alpha_y = fm_y$, which represent the focal length of the camera in terms of pixel dimensions in the x and y direction respectively. These parameters are shown on figure 2.3.
For generality, the parameter s is also added to the matrix which is referred to as the skew parameter. The skew parameter will be zero for most normal cameras.
In addition to the linear effects summarized in the K matrix, there are other nonlinear and second order effects such as lens distortion. The variables in K and the second-order effects can be approximated and compensated for via standard corrective warping techniques [37].
Each camera has a camera coordinate frame with origin in the centre of projection and the principle axis of the camera pointing down the z-axis. For a point X_{cam} expressed in this coordinate system its projection is calculated with the camera calibration matrix:

$$x = K[I|0]X_{cam} \tag{2.5}$$

Normaly a different Euclidean coordinate frame is used, known as the world coordiante frame. If $\widetilde{X}$ is the coordinates of a point in the world coordinate frame, and $\widetilde{X}_{cam}$ represents the same point in the camera coordinate frame, then the next equation is valid:

$$\widetilde{X}_{cam} = R(\widetilde{X} - \widetilde{C}) \tag{2.6}$$

where $\widetilde{C}$ represents the coordinates of the camera centre in the world coordinate frame, and R is a 3×3 rotation matrix representing the orientation of the camera coordinate frame.

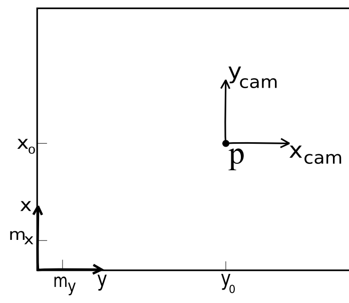

Figure 2.3: (x, y) is the coordinate system of the image, p the principle point of the camera. (x_{cam}, y_{cam}) is the coordinate system of the camera. m_x and m_y is the number of pixels per unit distance.

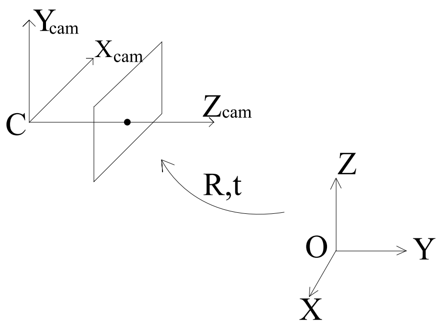

Figure 2.4: The left image is a representation of the camera coordinate frame with centre C. The right image is the world coordinate frame. The relation between them is the orientation R and the translation t [1]

These coordinate frames are presented in figure 2.4.
The next step is to find the projection matrix for this world coordinate frame. This is done using the *external parameters*. These parameters are R and $\widetilde{C}$ which relate the camera orientation and position to a world coordinate system. The projection matrix can then be written as:

$$P = KR[I| - \widetilde{C}] \tag{2.7}$$

Instead of using the camera centre explicit, the world to image transformation is represented as $\widetilde{X}_{cam} = R\widetilde{X} + t$. Here is $t = -R\widetilde{C}$. Therefore the projection matrix for the camera is defined as:

$$P = K[R|t] \tag{2.8}$$

Given this projection matrix for a camera and a three dimensional point, the projection can be calculated on the image.

$$x = PX \tag{2.9}$$

If all the cameras use the same world coordinate frame, then they are calibrated.

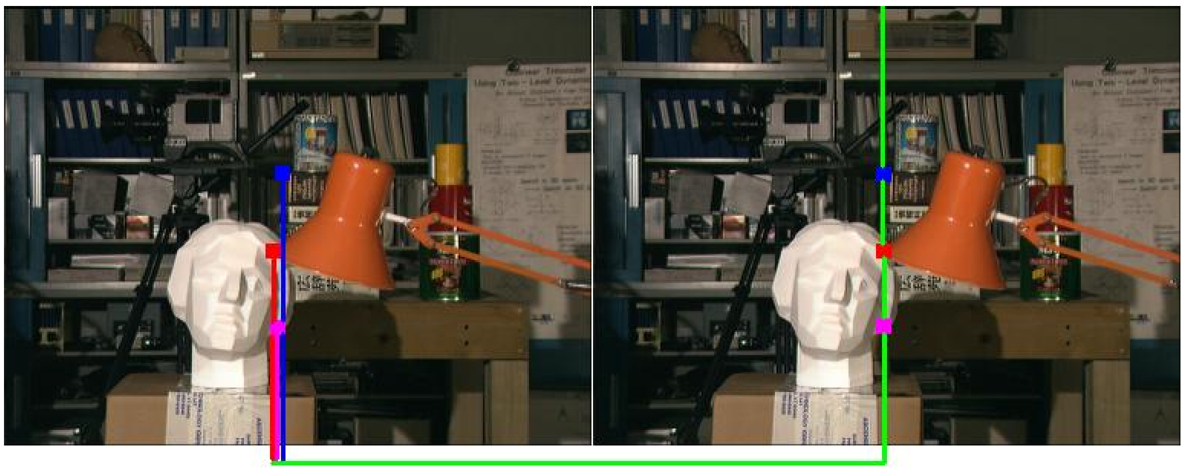

Figure 2.5: These images are from the same scene but the camera is translated over the x axis. The blue dot is a point on a book in the background, the red dot is the corner of the lamp and the purple dot is a point on the statue. They are marked in both images to show their disparity.

2.2 Depth Estimation

This is a three dimensional computer vision technique for calculating the depth of a scene given two-dimensional images [6]. Just like the human vision uses two eyes, multiple images can be made of the same scene as well. Then by searching for the displacements of objects in the two images the computer makes an estimate of the distance from the camera to the object. This displacement is often called *disparity* in human vision literature.
This is a very active research areas in computer vision. A lot of these algorithms only use two input images and are therefore called stereo depth estimation.
To make it easier to understand the concept of disparity calculation a small example is given in next section.

2.2.1 General Concept of Depth Estimation

Depth estimation is possible when two or more cameras are available by comparing the images. Figure 2.5 shows two images from the same scene. The two cameras that make the images were placed next to each other just like the human eyes. In the two images three specific points are marked. The blue mark is a point on a book in the background, the red mark the corner of the lamp which is close the cameras and the purple mark is a point on the statue. In the right image the three marks have the same x coordinate, but not in the left image. Objects closer to the camera have a larger translation in the two images then objects further away. So the displacement of the lamp is larger then the displacement of a book in the background. This concept is shown in figure 2.6. Here a point on the left image is back projected into space and as shown, the further it is away from the camera the more it is translated on the other image plane.

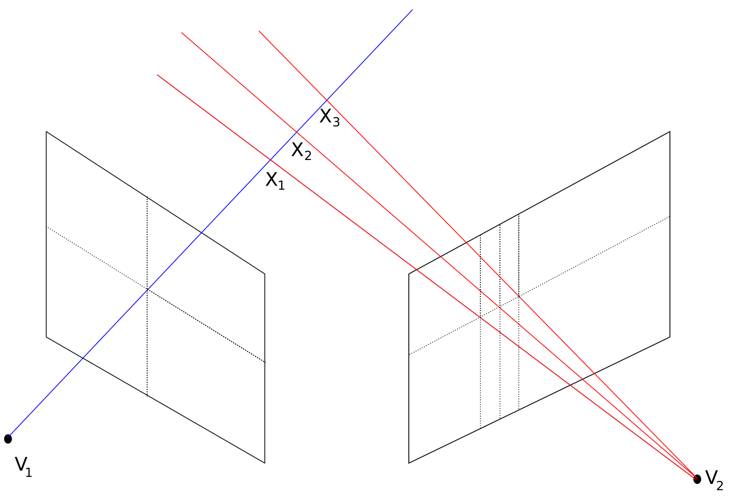

Figure 2.6: The three dimensional points X_1, X_2 and X_3 are projected on the images with viewpoints V_1 and V_2. The projections on the first image are on all the same point, but the further X_i is from the first image the further it is translated on the second image.

2.2.2 Representation

Most stereo correspondence methods compute a uni-valued disparity function $d(x, y)$ with respect to one of the input images meaning that every pixel, with coordinate (x, y), of the reference image has a single disparity value. The disparity is kept for each pixel of a reference image. Some other representations are:

- Multi-valued representation: instead of working in (x, y, d) space, (x, y, d, k) space is used where k refers to the index of the input image. The coordinates x, y and d are the coordinates with respect to a virtual camera. Szeliski et al [38] used this representation.
- Voxel-based representation: for example by identifying a set of invariant voxels which together form a spatial reconstruction of the scene (Seitz et al [39]).
- Layer-based representation: by representing the scene as a collection of approximately planar layers with a per-pixel opacity and depth offset relative to the plane (Szeliski [40]).
- Triangulated meshes: the scene is represented by a regular three dimensional triangulation like the algorithm by Fau et al [41].

This disparity function $d(x, y)$ is also called the disparity map and the (x, y, d)-space is therefore called disparity space. Several researchers have defined this disparity as a three dimensional projective transformation of 3D space (X, Y, Z). The enumeration of all possible matches in such a generalized disparity space can be achieved by projecting, for every disparity d, all images onto a common plane using a perspective projection, also called a homograph.
A common constraint is that the input images should be horizontal linear rectified. This means that the projection of a given point in the scene is on the same horizontal line in all

reference images. Therefore if (x, y) and (x', y') are projections of the same scene point the following equation is valid :

$$\begin{cases} x' = x + sd(x, y) \\ y' = y \end{cases} \tag{2.10}$$

where $s = \pm 1$ and the sign is chosen so that disparities are always positive.
Another concept to be introduced is the disparity space image (DSI). This is any image or function defined over a continuous or discredited version of disparity space. In practice, the DSI usually represents the cost or likelihood of a particular match implied by $d(x, y)$.

2.2.3 Structure of a Stereo Algorithm

Every stereo algorithm has a different approach for finding the disparity map, but most of them are divided into the same sub-problems with just another implementation. These sub-problems are:

- Matching cost computation
- Cost aggregation
- Disparity computation / optimization
- Disparity refinement

They will be described more in detail later. Based on the implementations of some sub-problems most of the stereo algorithms can be put into categories:

- Local algorithms (window-based) : the disparity computation at a given point only depends on intensity values within a finite window.
- Global algorithms make explicit smoothness assumptions and then solve an optimization problem.
- Iterative algorithms like hierarchical algorithms operate on an image pyramid, where results from coarser levels are used to obtain a more local search at finer levels.

Matching Cost Computation

This is a pixel-based method which defines how much two pixels match by calculating their difference. To do this, multiple methods exist.
Some examples :
Squared Intensity difference (SD) :

$$E = (P_1 - P_2)^2 \tag{2.11}$$

where E represents the error between the intensity of two pixels and P_1 and P_2 represent the intensity of pixel i.
Absolute intensity differences (AD) :

$$E = |P_1 - P_2| \tag{2.12}$$

Other examples could be for example binary matching costs, meaning that two are either a match or no match.
These matching cost computations have the constraint that the two cameras have to produce exact the same color for the same point. When this property is not met, a color calibration like histogram equalization can be used in a prepossessing stage [42]. But some costs are insensitive to these differences in camera gain or bias, for example gradient-based measures. To compare two gradient vectors g_L and g_R the following computations are required:

- Average magnitude : $\overline{m} = \frac{|g_L|+|g_R|}{2}$
- Magnitude of the difference : $d = |g_L - g_R|$

The evidence for a match is defined by Scharstrein [43] as

$$e = \overline{m} - d \tag{2.13}$$

These matching cost values are calculated over all pixels and all disparities from the initial disparity space image $C_0(x, y, d)$. The previous matching cost calculations are given for two input images, but it can easily be extended by summing up the cost for every image.

Aggregation of Cost

Local and window-based methods aggregate the matching cost by summing up or averaging over a specific area in the $DSI(x, y, d)$. This can be for example a box window or a Gaussian convolution.
Sum of Absolute Differences (SAD) : defined over a window with resolution = $[width, height]$

$$\begin{aligned} SAD(x,y,d) = & \sum_{i=-\frac{1}{2}(width-1)}^{\frac{1}{2}(width-1)} \sum_{j=-\frac{1}{2}(height-1)}^{\frac{1}{2}(height-1)} \\ & [|R_L(x+i, y+j) - R_R(x+i+d, y+j)| + \\ & |G_L(x+i, y+j) - G_R(x+i+d, y+j)| + \\ & |B_L(x+i, y+j) - B_R(x+i+d, y+j)|] \end{aligned} \tag{2.14}$$

Where R, G and B are the color components of the respective pixel, the index L stands for the left image and R for the right image. The parameters *width* and *height* define the window around the pixel and often are odd numbers.
Sum of Squared Differences (SSD) :

$$\begin{aligned} SSD(x,y,d) = & \sum_{i=-\frac{1}{2}(width-1)}^{\frac{1}{2}(width-1)} \sum_{j=-\frac{1}{2}(height-1)}^{\frac{1}{2}(height-1)} \\ & [(R_L(x+i, y+j) - R_R(x+i+d, y+j))^2 + \\ & (G_L(x+i, y+j) - G_R(x+i+d, y+j))^2 + \\ & (B_L(x+i, y+j) - B_R(x+i+d, y+j))^2] \end{aligned} \tag{2.15}$$

Disparity Computation and Optimization

Local methods : for every pixel the disparity is chosen with the best match without looking at the other pixels.
Global methods : these methods are trying to minimize a global energy defined by $E(d) = E_{data}(d) + \lambda E_{smooth}(d)$

- $E_{data}(d)$ is the measure how well the disparity function d agrees with the input image pair. $E_{data}(d) = \sum_{(x,y)} C(x, y, d(x, y))$ where C is the initial or aggregated matching cost DSI.
- $E_{smooth}(d)$ encodes the smoothness assumptions made by the algorithm. This term is often restricted to only measuring the differences between neighboring pixels' disparity. $E_{smooth}(d) = \sum_{(x,y)} [\rho(d(x, y) - d(x + 1, y)) + \rho(d(x, y) - d(x, y + 1))]$

Here is ρ some monotonically increasing function of disparity differences.
Once the global energy has been defined, it can be used to find a (local) minimum.

Refinement of Disparities

Depending on the further use of the disparity map, the result can be refined. For applications such as robot navigation or people tracking, these may be perfectly adequate. But for example when it is used for image-based rendering, such quantisation maps lead to very unappealing view synthesis results. Therefore many algorithms apply a sub-pixel refinement stage. This can be done for example by fitting curves.
Other methods like cross checking, meaning comparing left-to-right and right-to-left are used to find mismatches, and applying a filter can also increase the quality of the result. A popular filter herefore is the median filter.

2.2.4 Calculating the Disparity Map

The next algorithm was presented by Karsten Muhlmann et al [21]. His paper describes a fast algorithm, but it also has reasonable quality. The technique can be used as a starting point for implementing improved algorithms. If it is fully optimized it can even be used for real time applications.

Input

The algorithm requires a pair of horizontal linear rectified images as input. This means that a pixel and the matching pixel on the other image are on the same height (have the same y value). Therefore no calibration is needed resulting in a significant speedup compared to searching along epipolar lines of unrectified images. The representation of the disparity map will be :

$$\begin{cases} x' = x + d(x, y) \\ y' = y \end{cases} \tag{2.16}$$

Another requirement is a couple of input parameters like the lower and upper bound value of $d(x, y)$. It is important that these values are chosen as tight as possible. This interval is called the disparity search range. The last parameter is the size of the window used for calculating the matching cost.

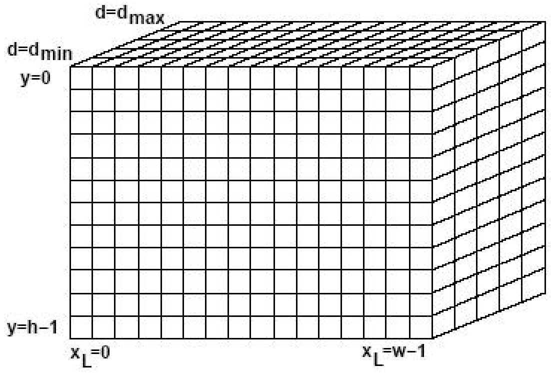

Figure 2.7: Volume defined by the dimensions of the input images width (w) and height (h) and the disparity search map ($d_{max} - d_{min}$)

Matching cost

For every value d in the disparity range and for every pixel, the matching cost between $I_1(x, y)$ and $I_2(x + d, y)$ must be calculated. In the paper this is done with the Sum of Absolute Differences (equation 2.14) of a window around the pixel, because it is slightly faster then the Sum of Squared Differences.
All the matching cost values are saved in a cuboid whose dimensions are given by the width and height of the input images and the disparity search range (Figure 2.7).

Creating the Disparity Map

Once the cuboid has been filled with values the disparity map can be made. This by selecting the best depth value for every pixel. Therefore can it be very easy by just looking in the cuboid for every pixel which depth-value has the smallest difference value.
The calculated disparity map will have a lot of noise in it due to "wrong" SAD values. This can happen because the matching pixel is not found in the other image, for example by occlusion. Another problem can be that the wrong corresponding pixel was found.
A standard way for dealing with noise is blurring the image, but this comes with a lot of quality loss. A more often used technique for noise reduction of disparity maps is a median filter. This filter will calculate for every pixel the median of a window around the pixel to reduce the extreme values.

2.2.5 Depth Estimation from Unrectified Images

When images are not rectified, calibration data for the cameras is required. Instead of searching for matching pixels along the x-axis, corresponding pixels can be found along the epipolare lines. Once the matching points are found in images their three dimensional position can be calculated. These epipolare lines are found by using the fundamental matrix of section 2.1.2, which describes the relationship between a point on one image ans its possible projections on the other image.
The calibration data often contains a projection matrix for every camera which describes the relationship between a point in the scene and a point on the view plane. A point in the scene

can be multiplied with this projection matrix to obtain the image coordinates. This function starts at a three dimensional space and ends in a two dimensional space and every point in space will be projected on exactly one point on the view plane.
Another situation is when the coordinates on the image planes are known and the three dimensional coordinate is unknown. Therefore the inverse of this function is required. However this inverse starts from a two dimensional space and ends in a three dimensional space so there is no one to one mapping, but an infinite collection of solutions. All points on the line starting in the centre of projection C of that camera and going through the selected point I on the view plane are being projected on I. So the inverse of the projection should be a line in the scene.
The projection matrix P_i of camera i describes the relation of a point in space X and his coordinates on the camera I_i. These camera coordinates I_i can be obtained by:

$$\begin{aligned} X_i{}' &= P_i.X \\ I_{i,x} &= \frac{X_{i,x}{}'}{X_{i,z}{}'} \\ I_{i,y} &= \frac{X_{i,y}{}'}{X_{i,z}{}'} \end{aligned}$$

Starting from the two dimensional coordinates I_i transforming it back to X_i' is :

$$\begin{aligned} X_{i,x}{}' &= I_{i,x}.z \\ X_{i,y}{}' &= I_{i,y}.z \\ X_{i,z}{}' &= z \end{aligned}$$

Where the z coordinate can have any value. Depending on this z value another point on the line is calculated. This point can be found by :

$$X = P_i^{-1} X_i{}'$$

Because a line is uniquely defined by two different points in space, this calculation must be done with at least two different z values to obtain the desired line.
If the projection of a point is known on multiple images they can all be back projected and the intersection of the lines is the point in the scene.
Here arises another problem, two lines in three dimensions generally do not intersect at a point. Often they only cross each other and only intersect after projecting on a plane. Therefore the shortest line segment, which is the shortest possible connection between the two lines, is unique and often considered to be their intersection in three dimensions.
When knowing two points on a line, Q_1 and Q_2, a parametric equation for the line looks like $Q_a = Q1 + t_a(Q_2 - Q_1)$. The same for the second line $Q_b = Q_3 + t_b(Q_4 - Q_3)$. The solution of the problem are the 2 points Q_a and Q_b of which the distance between them is the minimum distance possible. The value $||Q_b - Q_a||^2$ must be minimum.

$$||Q_1 - Q_3 + t_a(Q_2 - Q_1) - t_b(Q_4 - Q_3)||^2 \text{minimize} \tag{2.17}$$

An alternative approach [20] uses the property that the line Q_aQ_b will be perpendicular to the two lines (figure 2.8) and so this condition can be written down as dot products:

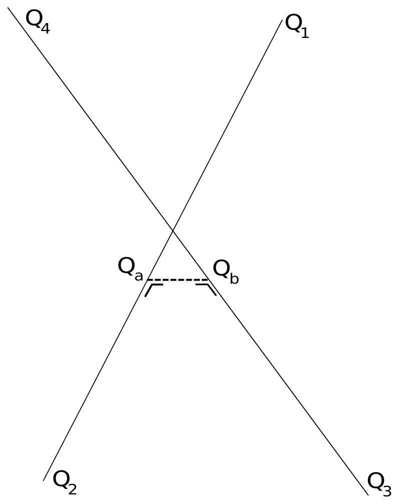

Figure 2.8: Two lines Q_1Q_2 and Q_3Q_4 cross each other, the line Q_aQ_b is considered to be the intersection of the two lines

$$\begin{aligned}(Q_a - Q_b) \cdot (Q_2 - Q_1) &= 0\\ (Q_a - Q_b) \cdot (Q_4 - Q_3) &= 0\end{aligned}$$

$$\begin{aligned}(Q_1 - Q_3 + t_a(Q_2 - Q_1) - t_b(Q_4 - Q_3)) \cdot (Q_2 - Q_1) &= 0\\ (Q_1 - Q_3 + t_a(Q_2 - Q_1) - t_b(Q_4 - Q_3)) \cdot (Q_4 - Q_3) &= 0\end{aligned}$$

$$\vdots$$

The result after expanding these terms is

$$\begin{aligned}d_{1321} + t_a d_{2121} - t_b d_{4321} &= 0\\ d_{1343} + t_a d_{4321} - t_b d_{4343} &= 0\end{aligned}$$

Where $d_{mnop} = (x_m - x_n)(x_o - x_p) + (y_m - y_n)(y_o - y_p) + (z_m - z_n)(z_o - z_p)$
Resulting in :

$$t_a = \frac{(d_{1343}d_{4321} - d_{1321}d_{4343})}{(d_{2121}d_{4343} - d_{4321}d_{4321})} \tag{2.18}$$

$$t_b = \frac{(d_{1343} + t_a d_{4321})}{d_{4343}} \tag{2.19}$$

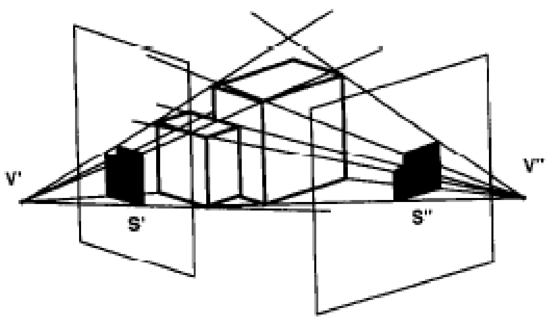

Figure 2.9: Volume intersection for object reconstruction [7]

2.3 Visual Hull

The problem discussed in this thesis is the interaction between objects or persons in the real world and objects in a virtual world. This could be done by placing a copy of the real object or person into the virtual world. To understand the 3D content of a scene is another problem in computer vision, since it can allow computer driven equipment to perform tasks such as navigation, manipulation and visual recognition. This is usually done by creating a visual hull from the object.

Visual hull is a geometric entity created by *shape-from-silhouette* three dimensional reconstruction techniques. A silhouette is a binary image containing all the pixels which are a projection of the three dimensional object on the image plane. This silhouette is therefore a perspective or orthogonal projection of the object (S). Creating a silhouette from images can be very simple when working with a known background, where the foreground is easily recognized. When working in an unknown or a dynamic scene it can be more difficult and more expensive to separate foreground from background.

Object reconstruction can be performed using volume intersection. The volumes are cones starting in the center of projection of the image (V) and going through the silhouette (figure 2.9). An example of such a cone is presented in figure 2.10. When the projection is orthogonal, the volumes are cylinders instead of cones. But this constructed object is not necessary the same as the real object. An example of a model which can not be reconstructed is shown in figure 2.11. This is explained more in detail in section 2.3.1.

The next section contains a precise definition of the visual hull and his general properties [7]. Knowing this definition, it is possible to define the objects which can be reconstructed and which are impossible. In section 2.3.2 some representations of the visual hull are given followed by a couple of possible ways to calculate them in sections 2.3.4 and 2.3.5.

The visual hull is then concluded in section 2.3.6 with some practical usage.

2.3.1 Definition and Properties of the Visual Hull

Let us start with defining some symbols. As mentioned before, S is the three dimensional model which has to be reconstructed and V is the viewpoint of the used camera. Another definition needed for defining the visual hull is the viewing region (R) which is a part of the three dimensional space that contains all allowed viewpoints, so $V \in R$.

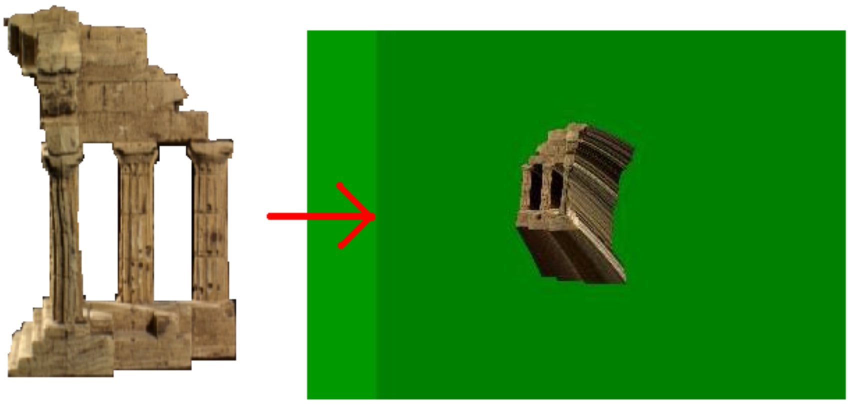

Figure 2.10: The left image is a silhouette of a temple, background pixels are colored white. This image is then back projected into the world. Every point in space projected on this silhouette is part of the back projected cone.

Laurentini [7] defined the visual hull as:

> *The visual hull $VH(S, R)$ of an object S relative to a viewing region R is a region in three dimensional euclidean space such that, for each point $P \in VH(S, R)$ and each viewpoint $V \in R$, the half line starting at V and passing through P contains at least a point of S.*

In other words, a ray starting in V and intersecting with the $VH(S, R)$ also means that the ray passes through the silhouette of the image plane with viewpoint V. This is trivial because the intersection point between the ray and the visual hull is projected on that intersection point. Knowing this, it is possible to create a cone starting in V and going to every point of the image plane silhouette. When S contains only one viewpoint, following the definition this cone is the visual hull. When S is bigger then only one viewpoint, by definition the intersection of all these cones must be calculated.
It is easy to understand now that $S \subseteq VH(S)$ because every point on S is projected on every silhouette and therefore part of $VH(S)$.
Because of the definition the visual hull has the following properties :

- Property 1 : $VH(S, R)$ is the maximal object silhouette equivalent to S
- Property 2 : $VH(S, R)$ is the closest approximation of S that can be obtained using volume intersection techniques with viewpoint $V \in R$. This is true because by intersecting only the points in space which are definitely not on S are eliminated.
- Property 3 : if $R > R'$ then $VH(S, R) \leq VH(S, R')$. This because of the previous property which defines that points only got eliminated, so when more viewpoints are available more points can fall outside a silhouette.

This means that the visual hull is the best possible way to represent a three-dimensional object from silhouette images.

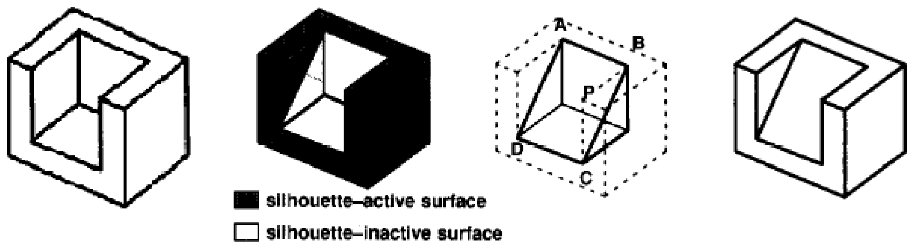

Figure 2.11: (1) The 3D object we want reconstruct (2) the silhouette-active and inactive surfaces (3) the silhouettes will not change when this part of the model is modified (4) visual hull of the first object [7]

But it is not always equal to the object because not every part is visible. For example look at figure 2.11. The first object is the three dimensional object to be reconstructed using silhouettes. The second figure shows the silhouette-active surface and the silhouette-inactive surface. The silhouette-active surface $sa(S, R)$ is the part of the boundary of the object S that is also part of the boundary of the visual hull $VH(S, R)$. The silhouette-inactive surface can have any shape inside the volume $VH(S, R) - S$ without changing any silhouette. The third figure shows the volume which is unknown because it does not affect the silhouettes. The visual hull, the closest approximation to the object, is eventually shown at the last figure.

2.3.2 Representation

Now the definition of the visual hull is given, it is still necessary to represent it in some kind of data structure. It is standard for images to be represented as two-dimensional arrays of pixels. But three dimensional models, however, are much more varied [9].
The most obvious technique is defining the model as a constructive solid geometry (CSG). In CSG the desired shape is specified in terms of a tree of elementary solids that are combined using simple set operations. The back projected cones or cylinders are the elementary solids and the operation between them is the intersection. So if it is possible to find out if a point is inside or outside each cone its easy to find out if that point is part of the visual hull or not. But it also have disadvantages that is not easy to render.
Enumerated space representation is another popular modeling description. It represents objects as a set of discrete volume elements. These volume elements are called voxels and can be compared as pixels in a two dimensional image. The word *voxel* is a combination of the words volumetric and pixel. One way of doing this is to create a spacial occupancy map using a number of voxels, for example cubes. For every voxel the CSG model can be used to define if it is part of the visual hull or not.
An optimisation for this is called an octree structure. The scene is created as an initial voxel, known as the universal cube. This universal cube is split into eight suboctants and each is iteratively subdivided until some predefined limit is reached. The subdivision can stop when the suboctant is completely inside or completely outside the visual hull or when a certain depth is reached. One way of creating an octree is by calculating every cubic voxel and combining eight neighbours with the same value to one. Ali Erol et al [8] described another method to create an octree without first creating a cubic visual hull. These methods typically have large memory requirements and are thereby restricted to low-resolution representations.

The next technique is called a boundary representation and defines the boundaries as a series of two-dimensional surfaces embedded within a three-dimensional space. A first-degree description might consist only of polygons. Higher-degree descriptions might describe a curved surface patch. An example is the visual hull surface estimation algorithm of Phillip Milne et al [10] which applies the marching cubes algorithm to all surface voxels of the visual hull. This representation uses less memory than enumerated space representation and the result is more smooth. Therefore it is more suitable for rendering.
The volumetric and boundary representation are presented in figure 2.12.

2.3.3 Background Subtraction

Before explaining how a visual hull can be created from silhouettes, it is important to know how the silhouettes can be obtained. Therefore a method to separate the foreground from the background is required.
For background subtraction, the algorithm needs to know what the background looks like. When this background information is available, it can be removed from every frame by just checking the difference between the current pixel and the background pixel. If this difference is below a threshold it is assumed that this pixel is part of the background. This background information can be obtained by making images of the scene before starting the game. To take into account the noise of the background images, more pictures of the background can be made and the average of them can be used as the background information.
This approach works fine, except when the background changes. These changes include moving objects in the back and different illuminations. Cheung [22] described a robust background subtraction algorithm used for urban traffic video. It dynamically adjusts the known background each frame to keep it up to date.
The computed silhouette is not always accurate, because some pixels can be tagged wrong, background pixels recognized as foreground or the other way around. These miscalculations can be partly corrected by applying erosion and dilation to the silhouette [44].
When a lot of cameras are available, the background subtraction algorithm must be applied to each input image, which can be very time consuming. Especially when erosion and dilation is used. Another used technique to save time is working with an easy to separate background. Therefore a green screen is often placed behind the person. The silhouette then only contains the non-green pixels.

2.3.4 Calculating the Visual Hull

The calculation of the visual hull often depends on the representation used. In this section two algorithms are given to calculate an enumerated space representation.

Cubic Voxel Visual Hull

Over the years a lot of algorithms are invented to create the visual hull from silhouettes. One of the most simple of them is creating a cubic enumerated space representation. The necessary requirements are that the foreground can be separated from the background for every image and that the projection matrix is known for every camera.
Space will be divided into small cubes, the voxels, which contain a boolean defining if the cube is part of the visual hull or not. If the voxel is part of the visual hull then the projection

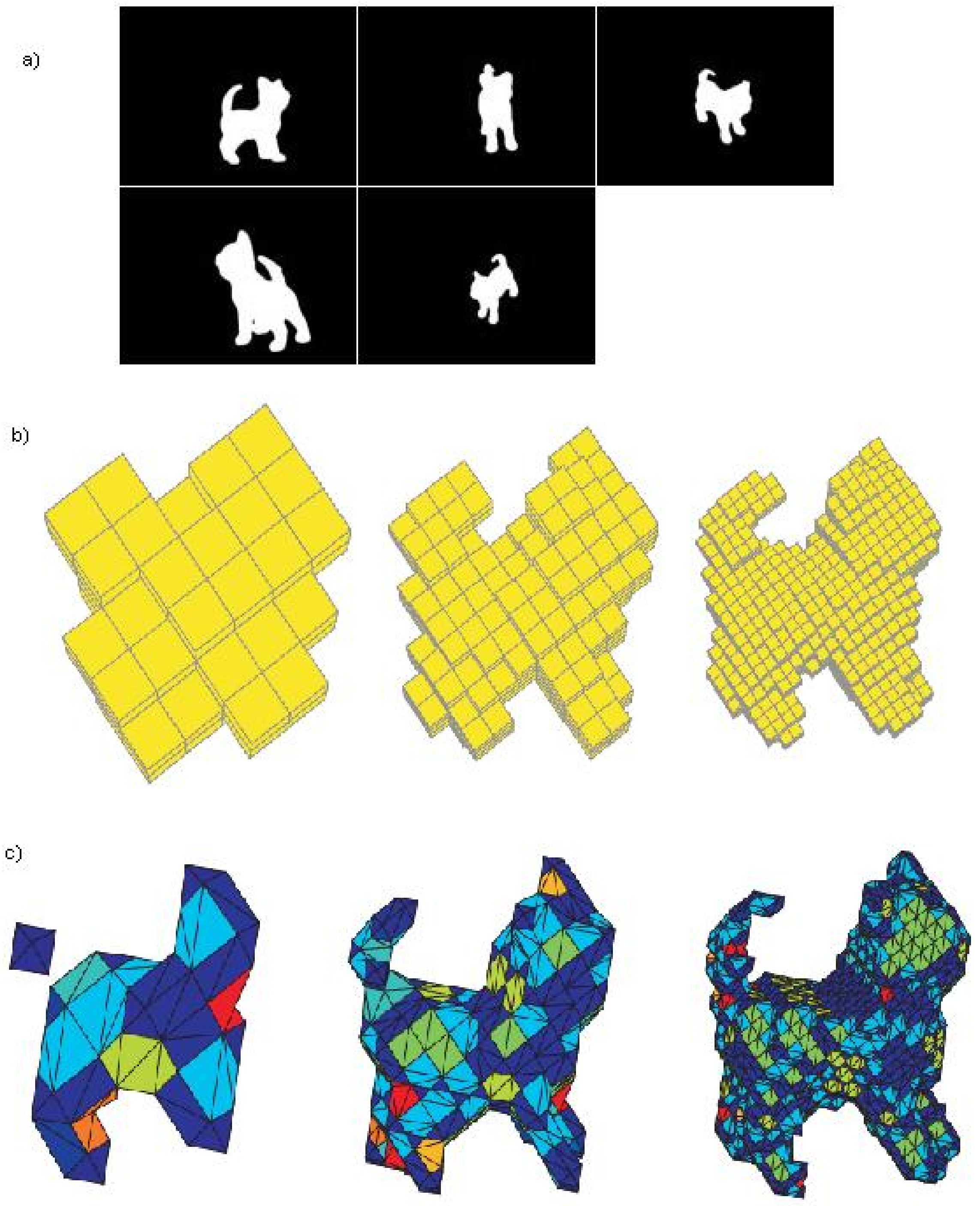

Figure 2.12: (a) 5 silhouettes from different views of the same dog (b) A volumetric representation of the dog at 3 different subdivision levels (c) boundary representation at 3 different subdivision levels [7]

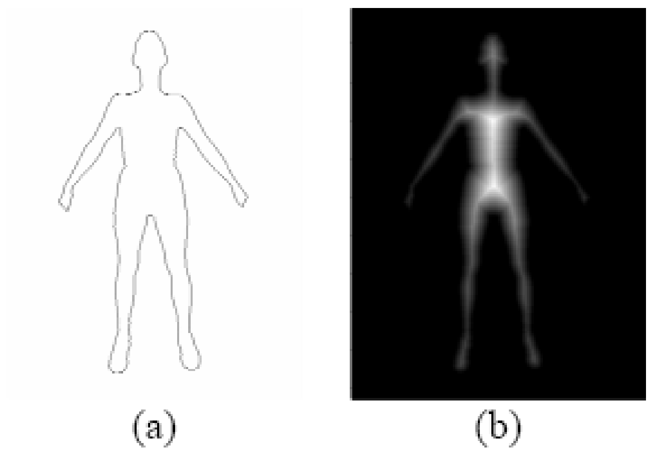

Figure 2.13: (a) The border of the silhouette (b) The Euler distance from the pixel to the border is stored in every pixel. The same can be done for the pixels outside the silhouette [23]

of it should be inside every available silhouette. Because the projection matrices are known it is straight forward doing this.
An example result is given in figure 2.12(b).

Octree Voxel Visual Hull

Starting from the cubic voxel representation it is possible to create an octree by just merging together eight cubes with the same value. But by doing this it is still necessary to calculate all the cubic voxels first. It should be more efficient to start from one voxel and divide it only if necessary. Because, when it is known that a voxel is entirely inside the visual hull, all its children are also entirely inside it. If the maximum depth of the octree is high, then a lot of time is saved when the value of the voxel can be determined at a low depth. Therefore the visual hull is calculated more in detail around the borders of the real object. Ali Erol [23] described a fast octree algorithm which can create this octree real-time. This algorithm is divided into two steps which are executed every time new images are available.
In the first step each silhouette is transformed into an image with the same resolution. The value of every pixel in those images correspond to the distance of that pixel to the borders of the silhouette as shown in figure 2.13. The higher the value the further it is inside the silhouette. Negative values describe the distance to the border outside the silhouette.
In the second step, the value of a voxel can be estimated by projecting their eight corners on those images and checking the value of the pixel on which the center of the cube is projected. By comparing the distance value of the pixel with the distance from the centre to the corners, an estimation can be made for the location of the cube to the visual hull. If the voxel can be defined as completely inside or completely outside every silhouette it is not necessary do divide them anymore, else the cube will be split into eight.

2.3.5 Visual Hull for Rendering

The two algorithms presented before are appropriate for real time creation of a visual hull, but they are not so useful for rendering. One way of rendering is to draw a cube at each location of a true voxel, but the color will be the same for every triangle. Therefore it is required to define the color of every voxel. Zhirkov [45] proposed a solution to color the voxels of an octree to use for image based rendering. Unfortunately this method is good for real time rendering, but not real-time creation.
A better algorithm for rendering a visual hull is presented by Yerex [24] and has some useful advantages. First of all it runs in real time, which is one of the requirements. Also it is executed entirely on the graphical processing unit (GPU). Therefore the CPU stays idle and can be used for other things like collision detection.
The idea of the algorithm is to project the object onto some planes in the scene and then render it with alpha blending. The input images also contain an alpha channel which defines if the pixels are part of the foreground or background. Background pixels have an alpha value of 0 and are thereby not drawn during rendering.
To explain this algorithm, it is necessary to understand how the visual hull is represented before knowing how it can be created. And eventually when it is created, how it can be colored. But first something more about background subtraction.

Background Subtraction

Like most visual hull algorithms the first step is some sort of background subtraction. The binary mask obtained after this step will become the alpha channel of the images. The images with their alpha values can be stored into a texture. This step can be done by the CPU or by the GPU depending on the further use of the images. If the silhouette is also necessary for further use like collision detection, it is better to calculate it on the CPU.

Scene Representation

In the next step the scene is created. It contains a serie of parallel surfaces on which the different input images will be projected. This is shown in figure 2.14. The images are projected on every slices with the use of the calibration data. When more planes are drawn close to eachother the scene will look like a three dimensional model instead of a series of individual planes. The downside is that when the scene contains more planes, it will take longer to render. The number of planes therefore defines the relationship between the speed of the algorithm and its result. This means that this algorithm, just like the other visual hull algorithms, can adjust its level of detail to reach its desired framerate.

Creating the Visual Hull

The images and planes are passed to the GPU, where a fragment shader can do all the calculations. Every fragment has multiple textures, one for every camera. If the fragment is part of the visual hull it should be projected onto every silhouette and therefore have an alpha value of 1 on every texture. If not, the fragment gets an alpha value of 0 and is therefore not drawn. This step does exactly what the definition of the visual hull described and thereby only visual hull fragments are rendered.

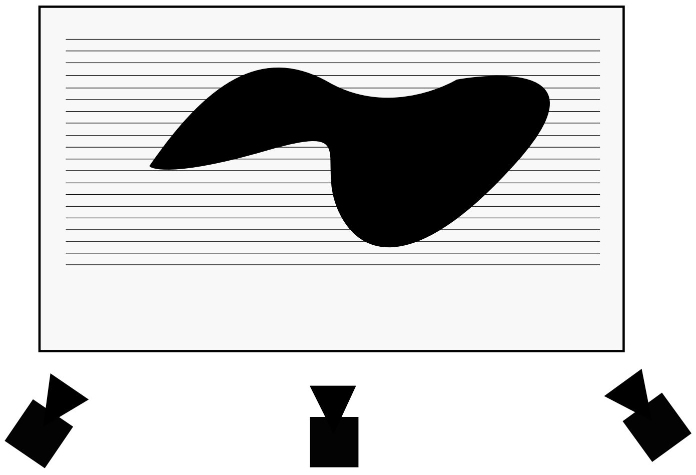

Figure 2.14: The scene containing the real object is divided into n parallel planes at the same distance of each other. The images recorded from the multiple cameras are then projected onto these planes.

Coloring the Visual Hull

The last step is to define the color of each fragment. It is logic that the colors from the textures are used. Therefore a weighted average of the input images is calculated. This weight should be high for input cameras close to the virtual camera and lower to others. Li et al [46] calculates these weights as:

$$W_{k,P} = \frac{1}{acos(d_k \cdot d_v)} \tag{2.20}$$

For every point P, the criterion to choose the weight is the angle deviation between the viewing direction of the reference view d_k and that of the target view d_v. A smaller angle gets higher weight. These terms are presented in figure 2.15.
The color of a point is then given by the formula:

$$C_P = \left[\sum_{k=1}^{N} V_{k,P}.W_{k_P}.T_{P,k}\right] / \sum_{k=1}^{N} V_{k,P}.W_k \tag{2.21}$$

where $T_{k,P}$ is the texture color from the k-th input image and $V_{k,P}$ is the visibility function for point P with regard to view k:

$$V_{k,P} = \begin{cases} 1 & \text{if P is visible from view k} \\ 0 & \text{otherwise} \end{cases} \tag{2.22}$$

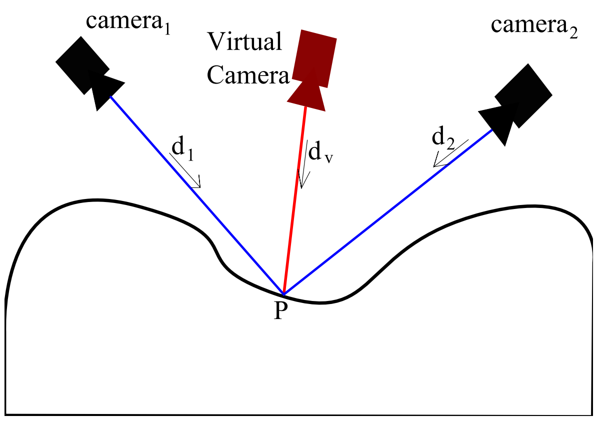

Figure 2.15: Camera 1 and 2 are the two reference views and the virtual camera the desired view. For point P, d_1 and d_2 are viewing direction of the reference views while d_v is the viewing direction of the desired view.

2.3.6 Use of the Visual Hull

The visual hull can be used in different ways in computer games. First and most obvious, it can be used for collision detection. This because a three dimensional model of the player is available. This model can be checked for collision with the virtual objects in the game. As explained before, the calculation is often restricted to a certain level of detail. By improving this level of detail the framerate will drop down, but the accuracy will raise. Most computer games are more concerned of keeping a high framerate, therefore a good balance between speed and accuracy is required.
A problem which has not been discussed before is creating an output for the games. The player has to be integrated in the game so he or she knows how to interact with virtual objects. One way of doing this is by showing the images of one camera on the screen and adding the game objects.
Another solution could be the other way around. First the game scene is created and the player is placed in there. The visual hull is a three dimensional representation of the player and can thereby be added to the virtual scene.

2.4 Object Tracking

Object tracking is a computer vision technique for defining the location of an object and tracking it. This is an important domain of computer vision because it enables several applications such as security and surveillance, people detection and recognition, medical therapy,... It can also provide a natural way of interaction between a human and a computer [13]. By locating an object or a person in the real world it is possible to place a corresponding object of it in a virtual world where it can interact with other virtual objects.
Because this technique is being researched a lot in the last decades there are a lot of different algorithms designed for this. These algorithms are based on statistics where the computer makes a prediction of the objects location in the next image. A common approach for this is the *Kalman filter* [11] or the *extended Kalman filter*, which is named after Rudolf E. Kalman who invented it in 1960. This filter is used for efficient estimating the state of a dynamic system from a series of incomplete and noisy measurements and is also very useful for general purpose object tracking [17].
Later the *particle filter* was invented whick is also very useful for general purpose object tracking. Over the years a lot of variants of the particle filter were introduced, for example the *condensation algorithm* by Michael Isard and Andrew Blake [25]. In the next section the the Kalman filter is shortly explained and an example is given how to use this technique to track a person. Section 2.4.4 contains the general idea behind the particle filter and the general particle filter. Section 2.4.5 describes the condensation algorithm derived from the particle filter. Section 2.4.6 contains some examples of how these general techniques can be used for object tracking.
Because object tracking does not need to start from a general-purpose technique, section 2.4.7 describes an algorithm for tracking hands and face of a person.
The tracked locations can directly be mapped into game or used for further calculations. Section 2.4.8 describes such a further step where the calculated locations can be used to define the pose of the player by using *kinematics*.

2.4.1 Kalman Filter

Originally developed for use in spacecraft navigation, the Kalman filter turns out to be useful for many applications. It is mainly used to estimate system states that can only be observed indirectly or inaccurate by the system itself [15, 16].
The Kalman filter is a tool that can estimate the variables of a wide range of processes. In mathematical terms we would say that a Kalman filter estimates the states of a linear system. A linear system is a process that can be described by the following two equations:
State equation:

$$x_{k+1} = Ax_k + Bu_k + w_k \tag{2.23}$$

Output equation:

$$y_k = Cx_k + z_k \tag{2.24}$$

Where A, B and C are matrices, x_k the state at time k, u_k is the known input to the system, y_k the measured output and w_k and z_k are noise.
The state x is a vector containing all of the information about the present state of the system. But this state cannot be measured directly. Instead y will be measured, which is a function of x that is corrupted by the noise z. These measures can then be be used to obtain an estimate

of x.
A good example to get a better idea of these terms is described by Simon [16]. Suppose that we want to model a vehicle going in a straight line. The state then contains the position p and the velocity v of the vehicle. The input u is the acceleration and the output y is the measured position. The acceleration can be changed every T seconds. In this case the update of the velocity is given by:

$$v_{k+1} = v_k + Tu_k \tag{2.25}$$

when the acceleration is constant during the time T. But the velocity will be perturbed by noise due to wind, put holes, etc. Therefore noise has to be taken into account.

$$v_{k+1} = v_k + Tu_k + \widetilde{v_k} \tag{2.26}$$

where $\widetilde{v_k}$ is the velocity noise. A similar equation can be derived for the position:

$$p_{k+1} = p_k + Tv_k + \frac{1}{2}T^2 u_k + \widetilde{p_k} \tag{2.27}$$

with $\widetilde{p_k}$ the position noise.
This can be put together into a linear system where the state x is:

$$x_k = \begin{bmatrix} p_k \\ v_k \end{bmatrix} \tag{2.28}$$

and the linear system equations are:

$$x_{k+1} = \begin{bmatrix} 1 & T \\ 0 & 0 \end{bmatrix} x_k + \begin{bmatrix} T^2/2 \\ T \end{bmatrix} u_k + w_k \tag{2.29}$$

$$y_k = \begin{bmatrix} 1 & 0 \end{bmatrix} x_k + z_k \tag{2.30}$$

z_k is the measurement noise due to such things as instrumental errors. To control the vehicle an accurate estimate of the position and velocity is necessary. This is where the Kalman filter comes in.

The Discrete Kalman Filter

The purpose of the Kalman filter is to use the available measurements y to estimate the state of the system x. This estimate must satisfy two requirements:

- The average value of the state estimate must be equal to the average of the real state: $E[x_k] = \hat{x}_k$
- The state estimate must vary from the true one as little as possible: $E[(x_k - \hat{x}_k)(x_k - \hat{x}_k)^T] = P_k$

where $E(\cdot)$ refers to the expected value [15, 16].
For the Kalman filter to satisfy these, an assumption about the noise must be made. The average value of w and z must be zero and there should be no correlation between the two. The noise covariance matrices S_w and S_z are defined by:

$$S_w = E(w_k w_k^T) \tag{2.31}$$

$$S_z = E(z_k z_k^T) \tag{2.32}$$

The estimate can be divided into two:

- $\hat{x_k}^-$: the *a priori* state estimate at step k given knowledge of the process prior to step k.
- $\hat{x_k}$: the *a posteriori* state estimate at step k given measurement y_k

The *a priori* and *a posteriori* estimate errors are then:

$$\begin{aligned} e_k^- &= x_k - \hat{x_k}^- && (2.33) \\ e_k &= x_k - \hat{x_k} && (2.34) \end{aligned}$$

and the *a priori* and *a posteriori* estimate error covariance:

$$\begin{aligned} P_k^- &= E[e_k^- c_k^{-T}] && (2.35) \\ P_k &= E[e_k e_k^T] && (2.36) \end{aligned}$$

The goal of the Kalman filter is to find an equation that computes the *a posteriori* state estimate $\hat{x_k}$ as a linear combination of the *a priori* estimate $\hat{x_k}^-$ and a weight difference between an actual measurement y_k and a measurement prediction $H\hat{x}_k^-$:

$$\hat{x} = \hat{x}_k^- + K(y_k - H\hat{x}_k^-) \tag{2.37}$$

The difference $y_k - H\hat{x}_k^-$ is called the *measurement innovation* and reflects the discrepancy between the predicted measurement $H\hat{x}_k^-$ and the actual measurement y_k. The $n \times m$ matrix K is chosen to be the gain or blending factor that minimizes the *a posteriori* error covariance (2.36). One popular form of K is given as:

$$\begin{aligned} K_k &= P_k^- H^T (HP_k^- H^T + S_z)^{-1} && (2.38) \\ &= \frac{P_k^- H^T}{HP_k^- H^T + S_z} && (2.39) \end{aligned}$$

As seen in (2.39) when the measurement error covariance S_z approaches zero, the gain K weights the innovation more heavily:

$$\lim_{S_{z,k} \to 0} K_k = H^{-1} \tag{2.40}$$

On the other hand, as the *a priori* estimate error covariance P_k^- approaches zero, the gain K weights the innovation less heavily:

$$\lim_{P_k^- \to 0} K_k = 0 \tag{2.41}$$

The Discrete Kalman Filter Algorithm

The Kalman filter estimates a process by using a form of feedback control: the filter estimates the process state at some time and then obtains feedback in the form of (noisy) measurements. Therefore the equations can be divided into two groups:

- The *time update* equations: these equations are responsible for projecting forward in time the current state and error covariance estimates to obtain the *a priori* estimates for the next step.

- The *measurement update* equations: for incorporating a new measurement into *a priori* estimate to obtain an improved *a posteriori* estimate.

The algorithm is thereby also divided into two steps: the *prediction step* using the time update equations and the *corrector step* using the measurement update equations. This algorithm is described in psuedocode below:

```
1. **Initial step**: initial estimate of $\hat{x}_0$ and $P_0$
2. **Predict step**:
      Project the state ahead: $\hat{x}_k^- = A\hat{x}_{k-1} + Bu_{k-1}$
      Project the error covariance ahead: $P_k^- = AP_{k-1}A^T + S_w$
3. **Correct step**:
      Computer the Kalman gain: $K_k = P_k^- H^T(HP_k^- H^T + S_z)^{-1}$
      Update estimate with measurement $y_k$: $\hat{x}_k = \hat{x}_k^- + K_k(y_k - H\hat{x}_k^-)$
      Update error covariance: $P_k = (I - K_k H)P_k^-)$
4. Increment k
   goto predict step (2)
```

2.4.2 The Extended Kalman Filter

The Kalman filter can estimate states for linear systems, but a lot of real problems are nonlinear. For solving these kind of problems there is the *extended kalman filter*. Instead of (2.23) and (2.24) the system can be described as:

$$x_k = f(x_{k-1}, u_{k-1}, w_{k-1}) \tag{2.42}$$

$$y_k = h(x_k, z_k) \tag{2.43}$$

Where f and h are some nonlinear functions.
The predict and correct step are given below without explaining the mathematical background. More information can be found in the articles of Welch and Ribeiro [15, 47].
Prediction step:

$$\hat{x}_k^- = f(\hat{x}_{k-1}, u_{k-1}, 0) \tag{2.44}$$

$$P_k^- = A_k P_{k-1} A_k^T + W_k S_{w,k-1} W_k^T \tag{2.45}$$

Correct step:

$$K_k = P_k^- H_k^T (H_k P_k^- H_k^T + V_k S_{z,k} V_k^T)^{-1} \tag{2.46}$$

$$\hat{x}_k = \hat{x}_k^- + K_k(y_k - h(\hat{x}_k^-, 0)) \tag{2.47}$$

$$P_k = (I - K_k H_k) P_k^- \tag{2.48}$$

where:

- A is the Jacobian matrix of partial derivative of f with respect to x
- W is the Jacobian matrix of partial derivative of f with respect to w
- H is the Jacobian matrix of partial derivative of h with respect to x
- V is the Jacobian matrix of partial derivative of h with respect to z

2.4.3 Kalman Filter for Object Tracking

The Kalman filter knows many applications like removing noise from electromagnetic signals, but it can be very useful in computer vision as well. The next algorithm, presented by Merven et al [17], is an example of the extended Kalman filter and is used for person tracking in a three dimensional space. This is possible when the cameras are static and calibration information is available.
First a representation of the person is required. Here the tracked person is assumed to be a three dimensional ellipsoid of known size and his/her feet must are on the ground plane. The tracking process then runs as follows, each time a new image of the scene is received:

1. Segmentation: foreground must be separated from background.
2. Prediction in world view: the location of the tracked person is predicted based on the previous estimate of his/her location.
3. Projection to image space: the three dimensional ellipsoid representing the person is projected to an ellipse in image space.
4. Observations in image space: starting from the predicted location, a search for the best elliptical-template-shaped image samples is performed.
5. Update: the Kalman gain can be computed using the best match from previous step and the associated quality. This matrix can then be used to make a new estimate of the persons location.

Segmentation

In the first step some kind of background subtraction is done to detect foreground regions and there are a lot of different methods to do this. Section 2.3.3 already described some of these methods which can be used here.
Since the tracker does not rely only on this segmentation, it is not crucial that this step is real good. However a better segmentation improves the measurement.

Prediction in world-view

Because the person is assumed to be on the ground plane the location and velocity can be defined with only two coordinates. Therefore the state x is defined as $(p_x, p_y, v_x, v_y)^T$. For every time step T, the system equations are:

$$\begin{aligned} x_{k+1} &= A_T x_k + T w_k && (2.49) \\ y_k &= b(x_k) + z_k && (2.50) \end{aligned}$$

where

$$A_T = \begin{bmatrix} 1 & 0 & T & 0 \\ 0 & 1 & 0 & T \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad (2.51)$$

b is the ground-plane to image plane transformation function and w_k and z_k are noise sequences with zero mean.
The prediction step of the Kalman Filter algorithm is executed here so:

$$\hat{x}_k^- = A_T \hat{x}_{k-1} \tag{2.52}$$

$$P_k^- = A_T P_{k-1} A_T^T + T W_k \tag{2.53}$$

Projection to image space

The three dimensional ellipsoid is projected to an ellipse on every image plane.

Observations in image space

Starting from the predicted image location from last frame, a search for the best match between the person and the image is performed. Only using the foreground regios identified in step 1 is not sufficient enough for robust tracking. To distinguish between the different foreground objects the colour information in the ellipse can be used. This is done by comparing the histogram of the ellipse pixels to the histogram of the person which is created in an initial step and updated each time sequence.
To find the best histogram match, n random location samples are tested. They are chosen from a normal distribution $N(\hat{x}_k^-, P_k^-)$. The location with the best resemblance is chosen as $\hat{y}_k$.

Update

Eventually the correction step of the Kalman filter must be executed to receive an optimal estimate of the persons location. Starting with the Kalman gain:

$$K_k = P_k^- B_k^T (B_k P_k^- B_k^T + V_k)^{-1} \tag{2.54}$$

Since $b_k(x)$ is non-linear, B_k is calculated by locally linearising b at $x = \hat{x}_k^-$. Finally the location and uncertainty is calculated by:

$$\hat{x}_k = \hat{x}_k^- + K_k(y_k - b_k(\hat{x}_k^-)) \tag{2.55}$$

$$P_k = P_k^- - K_k B_k P_{k-1}^- \tag{2.56}$$

The result of this algorithm is shown in figure 2.16.

2.4.4 Particle Filter

The general target tracking problem

Target tracking can be characterized as the problem of estimating the state of a system as a set of observations become available [12]. The state of time k is noted as X_k and the observation Y_k. With these parameters the following two densities can be defined :

- The transition priori : $p(X_k|X_{k-1})$. It is defined as a Markov process. At each time (k) the system may or may not change from the state it was before. The change of state is called transitions. A prior can be interpreted as a description of what is known about the state in absence of some evidence. Once the evidence is taken into account this conditional probability is called posterior probability [18].

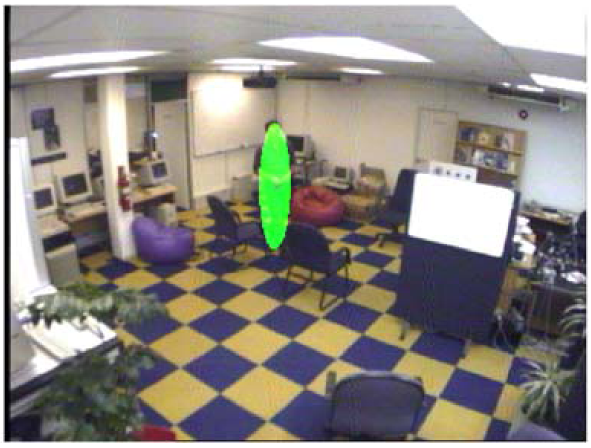

Figure 2.16: The result of tracking a person using the extended Kalman filter. An ellipsoid is drawn on the location of the person [17].

- Observation density (likelihood) : $p(Y_k|X_k)$

Where $p(k|l)$ defines the chance that k occurs when l is met.
The aim is to estimate recursively in time the filtering density $p(X_k|Y_{1:k})$ where $Y_{1:k} = \{Y_1, Y_2, ..., Y_k\}$, which is in target tracking the density of the state of the object at time k depending on all observations until time k. It is possible to estimate it from the filtering density of time $k-1$. This is done in two steps: the prediction step and the update step. In the prediction step, the filtering density $p(X_{k-1}|Y_{1:k-1})$ of the previous time is propagated via the transition prior as follows :

$$p(X_k|Y_{1:k-1}) = \int p(X_k|X_{k-1})p(X_{k-1}|Y_{1:k-1})dX_{k-1} \tag{2.57}$$

The update stage will use Bayes' rule when new data are observed to achieve the current filtering density :

$$p(X_k|Y_{1:k}) = \frac{p(Y_k|X_k)p(X_k|Y_{1:k-1})}{p(Y_k|Y_{1:k-1})} \tag{2.58}$$

The filtering density can then be used to compute the associated expectation for some integral function $f_k(X_k)$

$$E_{p(\cdot|Y_{1:k})}(f_k(X_k)) = \int f_k(X_k)p(X_k|Y_{1:k})dX_k \tag{2.59}$$

This strategy provides an optimal solution. But this solution includes a very expensive, or even impossible to compute high-dimensional integral. Therefore we rely on Monte Carlo methods. Monte Carlo methods are stochastic techniques, meaning that they use random numbers and probability statistics to investigate problems.
An example to understand the principle of Monte Carlo methods is shown in figure 2.17. The square is divided into two by some function $f(x)$ and we are interested in how much each part is of the total square. The Monte Carlo simulation creates a set of N uniform random values in the square and check if this point is inside the gray part of the square or not. The chance that a uniform random is under the function $f(x)$ is the same as that surface because the

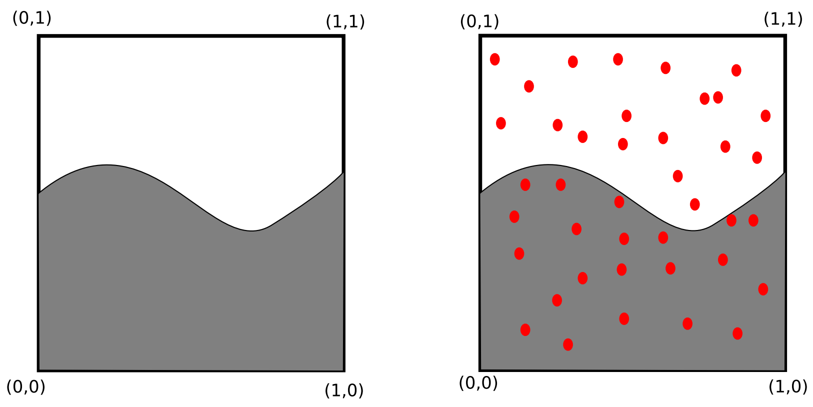

Figure 2.17: This square is divided into two and we want to find out how much of it is colored gray. The right images shows some random chosen points on the square used for Monte Carlo simulation

total surface of the square is equal to 1. The approximation for the gray part of the square is given by :

$$\int_0^1 f(x) \approx \frac{1}{N} \sum D(x) \tag{2.60}$$

where $D(x)$ is 1 if x is inside the gray part and 0 else. More about Monte Carlo methods can be found are described by Peter Shirley [48].

Such method can approximate the state with samples. Therefore a proposal density $\pi(X_k|Y_{1:k})$ is needed from which particles can be easily drawn and which is similar to the posterior density. To facilitate sequential estimation of the system state, the proposal density must satisfy the following equation :

$$\pi(X_k|Y_{1:k}) = \pi(X_k|X_{k-1}, Y_{1:k})\pi(X_{k-1}|Y_{1:k-1}) \tag{2.61}$$

Now the weight function $\omega(X_k) = \omega_k$ can be introduced which can be computed recursively :

$$\omega_k = \omega_{k-1} \frac{p(Y_k|X_k)p(X_k|X_{k-1})}{\pi(X_k|X_{k-1}, Y_{1:k}))} \tag{2.62}$$

The posterior distribution can be represented by this weight function up to a proportionality constant.
The estimation can be computed as :

$$E_{p(\cdot|Y_{1:k})}(f_k(X_k)) = \frac{E_{\pi(\cdot|Y_{1:k})}(\omega_k f_k(X_k))}{E_{\pi(\cdot|Y_{1:k})(\omega_k)}} \tag{2.63}$$

The open question now is which proposal distribution to choose. The optimal proposal distribution (OPD) minimizes the variance of the weights (ω_k), but its design is a non-trivial task. A much used proposal distribution is Gaussian :

$$\pi(X_k^{(i)}|X_{k-1}^{(i)}, Y_{1:k}) = N(\widehat{X}_k^{(i)}, \widehat{P}_k^{(i)}) \; i = 1, ..., N \tag{2.64}$$

This distribution is defined for each discrete particle $X_k^{(i)}$. $N(\widehat{X}_k^{(i)}, \widehat{P}_k^{(i)})$ denotes Gaussian with mean $\widehat{X}_k^{(i)}$ and covariance $\widehat{P}_k^{(i)}$.

The General Particle Filter

As explained in previous section the particle filter is a Monte Carlo simulation in which the posterior density is approximated by a set of particles and their associated weights $\{(X_{k-1}^{(i)}, \omega_{k-1}^{(i)}) \; i = 1, ..., N\}$. The new particles are achieved from the proposal distribution $\pi(X_k^{(i)}|X_{k-1}^{(i)}, Y_{1:k})$ and their weights are computed according to the particle likelihoods. If the weights are normalized, the posterior density can be represented by $\{(X_k^{(i)}, \omega_k^{(i)}) \; i = 1, ..., N\}$.
The standard particle filter can be divided into 4 steps which are described in pseudo code below :

```
1. **Initial step** : create a set of particles from the initial prior $p(X_0)$
   create : $\{(X_0^{(i)}, \omega_0^{(i)}), i = 1, ..., N\}$
   set $k = 1$

2. **Sampling Step** : create the new samples and define their weight
       a) For i = 1, ..., N
   Sample $X_k^{(i)}$ from the proposal distribution $\pi(X_k^{(i)}|X_{k-1}^{(i)}, Y_{1:k})$ (for example Gaussian)
       b) Calculate their new weights :
   $\omega_k^{(i)} = \omega_{k-1}^{(i)} \frac{p(Y_k|X_k^{(i)})p(X_k^{(i)}|X_{k-1}^{(i)})}{\pi(X_k^{(i)}|X_{k-1}^{(i)}, Y_{1:k})}, i = 1, ..., N$
       c) Normalize the weights :
   $\omega_k^{(i)} = \frac{\omega_k^{(i)}}{\sum_{j=1}^{N} \omega_k^{(j)}}, i = 1, ..., N$

3. **Output step** : outputs the set of particles $\{(X_k^{(i)}, \omega_k^{(i)}), i = 1, ..., N\}$
   This set can be used to approximate the posterior distribution and the estimate as :
   $p(X_k|Y_{1:k}) \approx \sum_{i=1}^{N} \omega_k^{(i)} \delta(X_k - X_k^{(i)})$
   $E_{p(|Y_{1:k})}(f_k(X_k)) \approx \sum_{i=1}^{N} \omega_k^{(i)}$

4. **Selection step** : resampling
   Resample particles $X_k^{(i)}$ with probability $\omega_k^{(i)}$ independent and identically distributed random particles $X_k^{(j)}$ approximately distributed according to $p(X_k|Y_{1:k})$

5. Increment $k$
   goto sampling step (2)
```

$\delta(x)$ is the Durac delta function which has the following properties :

$$\int_{-\infty}^{+\infty} f(x)\delta(x) = f(0) \tag{2.65}$$

This function is 1 where x is 0 and 0 elsewhere.

2.4.5 Condensation Algorithm

The condensation algorithm was presented by Michael Isard and Andrew Blake [25]. The name is a substitution for CONdensional DENSity propagATION. As explained before it is a variant of the particle filter, in which the proposal distribution is the transition prior :

$$\pi(X_k^{(i)}|X_{k-1}^{(i)}, Y_{1:k}) = p(X_k^{(i)}|X_{k-1}^{(i)}) \tag{2.66}$$

And the weights becomes :

$$\omega_k^{(i)} = \omega_{k-1}^{(i)} p(Y_k|X_k^{(i)}) \tag{2.67}$$

This algorithm can briefly be described as follows :

1. **Initial step**
2. **Sampling Step** : create the new samples and define their weight

   a) For i = 1, ..., N
   Sample $X_k^{(i)}$ from the proposal distribution $p(X_k^{(i)}|X_{k-1}^{(i)})$

   b) Calculate their new weights :
   $\omega_k^{(i)} = \omega_{k-1}^{(i)} \frac{p(Y_k|X_k^{(i)})p(X_k^{(i)}|X_{k-1}^{(i)})}{\pi(X_k^{(i)}|X_{k-1}^{(i)}, Y_{1:k})}, i = 1, ..., N$

   c) Normalize the weights :
   $\omega_k^{(i)} = \omega_{k-1}^{(i)} p(Y_k|X_k^{(i)}), i = 1, ..., N$
3. **Output step**
4. **Selection step**
5. Increment k
   goto sampling step (2)

2.4.6 Particle Filter for Object Tracking

When we follow the definition of the condensation algorithm described by Isard Blake, tracking an object in three dimensions using the condensation algorithm can be done in three steps for each frame. These steps are called selection, prediction and measure.

- Select : in this stage the particles are created using the particles and their weight of the last frame (as the selection or resampling stage described in 2.4.5)
- Predict : the particles are diffused using Gaussian random values (first stage of the sampling step)
- Measure : calculating the weight of every particle and normalize them (second and third stage of the sampling step)

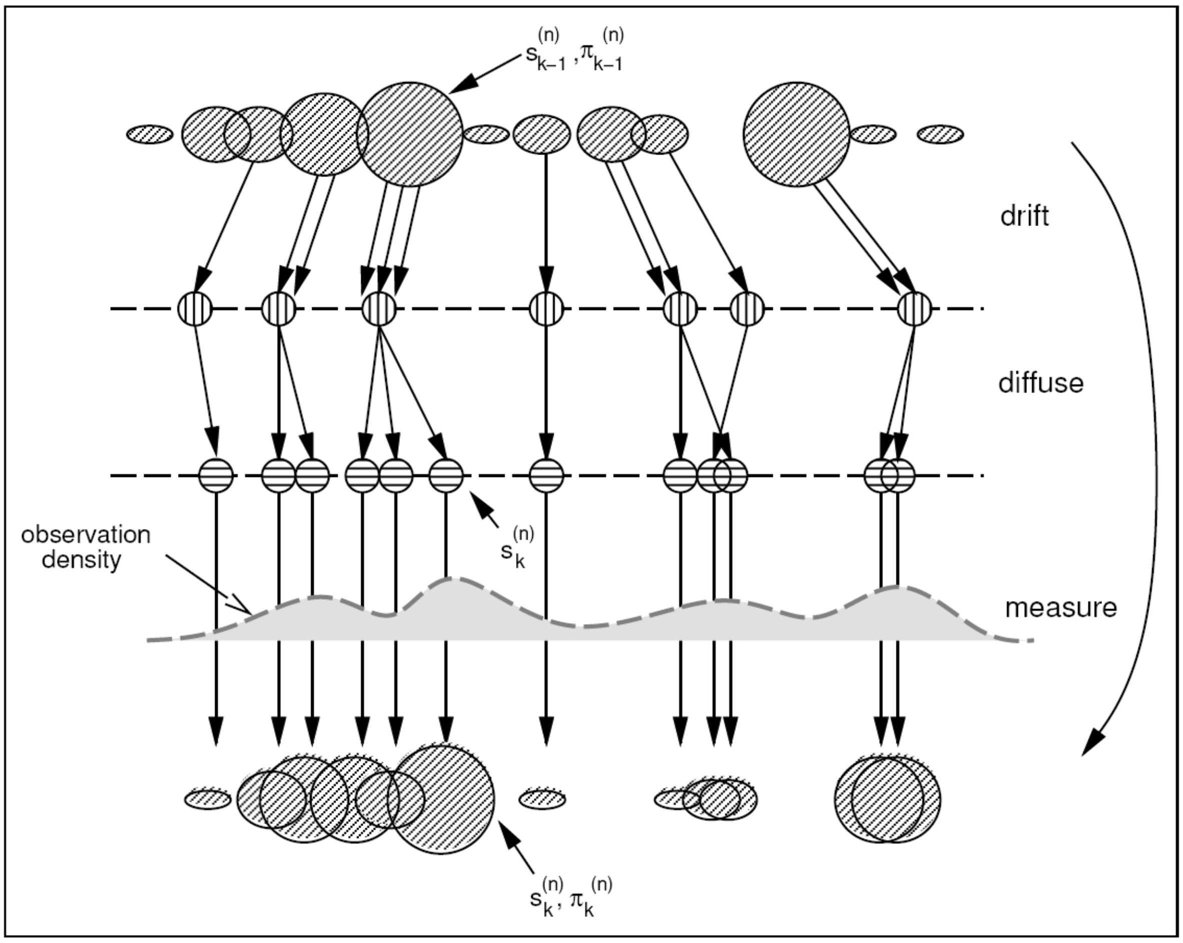

Figure 2.18: One step of the condensation algorithm (drift) in the selection step new particles are created using the weights of the last particles (diffuse) in the prediction step the particles are diffused (measure) in the measure step the weights of the new particles are calculated by observing the scene [25]

Initialize

A particle has coordinates (x, y, z) and a weight π as parameters. In the beginning of the algorithm, no particles are created yet. Therefore a set of N random particles are created in the scene. The location is uniform random and the weight is just $\frac{1}{N}$, so it is already normalized.

Select

In the select step N new particles are created, each with the same location as a particle from last frame. They are chosen according to the weight of the particle. If the weight of the particle is higher, than it will be chosen more often. This is done by creating a cumulative array of all the normalized weight values. A uniform random value between 0 and 1 is chosen and the first particle with a cumulative weight higher then this value is chosen. A one dimensional example is shown in figure 2.18 (drift). Particles with a higher weight are in this example sometimes rechosen twice or three times while particles with a low weight are not chosen again.

Predict

To predict the new location, the particles are diffused by a Gaussian model. A Gaussian random will be added to the x, y and z coordinate with average 0 and standarddeviation σ. Every particle has a unique location again and the density is higher around previous particles with a higher weight. This step is also shown in figure 2.18 as the diffusion.

Measure

Every particle will be measured using the input frames. This step is the only step that should be changed when tracking other kinds of objects because the measuring is done according to a property of the object. A simple example for measuring a particle is using the color. The particle can be projected onto every camera and the color at that pixel can be used as measurement. The better the color responds to the object the higher the value.
Once all the particles contain a weight they can be normalized and a cumulative array of the weights can be created which is used in the select step of next frames.

2.4.7 Tracking of Hands and Face

The next algorithm is not a general purpose algorithm like the Kalman filter or the particle filter. This one is specialized for tracking the head and hands of a person using the skin color. To create a player in a game it is not always necessary to calculate the entire pose of the player. When he is trying to interact he commonly does this with his hands. The location of the body is less important. The algorithm described below is based on an article written by Perales et al [14] to track the hands and face of a person with calibrated cameras.
It is also divided into 3 stages :

- Skin-Color Pixel Detection
- Data Association
- 3D Position Estimation

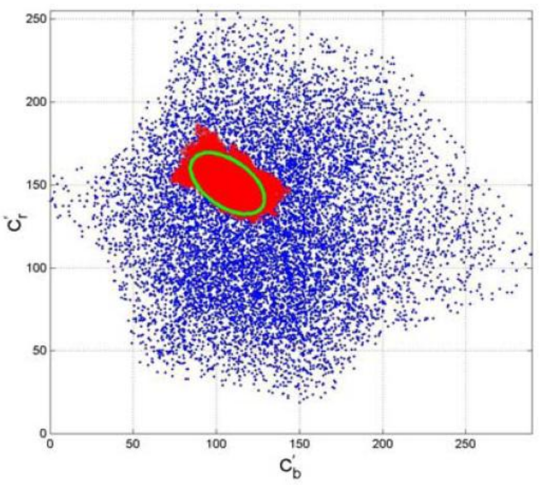

Figure 2.19: This ellipse defines the relationship between the combination Cb-Cr and the skin color [49]

The first stage detects which pixel in each image are skin-color pixels. The result of this stage is a set of blobs, which are groups of connected pixels with skin color. The second stage uses this set of blobs to detect the hands and the face on each image. Once this is done, the third stage can back project the found locations to calculate the three dimensional location of the hands and face.

Skin-Color Pixel Detection

The algorithm described by Perales [14] needs as input the color of the person, for example by clicking on some skin color on the recorded images. Once the color is known it can be used for detecting similar pixels using a segmentation algorithm to create blobs of skin colored pixels. This input is a list of n pixels of the persons skin.
After obtaining samples the pixels are transformed from the RGB-space to the HSL-space to take the hue and the saturation values, which is the chroma information. This is done because the difference between the pixels for the skin are smaller in HSL-space than in RGB-space. Next a two dimensional Gaussian model is built with these input samples. The model represents the probability that a pixel is skin colored. In order to detect the skin colored pixels in the image the probability for every pixel is computed. When this probability is higher than a predefined threshold, T_{max}, the pixel is marked as true, else false.
In the last stage of the algorithm the list of positive pixels is expanded by recursively adding those which are direct neighbours of true pixels and have a probability higher then T_{min}.
The result of this algorithm is very good but has the disadvantage of needing sample input. Another algorithm which does not need the skin color as input makes it easier for the player(s). This algorithm was found in an article by Senior et al about face detection [49]. Terrilon [50] found out by using statistics that the skin color can easily be detected in Y Cb Cr. The relationship between the combination Cb-Cr and the skin color is shown in figure 2.19. Every combination within the ellipse is marked as skin color, every outside as no skin.

Data association

In this step the skin colored pixels will be used to locate the face and hands on a two dimensional image.
The skin detection algorithm creates a set of blobs $b = \{b1, b2, ..., bn\}$. A blob is a set of connected skin colored pixels, which can be used to estimate the location of the three limbs $H = \{h_1, h_2, h_3\}$, the face and the two hands of the person. An ellipse $h_i = (c_x, c_y, A, B)$ is used to describe a limb. The parameters of this ellipse are (c_x, c_y), the center of the ellipse and A and B the lengths of the semi axes. Because of this definition it is easy to find out if the limbs contains a pixel or not. To calculate the distance from an ellipse to a pixel p the following formula can be used :

$$D(p, h) = \sqrt{(\frac{(p_x - h_{cx})}{h_A})^2 + (\frac{(p_y - h_{cy})}{h_B})^2} \tag{2.68}$$

If this distance is smaller or equal to one, the pixel is inside the ellipse.
First, the existing blobs from last frame are associated with the calculated limbs according to two rules :

- Rule 1: if the blob and the limb have a pixel in common, they are associated
- Rule 2: if a blob has no pixel in common with any blob, it will be associated with the limb which is the closest

Hereby a limb can have multiple blobs associated with it. In this case only the blob with the most common pixels is taken. Also the limbs center and axis have to be adjusted to the blobs.
If a limb has no blob with a common pixel, it will be removed. Therefore it is possible to have less than three limbs, in this case the largest blob which is not associated with any limb is chosen to be the new limb.
The location of a limb is predicted for the next frame, so there is more chance of finding the right blob in the next frame. The last replacement is used for this prediction.

$$\begin{aligned} C(t+1) &= C(t) + P(t) && (2.69) \\ P(t) &= C(t) - C(t-1) && (2.70) \end{aligned}$$

This assumes that a part will maintain the same velocity on the image plane.

3D Position Estimation

The previous two steps calculate the location of face and hands on a two dimensional image. When working with multiple input images it is possible to find the three dimensional locations. The method for doing this depends on the kind of information available. When the depth map or parts of it are available, it is easy to find the three dimensional location because the distance of the pixel from the camera is given. In this case the location of hands and face should only be calculated in one image.
In a more general case where no depth map is available, it is necessary to calculate the location of the limbs in two or more images. The two dimensional locations of the limbs can be back projected into space and the intersection of those lines give the three dimensional location as already explained in section 2.2.5.

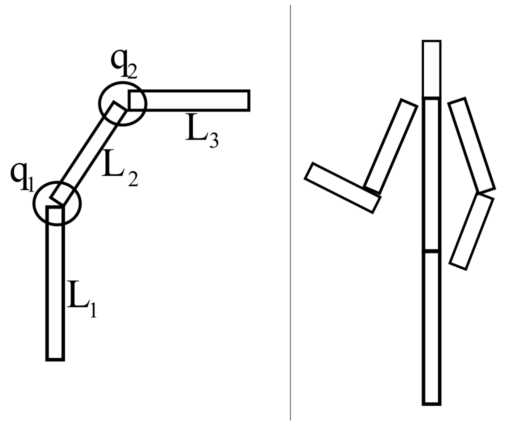

Figure 2.20: Left, a simple skeleton defined by three links L_1, L_2 and L_3 which are connected by the joints q_1 and q_2. Right, a simplified skeleton to represent a human.

2.4.8 Kinematics

One of the possible further uses of object tracking is to estimate the 2D or 3D pose of the player, who is modelled by a skeleton. A skeleton is a set of simple rigid objects, often called links, connected by joints [20]. The joints are usually rotational, but may also be sliding. Each rotary joint may allow rotation in 1, 2 or 3 orthogonal direction, which are its degrees of freedom. Two simple skeletons are given in figure 2.20. The pose of the skeleton is defined by all degrees of freedom of the joints.
Buades et al [51] proposed a method to estimate the entire pose using a particle filter for each degree of freedom. This technique is called *forward kinematics*, which involves explicitly setting the position and orientation of objects. The opposite techniques are called *inverse kinematics* [19]. These provide direct control over the placement of an *end effector* object at the end of a kinematic chain of joints, for example a hand of the player, calculating the intermediate joint rotations which place the object at the desired location. For example, Jaume et al [52] proposed a human pose estimation with inverse kinematics by only tracking the location of the hands and the face of a person.
To solve the inverse kinematic problem a lot of algorithms have been designed, much of them using the *jacobian matrix* J:

$$J = \frac{\partial f}{\partial q} \tag{2.71}$$

This matrix expresses the relationship between joint velocities $\dot{q}$ and the velocity of the end effector $\dot{x}$:

$$\dot{x} = J(q)\dot{q} \tag{2.72}$$

It is used to map changes in the joint parameters q to changes in the end effector position and orientation x. The i^{th} column of J represents the incremental change in the position and orientation of the end effector resulting from an incremental change in the joint variable q_i. The changes in position and orientation can be measured for the end effector resulting in the following equation to find the changes to the joint variables:

$$\dot{q} = J(q)^{-1}\dot{x} \tag{2.73}$$

But the Jacobian matrix is not always invertible, because it is singular or rectangled. Then the *psuedoinverse* of the Jacobian matrix can be calculated. But it is very expensive and numerical instable around singular configurations.
Alternative algorithms like the *Jacobian Transpose Method* and *Cyclic Coordinate Descent* are efficient enough, numerical stable and relatively simple [19]. The idea behind the first algorithm is to add a force F on the end effector in the direction of the desired location. This force in Cartesian space can then be transformed into an acceleration on the joint variables $\ddot{q}$ by using the transposed Jacobian. Because it is not necessary to acquire an accurate dynamic simulation, it suffices to use this acceleration as an estimate for the joint displacements $\dot{q}$:

$$\dot{q} = J^T F \tag{2.74}$$

This displacement is added to the joint variables, moving the end effector closer to the desired location. This procedure repeats until the end effector reaches its goal position or another stopping criterion is met.
The cyclic coordinate descent (CCD) is an iterative heuristic algorithm which varies one joint variable at the time in an attempt to move the end effector closer to the desired location. This is done by finding the joint variable q which minimizes the error measure. This error measure is a combination of a position error, the distance between the current location and the goal, and an orientation error. This minimize equation is solved for one joint at a time, from the end of the tree to the root. After each step the end effector will be moved closer to the desired location.

3 Implementation

In this chapter the implementation of a couple of computer vision algorithms are explained. These are divided into *depth estimation* in section 3.1, *visual hull construction* in section 3.2 and object tracking in section 3.3.

3.1 Depth Estimation

This section contains an implementation of the disparity calculation explained in section 2.2.4. The first subsection will start with explaining the general disparity calculation of the algorithm which has good results, but is far from real time. Therefore some optimizations are used to speed it up which are explained afterwards.

3.1.1 Disparity Calculation

The data set of Tsukuba University shown in figure 3.1 is commonly used for testing disparity computation. These images are rectified and will be used to present this algorithm.
First the matching cost for every x and y coordinate and for every possible depth value d will be computed. This is done with the sum of absolute differences. These calculations are then stored into a three-dimensional array.
The disparity map can be created by taking the disparity value for every pixel where the matching cost is the lowest. This straight-forward implementation gives a pretty good result as seen in figure 3.2, but it is very slow. To calculate a disparity map from a pair of images with resolution 320×240, a disparity range of 40 and a window size of 7×7 it takes 4703 ms on a Intel Pentium Mobile 1.5 GHz. More detailed results are given in chapter 4.

3.1.2 Optimalizations

Exploiting some properties of this algorithm is a good way to increase the speed and make it realtime. These speedups contain a better memory organisation and using a sliding window to save calculations.

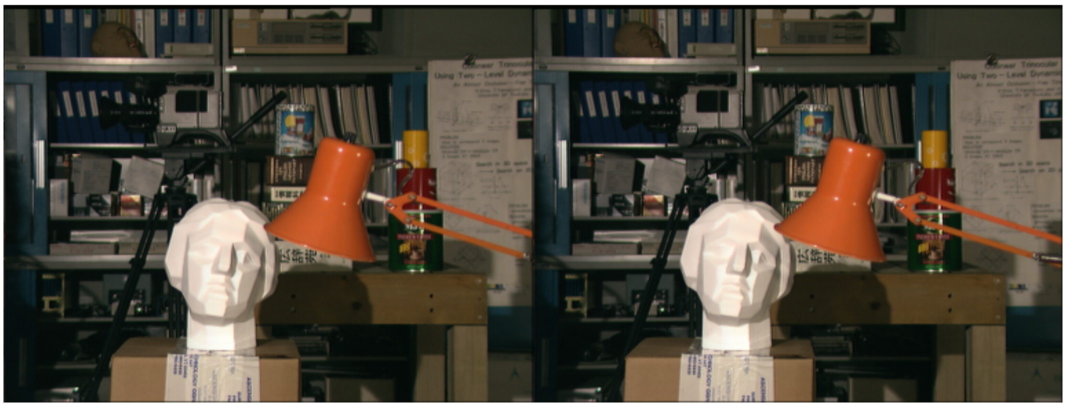

Figure 3.1: These rectified images are used to demonstrate the algorithm. The data set is presentated by Tsukuba University

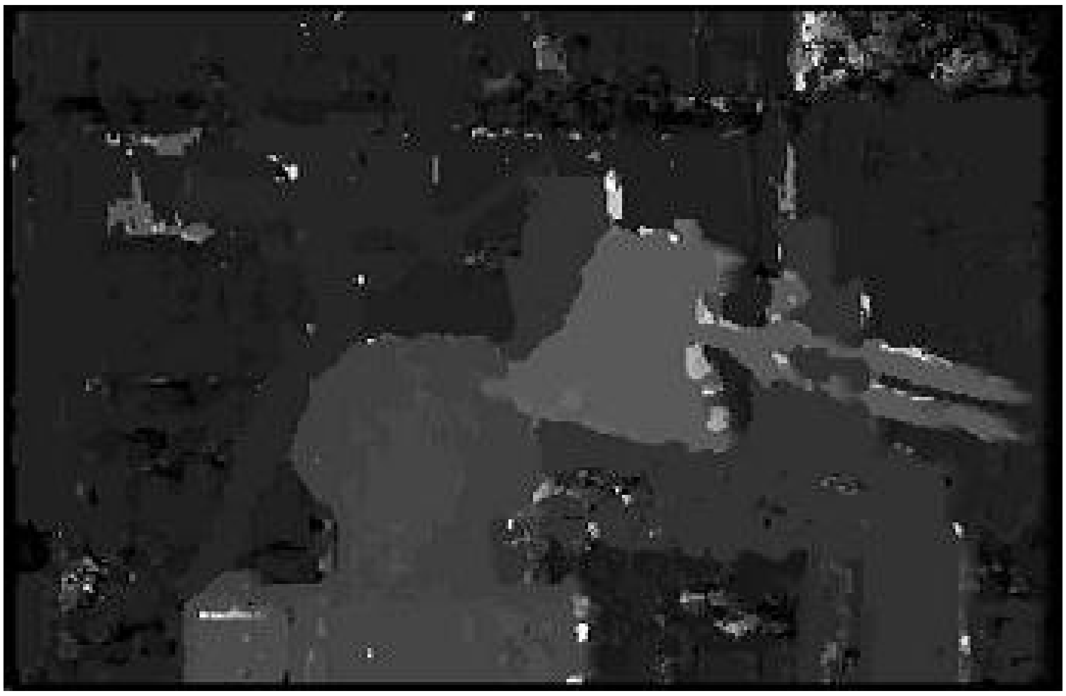

Figure 3.2: The result of the depth-estimation algorithm on figure 3.1

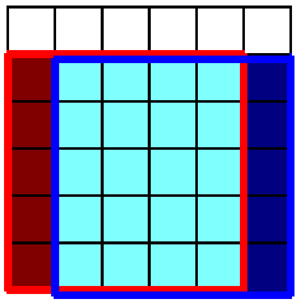

Figure 3.3: The red 5x5 window is the previous calculated window, the blue one is the next window to be calculated. The light-blue squares are the pixels calculated twice.

Memory Organisation

A first important thing is the order of filling up the cuboid, so the order of for-loops to go though every x, y and d value:

- $d - x - y$: this is the straightforward implementation, complete slices for constant d are computed. Hereby it is necessary to read the 2 images for every d-slice. The lowest SAD-value can only be defined when the whole cuboid is filled.
- $y - d - x$: more compact layout, since every y-slice is used only $win_y + 2$ times and can be discarded afterwards. This allows to implement a ring buffer in y, reducing the memory consumption significantly. Hereby this approach is a bit faster than the first one.
- $y - x - d$: this approach performs better than the last 2. Loading a cache line into a pentium processor involves 32 bytes [53]. Using one value in the disparity space volume brings 7 other values with it into the L1-cache. Because the 7 following values in the d-direction will be needed, a lot of time is saved.

Summing up the windows

An inefficiency of this algorithm is calculating the same values over and over again. If the size of the window is n by n, n^2 pixel comparisons are needed to calculate the matching cost for one pixel. But as seen on figure 3.3 $(n - 2) \times n$ of these comparisons are already calculated for the previous pixels because of the overlap. To use this property, calculating the difference between the SAD of the current window and the last one should be enough. This can be done by adding the right column to the SAD value and substract the left column. So instead of comparing n^2 pixels we only compare n pixels. The sum of the columns is stored into a queue for easy subtracting the left one.
This optimalisation increases the speed enormously and makes it possible for real-time applications. The calculation of the disparity map from a pair of images with resolution 320×240, a disparity range of 40 and a window size of 7×7 has been reduced from 4703 ms to 485 ms. The complete list of times is shown in next chapter.

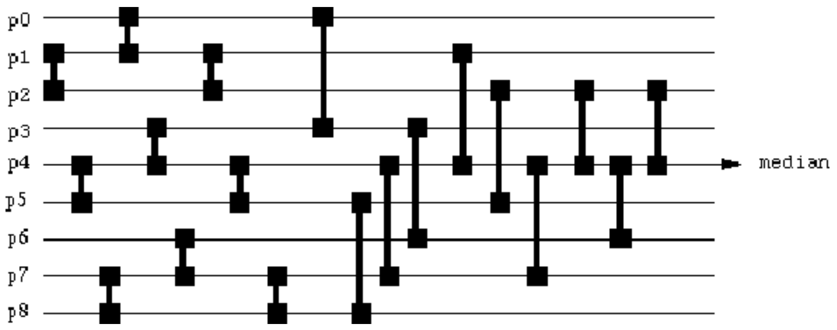

Figure 3.4: Minimum sorting network to calculate the median of nine elements [54]. If two elements are connected it means we have to compare them and swap them if the value on top is higher.

3.1.3 Noise reduction

As explained in section 2.2.4 noise reduction can be done with a median filter. This filter calculates the median over a window for every pixel to reduce extreme values. The window size for this filter is 3×3 and so the median of nine values has to be found. This can be done with a sorted list, but it is too expensive. A faster method is making use of a minimum sorting network to calculate the median of nine elements. Figure 3.4 shows this network in a scheme. $P0$ until $P8$ are the nine input values. When going from left to right, every time two values are connected they have to be compared. If a value with a lower index is higher than a value of a higher index the values must be switched. $P4$ will eventually be the median of all values.

The median filter gives a slightly better result which is shown in figure 3.5. In areas with the same real disparity some noise will be reduced as shown at the face. The improvement is better for low resolution disparity maps like 160×120.

This disparity map can be used to display the three dimensional scene by rendering the pixels at their depth value as shown in figure 3.6.

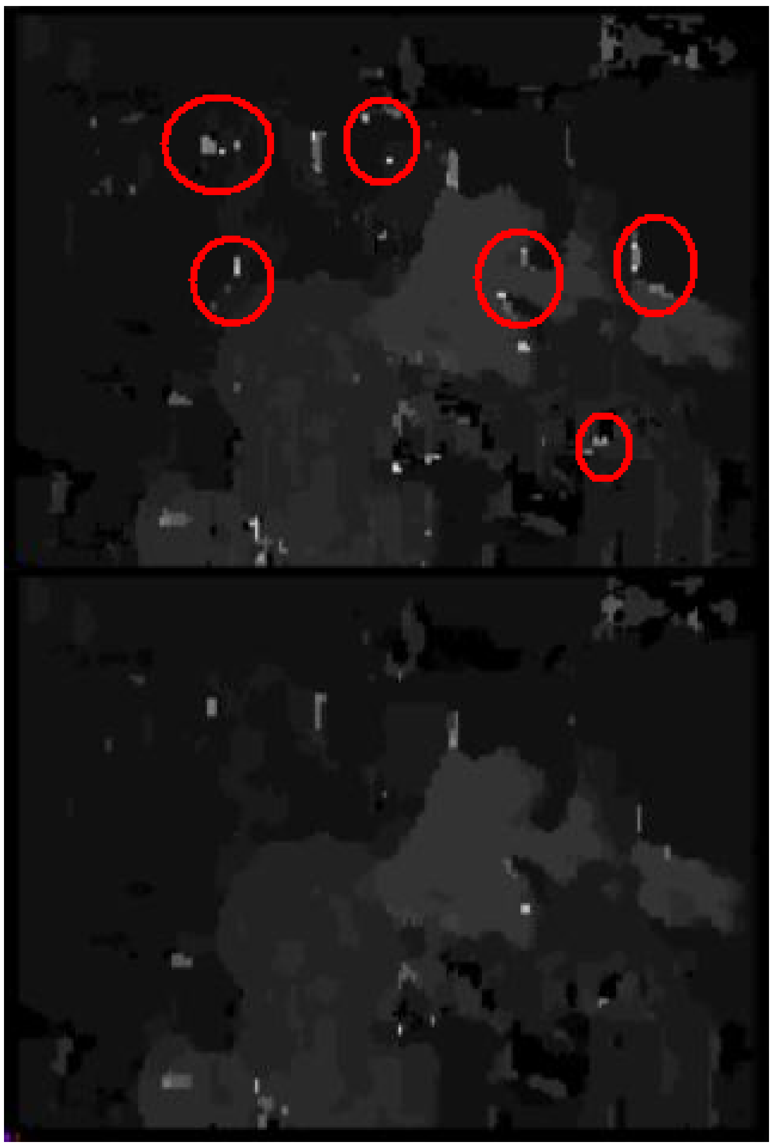

Figure 3.5: The top image is the disparity map without the median filter, the bottom disparity map is calculated with the median filter. Difference is minimal, but some of the extreme values can be reduced as shown in selected part.

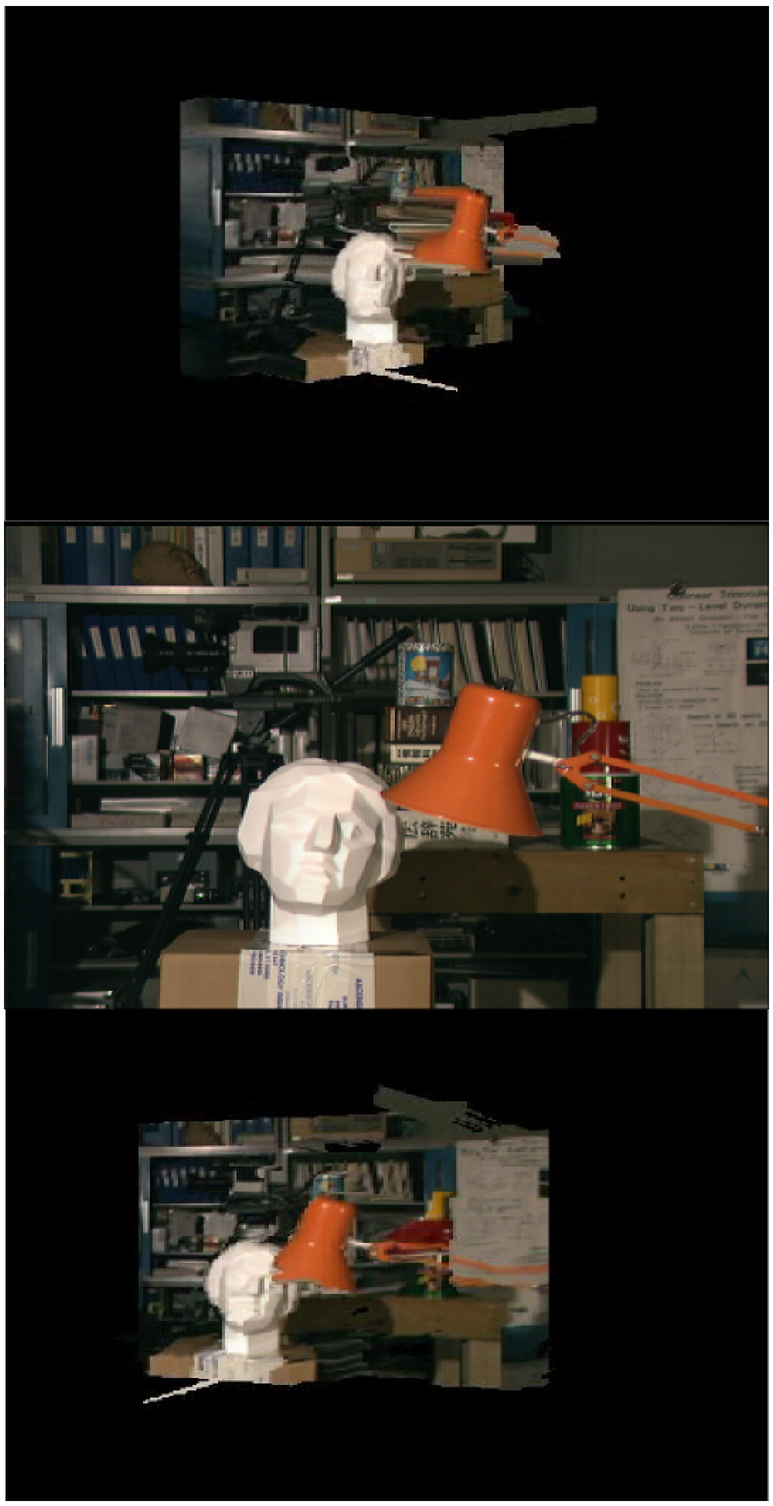

Figure 3.6: The reference image together with the depthmap can be used to display a three dimensional model of the scene

3.2 Visual Hull

The next sections contains the implementation of three kinds of visual hulls. The first two are visual hull data types calculated on the CPU for easy collision detection. These are cubic voxel visual hull and octree visual hull. The last algorithm creates a visual hull on the GPU and can be used for output. So there can be collision between a real person and the virtual scene and it is possible to place a three dimensional model of the player in the scene.

3.2.1 Cubic Visual Hull

This first algorithm is very straight forward. The space containing the object, will be divided n by m by k cubes. Each of them, also called voxels, represent a part of space. So for every voxel a calculation is required to check if it intersects with the object. If the object intersects, the voxel gets the value true, else false. The space is then transformed into a three dimensional array of booleans. Collision detection is then possible because only the voxels representing the other objects have to be checked for intersection to see if there is a collision. The checking can be done with direct access so this is very fast.
The calculation of the values of each voxel is also easy. For every voxel its center can be projected on every camera using the projection matrix of each camera. After projecting it has to be checked whether it is inside or outside the foreground. And by definition of the visual hull, all these projections should be inside all silhouettes to make part of the visual hull. Therefore the value of the voxel is the intersection of the projected points.
Only the voxels with value true are part of the visual hull. The calculated visual hull can be displayed by only drawing the true voxels as a cube. This visualisation is shown in figure 3.7.

3.2.2 Octree Visual Hull

An octree is another volumetric representation for visual hull. It uses less memory and is sometimes faster to calculate. Above this it can be more detailed than the cubic representation from the last section. This algorithm was defined by Erol et al [23]
The scene is initially one big voxel. It can then be divided into eight smaller voxels by dividing the width, height and depth of the voxel into two. This dividing process can go on recursively until a predefined depth is reached. It is not necessary do split all voxels every step. When it is known that a voxel is entirely inside or outside the visual hull, it is also know that all its children has the same property. Therefore only the voxels which are partly inside and partly outside will be divided. This approach saves a lot of time and can be calculated more detailed around the object.
The only thing that needs to be explained now is how to know if a voxel is inside, outside or contains boundaries of the object. A preprocessing stage is necessary for this. In this stage the silhouette is transformed into an integer image. Every pixel of this new image has the value of the euler distance from this pixel to the edge of the objects silhouette. This means that further inside the silhouette the pixel value gets higher. The pixels outside the silhouette have a negative value representing the distance to the edge of the silhouette.
This can be done by first detecting the edges and giving those pixels value 0 and then putting their neighbour pixels in a queue. Every pixel inside this queue, inside the silhouette and that without a known distance value gets the value of its closest known neightbour plus the distance to it. All his neighbours without a known distance are then put into the queue. The

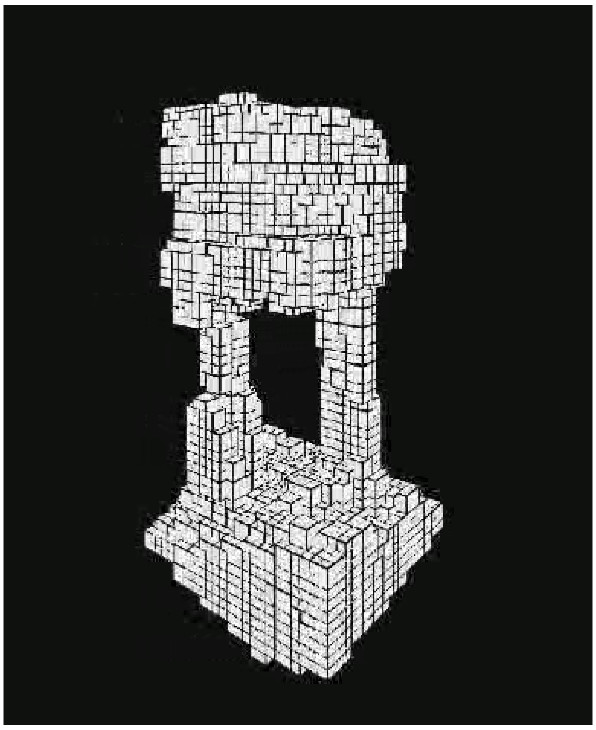

Figure 3.7: A cubic visual hull of a dataset from a temple [55]

same thing can be done for the pixels outside the silhouette. The result of this preprocessing step on the temple dataset is given in figure 3.8.

After the preprocessing stage it is possible to check if a voxel is inside or outside the object. This is done by projecting the center of the voxel on the images and looking at the distance value at that location. If this distance value is larger than the distance between the projected center of the voxel and a projected corner it means that the voxel is completely inside the object. The same can be done to check if the voxel is completelty outside the silhouette.

This can be visualised with the same method as the cubic visual hull, but instead of drawing cubes of the same size, the size will depend on the depth of the cube in the tree (figure 3.9). Collision detection can now be done by traversing through the tree and only use the branches that contain the same volume.

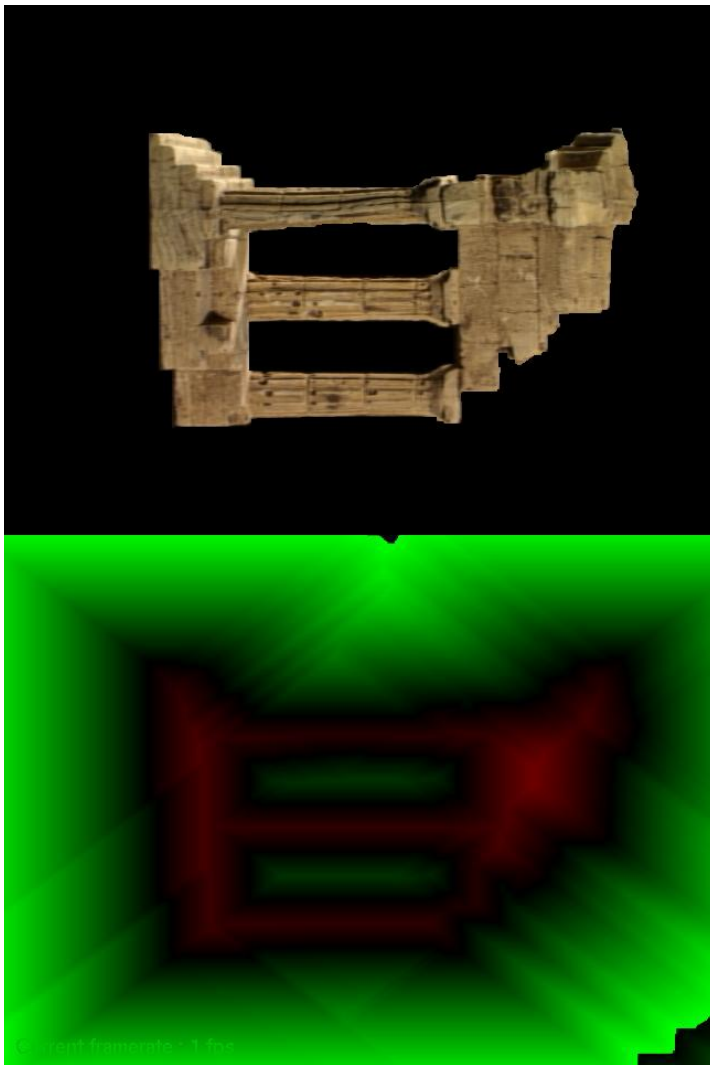

Figure 3.8: The result after the preprocessing stage. Red pixels indicate how far inside the silhouette while green pixels indicate how far outside the silhouette

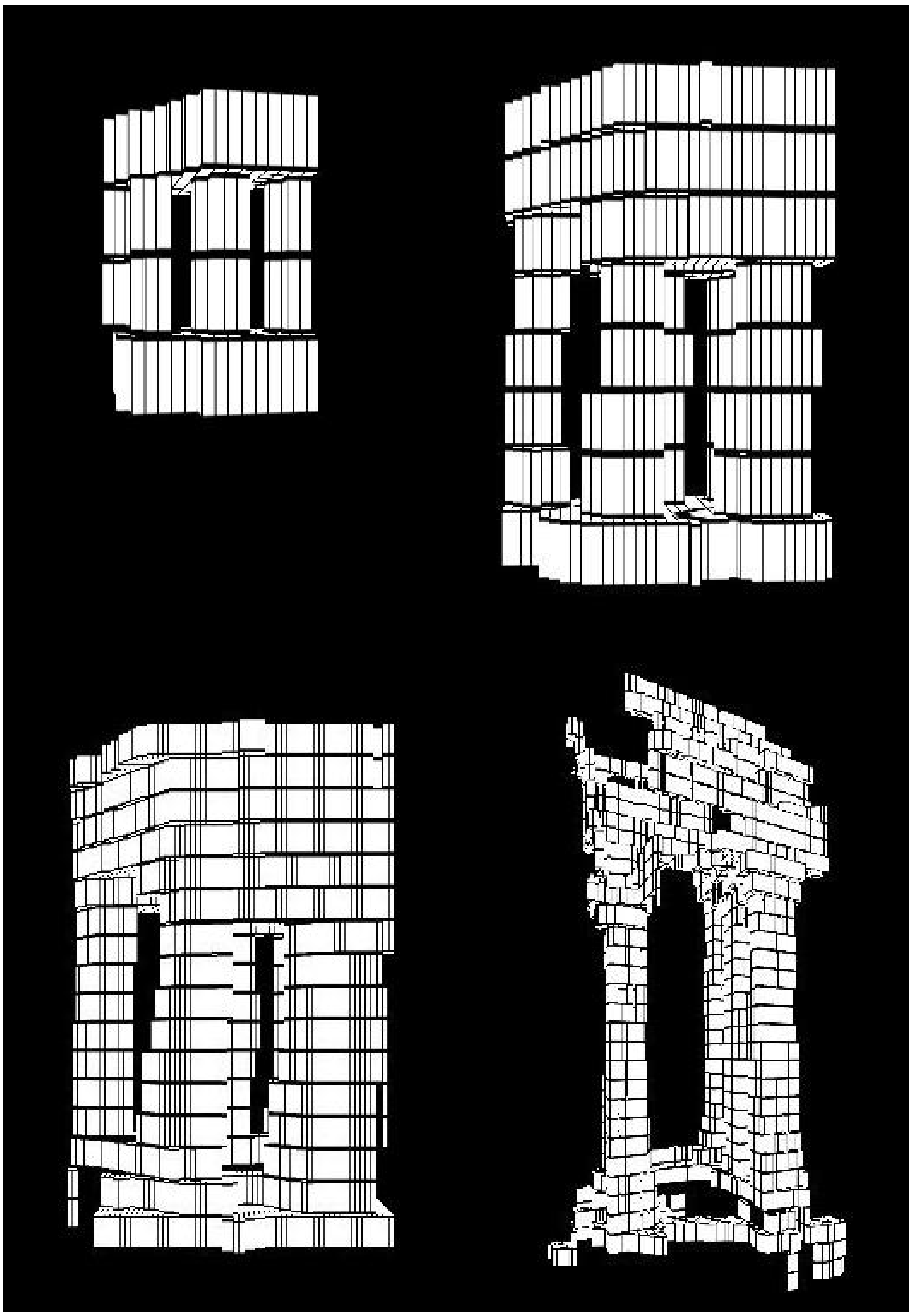

Figure 3.9: An octree visual hull of the same dataset with depth 5, 6, 7 and 8

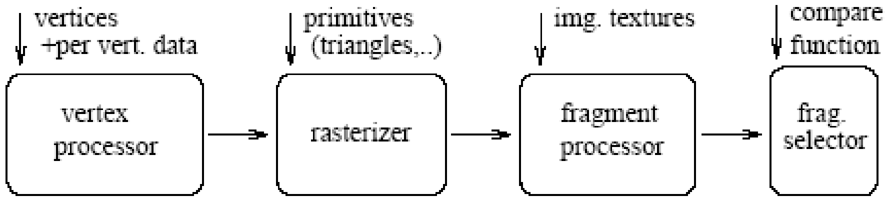

Figure 3.10: Simplified flow of data though GPU pipeline [29].

3.2.3 Sliced Visual Hull

The next algorithm will create an image hull of the object so it can be used to display a three dimensional model of the player in the game. It contains a set of slices of the visual hull, but so close together that it looks like a complete three dimensional model when rendering. This algorithm was presented by Keith Yerex [24].
The graphical processing unit (GPU) is becoming more and more interesting for computer graphics algorithms to speed them up. Therefore a brief introduction to the modern GPU is necessary before going on. This is done in the next section followed by an implementation of a sliced visual hull on the GPU.

GPU : Graphical Processing Unit

Today's graphics architecture (fourth generation) can be used as a fully programmable hardware. It is optimized to work with vertices and images. This creates new possibilities for image processing because it can do some work instead of the CPU.
It is important to understand the concept of current GPU-architecture to develop and adapt general purpose algorithms for implementation on GPU hardware. First of all, the GPU supports vector operations, Single Instruction - Multiple Data (SIMD). An other advantage is that the architecture is divided into four stages and is pipe lined (figure 3.10). This pipeline is used to calculate in parallel. Even low-cost graphical cards have 8 processor units working completely independent in parallel. The four stages of the GPU are :

- Vertex processing: this stage gets as input a vertex with his parameters and generates exactly one output vertex. For example it gets a three dimensional vertex as input. The output can be the projected two dimensional vertex, for example multiplied by the model view- and projectionmatrix.
- Rasterizer: this stage will group the vertexes into fragments, which are potential pixels for a given viewing frustum projection.
- The fragment processor: gets as input a fragment, received from the rasterizer, and outputs the color of the fragment. An example is given the color to a pixel by knowing the texture and his texture coordinates.
- Fragment selector: will select which fragment will be chosen to be drawn at each pixel. The best example is selecting the nearest fragment for every pixel, called depth testing.

The enormous processing power of graphics cards opens new opportunities. This power can be seen by the number of transistors. A pentium 4 processor has approximate 55 million transistors in comparison to approximate 125 million transistors of Nvidia Geforce FX graphics cards. Also it is optimised for image processing. A lot of image problems have been solved on the GPU with a high speedup factor compared with the CPU [56, 57]. Another big advantage is that when you let the GPU calculate the solution or a part of the solution, the CPU is idle and free to use. This way the CPU and the GPU can execute different algorithms or part of algorithms in parallel.
Even though the GPU has four stages only two of them are programmable: the vertex and fragment processor. A vertex program has as input one vertex and as output one vertex, it can for example be used to transform the scene by applying an extra transformation to every vertex. A fragment program has one input fragment and one output fragment and can be used to specify the color for each fragment.
A language developed by NVidia can be used to program these processors. This language is called Cg, which stands for "C for graphics" [58]. This language removes the need to program directly to the graphics hardware assembly language. It is based on C, this because of its popularity, and therefore easier to use. But it is very specialized for graphical purposes, so not everything is possible with it. It is a cross-platform language and can be used with OpenGL or DirectX.
More information about the current graphical processing units and Cg can be found in the Nvidia Cg manual.

GPU Visual Hull

The idea of this algorithm is exactly the same as the definition given in the introduction of the visual hull. The visual hull can be obtained by back projecting every silhouette into the scene and intersects the responding cones.
The algorithm presented by Keith Yerex [24] does this by drawing a series of parallel squares in the scene and projecting the three camera images on it. The textures are all in RGBA-format and the pixels that are in the silhouette are drawn with alpha value = 1, others with alpha value = 0. Projecting the image of one camera on all of those slices with alpha-blending creates a result as shown in figure 3.11.
For more then one camera placing multiple textures on the same slice is necessary. The GPU can then calculate the intersection of the textures.

To implement this OpenGL and Cg are used. The OpenGL extensions *glMultiTexCoord2fARB* gives the possibility to connect multiple textures with every slice. By projecting the corners of every slice onto the cameras the texture coordinates of the image can be found. Next, the slices are drawn with alpha blending enabled.
The last stage is done on the gpu by a fragment-program written in Cg. Only fragments which are part of the visual hull have an alpha value equal to one for every texture. If not, its projection is not inside every silhouette and therefore the alpha value of his color must be set to zero so it becomes invisible.
To define the color of fragments inside the visual hull, the program uses the location of the fragment and the refence cameras. These locations can then be used to calculate the weight for every camera. The color of the fragment is eventually calculated as a weighted sum of the colors of the textures and the weight of the cameras. The algorithm has been executed on two kinds of graphical cards. The first card is a Mobile ATI radeon 9700, the other is a GeForce

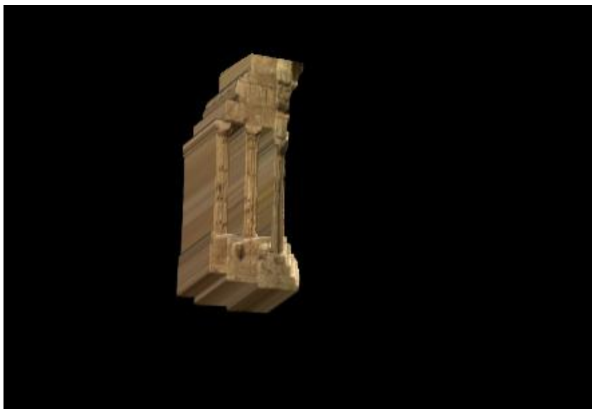
Figure 3.11: The projection of one camera on every slice

7300 GT. The first card created the visual hull in 110 ms with 40 slices. The GeForce card got a framerate of 25 fps (40 ms per frame) when using 80 slices (figure 3.12).
After trying different views, one of the conclusions of this algorithm is that it works better if the cameras are all at the same side of the object and the angle between them is not too large, approximate 30 degrees works just fine.

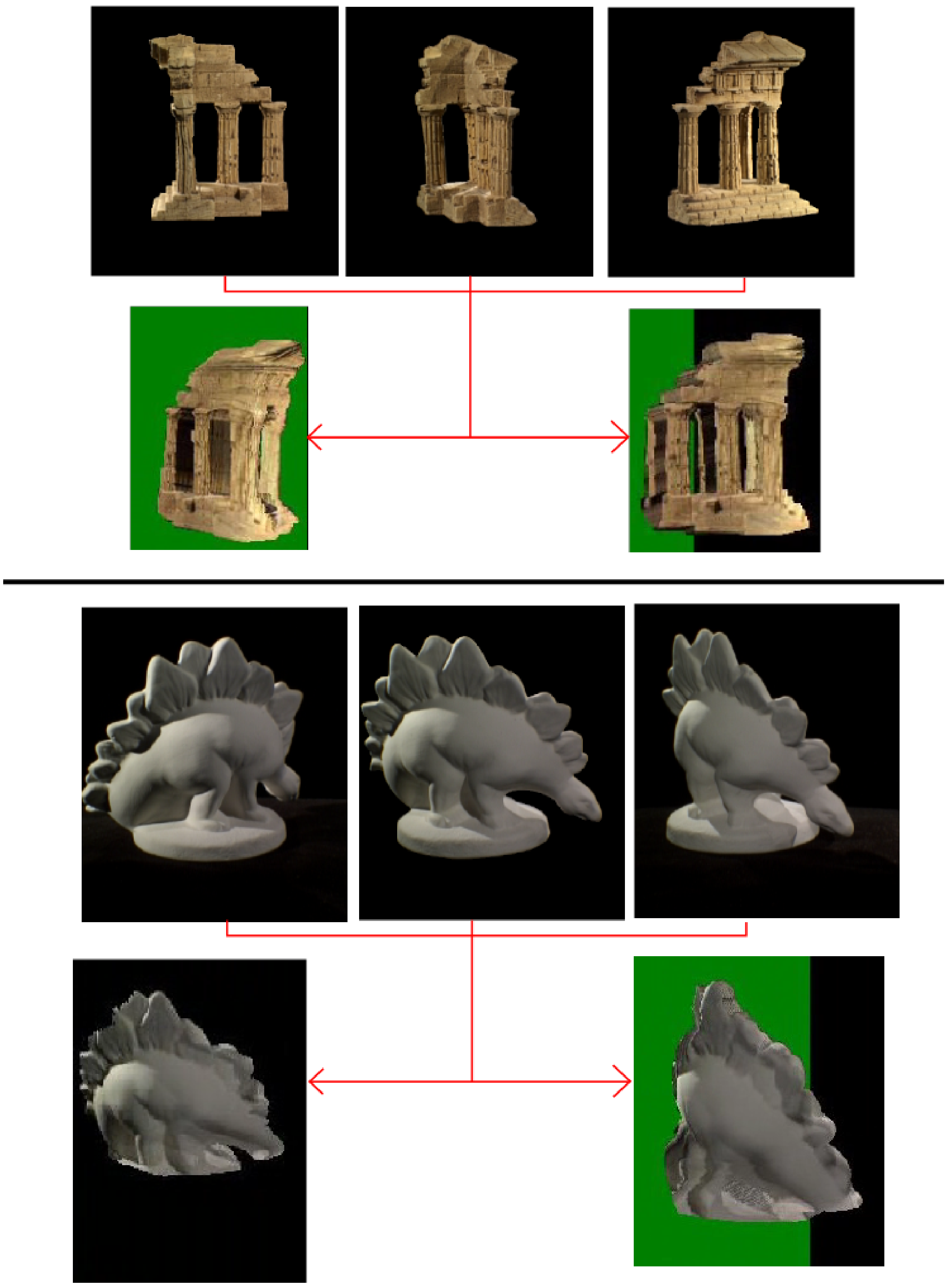

Figure 3.12: An example of the sliced visual hull on the temple and dino data set

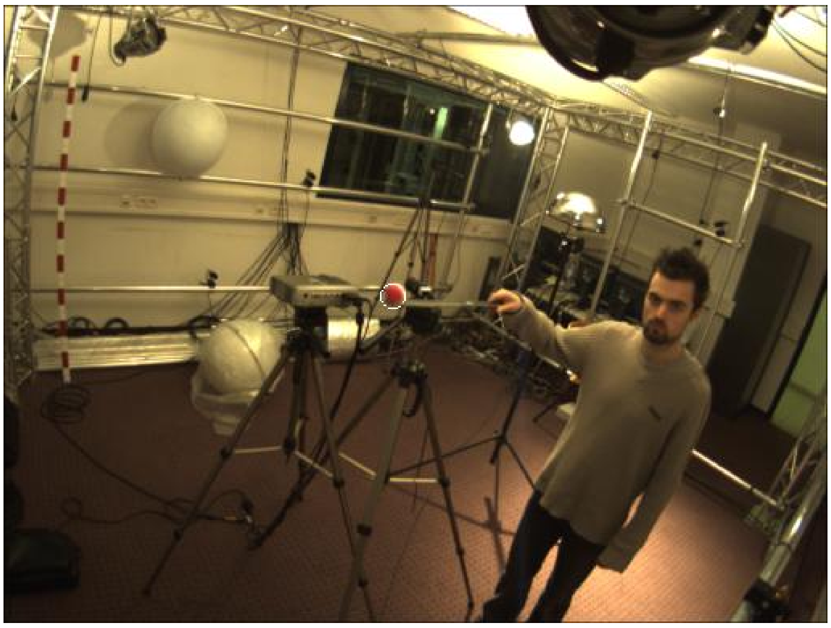

Figure 3.13: The condensation algorithm executed to track a red ball in the scene [27].

3.3 Object Tracking

3.3.1 Condensation Algorithm

The first algorithm for object tracking used in this thesis is the condensation algorithm. It is used to track the location of an object based on its color. The object used here is a red ping pong ball. Just like the algorithm described by Isard and Blake [25] this implementation is divided into three steps for every new input. These are the selection, prediction and update step and will be explained next. First an initiation step is done to start from.

Initiation Step

A particle contains a three dimensional location, a weight and a cumulative weight and can thereby be stored as 5 floating points. The cumulative weight is the sum of his weight and the weights of all other particles before him in the list. At the start of this algorithm n particles are created at random locations in the scene. The weight is the same for every particle.
The number of particles created here can affect the speed of the tracking mechanism. When the number is high, more particles must be created so the framerate will drop. On the other side, when n is chosen very low, less particles are created so the framerate will raise. Less particles makes the accuracy lower.
These particles are stored in a vector because a vector offers direct access and is easier to adjust its size.

Selection Step

In this step, new particles are created based on the location and weight of the current particles. The number of particles created does not have to be exactly n. In this implementation, the

number of particles is adjusted to the current framerate. So one of the input parameters is the desired framerate. When the calculation time for each frame is lower then this desired time, n can be raised by 5%. The same when the calculation time is too high, the amount of particles can be lowered by 5%.
Not only the new particles are created here, but their location as well. This is done based on the weights of the current particles. For each of them a random value is created between 0 and the highest cumulative value. Then by binary search the first particle is found in the current vector. The location of the new particle is set to the value of the found particle. When the weight of a particle is high, the probability is higher to choose his location again. Therefore the locations of particles with high weights are chosen more often and locations with weight 0 are not chosen again.

Prediction Step

The difference between the location in the last frame and the current frame is estimated here. The probability for the new location is a Gaussian model with the current location of the particle as origin and some standard deviation σ. This deviation is chosen based on the size of he scene and the time between frames. When the framerate is high, the object can not move too far between the frames and therefore the σ can be chosen lower.
This prediction step can be implemented by adding Gaussian random values with average 0 and standard deviation σ to the x, y and z coordinates of the particles. After this step the particles have a unique location again.

Update Step

In the last step, the weights and cumulative weights of the particles are calculated. Barrera [26] described a method to calculate these weights based on the color of the pixels on where the particle is projected on.
First the location is projected onto the input images. Then in a window of 5 by 5 around the projected point, the pixels with a similar color to the color of the object are counted. Barrera does this in the RGB color space, but because of the variable illumination in the scene Y Cb Cr is used.
Once this is done for all input images, the counted values will be multiplied with each other and assigned as the weight for the particle. This has one disadvantage when the object is not visible from all cameras because of occlusions. On the camera where the object is not visible the pixel count is 0, resulting into a 0 weight for the particle, even though the particle has the right location. Therefore the minimum count must always be 1.
The location of the particle with the highest value is chosen to be the current location of the real object. A screenshot of the result is shown in figure 3.13.

3.3.2 Tracking of Hands and Face

The next algorithm is not used to track general objects but is specialized for the face and hands of a person. It is based on the algorithm of Perales et al [14], but some steps have a different implementation. For each frame three steps are executed to obtain the locations.

- Skin-Color Pixel Detection: the skin color is used as reference for detecting the hands and face. In this stage the skin colored pixels are detected to use in the next stage.

- Data Association: the location of the limbs in image space are calculated based on the skin-colored pixels and the location from last frame.
- Three Dimensional Position Estimation: when the locations are found on the multiple images, the three dimensional locations can be estimated.

Skin-Color Pixel Detection

The algorithm here to detect skin color is a combination of two different ones described in section 2.4.7. The first is described by Perales et al [14] and requires samples of the players skin color. The second method was described by Senior et al [49] and gives a slightly worse result, but requires no input.
First, the pixels are converted to YCbCr to receive the chroma values:

$$Cb = -0.1687R - 0.3313G + 0.5B + 128 \tag{3.1}$$
$$Cr = 0.5R - 0.4187G - 0.0813B + 128 \tag{3.2}$$

Senior described the relationship between Cb, Cr and skin color based on statistical measurements. This connection is defined as an ellipse and is shown in figure 2.19. The found skin colored pixels can then be expanded by adding their direct neighbours with a small color difference.
The segments with the most skin colored pixels in the first n frames are used as input for the second algorithm. A Gaussian model is built with this input to define the skin. This means that the average and standard deviation for the Cb and Cr components is calculated. The probability of a color being a skin color is then given by:

$$p_{Color} = p_{Cb} \times p_{Cr} \tag{3.3}$$
$$p_{Cb} = \frac{1}{(Cb_\sigma * \sqrt{2\pi})e^{-\frac{(Cb-\hat{Cb})^2}{2Cb_\sigma}}} \tag{3.4}$$
$$p_{Cr} = \frac{1}{(Cr_\sigma * \sqrt{2\pi})e^{-\frac{(Cr-\hat{Cr})^2}{2Cr_\sigma}}} \tag{3.5}$$

where $\hat{Cb}$ and $\hat{Cr}$ is the average Cb and Cr of the input samples and Cb_σ and Cr_σ their standard deviation. These probabilities can be calculated for every Cb-Cr combination in a preprocessing step and put into an 256×256 matrix, which is shown in figure 3.14.
Every pixel with a probability higher then T_{max} and their neighbours with a probability higher then T_{min} is tagged as skin colored. Good values for these thresholds are $T_{min} = 0.05$ and $T_{max} = 0.15$.
Figure 3.15 shows the result of this algorithm in image C. As seen, it is better than the parameterless algorithm of Senior et al [49] (image B), but slightly worse than the parameter required method of Perales et al [14]. The reason to choose an algorithm with medium result is that it does not require to know the color of the players skin in advance which is more user friendly.

Data Association

This step starts by finding all blobs. A blob is a group of connected skin colored pixels. These are used to find or update the location of the three limbs, two hands and the face. Such a

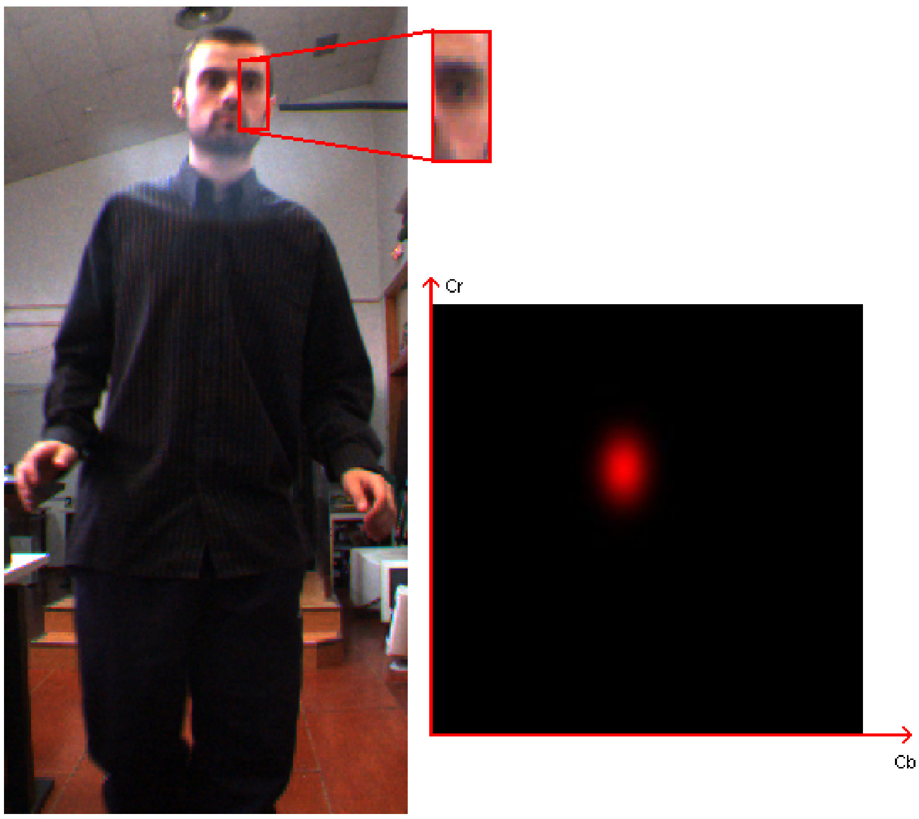

Figure 3.14: A part of the face is used as skin samples to create a Gaussian probability for skin color which is shown in the right image. The probability for every Cb-Cr combination is described. The higher the red value the higher the probability.

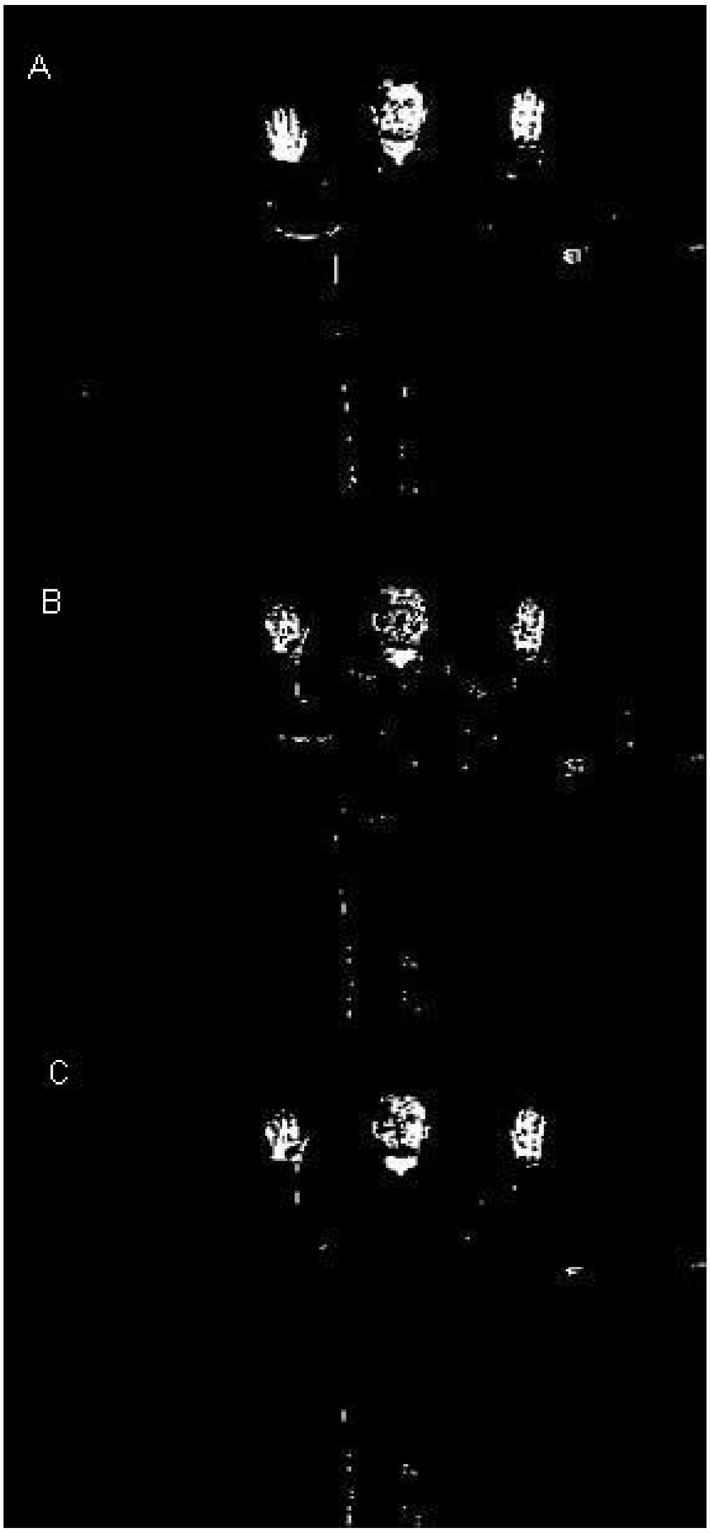

Figure 3.15: Result A shows the algorithm of Perales et al [14], B is the result of Senior et al [49]. The combination of both is shown in C.

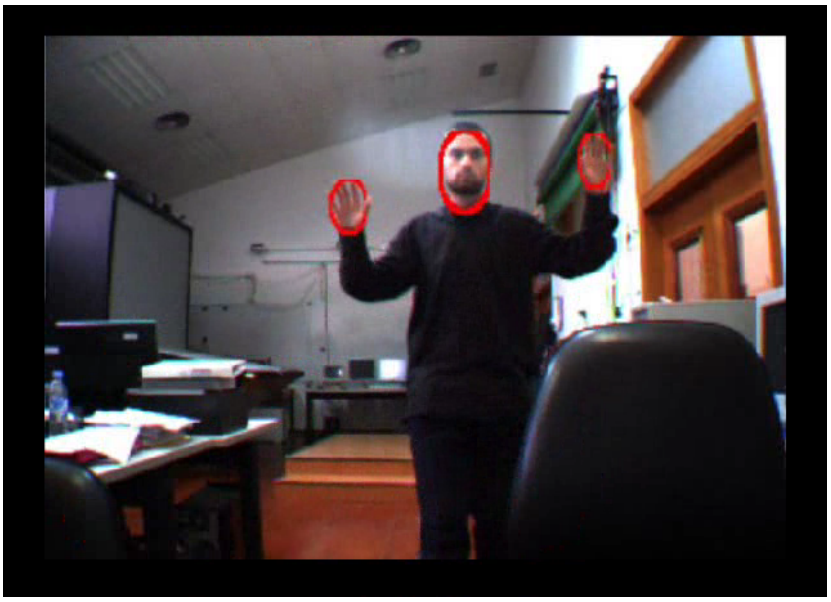

Figure 3.16: Face and hands detection algorithm

limb is defined as an ellipse with four degrees of freedom, the centre and the semi axis. In the first frame there are no limbs found yet, so the three blobs with the most skin colored pixels are used to create them.
In the following frames the current limbs are going to be updated using the new input. This is done as follows:

- Each limb is updated using its last translation, making the assumption that it is moving with a constant speed.
- The intersection between a limb and all the blobs are calculated and the limb is moved to the blob with the most pixels in common.
- If a limb has no pixel in common with a blob, it will be deleted.
- When there are less then three limbs, new ones are created using the largest blobs which are not associated with a limb yet.

3D Position Estimation

The previous steps are executed on two images and the location of each limb is found on the image planes. The three dimensional location is found by back projecting these positions and calculating the shortest line segment between the back projected lines.
Figure 3.16 shows the result of this algorithm on one image plane.

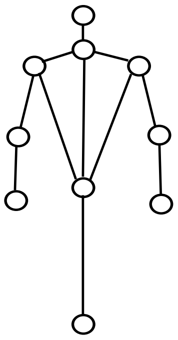

Figure 3.17: A skeleton to represent a human player, assuming that he stays at the same location.

3.3.3 Inverse Kinematics

After tracking the hands and face of the player, his pose can be estimated by using inverse kinematics [52]. An important assumption made here is that the player stays at the same place. The skeleton used to represent the human is shown in figure 3.17. The inverse kinematics algorithm is the Jacobian transpose method.

In this implementation it is only important that the location of the end effector gets to the desired location and therefore the orientation of the end effector is not used. Given a vector of joint variables q, the end effector frame is specified by a position $P(q)$. The Jacobian column entry for the i^{th} joint variable is then:

$$J_i = \begin{bmatrix} \partial P_x \\ \partial P_y \\ \partial P_z \end{bmatrix} \tag{3.6}$$

Every joint has a global transformation matrix M_i, which transforms the local joint frame to the world frame. The rotation axis of the joint is one of the principle axes u_i in the local joint frame. The axis in world coordinate frame can be found by:

$$axis_i = M_i u_i \tag{3.7}$$

Then the the Jacobian entry can be written as

$$J_i = [(p - j_i) \times axis_i]^T \tag{3.8}$$

where p denotes the position of the end effector and j_i the position of joint i in world coordinates [19].
The force F added to the end effector is the vector from its position x_c to the desired location x_d:

$$F = x_d - x_c \tag{3.9}$$

The changes of the joint variables are eventually calculated as

$$\dot{q} = J^T F \tag{3.10}$$

This update is performed in a loop until the distance between the end effector and the desired location is below a threshold. To avoid infinite loops and expensive calculations, a maximum number of updates is defined. This control loop is visualized in figure 3.18.

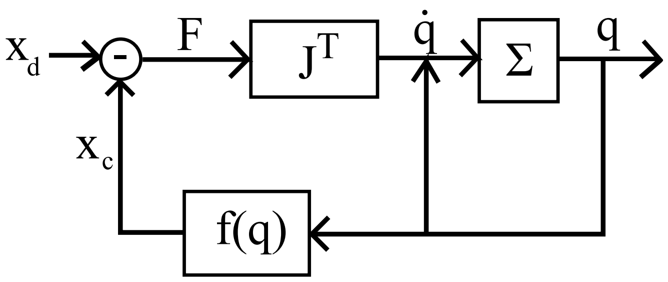

Figure 3.18: Interactive control loop model for Jacobian transpose methode [19].

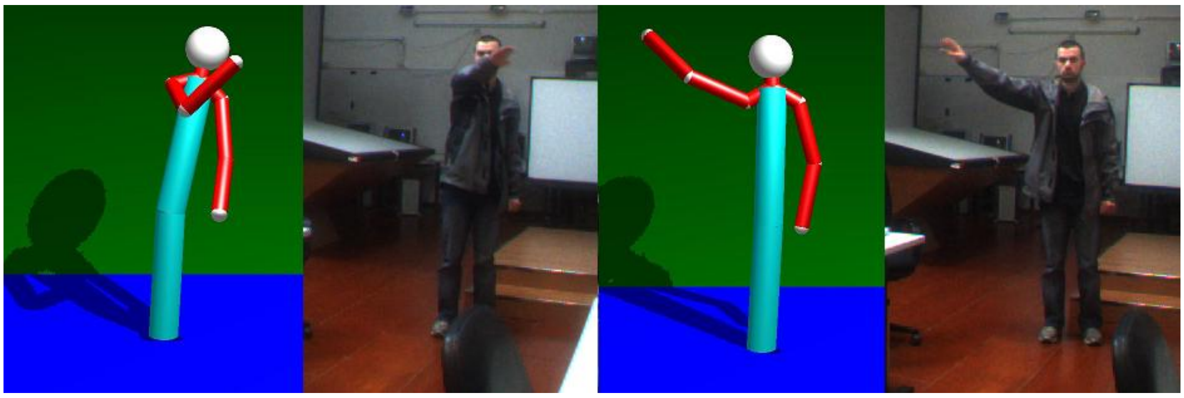

Figure 3.19: After locating the hands and the face of the player the pose is estimated using the Jacobian transpose method.

4 Results

This chapter contains some of the results of the algorithms explained in chapter 3. The advantages and disadvantages of each algorithm are explained based on the results in different situations. These situations can be the setup of the cameras, the number of cameras, their resolution, ... Therefore no unique system for interactive computer games can exists. This chapter is thereby divided into different situations, each describing the algorithms with their benefits.

4.1 Camera Setup

A first important decision the developer has to make is the setup of his cameras. The cameras can be put close together or far from each other. A camera setup with the centre of projection close to each other is called a *small baseline* setup and when the distance between the centres is high the setup is called *wide baseline*. These two are shown in figure 4.1.
Most of the algorithms make an assumption about this setup and therefore do not work good with the other possibility. For example the group of stereo vision depth estimation algorithms of section 2.2. There a small baseline setup is assumed. The principle is to find for every pixel in the first image a corresponding pixel in the second image that looks the same. This is off course only possible when the cameras are close together. With a wide baseline setup the cameras will record the scene from a total different angle so the chance that the corresponding pixels look the same is very slim.
Another example is the family of shape-from-silhouette algorithms. The silhouettes are back projected into space and the intersection is calculated. When the cameras are close to each other the back projected cones will be similar and therefore the intersection will be too large. Visual hull construction works better with a wide baseline camera setup. An example is shown at figure 4.2.
The choice of the camera setup does not directly depend on the used algorithms, but can have other reasons as well. Working with a small baseline camera setup has the advantage that a construction can be created where the two cameras are attached to. The orientation between

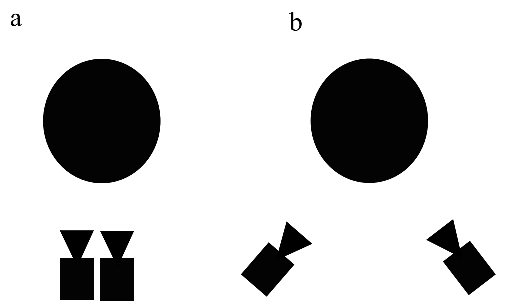

Figure 4.1: The left figure is a sketch of a small baseline camera setup while the right is a wide baseline camera setup.

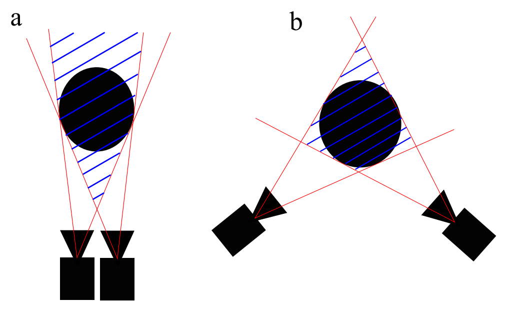

Figure 4.2: The visual hull is more tight when the cameras are further from each other then for a small baseline setup.

the cameras remains and therefore also the calibration parameters. This can make it more attractive for commercial purposes, because the cameras can be calibrated before delivering. If the game is based on a wide baseline setup the player needs to calibrate the cameras before he can start playing.

4.2 Resolution of the Images

The resolution of input images definitely has an impact on the framerate. An image of 320×240 contains four times less data then an image of 640×480. This means that it takes longer to load this image into the main memory. But the time to load the images is the same for all algorithms and are thereby not counted into the results. The calculation time of the used algorithm can also depend on the resolution.
An example is calculating a disparity map. The disparity must be calculated for every pixel and so a smaller resolution will lead to a higher framerate. This is shown in table 4.1. In the forth column the calculation time is given for the disparity calculation without optimisations,

Resolution	Windowsize	Depthrange	Normal	Optimal	Quality
384x288	7	40	6438 ms	735 ms	good
384x288	5	40	3000 ms	625 ms	good
320x240	7	40	4703 ms	485 ms	good
320x240	5	40	2078 ms	422 ms	normal
320x240	3	40	643 ms	328 ms	much noise
160x120	7	40	984 ms	109 ms	normal
160x120	5	40	456 ms	94 ms	bad
160x120	3	40	140 ms	78 ms	very bad
384x288	7	30	4937 ms	547 ms	good
384x288	5	30	2250 ms	469 ms	good
320x240	7	30	3437 ms	375 ms	good
320x240	5	30	1594 ms	297 ms	normal
320x240	3	30	469 ms	250 ms	much noise
160x120	7	30	765 ms	78 ms	normal
160x120	5	30	390 ms	78 ms	bad
160x120	3	30	109 ms	47 ms	very bad
320x240	9	30	5281 ms	422 ms	very good
320x240	11	30	8016 ms	578 ms	good
320x240	13	30	11297 ms	562 ms	good
320x240	15	30	14969 ms	765 ms	blurry
320x240	21	30	29375 ms	1656 ms	too blurry

Table 4.1: Calculation Time of the depth maps : (a) Resolution of the input images (b) Window size to calculate the SAD value (c) Depth range is the depth disparity range (d) Time to calculate the depth map without optimizations (e) Calculation time with optimizations (f) Quality of the depth map (how useful it can be). Calculated on a Pentium Mobile 1.5 GHz

the fifth column takes those improvements into account. As expected, by looking at the difference between 320×240 and 160×120, when the number of pixels is four times less, the calculation is also approximately four times less. The window size and depth range have a great impact on the calculation time as well, but their value can also be chosen according to the resolution. The disparity range is smaller in low resolution images then in high resolutions. If the maximum disparity of an 320×240 image is 40, then it is definitely not higher then 20 if the image was 160×120.
Off course, it not not necessary to calculate the depth map for the entire image. After background subtraction a lot of pixels can be eliminated, increasing the speed.

Not all of the algorithms described before that calculate a visual hull are that much dependent on the image resolution. Table 4.2 gives an overview of the calculation for a cubic voxel and octree visual hull. The first algorithm depends only on the amount of voxels the scene is divided in. This is shown in the results as depth. The width, height and depth are divided into 2^{depth}, so the number of voxels is $2^{3 \times depth}$. Because of the accuracy of the timer, the times for a lower detail than depth 5 are not shown in the table, but they are still very usefull for games.

Res	depth	preproces	octree	cubes	#voxels	#cubes
640x480	5	343 ms	343 ms	16 ms	118	157
640x480	6	344 ms	364 ms	174 ms	1252	1943
640x480	7	343 ms	531 ms	1240 ms	7927	15086
640x480	8	345 ms	1720 ms	10138 ms	29026	120749
640x480	9	343 ms	34140 ms	80719 ms	52337	too large
320x240	5	78 ms	78 ms	16 ms	84	
320x240	6	78 ms	109 ms	174 ms	967	
320x240	7	78 ms	266 ms	1268 ms	5793	
320x240	8	78 ms	14222 ms	1112 ms	12078	

Table 4.2: Time results for calculating a visual hull, calculated on a Pentium Mobile 1.5 GHz, 4 input images are used (a) resolution of input images (b) level of detail of the visual hull (c) time for the preprocessing stage of the octree (d) time to calculate the octree (e) time to calculate the cubic voxel representation (f) number of voxels the octree has (g) number of cubes the cubic voxel representation has

Res	# cameras	depth	preproces	octree	cubic
640x480	4	5	343 ms	343 ms	16 ms
640x480	4	6	344 ms	364 ms	79 ms
640x480	4	7	343 ms	531 ms	620 ms
640x480	2	5	157 ms	157 ms	15 ms
640x480	2	6	156 ms	187 ms	87 ms
640x480	2	7	156 ms	328 ms	682 ms
320x240	2	5	40 ms	40 ms	16 ms
320x240	2	6	40 ms	72 ms	170 ms
320x240	2	7	40 ms	196 ms	1245 ms

Table 4.3: Time results for calculating a visual hull (a) resolution of input images (b) number of input images (c) depth of the octree (d) time for the preprocessing stage of the octree (e) time to calculate the octree (f) time to calculate the cubic voxel visual hull

4.3 Number of Cameras

To obtain three dimensional information at least two cameras are required. For most of the algorithms in this work, two cameras suffice the need but it is extendable for more input. This can especially be demonstrated for the visual hull where more viewpoints mean a better estimate of the real object. Unfortunately using more cameras often means a lower framerate. A list of times is shown in table 4.3.

Another advantage of using more then two images is handling occlusions. Sometimes a point is not visible on one or more cameras bacause the player or another object is blocking the sight. In this situation the projection of the point is unknown. But to calculate the exact location of a point in space at least two projections of it are required. So if a point is invisible in one camera and visible in two others, the exact location can still be found. Figure 4.3 shows an example of the condenstion algorithm with three cameras and one camera is partly

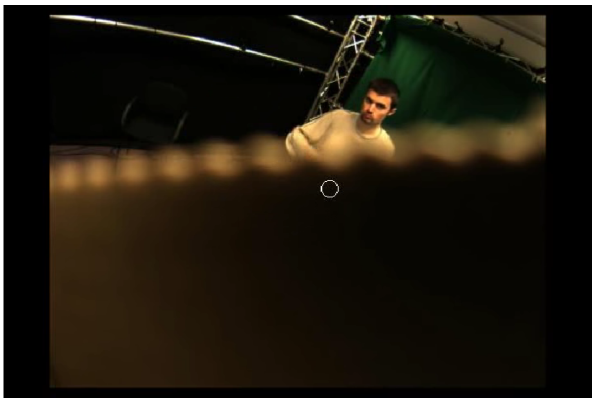

Figure 4.3: The projection of the red ball is invisible on this camera, but because it is visible at two other cameras the location can still be calculated [27].

taped. Even though the object is invisible from this camera, the location can be found with the other two cameras.

4.4 Desired Framerate

As described before in the thesis, all of the algorithms have parameters defining the accuracy of the outcome. The more accurate something is calculated, the more time it requires. For example the visual hull where it takes more time to create a detailed three dimensional model of the player compared to a low detailed visual hull. This parameter is ofcourse the number of cubic voxels or the depth of the octree. For games it is often more important to have a high framerate then to have a high accuracy, because it has to be interactive. Therefore these parameters can be chosen according to a desired framerate.
The visual hull for output created on the gpu has the number of slices as level of detail. The more slices, the more the gpu has to calculate and the lower the framerate can get.
The level of detail parameter of the depth estimation algorithm is mainly the input resolution. When the image contains four times less pixels, the calculation time will be approximatly four times less. Another parameter that can be adjusted is for example the windowsize for the pixel match cost.
The calculation time of a particle system depends on the number of particles used. Using more particles result in a higher calculation time for each frame. This is shown in table 4.4.
The algorithm for tracking hands and face has just like the depth estimation one level of detail parameter, the image resolution. This because the first step to find skin colored pixels requires to do a calculation for every pixel. A trackingtime of 50 ms is reached for images of 320×240.

4.5 Type of Game

The end discussion of the algorithms to use for the interactive videogames is offcourse the type of game. It would be useless to calculate a visual hull if you are only interested in the distance of the player to the camera. This section contains some final results of games using

cameras	particles	tracking time	correctness
2	1300	10 ms	25%
2	2400	20 ms	95%
2	3400	30 ms	98%
2	6000	50 ms	99%
2	7200	66 ms	99%
3	1000	10 ms	18%
3	1900	20 ms	98%
3	2900	30 ms	99%
3	4500	50 ms	99%
3	6000	66 ms	99%

Table 4.4: Time results for calculating a particle filter to track a pingpong ball (a) number of input images (b) number of particles (c) calculation time per frame (d) in how many frames has the algorithm found the right location

the previous algorithms. Each game makes use of a different algorithm or uses it different. Each game also has a different output to give an overview of some representation methods.

4.5.1 Drawing Game

The first game is the most simple one. It is a drawing game that makes use of depth information. First a motion detection is executed on a reference image. Each motion pixel is then tested with the depth map. If the motion is in front of a certain threshold, it will be recorded, else rejected.
The player is now standing somewhere in the room behind the threshold distance, so his motion will not be recorded. When he put his arm in front of him, his hand is in front of the threshold distance. The location of his hand is then recorded and drawn on the screen.
The output of this game is just the reference image on which the recorded pixels are drawn red. A screenshot is shown in figure 4.4.

4.5.2 Frogger

The next game makes use of depth information as well. It is a version of an old game named Frogger. First a background subtraction is executed to find the silhouette of the player. In a following step the silhouette is used to find the centre of mass, meaning the middle of the silhouette. This is done by calculating the average x and y values of all silhouette pixels. The z value is found by using the depth value of the centre of mass on the refence image. This means that only the depth information of one point is required and not the entire depthmap which results in an increase of speed.
This is a two dimensional game in which the player has to try to cross the street without getting hit by a car. The player can move to the left or right and up and down to avoid the cars. This game is shown on figure 4.5.

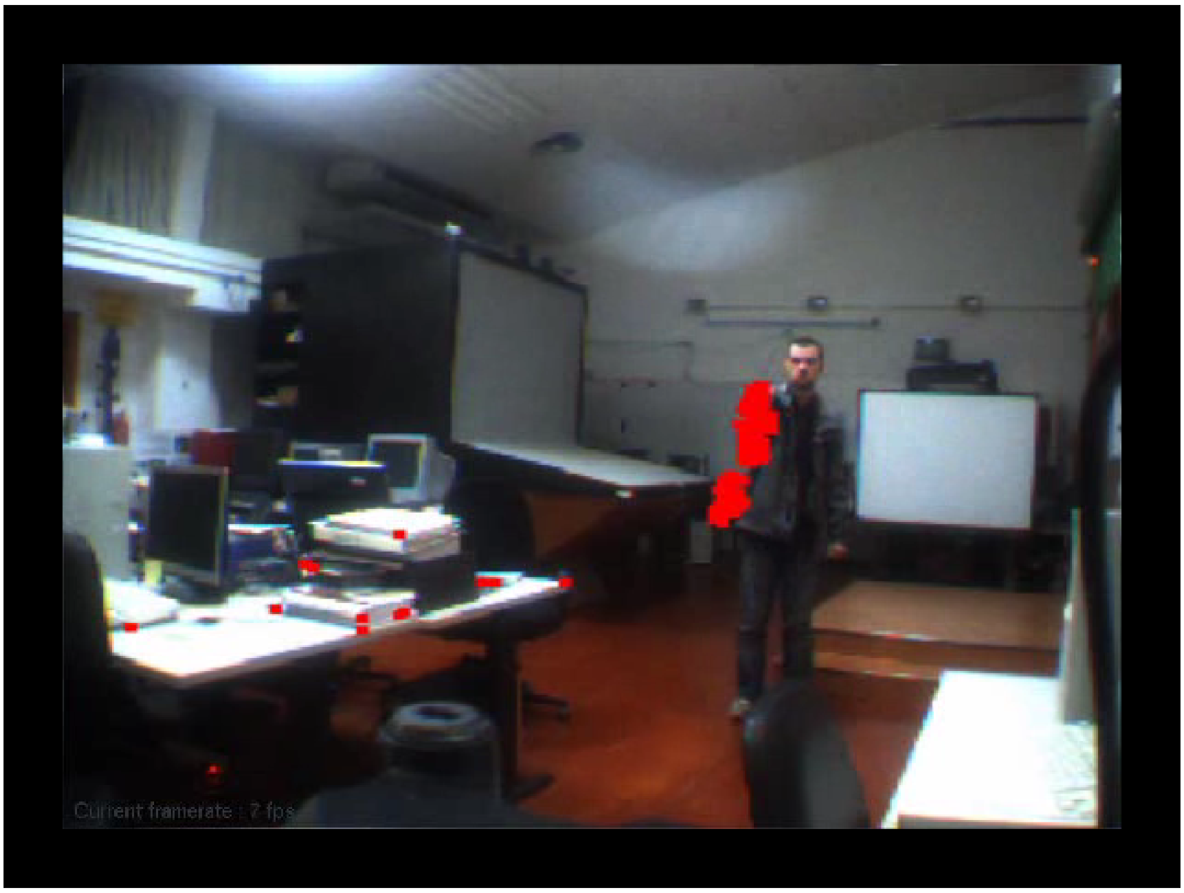

Figure 4.4: A 3D drawing game which records motion and use the depthmap to define the location of the motion in space. When the player moves further than a predefined distance from the camera, the motion is ignored. When he puts his hand in front of him, the motion is closer to the camera and the pixels are colored red. The red dots on the leftside of the image are created because of noise.

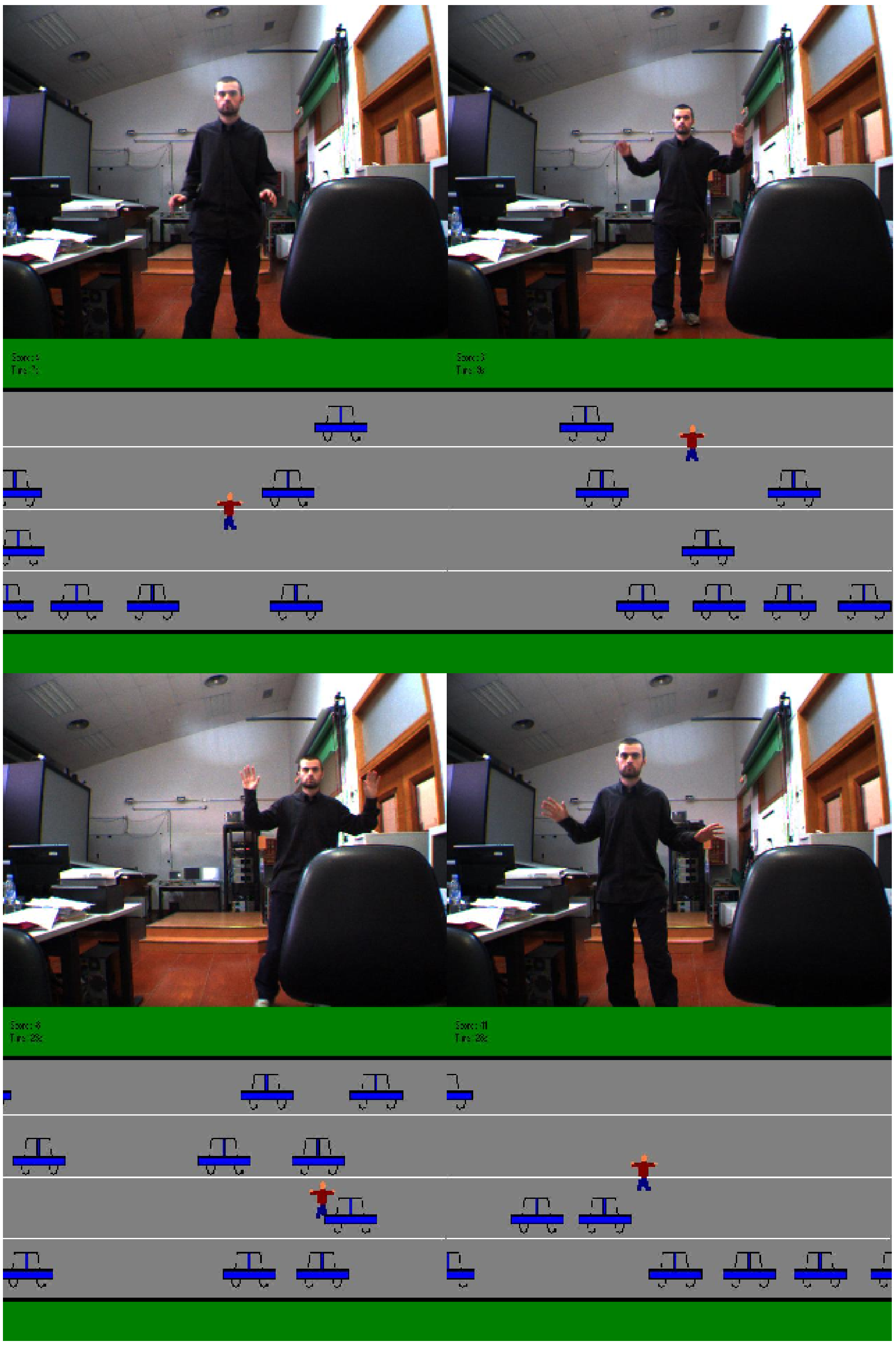

Figure 4.5: The game frogger where the location of the avatar in the game is defined by the location of the player.

4.5.3 Mapping Player to a Game Engine

The next result shows one of the usage of a visual hull. Herefore three input cameras were used to calculate a low detailed visual hull. This model can also be used to calculate the centre of mass. This calculated three dimensional point can then be used to move around an avatar in the game, but can also be used to define some animation of it. For example based on the height of the centre of mass the game can presume that the player is jumping or crowling. This is illustrated in figure 4.6 where the location of the player is mapped to a game. The game engine used here is *irrlicht* [28]. The exact purpose of this game is still undefined, but a possible goal could be to avoid getting hit by other objects in the game. Calculation time per frame is 70 ms.

4.5.4 Collision Detection Game

In the next game the visual hull of the player is used for collision detection with the virtual objects. The player is placed into a three dimensional world in together with virtual objects. To show the player a billboard or a three dimensional model can be used depending on the location or orientation of the camera. In the game shown in figure 4.7 a billboard is used. Objects are added to the game using *ode* [59] as physics engine. Shadows are added because it gives a better feedback of three dimensional information to the player [60].
The purpose of this game is to avoid getting hit by an object. Another goal could be to move the object from one place in the room to another. The calculation time for each frame is 76 ms.

4.5.5 Spiral

Spiral is an old game where the player has a metal disc which is placed around a metal pipe. The pipe creates a path which the placer must follow with the disc with touching it. The following game is a digital version of it. A particle filter is used to track a known object which the player is holding. It contains three colored pingpong balls and by tracking the three locations the position and orientation of the disc can be found. These calculations are mapped to a digital version of spiral. The result is shown in figure 4.8. Tracking time per frame is 80 ms.

4.5.6 Roboshoot

The last game is called Roboshoot and also makes use of an object tracking algorithm. The algorithm of section 3.3.2 is used to track the face and the hands of a player. In a next step, these calculated locations are used to estimate the pose of the player using invers kinematics. The invers kinematics algorithm used for this game is called the jacobian transpose method [19].
The pose of the player is rendered as some kind of robot as shown in figure 4.9. The arm is replaced by a canon that fires a cannonball each frame. The direction in which this ball is fired is defined by the vector from the elbow to the hand. The purpose of this game is to hit objects in the scene using your arms as a fireweapon. For input images of 320×240, the calculation time for each frame is 47 ms. When the images have a resolution of 640×480 the time per frame is 167 ms. Depth estimation is done with a Bumblebee 2 camera of Point Grey Research [61].

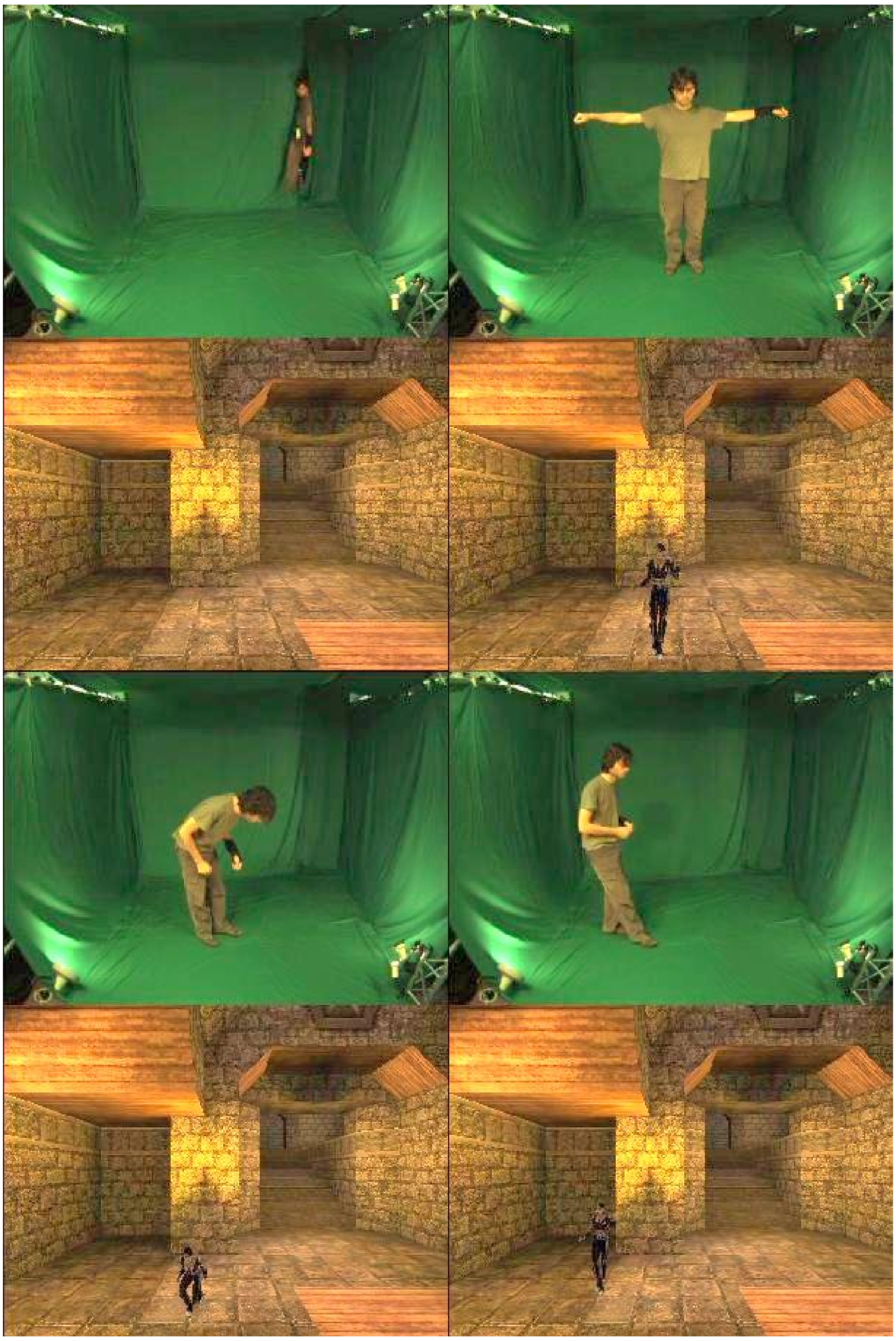

Figure 4.6: The centre of mass of the player is used to place and animate the avatar in a game. This game is made with the game engine *irrlicht* [28]

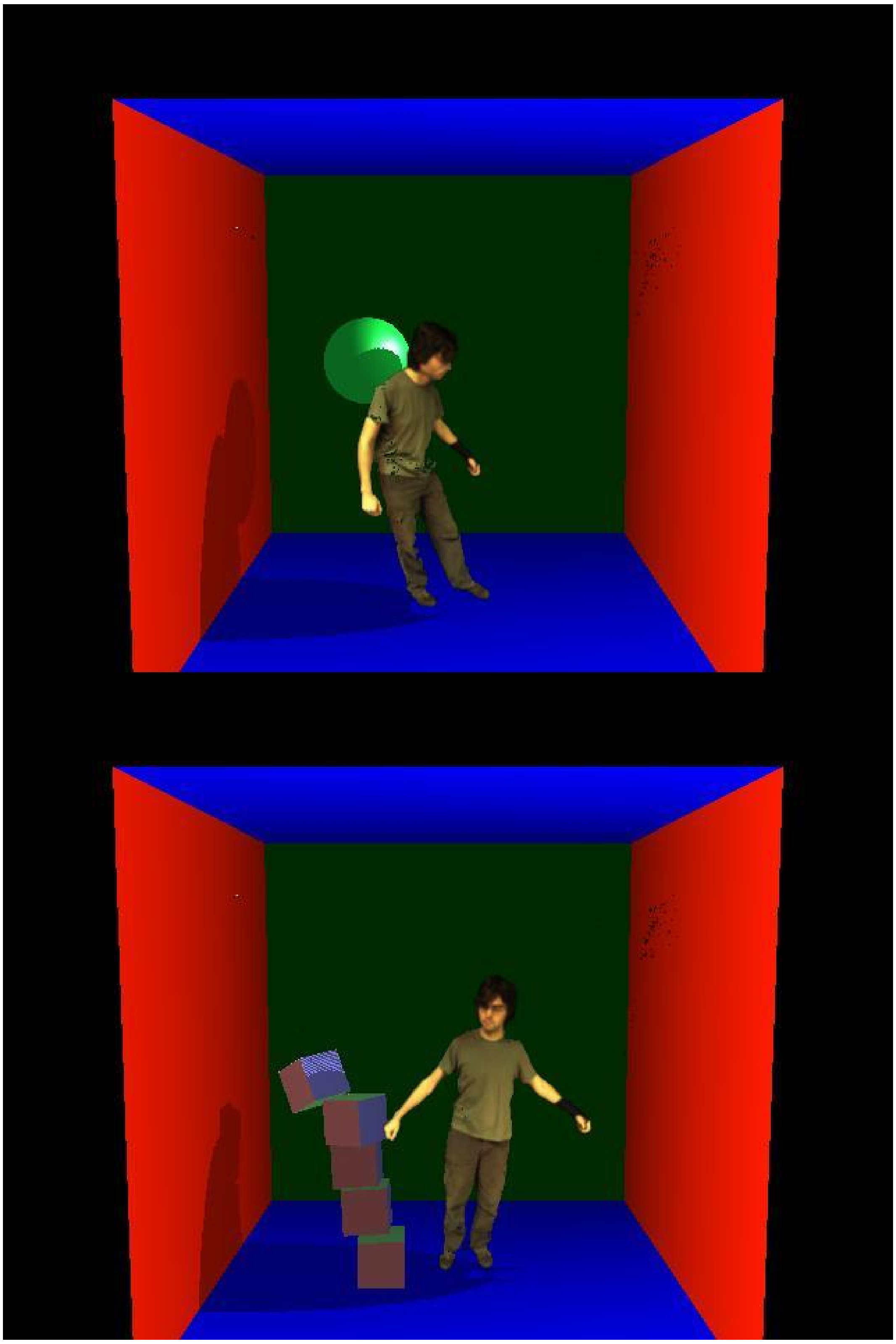

Figure 4.7: The player is placed into a virtual room where he can interact with the virtual objects.

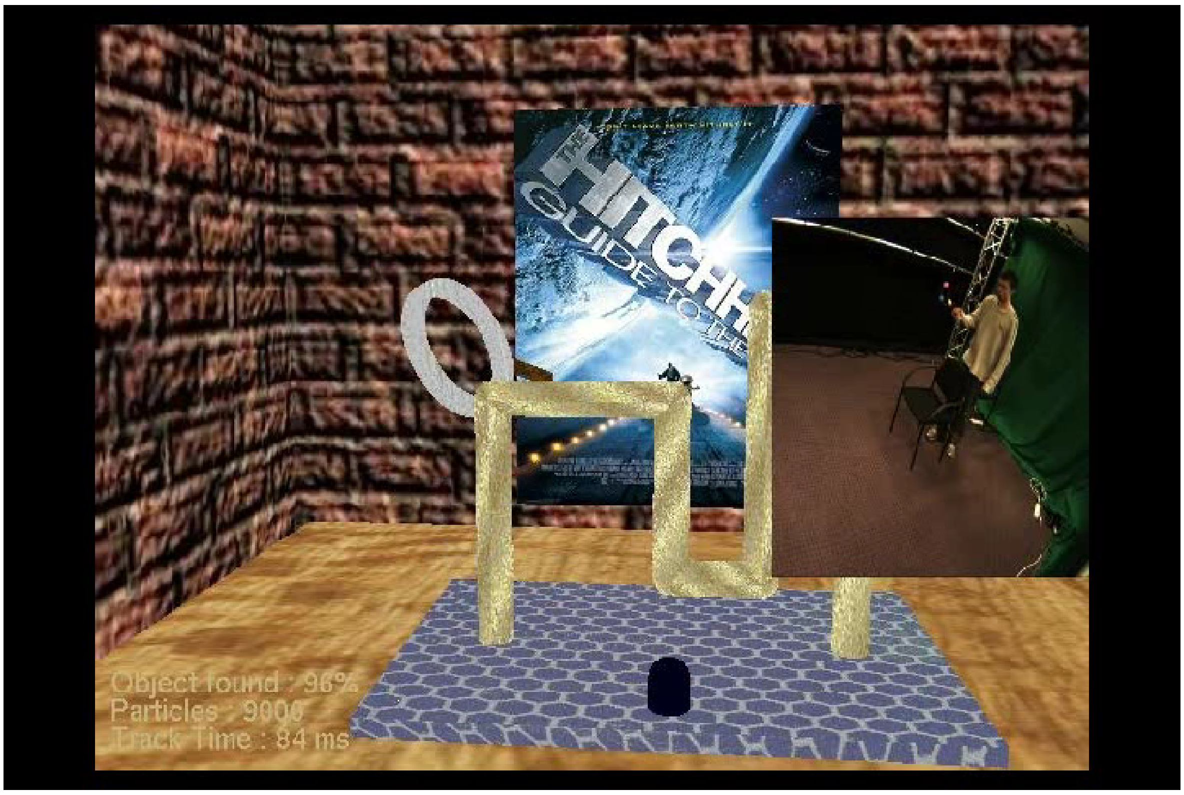

Figure 4.8: The location and orientation of the object in the players hand is tracked and mapped to the game named spiral. The purpose is to follow a path without touching the metal pipe [27].

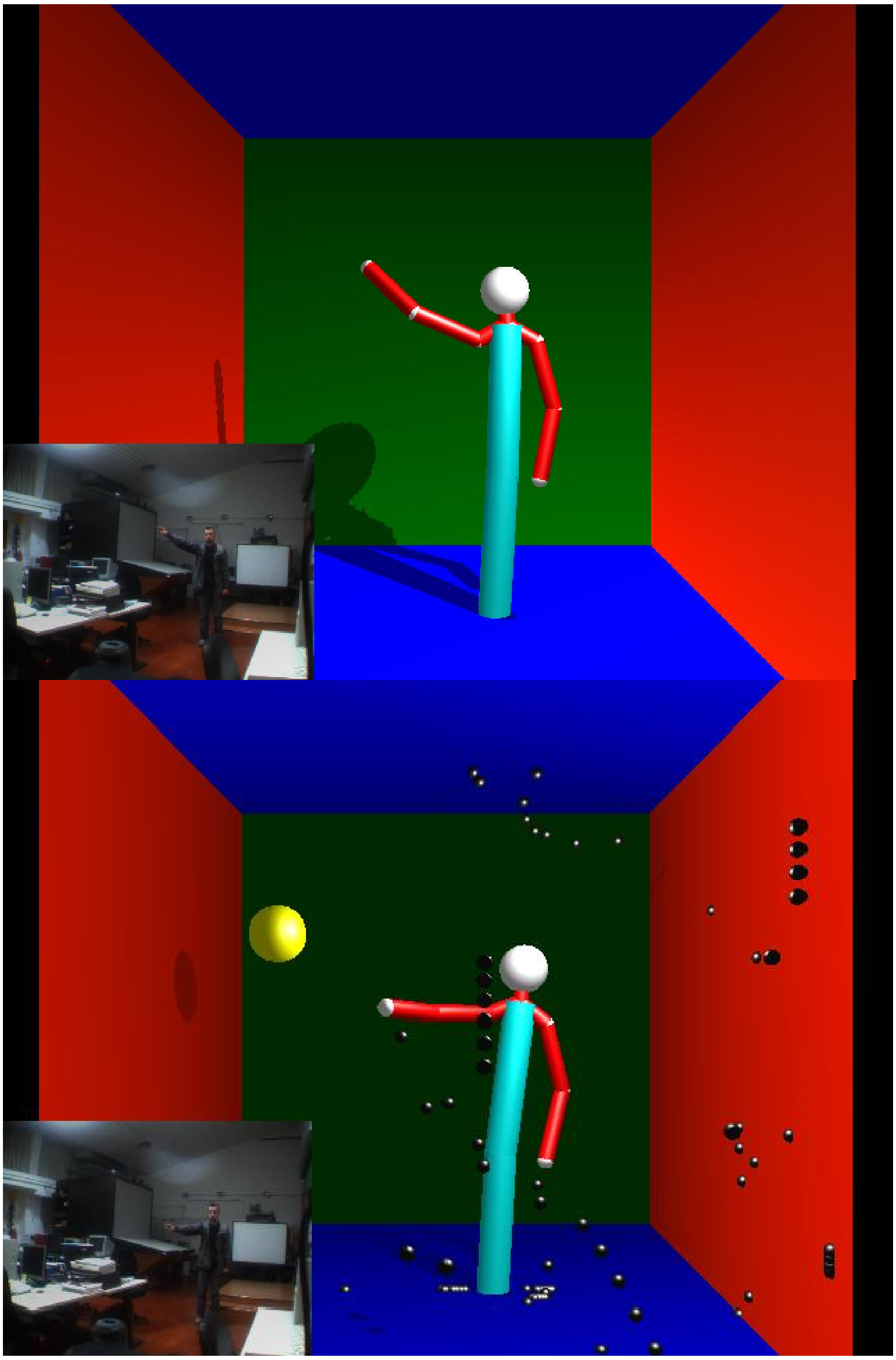

Figure 4.9: The location of the face and the hands of the player is used to control a skeleton with invers kinematics. In the lowerimage the arms of the robot are used as cannons firing a cannonball each frame with the purpose to hit all yellow balloons.

5 Conclusions

In this work, six different interactive multi-camera videogames are presented. These are videogames where two or more calibrated cameras are used as interaction device. To achieve this, depth estimation, visual hull construction and object tracking techniques were used to obtain three dimensional information from these cameras. Depth estimation and object tracking were used to calculate the location of the player or specific objects in the scene. The purpose of these locations were to place the avatar in the game at the same location as the player in reality or to estimate the players pose. Visual hull construction on the other hand provides a three dimensional model of the player allowing collision detection between the player and the virtual objects.

Because these games have to be interactive they require a high framerate. Therefore the algorithms are more focussed on speed than accuracy. The visual hull for example is only an estimated model of the player and because of the speed it is calculated with low detail. Still it has shown to be very useful to provide a realistic collision detection and response. The skeleton to represent the player is also just an estimation because only the locations of the face and the hands are known, other bodyparts are estimated. But just like the visual hull, this estimate is good enough to provide a realistic representation of the human player.

The advantage over games based on only one input camera is of course the extra depth information that can be obtained. It can make the game look more realistic, like the collision detection game of section 4.5.4, where objects bounce realisticly on a three dimensional model. When one camera is used, only two dimensional collision detection is possible. A comparison with some games of Sony Playstation Eyetoy [4] is shown in figure 5.2. The extra depth information also results in a more extended collection of possible games. Most of the games in this work can probably not be created with only one camera.

In these kind of games the player has no physical contact with the gaming device. This can result in some unrealistic situations like a player walking through a wall because there is no wall in reality. Figure 5.1 illustrates such an unintuitive scenario. But on the other hand, these games seems very intuitive. By stepping to the left, the avatar will move to the left as well. Hereby it is possible for both young children as well as adults to play these games without the requirement of knowing all the buttons on the joystick. The player can also

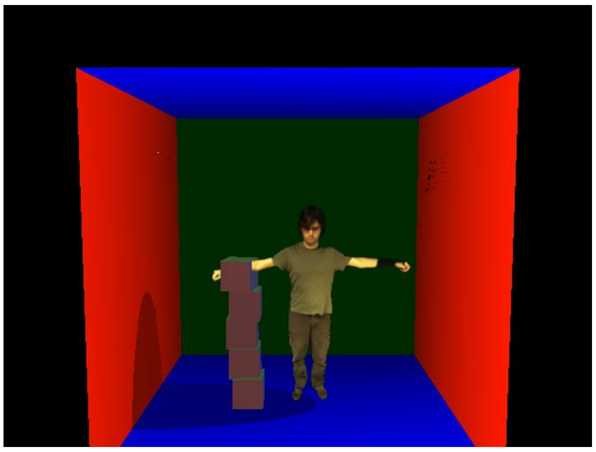

Figure 5.1: The player is pushing on top of the box. In real life he can not push any further because the boxes are placed on the floor. In the game, the player does not receive these physical feedback and can therefore put his hand inside the box.

control more degrees of freedom. An example is roboshoot where the two arms and the body can be manipulated by the player at once.
After creating these games we can conclude that the computer vision area provides an answer to create interactive games based on two or more input cameras. In the last couple of decenia, much research has been done in the area of computer vision, resulting in a large collection of techniques and algorithms. Even though computer games often were not the initial purpose of these techniques, it is shown that some of them can be very useful in this context. In this work some of them were described and later on illustrated in simple games. Of course there are a lot of different alternative real time algorithms invented which can also be used to serve the same goal. The algorithms in this work were chosen based on its speed and result. Depending on the used algorithms and the required framerate, more sophisticated games can be be developped.

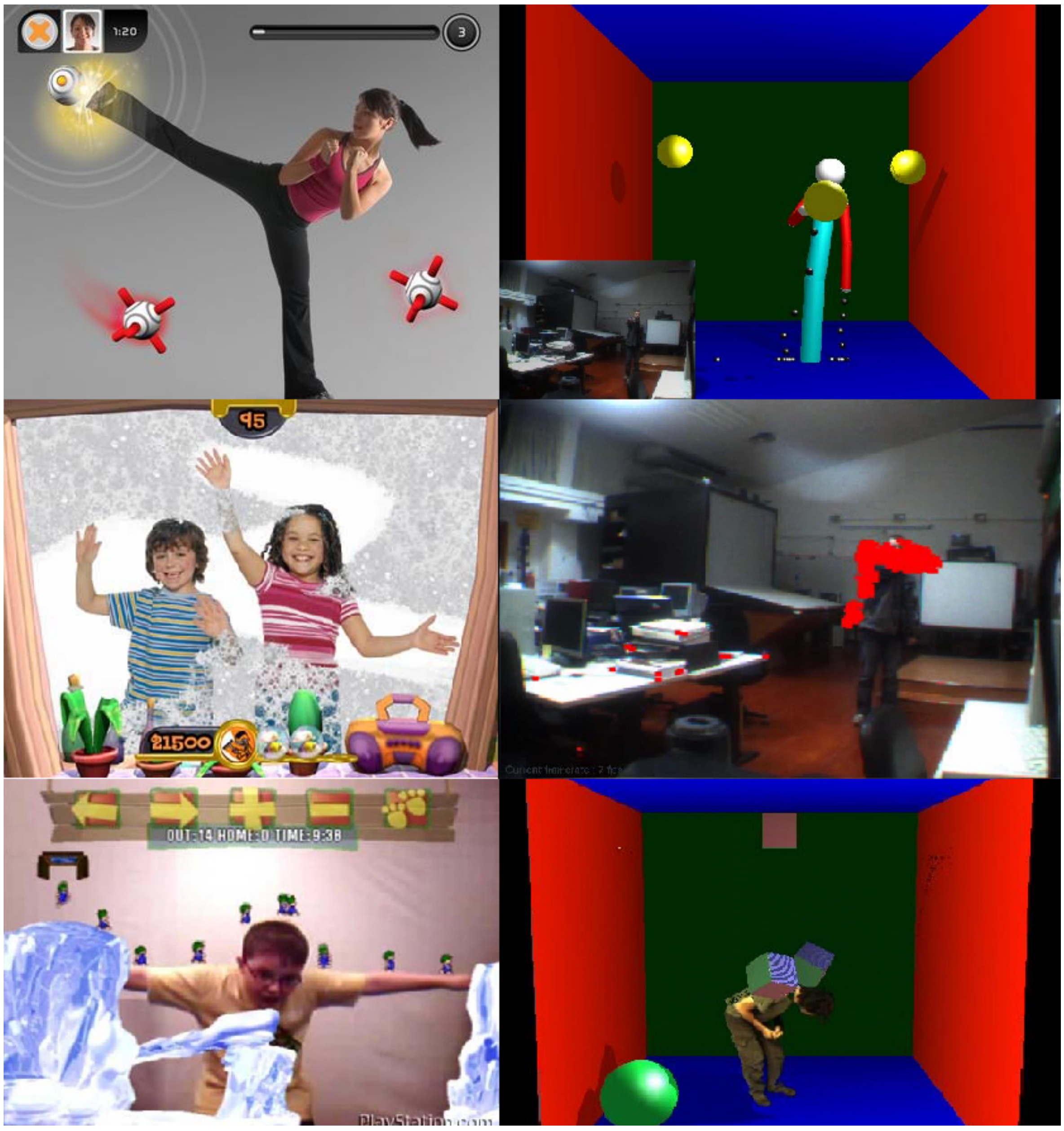

Figure 5.2: On the left games of Sony Playstation Eyetoy where the information of objects are only defined in two dimensions. On the right a couple of simular multi camera video games using three dimensional information [4]. The goal of the top games is to hit the objects which are located in two dimensional (left) or three dimensional space (right). The games on the second row show the result of motion detection. The bottom games show a collision response in the two dimensional and three dimensionl space.

6
Future Work

This work presents a complete system for multi-camera interactive videogames containing possible computer vision algorithms and interactive games. These algorithms are efficient and accurate enough to control interactive videogames. But looking into the future, some expansions can be made. These computer vision problems could be solved differently by using for example different hardware or a different approach.

For example, to obtain more efficient and accurate results, the graphical processing unit can be exploited. It is capable of processing a large amount of similar data and it seems to be very useful in the area of computer vision. Depth estimation is one of the techniques which is often implemented on graphics hardware and it has been proven to be very accurate and fast [29, 30]. Collision detection is another example of a technique that can be solved using the GPU [31, 32].

But instead of just optimizing current techniques, they can sometimes also be solved with different methods as well. De Decker et al [33] described another new technique for collision detection and collision response without the construction of a visual hull first. It is executed in imagespace and could be a nice supplement to this work.

Other possibilities to expand this work is to increase the degrees of freedom of the games. For example the human pose estimation of section 3.3.3 assumes that the player remains at the same location during the game limiting his degrees of freedom. An extension to this technique is to allow him to walk around in the scene. This could for example be done by combining some methods like first detecting the location of the player using a visual hull and then estimating the pose with object tracking and inverse kinematics.

Another assumption made in these games is that only one player is present in the scene, which can be expanded to a multiplayer game in two different ways. The first is to allow multiple players to be in the same scene meaning that they have to be tracked individually. Here arises the problem to separate the players because they are filmed in the same room with the same cameras. Another way to expand this to a multiplayer game is to track players in different rooms and communicate through the network. So the network aspect has to be investigated. The communication between the computers introduces an extra delay, therefore it is important to send only important data. An example of such a multiplayer game can be an extension of the collision game in section 4.5.4 where the players have to put as many as possible objects into the goal of the other player. This can be done by letting objects bounce on the persons.

Bibliography

[1] Richard Hartley and Andrew Zisserman. *Multiple View Geometry in Computer Vision.* Cambridge University Press, New York, NY, USA, 2003.

[2] J. Ohta W.T. Freeman, K. Tanaka and K. Kyuma. Computer vision for computer games. *http://www.merl.com/projects/cvcg/*, 1996.

[3] S. Balcisoy and D. Thalmann. Interaction between real and virtual humans in augmented reality. In *CA '97: Proceedings of the Computer Animation*, page 31, Washington, DC, USA, 1997. IEEE Computer Society.

[4] http://www.eyetoy.com. Eyetoy - sony.

[5] W. T. Freeman, D. B. Anderson, P. A. Beardsley, C. N. Dodge, M. Roth, C. D. Weissman, W. S. Yerazunis, H. Kage, K. Kyuma, Y. Miyake, and K. Tanaka. Computer vision for interactive computer graphics. *IEEE Computer Graphics and Applications*, 18(3):42–53, /1998.

[6] D. Scharstein and R. Szeliski. A taxonomy and evaluation of dense two-frame stereo correspondence algorithms. *Int. J. Comput. Vision*, 47(1-3):7–42, 2002.

[7] A. Laurentini. The visual hull concept for silhouette-based image understanding. *IEEE Trans. Pattern Anal. Mach. Intell.*, 16(2):150–162, 1994.

[8] A. Erol, G. Bebis, R. D. Boyle, and M. Nicolescu. Visual hull construction using adaptive sampling. *Proceedings of the IEEE Workshop on Applications of Computer Vision*, 2005.

[9] L. McMillan. *An Image-Based Approach to Three-Dimensional Computer Graphics.* PhD thesis, University of North Carolina, Apr 1997.

[10] Fred Nicolls ans Gerhard de Jager Phillip Milne. Visual hull surface estimation. *PRASA2004*, pages 13–18, 2004.

[11] R. Kalman. A new approach to linear filtering and prediction problems. *Transactions of the ASME–Journal of Basic Engineering*, 82(Series D):35–45, 1960.

[12] P. Li, T. Zhang, and A. Pece. Visual contour tracking based on particle filters. *Image Vision Comput.*, 21(1):111–123, 2003.

[13] Mitsubishi Electric Research Laboratories. http://www.merl.com/projects/objecttracking/.

[14] F. J. Perales J. Baudes and J. Varona. Real time segmentation and tracking of face and hands in vr applications. *Articulated mation and Deformable Objects*, pages 259–268, 2004.

[15] G. Welch and G. Bishop. An introduction to the kalman filter. Technical report, Chapel Hill, NC, USA, 1995.

[16] D. Simon. Kalman filtering. *Embedded Systems Programming*, 14:72–79, 2001.

[17] F. Nicolls B. Merven and G. de Jager. Multi-camera person tracking using an extended kalman filter. -, 2003.

[18] T. A. Williams D. R. Anderson, D. J. Sweeney. *Statestiek voor economie en bedrijfskunde.* Academic Service, Schoonhoven, 1998.

[19] C. Welman. Inverse kinematics and geometric constraints for articulated figure manipulation. Master's thesis, B.Sc. Simon Fraser University, 1989.

[20] R. Parent. *Computer animation: algorithms and techniques.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[21] K. Muhlmann, D. Maier, J. Hesser, and R. Munner. Calculating dense disparity maps from color stereo images, an efficient implementation. *Int. J. Comput. Vision*, 47(1-3):79–88, 2002.

[22] S. S. Cheung and C. Kamath. Robust techniques for background subtraction in urban traffic video. *Video Communications and Image Processing*, 2004.

[23] A. Erol, G. Bebis, R. D. Boyle, and M. Nicolescu. Visual hull construction using adaptive sampling. In *WACV-MOTION '05: Proceedings of the Seventh IEEE Workshops on Application of Computer Vision (WACV/MOTION'05) - Volume 1*, pages 234–241, Washington, DC, USA, 2005. IEEE Computer Society.

[24] K. Yerex. Real-time visual hulls. *CMPUT 605 Project Report*, 2004.

[25] M. Isard and A. Blake. Condensation – conditional density propagation for visual tracking. *International Journal of Computer Vision*, 29(1):5–28, 1998.

[26] V. Matellan P. Barrera, J.M. Canas. Multicamera 3d tracking using particle filter. *Proceedings of Int. Conf. on Multimedia, Image processing and Computer Vision (IADAT-MICV2005)*, pages 193,197, 2005.

[27] T. Cuypers. Tracken van een eenvoudig object met meerdere gecalibreerde cameras. Bachelor thesis, University Hasselt, 2006.

[28] Irrlicht. http://irrlicht.sourceforge.net/.

[29] J. Woetzel and R. Koch. Real-time multi-stereo depth estimation on gpu with approximative discontinuity. *1st European Conference on Visual Media Production (CVMP 2004), London, United Kingdom*, March 2004.

[30] R. Yang and M. Pollefeys. Multi-resolution real-time stereo on commodity graphics hardware, 2003.

[31] M. Malmsten and S. Klasen. *Practical Collision Detection on the GPU.* PhD thesis, Department of Computer Science, Lund Institute of Technology, Sweden, 2005.

[32] M. C. Lin N. K. Govindaraju, S. Redon and D. Manocha. Cullide: Interactive collision detection between complex models in large environments using graphics hardware. *ACM SIGGRAPH/Eurographics Graphics Hardware*, 2003.

[33] B. De Decker, T. Mertens, and P. Bekaert. Interactive collision detection for free-viewpoint video. *GRAPP 2007: Proceedings of the Second International Conference on Computer Graphics Theory and Applications*, 2007.

[34] W. T. Freeman. http://www.siggraph.org/ publications/newsletter/v33n4/contributions/freeman.html.

[35] M. E. Latoschik. A gesture processing framework for multimodal interaction in virtual reality. In *AFRIGRAPH '01: Proceedings of the 1st international conference on Computer graphics, virtual reality and visualisation*, pages 95–100, New York, NY, USA, 2001. ACM Press.

[36] http://merl.com/projects/cvcg. Computer vision for computer games.

[37] A. Azarbayejani T. Jebara and A. Pentland. 3d structure from 2d motion. *3D And Stereoscopic Visual Communication*, 1999.

[38] R. Szeliski and P. Golland. Stereo matching with transparency and matting. *Int. J. Comput. Vision*, 32(1):45–61, 1999.

[39] S. M. Seitz and C. R. Dyer. Photorealistic scene reconstruction by voxel coloring. *Int. J. Comput. Vision*, 35(2):151–173, 1999.

[40] S. Baker, R. Szeliski, and P. Anandan. A layered approach to stereo reconstruction. In *CVPR '98: Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, page 434, Washington, DC, USA, 1998. IEEE Computer Society.

[41] P. Fua and Y. Leclerc. Object-centered surface reconstruction: combining multi-image stereo shading. In *Image Understanding Workshop*, pages 1097–1120, 1993.

[42] I. J. Cox, S. Roy, and S. L. Hingorani. Dynamic histogram warping of image pairs for constant image brightness. In *ICIP*, pages 2366–2369, 1995.

[43] D. Scharstein. Matching images by comparing their gradient fields. In *ICPR94*, pages A:572–575, 1994.

[44] R. C. Gonzalez and Richard E. Woods. *Digital Image Processing.* Prentice-Hall, Inc, 2002.

[45] A. Zhirkov. Binary volumetric octree representation for image based rendering, 2001.

[46] M. Magnor M. Li and H. Seidel. Hardwareaccelerated visual hull reconstruction and rendering, 2003.

[47] M. Ribeiro. Kalman and extended kalman filters: Concept, derivation and properties. *Mobile Robotics Lab*, 2004.

[48] P. Shirley and R. K. Morley. *Realistic Ray Tracing.* A. K. Peters, Ltd., Natick, MA, USA, 2003.

[49] A. Senior, R. Hsu, M. Abdel Mottaleb, and A. K. Jain. Face detection in color images. *IEEE Trans. Pattern Anal. Mach. Intell.*, 24(5):696–706, 2002.

[50] J. Terrillon and S. Akamatsu. Comparative performance of different chrominance spaces for color segmentation and detection of human faces in complex scene images, 2000.

[51] J. M. Buades Rubio, F. J. Perales López, M. G. Hidalgo, and X. Varona. Upper body tracking for interactive applications. *AMDO*, 2006.

[52] A. Jaume, J. Varona, M. Gonzalez, R. Mas, and F. J. Perales. Automatic human body modeling for vision-based moion capture. *WSCG*, 2006.

[53] intel. http://download.intel.com/design/processor/datashts/31327802.pdf, 2006.

[54] N. Devillard. http://www.eso.org/projects/dfs/papers/jitter99/node27.html.

[55] S. Seitz, J. Diebel, D. Scharstein, B. Curless, and R. Szeliski. http://vision.middlebury.edu/mview/data.html.

[56] Knut-Andreas Lie. http://www.math.sintef.no/gpu/conslaws.html.

[57] S. Yazheng. Performance comparison of cpu and gpu silhouette extraction in a shadow volume algorithm, 2006, 2006.

[58] NVidia. http://developer.nvidia.com/page/cg main.html.

[59] ODE Open Dynamics Engine. http://www.ode.org/.

[60] Paul Baker. http://www.paulsprojects.net/tutorials/smt/smt.html.

[61] http://www.ptgrey.com/products/bumblebee2/bumblebee2_xb3_datasheet.pdf. Point grey research.