Advanced Real-Time Video Preprocessing on Graphics Hardware for Image-Based Rendering Systems

Steven Schuermans

2 juni 2008

Voorwoord

Ik ben momenteel laatstejaarsstudent aan de De Nayer Hogeschool [1] in Sint-Katelijne-Waver, die mij deze Meesterproef heeft toegekend. Zij maakt deel uit van de Associatie voor Wetenschap en Kunst (WENK), wat een conglomeraat is van verschillende hogescholen in België. De Nayer biedt, net zoals verscheidene andere scholen, een opleidingsprogramma voor het behalen van de graad "Master in Industrial Sciences" in een van de richtingen Bouwkunde, Scheikunde, ElektroMechanica of ICT. Ik heb gekozen voor het opleidingsprogramma van de richting voor Information and Communications Technology, kortweg ICT. Het is mijn promotor op De Nayer geweest, Herman Crauwels, die me in contact heeft gebracht met de verschillende thesisonderwerpen volgens mijn interesse, en later ook met Sammy Rogmans, die toen zelf nog laatstejaars was op ons college. Ik ben uiteindelijk ingegaan op een thesisvoorstel van Sammy en wil hem bij deze weg dan ook bedanken om mij de kans te geven zo'n fijn project tot een goed einde te brengen. Dit was natuurlijk nooit gelukt zonder zijn deskundige ervaring en bijstand. Vooral omdat hij ook erg met het onderwerp verweven is, wist hij vaak heel goed wat ik doormaakte als ik het even niet meer zag zitten, en kon hij me er altijd snel weer bovenop helpen. Natuurlijk wil ik ook Herman bedanken, die mij de richting van de thesis heeft ingestuurd en mij heeft aanbevolen bij Sammy, omdat hij op de hoogte was van mijn capaciteiten en interesses. Ook bij hem mocht ik langsgaan als er problemen waren, en hij heeft enorm geholpen bij het tot stand komen van dit werk door zich toe te spitsen op de vormelijke en taalkundige aspecten en regels. Niet in het minst zou ik ook mijn mede-student Niels Gabriels, die ook zijn eindwerk bij Imec heeft gedaan, willen bedanken, want zonder hem waren de lange dagen en de uren achter de bureau heel wat minder aangenaam geweest. En tot slot wil ik kort ook mijn ouders,broer en familie bedanken voor hun continue steun en oprechte interesse in mijn verhaal gedurende het hele jaar.

Abstract

Free Viewpoint Video (FVV) is a revolutionary technique that allows a spectator to freely choose a custom camera viewpoint. Based on information of several fixed-point cameras, the requested intermediate viewpoint is synthesized without the need of a physical camera. These calculations heavily rely on stereo matching algorithms, which extract depth information out of a pair of available input cameras. However, these algorithms cannot process the live feed instantly and therefore require the feed to be preprocessed adequately. The preprocessing mainly involves correcting barrel distortion that occurs by the use of practical camera lenses, and rectifying the stereo vision, i.e. aligning object features to the same scan line between the two images. This enables image-based rendering systems to synthesize any viewpoint in real-time from a practical stereo camera setup.

The core topic of this thesis book focuses on advanced video preprocessing, implemented on graphics hardware. We have expanded an existing imaged-based rendering framework – codenamed aLive – , that originally takes in ideal images with no regards to practical imperfections whatsoever. To correct these imperfections, the fixed-point cameras are also used in a self-calibration process, exposing the cameras' relative pose and specific camera properties. Using the obtained calibration data, an adequate amount of pincushion can be applied to the images, to restore the barrel distortion caused by practical camera lenses. The acquired geometrical information is then used to rectify the self-calibrated stereo pair. Since the rectification causes the object features to align on the same scan line of the images, the stereo matching process can efficiently compute the depth information needed to synthesize the requested viewpoint. Moreover, real-time performance is still maintained by offloading the high amount of computations from CPU to the Graphics Processing Unit (GPU).

The advanced preprocessing is implemented by developing additional modules in the existing framework that use the same environment, resulting in a stand-alone performance of 3691 fps for 450×375 image resolutions. The full functionality thereby only suffers from a minor frame rate drop of 1.32% , to a total speed of 43.9 fps.

Abstract

Free Viewpoint Video (FVV) is een revolutionaire techniek die een kijker toelaat om vrij zijn eigen camerastandpunt te kiezen. De gevraagde kijkhoek wordt berekend op basis van de informatie van een aantal vaste camera's, zonder dat daarvoor extra camera's nodig zijn. De berekeningen hiervoor zijn grotendeels gebaseerd op *stereo matching* algoritmes, die diepte informatie uit een koppel beschikbare camera's kunnen halen. Deze algoritmes kunnen echter geen rechtstreekse berekeningen op live beelden uitvoeren, daarvoor moeten de opgenomen beelden eerst voldoende bewerkt worden. Deze voorbewerkingen omvatten hoofdzakelijk het corrigeren van radiale distortie op de beelden, die ontstaat door het gebruik van echte cameralenzen, en het rectifiëren van de stereo visie, wat wil zeggen dat er moet voor gezorgd worden dat karakteristieke punten op dezelfde horizontale *scanlijn* komen te liggen. Dit zorgt ervoor dat beeldgebaseerde verwerkingssystemen vanuit een praktische stereo camera opstelling zo goed als elke kijkhoek in real-time kunnen synthetiseren.

De essentie van dit thesisboek focust op geavanceerde videovoorbewerking, geimplementeerd op de grafische kaart. Wij hebben een reeds bestaand beeldgebaseerd verwerkingssysteem – genaamd aLive– uitgebreid. Dit verwacht zich enkel aan ideale beelden, zonder rekening te houden met enige praktische onvolkomenheden. Om zulke onvolkomenheden weg te werken worden de vaste camera's ook gebruikt in een zelfkalibratieproces, waarbij de intrinsieke eigenschappen en de relatieve positie van de camera's wordt bepaald. Gebruikmakend van deze kalibratiedata, kan de benodigde hoeveelheid *pincushion* distortie op de beelden aangebracht worden, die de radiale of *barrel* distortie, veroorzaakt door praktische cameralenzen, corrigeert. De verworven geometrische informatie kan dan gebruikt worden om de rectificatie van het zelf-gekalibreerde paar camera's te bekomen. Aangezien rectificatie van de beelden ervoor zorgt dat de karakteristieke punten op dezelfde scanlijn komen te liggen, kan een stereo matching proces de diepte informatie, die nodig is om de gevraagde kijkhoek te synthetiseren, accuraat bepalen. Bovendien wordt de real-time performantie behouden door het hoge aantal berekeningen over te zetten van de CPU naar de GPU.

Het geavanceerde voorverwerkingssysteem is in het bestaande *framework* geimplementeerd door de ontwikkeling van aanvullende modules die dezelfde programmeeromgeving gebruiken, wat resulteert in een alleenstaande performantie van 3691 fps voor beelden met een resolutie van 450×375 . De volledige functionaliteit wordt hierdoor maar een klein snelheidsverlies van 1.32% toebedeeld, wat de totale snelheid herleidt tot 43.9 fps.

Contents

List of Figures

Acronyms

API Application Programming Interface

DIBR Depth Image Based Rendering

CCD Charged Coupled Device

CSS Curvature Scale Space

Cg C for graphics

CPU Central Processing Unit

EIDE Enhanced Intelligent Device Electronics

EDM Expertisecentrum voor Digitale Media

FVV Free Viewpoint Video

GPU Graphics Processing Unit

GPGPU General Purpose computing on the Graphics Processing Unit

HLSL High-Level Shading Language

ICT Informations and Communications Technology

IEEE Institute of Electrical and Electronics Engineering

IMEC Inter-university Micro Electronics Center

NES Nomadic Embedded Systems

NOVA Nederlandse Onderzoeksschool Voor Astronomie

PPM Perspective Projection Matrix

RANSAC RANdom SAmple Consensus

ROI Region of Interest

SCSI Small Computer System Interface

SUSAN Smallest Univalue Segment Assimilating Nucleus

SVD Singular Value Decomposition

WENK Hogeschool voor Wetenschap & Kunst

Nederlandse samenvatting

Free Viewpoint Video (FVV) is een techniek die het mogelijk maakt voor een digitale kijker om het camerastandpunt vrij te kiezen in een digitale video toepassing. De opnames worden gemaakt met een beperkt aantal vaste, fysische cameras, en de gevraagde camerastandpunten of kijkhoeken worden dan berekend aan de hand van die informatie, zonder dat we daarvoor nood hebben aan een extra camera. De verwerking van de binnenkomende informatie gebeurt door beeldverwerkingsalgoritmes die zich baseren op dieptebeelden. Uit een paar beelden kan namelijk diepte-informatie gehaald worden met behulp van een proces dat we stereocorresponderen noemen. Aan de hand van de dieptebeelden die hiermee bepaald worden, kan het tussenliggende camerastandpunt, dat door de kijker was aangevraagd, berekend worden.

Diepte-informatie zit in beeldenparen vervat door de verplaatsing die de objecten maken tussen de twee beelden. Voorwerpen die korter bij de camera gelegen zijn, zullen een grotere verschuiving maken dan voorwerpen die veraf gelegen zijn. Door deze verschuiving te bepalen voor elke pixel van een beeld kan er een dieptebeeld gecreëerd worden. Als dan een camerastandpunt wordt aangevraagd dat niet samenvalt met de positie van een fysische camera, kan het dieptebeeld eenvoudigweg geschaald worden om te bepalen hoeveel de voorwerpen zullen verschuiven als er vanuit de virtuele positie wordt gekeken, zodat een realistisch beeld wordt gesynthetiseerd. Mogelijke praktische toepassingen van deze techniek zijn 3D cinema zoals 'Beowulf' of vrije keuze van camerastandpunt bij het bekijken van een voetbalwedstrijd via digitale televisie.

Deze stereocorrespondentie algoritmes zijn er echter op voorzien om ideale beelden te verwerken, zonder rekening te houden met eventuele afwijkingen door gebruik van niet gemodelleerde objecten. Zo zal er hoofdzakelijk radiale beelddistortie optreden doordat er met echte lenzen wordt gewerkt die afwijken van het theoretische model. Om stereocorrespondentie mogelijk te maken, wordt ook verondersteld dat de camera's parallel uitgelijnd zijn bij het maken van de beelden. Aangezien in realiteit geen garantie kan gegeven worden dat dit altijd zo is, moeten de gefilmde beelden dus eerst nog getransformeerd worden zodat het lijkt alsof ze van een parallelle cameraopstelling afkomstig zijn. Deze operatie noemen we het rectifiëren van beelden. Deze voorverwerkingen vormen het hoofdonderwerp van dit werk en zijn essentieel voor het ontwikkelen van een reële FVV applicatie, aangezien stereocorresponderen op echte beelden niet mogelijk is zonder de nodige voorverwerking.

Om zulke afwijkingen van de modellen te kunnen corrigeren, hebben we nood aan de eigenschappen van de camera's en hun relatieve posities. Deze eigenschappen worden de intrinsieke en de extrinsieke parameters van de camera's genoemd en kunnen gegroe-

peerd worden in een matrix, de perspectief projectiematrix (PPM). De parameters kunnen bepaald worden door middel van camerakalibratie, waarbij enkel aan de hand een paar beelden, deze parameters berekend worden. Wij hebben geopteerd voor fotometrische zelf-kalibratie, waarbij met een makkelijk herkenbaar object in de beelden wordt gewerkt, om ze eenvoudiger met elkaar te kunnen linken.

De eerste stap in het bepalen van de perspectief projectiematrix en dus ook de camera parameters, is het detecteren van karakteristieke punten in de verschillende beelden. Omdat we niet elk beeld pixel per pixel kunnen gaan vergelijken om eventuele overeenkomstigheden te vinden, wordt er gewerkt met karakteristieke punten, bijvoorbeeld hoeken, die gemakkelijk herkenbaar zijn in verscheidene beelden. Voor de detectie hiervan maken we gebruik van karakteristieke puntsdetectie of ook wel hoekdetectie. Wij hebben geopteerd voor de Harris hoekdetector, die aan de ene kant heel wat rekenkracht vraagt, maar aan de andere kant ook heel robuust en nauwkeurig blijkt te zijn. Deze hoekdetector maakt gebruik van metingen van de intensiteitsvariatie in de buurt van een punt en definieert karakteristieke punten dus als plaatsen met een grote variatie in de intensiteit. Om de vereiste rekenkracht op te vangen die bij zulke beeldverwerkingstoepassingen hoort, wordt er gewerkt met de Graphical Processing Unit (GPU) oftewel de grafische kaart.

De grafische kaart was een van de eerste hardware onderdelen die een groot aantal betaalbare parallelle processoren in huis had, en is wegens zijn grote mogelijkheid tot parallelisatie een zeer nuttige bron van rekenkracht voor beeldverwerking. In vele gevallen is het voor een beeldverwerkingsapplicatie noodzakelijk om operaties op iedere pixel apart uit te voeren, zodat een parallelle verwerking enorm veel snelheidswinst kan opleveren. De GPU is eigenlijk een pijplijn met verschillende tussenstations, waar dan op elk tussenstation het werk in parallel wordt uitgevoerd. Naargelang het type GPU dat wordt gebruikt zijn twee of meer van zulke tussenstations programmeerbaar, wat wil zeggen dat je je eigen code kan laten uitvoeren op de grafische kaart. Wij hebben gewerkt met een grafische kaart (Nvidia GeForce 7900 GTX) die twee programmeerbare tussenstations bezit, namelijk de *vertex* processor en de *fragment* processor. Door gebruik te maken van de Direct3D programmeerbare interface van Microsoft, kunnen we onze stukken code omzetten naar laag-niveau assembly-code die dan in de GPU kan worden uitgevoerd.

Omdat niet elk stuk code geschikt is voor de parallelle structuur van de GPU, wordt er ook steeds meer gezocht naar manieren om sequentiële code om te zetten naar parallelle code om ze zo toch te kunnen inladen in de grafische kaart. We noemen dit principe General Purpose computations on the GPU (GPGPU) en dit is een belangrijk onderzoeksgebied omdat grote snelheidswinsten kunnen geboekt worden als bepaalde algoritmes van het manusje-van-alles, de CPU, naar de gespecializeerde GPU kunnen worden overgedragen. Zo hebben wij ook de Harris hoekdetector volledig op de GPU geprogrammeerd, hoewel lang niet alle berekeningen 'echte' grafische berekeningen zijn zoals het opslaan van een berekende correlatie waarde in een textuur. De verdere camerakalibratie hebben we echter niet geimplementeerd aangezien dit teveel onwikkeltijd zou innemen en dit voor onze vaste cameraopstelling niet strikt noodzakelijk was. Ons hoofddoel ligt bij de voorverwerking van echte beelden.

Nadat de cameraparameters zijn bepaald, kunnen we ons toespitsen op de eerste voorverwerkingsstap, de correctie van radiale distortie (ook 'barrel' distortie). Dit fenomeen is makkelijk zichtbaar in bijvoorbeeld foto's van groothoeklenzen, waarbij het beeld meer

en meer vervormt naarmate de afstand tot het centrum van de distortie groter wordt, vandaar dat het radiale distortie heet. Maar ook bij gewone cameralenzen is dit effect al zo goed zichtbaar dat we in vele gevallen niet verder kunnen zonder het weg te werken. Die correctie kan gezien worden als de inverse operatie toepassen op het vervormde beeld waardoor er een gecorrigeerd beeld wordt gecreëerd. Wij hebben deze correctie geïmplementeerd met de architectuur van de grafische kaart in ons achterhoofd. Daarom berekenen we niet de transformatie van het vervormde naar het ideale beeld, maar net andersom. Omdat in de GPU altijd eerst een textuur moet geladen worden om kleur te geven aan de beelden, berekenen wij een omzetting van het niet-gekleurde, ideale beeld naar het vervormde en gekleurde beeld. Door de ideale pixel de kleur te geven die bij zijn vervormde waarde hoort, wordt uiteindelijk dezelfde correctie doorgevoerd.

De laatste stap voordat de verkregen beelden kunnen gebruikt worden voor stereocorrespondentie, is rectificatie van de beelden. De bedoeling is om het zo te laten lijken alsof de beelden opgenomen zijn met twee parallel uitgelijnde camera's, want dit zorgt ervoor dat alle overeenkomstige karakteristieke punten in de twee beelden op gelijke hoogte komen te liggen zodat het zoeken naar corresponderende punten in de stereocorrespondentie algoritmes wordt beperkt tot het zoeken op een lijn in plaats van het hele beeld. De transformatie wordt gerealiseerd door de berekende perspectief projectiematrix van elke camera, om te zetten naar een nieuwe PPM die de camerapositie voorstelt in een parallelle opstelling. Dit betekent dat, eens de nieuwe PPM's voor beide camera's bekend zijn, dat we enkel nog de transformatie moeten zoeken die pixels van op het cameravlak dat bij de oude PPM hoort, afbeeldt op het nieuwe cameravlak dat bij de nieuwe PPM hoort. Dit is een per-pixel transformatie die in een matrix kan gebundeld worden, zodat deze omzetting maar een matrixvermenigvuldiging vereist, eens de nieuwe PPM's gekend zijn. Alle andere beelden die door dezelfde camera's worden gemaakt kunnen nu met diezelfde transformatie worden omgezet, aangezien we veronderstellen dat de camera's vast staan. Dit wil zeggen dat de berekening van een nieuw paar PPM's slechts eenmalig is als het om een gefixeerde cameraopstelling gaat. Eens de omzetting gedaan is, zijn de beelden nu de best mogelijke benadering van de gevraagde ideale beelden en kan er aan de stereocorrespondentie begonnen worden.

Aangezien wij enkel de voorverwerking van de camerabeelden uitvoeren, en niet de achterliggende correspondentie, moeten deze operaties zo snel en efficiënt mogelijk verlopen zodat de eindtoepassing er zo weinig mogelijk hinder van ondervindt. Onze ontwikkelingen hebben geleid tot een applicatie die de beelden voorverwerkt aan een snelheid van 3691 beelden per seconde (fps). Dit zorgt slechts voor een snelheidsdaling in de eindapplicatie van zo'n 1.32%, wat de eindsnelheid herleidt tot 43.9 fps, nog steeds ruim voldoende voor een live video verwerking, die minimum 25 fps moet bedragen.

Chapter 1

Introduction

3D is a booming business in a very broad range of sectors, from the industry over the medical sector to the entertainment sector, everybody is talking about 3D nowadays, and that is also one of the motivations of this work, trying to set the pace in a yet relatively unexplored domain. Right now, the possibilities are unlimited, numerous new applications, goods, merchandizing stuff and many more see the light of day. Thanks to strong evolution in silicon technology, the average Joe can take part in this as well, because the desktop PCs we buy for gaming or other joyful purposes contain graphics cards, that are the supercomputers of the future. In this work we want to introduce the reader to this new world and try to make a difference ourselves, developing a customizable viewpoint application on commodity hardware.

1.1 Background information

1.1.1 The related company

The "Inter-university Micro Electronics Center", IMEC abbreviated [2], has been my home base for the past six months to develop a Master's thesis. Here, in Leuven, resides the largest independent European research center in the field of microelectronics, nanotechnology, design methods and technologies for ICT systems. IMEC's research bridges the gap between fundamental research at universities and technology development in industry.

Figure 1.1: The new IMEC corporate logo

Since its foundation in 1984, IMEC has evolved into a corporation that employs an international group of over 1500 people, mainly researchers, PhD students, industrial

residents and interns. Because of its physical location close to the univerity campus of the Catholic University of Leuven, it offers a lot of opportunities and serious motivation for several academic essays. This wide variety of people contributes to IMEC's main mission statement: "*To perform Research and Design, ahead of industrial needs by 3 to 10 years.*"

In the meantime, IMEC can count itself among the top research centers in the world concerning semiconductor production and it maintains healthy relationships with big international corporations such as Intel, Samsung, Texas Instruments and TSMC. This global network of clientele results in several headquarters located abroad, with the recent opening of an IMEC representative office in Taiwan as the latest newcomer.

1.1.2 Personal motivation

I have always had a fascination for 3D and cinematography in general, so when this thesis topic came along, combining both my interest in three dimensional media and the ICT sector, I felt suited for the job. As a Master's thesis researcher I have been brough under the Nomadic Embedded Systems (NES) division, in the MultiMedia (MM) group, to be more precisely. Being part of such an internationally recognized environment did also contribute a great deal to my personal motivation to produce a high standard Master's thesis.

My predecessor's (Mr. Sammy Rogmans) Master's thesis was a big inspiration to me as well, since he proved that since the Bologna declaration of June 19th, 1999, college students could, just as university students, get their hands on a project of academic level and quality. Knowing that he would be my corporate promotor gave me faith that I too, with his and his colleagues help, would be able to accomplish the task that was set.

1.1.3 Starting point

I was given a slight head start in my Master's thesis as I would not be working on an entirely new project, but I would be delivering some of the necessary pieces to complete a project started in 2006 (see chapter 1.2). I was handed over a framework that had been build the past year and I was expected to expand this existing framework while also creating some new elements from ground zero. I believe that this reuse of intellectual property came in real handy, because it took me immediately one step in the right direction. I could use it as a stepping stone, something to rely on.

1.2 Thesis objectives

1.2.1 Free Viewpoint Video

This Master's thesis classifies under the umbrella concept of Free Viewpoint Video (FVV). The global approach of this concept is to allow any live stream spectator to customize its viewpoint according to his desire. For example, FVV will make it possible for a digital

television customer to choose the angle of incidence at which he wants to witness e.g. a game of soccer.

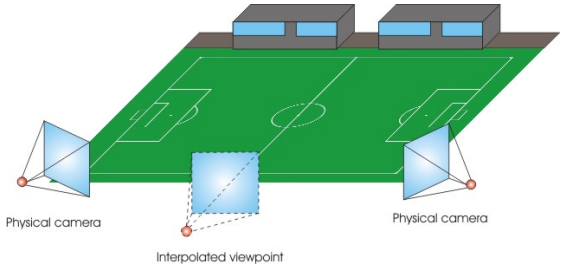

Figure 1.2: The interpolated view sits mid-court

With the information that is captured from a couple of physical cameras on the pitch, for instance one in each corner, a virtual intermediate view will be constructed to allow the spectator to experience the game as if he is sitting mid-court. This effect can be applied on a live stream without the need of an extra physical courtside camera. Considering that every digital spectator might request a different angle of incidence, it is essential that we can create these intermediate viewpoints without additional cameras. There are several techniques and possibilities to create this custom intermediate viewpoint, but the technique we use is known under the name of *Depth Image Based Rendering* (DIBR) and has been the subject of the Master's thesis of Rogmans [3]. In this technique, depth information is used together with available imagery to obtain the requested viewpoint.

However, these algorithms cannot process the live feed instantly and therefore require the feed to be preprocessed adequately. This involves correcting barrel distortion that occurs by the use of practical camera lenses, and rectifying the stereo vision, i.e. aligning object features to the same scan line between the two images. That is why image preprocessing is one of the key objectives of this thesis.

1.2.2 High speed adaptive preprocessing system

The preprocessing system that we want to construct has to be able to fit into a live streaming application, so it will have to exceed the limitations of live video feed as a minimum requirement. Since we can only speak of moving images or video applications when the achieved frame rate tops 25 frames per second, also referred to as *genuine real-time*, we could demand nothing less of our preprocessing system. As the necessary preprocessing steps may vary depending on the quality and specifications of the camera itself, the need arose for a flexible system that would easily adapt to the needs of the user. For instance, not every camera shoots with serious lens distortion and most camera setups will not need constant recalibration, so depending on the situation, certain steps can and will be skipped.

1.2.3 Contribution to GPU parallelization

With the rise of the Graphics Processing Units (GPU), the first inexpensive, massive data-parallel processors became accessible for researchers, which caused them to change their

focus from working with the Central Processing Unit (CPU) as main supplier of arithmetic horsepower to the more agile and higher capability GPU. Our work can contribute to help create and stimulate this change in perspective. The reason is simple, because Lu and Rogmans' interpolating algorithm [3] had already been implemented on the GPU, we would build on that experience and try to implement the full application on the GPU, thereby offloading the end user's CPU, leaving it open for tasks not suited for the GPU and other user applications as the Internet, chatting, playing music and many more. This offloading to the GPU can lead to significant speedups thanks to its specialized architecture.

Obviously not every task is suited for the GPU, but as it happens, a lot of image processing tasks are well-suited and yet more and more conventional CPU tasks can be *transformed* to fit the needs of a good GPU task. In short, the GPU needs parallelizable tasks since it is built out of multiple processing units, all running concurrently. It is not so hard to understand that a lot of image processing tasks get excellent results on the GPU because they are pixel based and thereby can be easily parallelized. But more challenging for future scientific research is the transformation of sequential CPU to parallel GPU tasks. This book is not a guide to transforming every possible task to a task that suits the GPU, but we do believe that every step, even the little ones, towards greater GPU utilization and exploitation, will help others to overcome this challenge.

To illustrate the importance of parallelization, an off-topic example is presented here. Just recently, scientists of the *Nederlandse Onderzoekschool voor Astronomie* (NOVA) were able to realise a major breakthrough in their research thanks to NVIDIA's GeForce 8800 GTX graphics card. As they were trying to reproduce astronomical fenomena on a PC, they discovered that they were able to speed up these intense calculations by a factor twenty, using the computers graphics card instead of the usual CPU. Thanks to this discovery "The simulation of star clusters and the solar system suddenly becomes a piece of cake" [4].

1.2.4 Satisfying the 3D hunger

The hunger for 3D information in the present is greater than ever, there are already numerous applications and projects that would not have seen the light of day if it had not been for their use three-dimensional information. We make a random pick, out of the various possibilities to illustrate what an impact 3D can already have on your every day life:

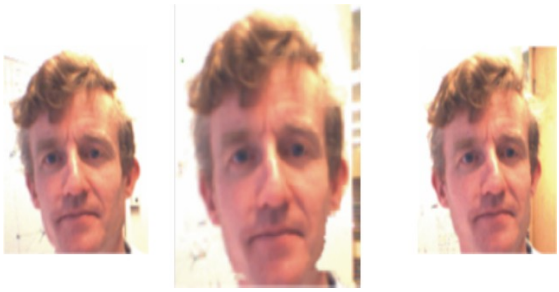

Figure 1.3: Center image with corrected gaze

- **Eye-gaze corrected video chatting**: With the help of a multiple (web)camera setup around your desktop screen, your daily web-based video chatting experience will get a serious improvement. Since you cannot look at your screen and look into the lens of your webcam simultaneously, no real eye contact is present and your communication does not feel as real as it should [5]. Thanks to the information of mutilple cameras, an intermediate view can be constructed to make it look on the other side of the communications line as if you were looking straight into the camera. This makes your web based contact with other people much more personal and more resembling a live conversation.

Figure 1.4: Examples of autostereoscopic displays

- **Autostereoscopic displays**: To create a more realistic view of objects on your flat and inherently 2D screen, depth information is inevitable when we are talking 3D. Thanks to this information it will be possible to let your specialized screen, equipped with what is called a 'lenticular sheet', emit light rays that differ depending on your point of view, thus creating the posibility to virtually look around an entire object. More precisely this means, that, instead of having your PC rotate the object to be able to see its other side, it would suffise to move your head a little to see sides of the object you could not see before.
- **Future medical and AI applications**: Even nowadays, in the sectors of medics and Artificial Intelligence, a lot of applications, integrating 3D technology, already exist. For medical purposes, it might come in handy for the surgeon to get a better, in-depth view of the human body when performing an endoscopy. Using one pair of cameras the effect of the human eye can be reproduced where two images are merged into one image with depth vision.

  The same principle might easily be applied to the robots equipped with AI. Because, how can you make a robot better resemble a human being, if you can give him the same pair of eyes to extract information from its surroundings? Again, thanks to rapidly improving 3D algorithms, nothing stands in the way of developing a more human, more realistic approach of an AI robot.
- **3D cinematography**: Perhaps not many of you have experienced the effects described above in real life, but this is a more contemporary example. In the entertainment industry, 3D is a booming business. Although the concept itself exists

Figure 1.5: Dodging a bullet in The Matrix

merely a century, it is not till the late nineties that there has been a commercial expansion. One of the most well known examples is the principle of *bullet-time*, first used in the Matrix trilogy. This process is not entirely 3D cinema, it is in fact cinema post-production where the camera rotates around the main character while he is dodging a bullet. In past times this effect was created with images from a series of no more than 270 cameras, but with recent DIBR algorithms like ours, it is possible to shoot these sequences with a limited number of cameras, leaving it up to the processing horsepowers to create the intermediate images.

As The Matrix set the pace, many other developments followed in this sector, such as full 3D movies, that use the principle of 'stereo rendering', where they make you wear special glasses to experience the 3D effect, the most well known are probably "Beowulf", "Meet the Robinsons" and the Belgian production "Fly me to the moon". As you can see, the 3D business is alive and kicking, and there is no foresight whatsoever that that is going to change in the nearby future.

1.3 Thesis disposition

1.3.1 Basic graphics hardware

In this chapter, the basic principles of the graphics hardware are explained. We give an overview of the pipelined graphics architecture and focus on the programmable elements of the Graphical Processing Unit (GPU), as they are the most important to us. As working with a GPU is very different from working with the CPU, basic graphics programming principles are given as well, describing how a programmer can learn to think in GPU programming terms, which is quite an adaptation. Since more and more graphics cards tend to be exploited for non-image processing, we introduce the reader to the world of General Purpose computations on the GPU (GPGPU).

1.3.2 Camera model and usage

This chapter handles all elements related to the camera whatsoever. First, the camera model is described. Starting from one of the simplest representations, to be gradually built up to a model that represents reality as good as it gets. So from then on, we can work with cameras in theory as well, since they will follow a defined model. Secondly, we take a closer look at the cameras we use to develop our application. We investigate why they were chosen and what properties give them an advantage. That means that synchronization, stability, color images and more are part of this section, to learn what

qualities and flaws real cameras can have. And finally, we dive in to the programming interface, that is used to configure the camera. We explain how these cameras can be manipulated to execute the desired functionality.

1.3.3 Feature point detection

This entire chapter is devoted on the workings and implementation of a feature point detector. Such a detector is very useful, since it will releave some computational work from other algorithmic units. When using feature points, we can relate corresponding images faster, because we do not have to match entire images when looking for correspondences. Some key properties of the corner detector, like repeatability rate, localization and computational speed are explained as well. Finally, the implementation approach we use to execute our functionality on the GPU, is presented. Using this design, various visual results were obtained and are summarized at the end of the chapter.

1.3.4 Camera calibration

Feature point detection is part of the process of camera calibration, explained in this chapter. As camera calibration obtains the intrinsic and extrinsic parameters of the camera, we demonstrate what these properties are and why it is so important to calculate them. These parameters can be extracted from the fundamental matrix, that describes the mapping from one image plane onto another. So essentially, we calulate the fundamental matrix first, and then we retrieve the camera parameters. But to calculate the fundamental matrix, we need a large number of corresponding points, detected with a corner detector, from which we can retrieve a number of accurate points with which we can compute a proper fundamental matrix. The estimation of this set of points is performed using an iterative optimization called Ransac. Once a good set of points is found, a fundamental matrix is calculated and the camera properties are retrieved, thereby solving camera calibration.

1.3.5 Distortion correction

This chapter focuses on the removal of distortion correction, which is an image deformation, induced by the use of real camera lenses. The distortion is modeled and an approach for a solid solution is presented. To remove distortion, the inverse operation is applied to the image to result in a corrected image. When implementing this on the GPU, we will use a reverse approach, because of the GPU's pipelined architecture. Calculating the undistorted image, causes bent lines to run straight again and gives a better approximation of the reality. Without this correction, it is not possible for the stereo matching algorithms to deliver an accurate intermediate viewpoint, if the received input does not even represent reality.

1.3.6 Rectification

In the final chapter, we explain why this last step of image preprocessing is necessary for the stereo matching algorithms. To correct the images, a special geometric setup is used, namely epipolar geometry. This leads to an easy understanding of what the rectification does with a pair of camera images. Because this is a very important step, just like distortion correction, we have subjected this stage to extensive testing on available datasets. The results are promising, as you will see, and the images are now ready to be used as the input of a stereo matching application.

1.4 Tools and experimental environment

The code described in this book is developed in the existing aLive framework which has been build for commodity hardware by a combination of various programming languages, each with a specific purpose. We will only provide a simple list of the tools and languages used in this thesis, providing the interested reader with limited but sufficient overview.

The aLive framework uses the object oriented programming language C as a host language for all the others. We use Microsoft DirectX version 9.0c and the extension library D3DX for communicating with the graphics card. The graphics card in our system is a NVIDIA GeForce 7900GTX with 512 MB GDDR3 memory [6] housed in a 3.2 GHz dual core PC with 1 GB system memory. Application programming on the GPU is done by developing and compiling the High-Level Shading Language (HLSL) to Shader Model 3.0 assembly languages. The GPU programming was structured by the use of the Effect Format, bundling all of the Shader programs in one collection.

Chapter 2

Basic graphics hardware

Graphics hardware is the core of our application since all the image processing that needs to be done, is offloaded to the supercomputing power of the Graphics Processing Unit (GPU). Proper use of its computational power can decide over your application's success or failure, we try to understand the way things work in there, so we can exploit some of its qualities to our own benefit. Different factors play a role in this. The possible outcome of your use of the graphics hardware depends on the amount of parallel programming you apply, the load balancing of the GPU pipeline, localizing the data transfer bottleneck and to which extent you are able to exploit the special qualities in your advantage.

2.1 The graphics pipeline

The GPU is not the average day system, handling tasks in a sequential order as fast as possible. Instead, it is built analogous to the assembly lines of the automotive industry, resulting in a pipelined design. Just as in these assembly lines the goal is to keep the data that needs processing equally long at each stage, to avoid that a stage would have to wait for the previous one or that all of the data piles up at one particular processing step (a bottleneck). If this criterion is reached, a balanced pipeline is constructed and will deliver the highest throughput. In GPUs we are not aiming on the construction of vehicles, but we want the hardware to process a high amount of vertices, geometric primitives and fragments. High throughput is the key feature of a balanced pipeline, as long as there is a lot of data to process, to keep the overhead minimized.

2.1.1 The contemporary model

In this section we present a simplified version of today's GPU's processing pipeline. Notice that in reality, all of the elements mentioned are far more complex and have more extensive capabilities than what is shown. But since we want to accustome the reader with the principal working of the GPU, to underbuild the explanations given in the chapters ahead, a simple but clear overview will suffice.

In figure 2.1 a simplified model of the graphical pipeline is shown. As you can see, the graphics hardware can be subdivided into four basic computational stages.

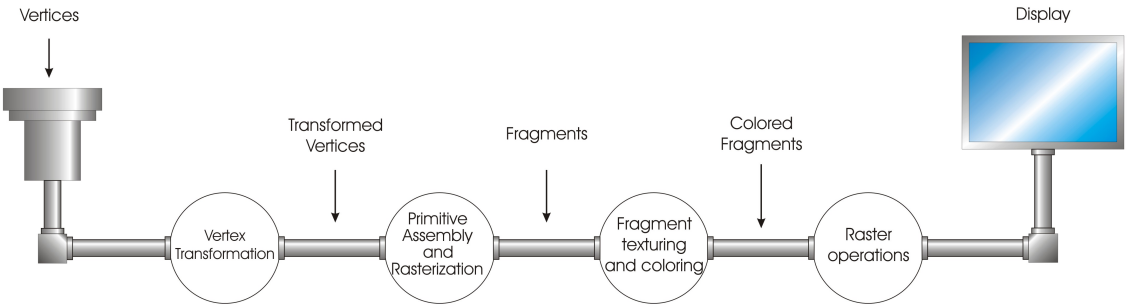

Figure 2.1: The basic graphics hardware pipeline

1. The first stage receives the input data and performs the *Vertex transformation*. The input data provided, merely exists from a couple of vertex coordinates, that describe a model in three dimensions. By perfoming the vertex transformation onto these vertices, their model coordinates are transformed to view coordinates, that can simply be seen as the screen positions, if you would look at the object through a camera. This transformation is in fact a projection from 3D world space to 2D screen space, and can be modeled through a matrix multiplication. These mathemathical operations are performed on each vertex seperately so the order of processing does not influence the result.

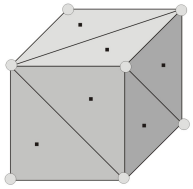

Figure 2.2: A cube subdivided in triangle primitives

2. *Primitive assembly and Rasterization* is performed in the stage following the vertex transformation. First, the hardware computes which coordinates form a primitive geometric surface, using additional input information describing the object(s) in the scene. Possible primitives are either triangles,lines or points, all other primitives can be constructed out of combinations of these basic types (e.g. the cube in figure 2.2). Primitives that do not completely fall into the view's visible region of 3D space (*the view frustrum*), are partially or entirely discarded in a process that is called *clipping*.

   Secondly, the rasterizer will determine for each of the polygons that survived the previous step, what set of pixels is covered by this geometric primitive. This process is called rasterization and the output elements are called *fragments* instead of pixels. Try not to confuse pixels and fragments as they are not the same. A pixel stands for '*pix*el *el*ement' and is part of the content of the frame buffer at a specific location. A fragment can better be described as a 'potential pixel', because only if it survives the various raster operations at the end of the pipe, it will update an element of the frame buffer and become a pixel.

3. *Fragment texturing and coloring* can then be performed on this collection of fragments that is outputted by the rasterizer. This is the stage in the pipeline that gives the fragments their final color, based on their initial color and various calculations (for shadows, depth polling, etc.). A texture is an image that is loaded into GPU memory and can be accessed from within this stage to determine the color of each fragment. As not every image will have the same dimensions as the polygon it will color, some filtering (e.g. sub- or supersampling) may be necessary.

4. Then, a final sequence of *raster operations* is executed on the fragments before they are sent to the frame buffer, ready to appear on your screen. In this stage, hidden surfaces are eliminated. This means that when new fragments lie behind the present pixels, which are already in the frame buffer, they have to be discarded. This is only one example of all the operations that the fragments have to undergo, like blending, dithering or anti-aliasing, just to name a few. But, because these other operations are not of utmost important to get a good overview of the inner workings of the GPU, we will round our story up here, and refer to other work [3, 7] for more in-depth information.

To illustrate these key processing stages, an example is quite useful, so we have spread the entire process out in figure 2.3. The input the user sends, are the vertices that define a couple of triangles, and what comes out of the pipe are the pixels to be displayed on the screen.

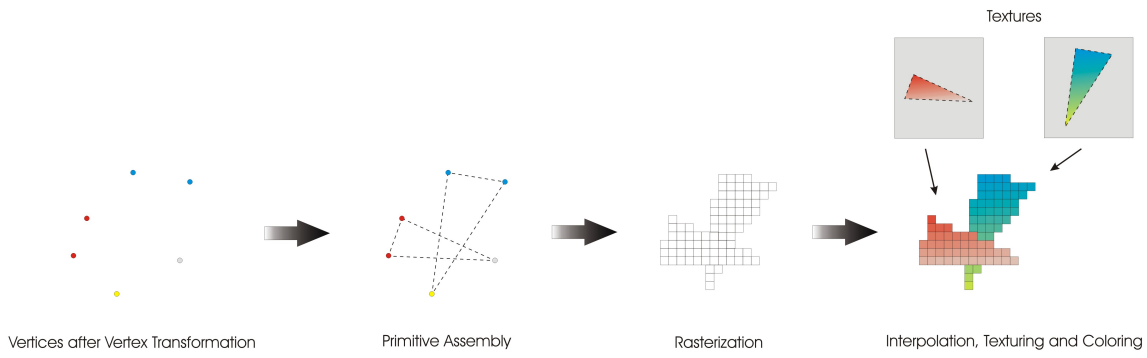

Figure 2.3: Visualizing the graphics pipeline

2.1.2 CPU - GPU interface

Although graphics hardware nowadays offers a lot of flexibility and programmability as we will see in section 2.2, its functionality cannot be directly invoked by an application running on the CPU. To protect the hardware and the system user against unwanted errors or crashes in the GPU's functionality, the GPU programs can only be invoked through an Application Programming Interface (API). In our system setup we will be using Microsoft's Direct3D programming API [8, 9] and the DirectX 9.0c additional library to communicate transparantly between CPU and graphics hardware. That means that, to invoke GPU functionality, an application issues a standard DirectX function call and

in the background, the library then performs the necessary steps to get the GPU program (also called *shader*) up and running. This allows a user to send and receive data from the graphics card in a transparant way (figure 2.4).

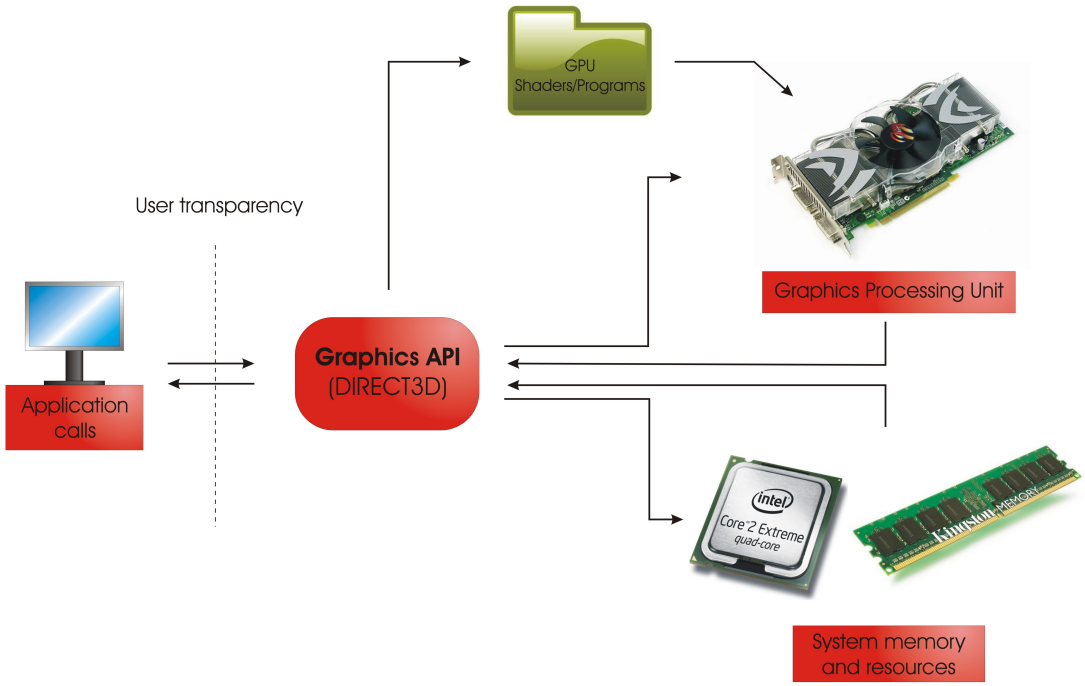

Figure 2.4: The graphics API

Not only provides the use of an API user transparency, it also allows us to divide an application that uses GPU resources up in two parts, one application-specific and one hardware-specific. This decoupling between the part on the CPU, for control and high-level tasks, and the part on the GPU, with hardware specific assembly programming, makes it possible for an application to run on two systems with different graphics hardware and different drivers without the need of recompilation. This increases the application's portability a great deal.

2.2 Programming graphics hardware

In the beginning, there was nothing, so a graphics card did not exist and we could not speak of pipelines at all. The earliest graphics hardware on a computer had the sole purpose to buffer the data that would be displayed on the screen. All of the per-pixel calculations and other imaging operations was executed on the CPU. But the generic buildup of the CPU made it much more suited for all kinds of different tasks, but not so much for the high amount of computations and the specialized functionality that was required for proper 3D effects. Therefore specialized hardware was introduced into the system and replaced the much too slow CPU to perform the needed graphics computations [7, 10]. The first generations of graphics hardware were fixed functionality pipelines and

were called *3D acceleration boards*. They were the first to have vertex acceleration and were also developed to fulfill the increasing demands of the gaming community.

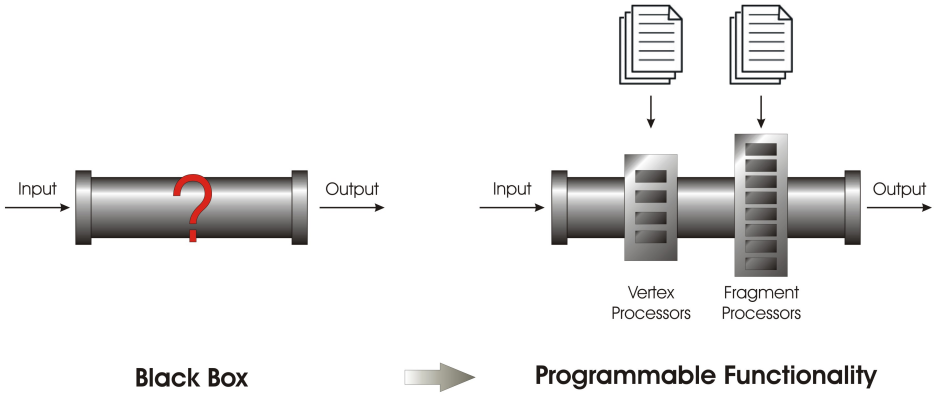

Figure 2.5: Evolution from fixed functionality (black box) to flexible and highly programmable hardware

This graphics hardware of the early ages was built out of configurable hardware. Up untill now, the dominant trend in graphics hardware design has been the effort to expose more programmability within the GPU (figure 2.5). This process started with the introduction of a load of parallel processors on the GPU, first in the fragment processing stage, later in the vertex shader stage and contemporary GPUs have introduced geometry shaders as well [11]. So not only the raw speed, the high level of parallelization and the increasing precision, but especially the rapidly expanding programmability make the modern GPU so flexible and powerful. As a consequence, programming graphics hardware has become a hot topic in the development programmer's world, either for image processing tasks, either for general purpose computations (GPGPU) [10] as we will see in section 2.3.

2.2.1 Multi-level parallelism

Just as everything else, technology evolves, and when we talk about silicon die technology, it evolves extremely fast. Nowadays we are able to fit hundreds of thousands of computational units on a single die. To optimize performance with such a high amount of computational units, the objective should be to let as much as possible units process useful data and to have as few as possible units process control data or storage data. To achieve this goal of maximizing the computational performance, the units have to process lots of data at the same time, or in other words, in parallel. Graphics hardware designers grasped this concept and have tried to exploit it to the fullest. This has lead to four different levels of parallelism [12, 3], all an essential part and contributing to the high-speed processing of the modern GPUs. Although not directly necessary for hardware programming, knowing what these levels of parallelism are about, leads to a better understanding and thus a more efficient use of the hardware. Therefore we summarize:

1. *Task parallelism* is visible at a first glance at the GPU pipeline. Every stage is

working concurrently with the other stages, each performing different operations, thus working in parallel and increasing throughput.

2. When we zoom in at a single stage of the pipe, *high-level data-parallelism* can be descried. Each stage is composed out of multiple processing units and thanks to data independency (a primary demand of GPU programming) tasks can be run on different data at the same time.

3. Taking a closer look at the processors individually, we will learn that the input data is a vector format. Using vector processors, this means that computations will be perfomed on upto four different data elements at the same time. This concept is better known as *low-level data-parallelism*.

4. On each of these data elements, in one action multiple simple instructions can be executed. We will refer to this as *instruction-level parallelism*.

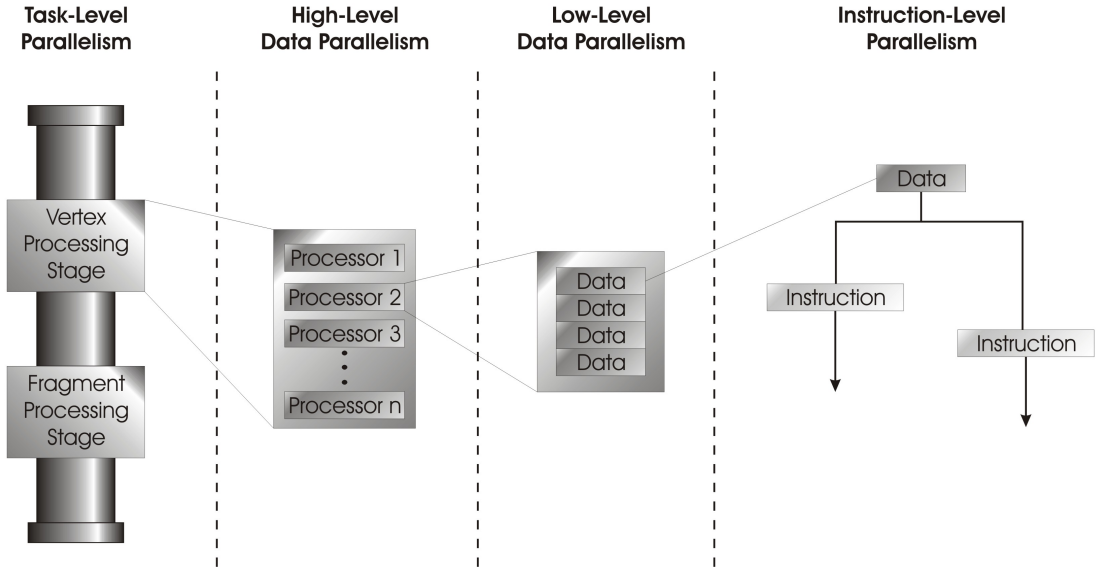

Figure 2.6: The different levels of parallelism in the GPU

This high degree of parallelization is also the main cause why GPUs are currently able to outperform even numerous CPUs. Thanks to the parallel nature of the GPU, it can profit to the maximum of new developments in the silicon die technology. As more and more computational units can be placed on a single die, the CPUs run into trouble with their high amount of unparallelized control data. The only way of increasing performance is to increase clock speed as well, while GPUs can simply increase the number of parallel processors they are using. That results directly in much higher computational horsepower, since most of the computational units of a GPU spend their time processing useful data instead of control data.

Now, you might think that, if we would just make everything in the GPU programmable, we would be able to perform any task. That is not the case, as many sequential programming functionalities cannot be implemented 'as is' in the GPU, for instance data independency is crucial and only in the very latest GPU generations branching in the

fragment shader is possible, just like vertex texture accessing. Plus, the obtained return lies much higher when we leave some of the pipe's hardware specialized for certain tasks. Triangle rasterization is in that case a perfect example where specialized hardware will be the better of full programmability of the stage. That is why, when we are going into detail on hardware programming in the upcoming section, we will only cover the vertex shader and the pixel shader and not the entire GPU pipe.

2.2.2 Shader programming

When we want to program either the vertex shader or the fragment shader, we have to get our customized functionality into the GPU. As we have discussed before, any communication between the user or the CPU and the graphics card has to go through the graphics API (DirectX or OpenGL). But since the graphics program itself does not have anything to do with intermodule communications, it is written in another language. This used to be simple assembly code, but as the 3D industry evolved so quickly a new language was developed to lighten the burden of the effect programmer. Microsoft and Nvidia combined their forces and developed a C-like shading language, called *C for graphics* (Cg) [7]. Later on, the partnership split up and Microsoft started selling the shading language under the name of *the High-Level Shading Language* (HLSL) [13]. They are both still very alike, but in this work we will only address HLSL.

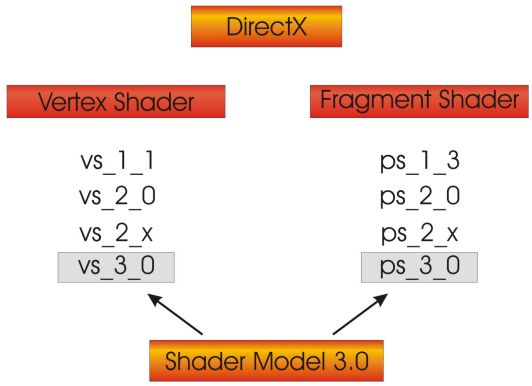

Figure 2.7: Shader Profiles

The Effect framework was introduced by Microsoft and allows to store shader programs, settings and more in one global *.fx* file. Within the DirectX 9.0 API an fx loader is integrated, so your self-written piece of code, that has to be executed in a stage of the GPU, will get loaded into the programmable processors through the API after passing the FX-compiler. Yet there are still restrictions on what you can program on a vertex or fragment shader, for instance the vertex shader can not access other vertices just as the pixel shader can not access other pixels (data independency). And, for programming comfort, different profiles can be used to compile a vertex or fragment program (see figure 2.7), allowing it to run on different generations of GPUs supporting older or newer profiles. In DirectX these profiles are called *Shader Models* (SM) and in this work Shader Model 3.0 is used. Depending on how demanding your piece of code is, you may want to use a more advanced profile, risking that your program will not work on older hardware. To get the

best of both worlds, that is, broad hardware support and still allowing the use of the latest hardware, you could write different versions of your program. One backup version, to run on almost every GPU, and another advanced version for greater functionality hardware. Having all of these different pieces of code work together, remember that they are all still in the same .fx file, is covered in the next section.

2.2.3 High-level programming

To get the shader programs you have written to work on the GPU it is not sufficient to just group them together in a single file. Three quintessential elements ensure the smooth cooperation of all these pieces and make sure that everything that is needed to apply a rendering effect, is present in the .fx file. Only by wrapping the vertex and pixel shader programs together with render state, texture state and graphics pipeline *pass* information, thus configuring the *entire* pipeline, a developer can describe a complete rendering effect.

- Each .fx file contains one or more *techniques*, which all describe a way to create an effect. A technique always applies for a specific profile, thus aiming for a certain level of GPU functionality. By making use of different techniques, a fallback version of your effect to work on older hardware can be provided together with a more advanced version (see listing 2.1).

Listing 2.1: Using techniques to combine shaders

```
technique Example
{
    pass P0
    {
        VertexShader = compile vs_3_0 vs_std();
        PixelShader = compile ps_3_0 ps_std();
    }
}

technique Example_Backup
{
    pass P0
    {
        VertexShader = compile vs_1_1 vs_std();
        PixelShader = compile ps_1_3 ps_std();
    }
}
```

- Each technique contains *one or more passes*. Each pass represents a set of render states and shaders to apply for a single rendering pass within a technique.

Listing 2.2: An example shader and environment setup

```
//Definition of structures used in the shader
struct VS_OUTPUT_STD
{
    float4 viewCoordinate: POSITION;
  float2 texelCoordinate: TEXCOORD0;
};
```

```
struct PS_OUTPUT_STD
{
    float4 computedValue: COLOR;
};

//Input image is loaded in GPU memory as a texture
//and texture filtering modes are set
texture inputImage;
sampler inputSampler = sampler_state
{
    Texture     = <inputImage>;
    MipFilter   = NONE;
    MinFilter   = LINEAR;
    MagFilter   = LINEAR;
    AddressU    = BORDER;
    AddressV    = BORDER;
    AddressW    = BORDER;
};

//Standard pixel shader that samples a texture to color the output pixels
PS_OUTPUT ps_std(VS_OUTPUT vertexOutput)
{
    PS_OUTPUT_STD pixelOutput;

    float4 sampledTexel = tex2D(InputSampler, vertexOutput.texelCoordinate);
    pixelOutput.computedValue.rgba = sampledTexel.rgba;

    return pixelOutput;
}
```

- Within each pass you have to provide some information about the programs environment, what we describe as *pipeline states*. Alpha blending, depth writes or texture filtering modes (see listing 2.2) are some of the most common examples.

2.2.4 Schematic overview

Now that we have described most of the essential elements of the graphics hardware, leading us to a better understanding and more efficient programming of the graphics pipeline, it is time for a short overview.

In figure 2.8 a brief summary is given of the position of the GPU in the overall system architecture. You can see that when a lot of data transfer is necessary between system RAM and graphics hardware or when a lot commmunication between CPU and GPU is inavoidable, this will seriously downgrade hardware performance since the transfer bus speed will create a bottleneck.

A brief overview of the essential elements that built up the core of our graphics card, the NVIDIA GeForce 7900GTX, is given in figure 2.9. This is a perfect proof of what we have tried to explain in the previous sections. The graphics hardware is built out of several pipelined stages and these stages are capable of processing lots of data in parallel through the use of a parallel processing architecture.

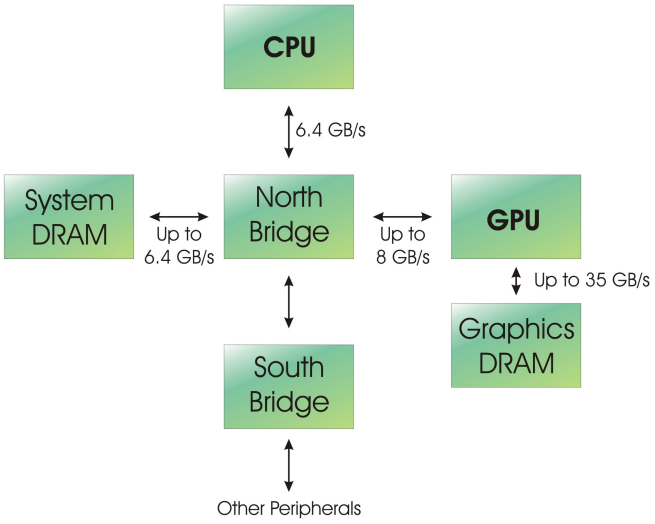

Figure 2.8: Overall system architecture

2.3 Exploiting graphics hardware (GPGPU)

The graphics hardware sector is probably the fastest growing sector in silicon technology, as mentioned before. Modern GPUs have become very flexible systems with tremendous computational horsepower and fully programmable vertex and pixel shaders. No wonder that these processors are now capable of much more than the graphics applications they were originally designed for. Under the name of General Purpose computations on the GPU (GPGPU) [14] researchers and developers have become interested in harnessing this horsepower for general purpose computations. They have found that exploiting the GPU in such a way can accelerate some problems by over an order of magnitude compared to the CPU.

However, it is not possible for any programmer to just make the switch to GPU programming. Mapping algorithms, that were designed for the sequential and low latency based CPU, to the highly parallel and high throughput GPU requires good knowledge of the graphics functionality. To introduce this new concept and to allow you to understand some of the GPU exploitation that is explained in other sections of this book, we sum up some of the most basic conversions that a program developer has to think of, to make his functionality run on the GPU's programmable processors. You can think of it as CPU-GPU analogies. Note that this is not a complete summary, as this field of graphics programming evolves every day and we did not need the general purpose mappings in this book to its full extent. That is why concepts as scatter and gather, reduce, sort or scan [10] are not discussed here.

2.3.1 Stream programming

Data does not appear under the same form in the GPU as we are used to it on the CPU. In the graphics architecture, data is represented by textures. So an array of data will become a 2D texture when ported to the GPU as you can see in figure 2.10. Since

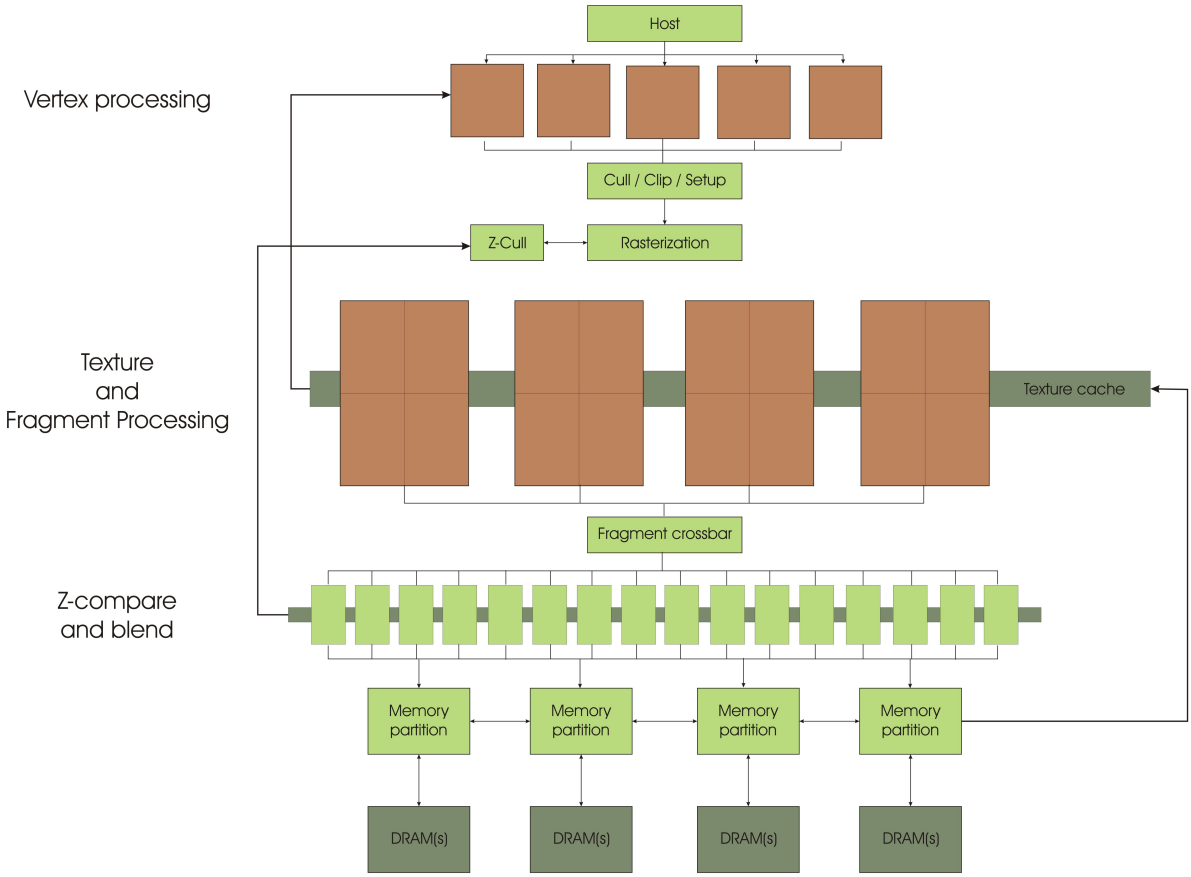

Figure 2.9: A block diagram of the Nvidia GeForce 7 Series

two major issues concerning data on GPU are data parallelism and data independency, we can think of processing an array of data as processing a stream, where the same computations are performed on each element of the stream and no dependancy between various elements is allowed. It should be clear that, to invoke this computation, we have to send a quadrilateral to the start of the graphics pipeline. It is important to make sure that there is a one to one mapping between the data stored in the texture and the fragments that come out of the rasterizer. Otherwise data values will get interpolated when using a texture sampler and results will become corrupted.

2.3.2 Fragment shaders

We call the sequence of computations that is performed on the elements of a stream the *computational kernel*. When we would perform these kernel computations on an array of data on the CPU, we would use nested loops to iterate through all the elements of the array. The kernel itself would then be the statements inside the inner loop. On graphics hardware, we write the demanded computations or kernel in a fragment shader, which is loaded into all the processors available in the stage and executed on all the elements of the sampled texture, so no nested loops are used. The amount of parallelism in this stage depends on the number of processors and on how well we exploit the data parallelism,

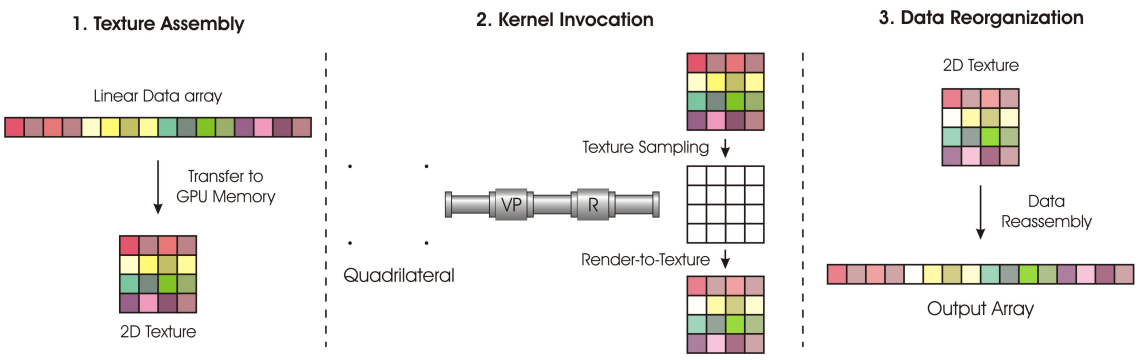

Figure 2.10: The Stream Programming Model

since processors work with four-component vectors. Making good use of this data packing, when working with scalar values, can greatly reduce the time of computation [12].

2.3.3 Render-to-texture

In CPU programming, feedback is inherently present since the CPU uses a memory model where variables and arrays can be written to and read out from all the time. On the GPU this is not possible, the fragment shader has to write its results to the frame buffer, but it cannot use immediate feedback. There are two ways to get output data into a texture that can be read during the following pipeline pass. Either indirectly, by copying it from the frame buffer (*copy-to-texture*) or directly, by rendering the results immediately on an offline texture (*render-to-texture*).

These three concepts, using textures for data storage, fragment shaders as computational kernels and output feedback through texture storage, are the basics of GPGPU programming. It should give you the insight that GPGPU programming is not as easy as it may sound, but the possible outcome, a leap forward in computational capability and a possible growth quickly exceeding CPU limits, makes all the programming efforts and challenges worthwhile.

Chapter 3

Camera model and usage

Cameras provide our application with its own set of eyes, they are irreplaceable because they have to transform the surroundings to an input for our image processing application. And just like the human eye, cameras are complicated structures, so models are used to describe the cameras' behaviour and gradually we build up towards a general description the cameras' inner workings. Such models describe the camera properties or intrinsic parameters as well as the cameras pose and relative orientation or extrinsic parameters. Once a model is set, a reality check has to be performed so we analyse why we opted for synchronizable, stable, color cameras. If we determine what the camera should do, with our model, and analyse what it does in reality, we are ready to make it do what we want, we program the eyes of the application to capture exactly that information that subsequent image processing needs.

3.1 Camera model

A camera is a mapping between the 3D world and its image plane. There are several camera models that describe this mapping, the one we will be using is the finite projective camera model. The model actually represents a matrix multiplication that transforms a point in the 3D world coordinate system to an image point. In fact, this equals a single multiplication with the 3×4 *perspective projection matrix* (PPM). The fastest way to understand this transformation is to build the model from the bottom up. We start with a very specialized and simple camera model, *the pinhole camera model*, and go all the way to a *generic projective camera model* through a series of gradations.

3.1.1 The pinhole camera model

The pinhole camera model originates from the age-old camera obscura (see figure 3.1), where all light rays converge at the hole in the camera case to construct an inverted image on the camera's backplane. In our model we call this point where light rays join, the camera center or optical center. Since we do not like it so much to work with inverted images, we make life a little easier and put the image plane or focal plane in front of the camera, at a distance equal to the *focal length f*.

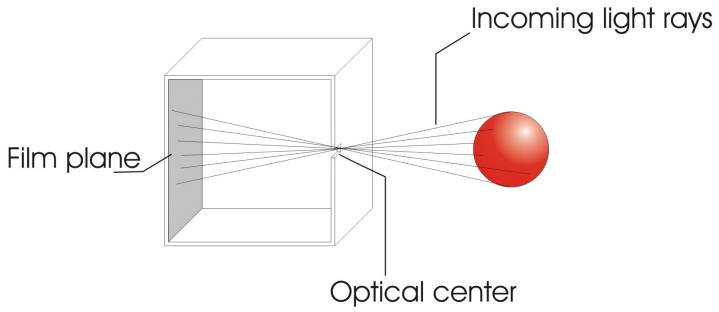

Figure 3.1: The camera obscura

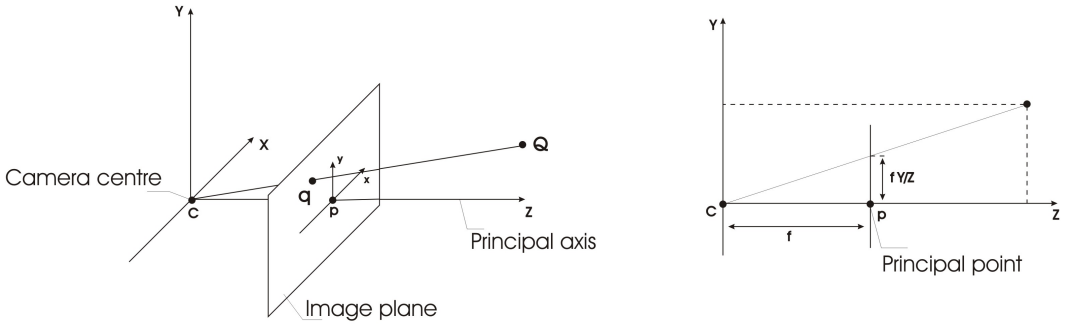

Figure 3.2: The pinhole camera model

Further, we let the center of projection coincide with the origin of a Euclidian coordinate system and choose an arbitrary point $Q = [X\ Y\ Z]^T$ in 3D space. In this model, Q is mapped onto the point on the image plane where the line between Q and the optical center instersects the plane. Using similar triangles we can calculate that the 3D point $Q = [X\ Y\ Z]^T$ is mapped onto the image point $q = [fX/Z\ fY/Z\ f]^T$, as you can see in figure 3.2. For all image points of the focal plane, the third coordinate is the same, so it can be negligated. This leads to the mapping

$$[X\ ,\ Y\ ,\ Z]^T \rightarrow [fX/Z\ ,\ fY/Z]^T \tag{3.1}$$

which proves that the pinhole camera model is in fact a mapping from world coordinates (3D) to image coordinates (2D). We call the point of intersection between a perpendicular line starting at the optical center and the focal plane *the principal point*.

When we express both points Q and q in *homogenous coordinates*, the pinhole model mapping can simply be described in matrix form as a linear mapping between their homogenous coordinates. This means that we can rewrite equation (3.1) as follows

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \rightarrow \begin{bmatrix} fX \\ fY \\ Z \end{bmatrix} = \begin{bmatrix} f & & & 0 \\ & f & & 0 \\ & & 1 & 0 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \tag{3.2}$$

Now when we call $\tilde{Q}$ the notation of a point in homogenous world coordinates $\begin{bmatrix} X & Y & Z & 1 \end{bmatrix}^T$ and $\tilde{q}$ the notation for an image point represented by its homogenous 3-vector $[fX\ fY\ Z]^T$, we can define P as the 3×4 homogenous *camera projection matrix*. Equation (3.2) can be condensed to

$$\tilde{q} = P\tilde{Q} \tag{3.3}$$

The mapping of the pinhole projection model from homogenous world coordinates to homogenous image coordinates is now described by the projection matrix P

$$P = \begin{bmatrix} f & & & 0 \\ & f & & 0 \\ & & 1 & 0 \end{bmatrix} \tag{3.4}$$

3.1.2 Principal point offset

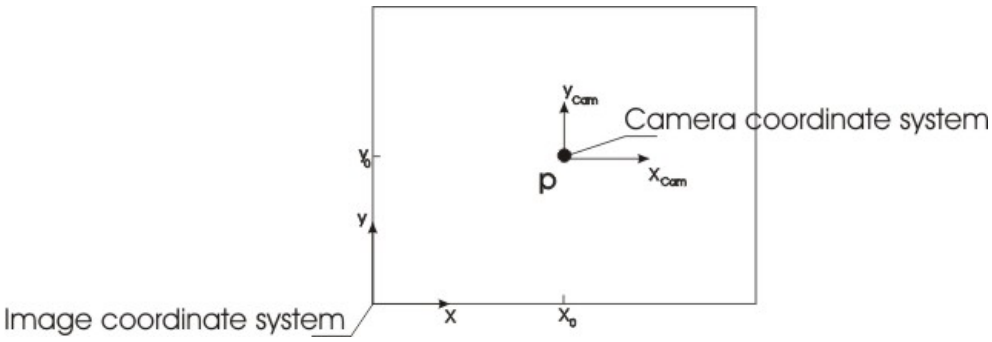

Figure 3.3: Principal point offset

In equation (3.1) we have assumed that the origin of the image coordinate system coincides with the principal point p. In reality though, the image coordinate system will have its reference in the bottom left corner of the image plane, which induces a shift in equation (3.1)

$$[X\ ,\ Y\ ,\ Z]^T \rightarrow [fX/Z + p_x\ ,\ fY/Z + p_y]^T \tag{3.5}$$

when we consider the couple (p_x, p_y) to be the image coordinates of the principal point. Just like before, we can rewrite this equation in an equation with homogenous coordinates, leading to the matrix transformation

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \rightarrow \begin{bmatrix} fX + Zp_x \\ fY + Zp_y \\ Z \end{bmatrix} = \begin{bmatrix} f & & p_x & 0 \\ & f & p_y & 0 \\ & & 1 & 0 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \tag{3.6}$$

Now, when we define

$$K = \begin{bmatrix} f & & p_x \\ & f & p_y \\ & & 1 \end{bmatrix} \tag{3.7}$$

equation (3.6) can be abbreviated to

$$\tilde{q} = K\tilde{Q}_{cam} \tag{3.8}$$

and we call K the *camera calibration matrix*.

3.1.3 Camera rotation and translation

You have probably noticed that we used the notation Q_{cam} in equation (3.8). Doing this, we want to stress that these coordinates, of the 3D point Q, are still under the assumption that the camera is located at the origin of the Euclidian coordinate system with the principle axis aligned with the Z axis. We call Q_{cam} a point in the *camera coordinate frame*. In reality though, most of the time a point in world space will not be expressed in camera coordinates, but in coordinates of a different Euclidian coordinate system what we call the *world coordinate frame*. These two coordinate systems are related to each other through a rotation R and a translation t as you can see in figure 3.4.

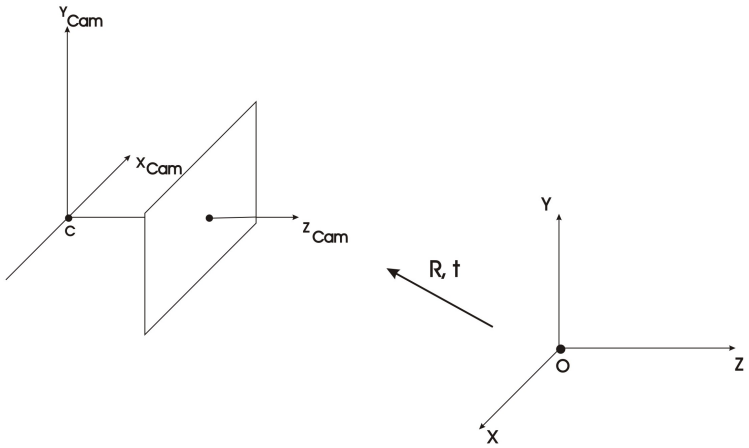

Figure 3.4: R,t relation between the world reference frame and the camera frame

Let Q_{cam} be an arbitrary inhomogenous point in the camera frame and let Q be that same point, expressed in inhomogenous world coordinates. Now we can write $Q_{cam} = R(Q - C_{world})$ with C_{world} the camera center expressed in world coordinates and R a 3×3 matrix that represents the orientation of the camera coordinate frame. Again, we can rewrite this equation in homogenous coordinates which gives us

$$\tilde{Q}_{cam} = \begin{bmatrix} R & -R\tilde{C}_{world} \\ 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} = \begin{bmatrix} R & -R\tilde{C}_{world} \\ 0 & 1 \end{bmatrix} \tilde{Q} \tag{3.9}$$

Combining equation (3.9) with equation (3.8) gives

$$\tilde{q} = KR\left[I \mid -\tilde{C}_{world}\right]\tilde{Q} \tag{3.10}$$

$\tilde{Q}$ is now expressed in world coordinates and this is the general projective mapping for the pinhole camera model. The elements of matrix K are called the *intrinsic camera parameters* and R and $\tilde{C}_{world}$ who define the orientation and position of the camera to the world reference frame, confine the *external camera parameters*. Usually we do not express the camera's position in terms of the camera center, so more often the world to image transformation is represented by $\tilde{Q}_{cam} = R\tilde{Q} + t$, where t represents the translation from the world origin to the camera center. The 3×4 perspective projection matrix P, or the 3D-2D mapping camera matrix, becomes simply

$$P = K\left[R \mid t\right] \tag{3.11}$$

where we know out equation (3.10), that $t = -R\tilde{C}_{world}$.

3.1.4 CCD camera imperfections

The pinhole camera model defines that the image coordinate system is a Euclidian coordinate system just as the world coordinate system with equal scales on both x and y axes. But when we are working with real Charged Coupled Device (CCD) cameras, there is a possibility that there are non-square pixels. That would mean pixels are longer in the x-direction than in the y-direction or vice versa. So if image coordinates are expressed in pixels, this demands for an extra scaling of the pinhole model. To introduce this additional generality into our model we will use m_x and m_y to express the number of pixels per unit distance along the corresponding axes. To get the transformation from world coordinates to *pixel* coordinates, we have to multiply equation (3.7) on the left with $diag(m_x,\ m_y,\ 1)$. This is a 3×3 matrix with the latter numbers on the diagonal. This lead us to

$$K = \begin{bmatrix} \alpha_x & & x_0 \\ & \alpha_y & y_0 \\ & & 1 \end{bmatrix} \tag{3.12}$$

The elements α_x and α_y equal fm_x and fm_y respectively and represent the focal length of the camera in terms of pixel dimensions in x and y directions. $\tilde{x}_0 = (x_0, y_0)$ is the representation of the principal point in pixel dimensions and its coordinates are $x_0 = m_x p_x$ and $y_0 = m_y p_y$. To introduce even more generality, we can take the possible skewness of the pixels into account as well. The *skew factor* s models the non-orthogonality of the x and y pixel axes (see figure 3.5). This value is almost always zero, but in certain cases it may differ, so adding this factor increases the generality of our model as well.

Since skew has no effect on the orientation or position of the camera, it is also an intrinsic parameter, one of the camera's properties and that means that finally, with the parameter s defined as the skew factor, we get

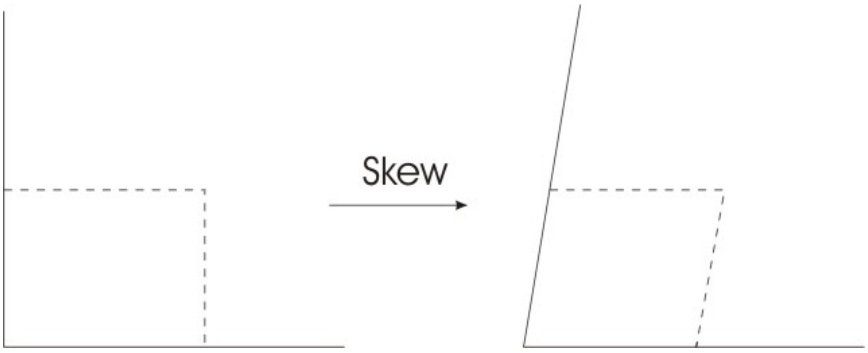

Figure 3.5: skew effect s

$$K = \begin{bmatrix} \alpha_x & s & x_0 \\ & \alpha_y & y_0 \\ & & 1 \end{bmatrix} \tag{3.13}$$

as the most general description of the intrinsic matrix of the finite projective camera model.

3.2 Camera specifications

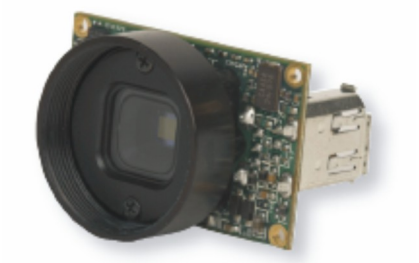

Figure 3.6: The Point Grey Firefly MV Camera

One of the thesis' objectives is to create a real-time preprocessing environment for the aLive framework, so we are able to test the used algorithms on live feed and perform checks on the achievable frame rate, the processed image quality and are able to work with a large baseline. In order to do so, we need cameras that are easy to use, but still complex enough to allow user-specified programs to run on it. As you can see, we are not talking about a couple of ordinary webcams, but this implies working with an industrial type of camera.

I was not granted to choose the type of cameras, the manufacturer, the price or the number of cameras. That choice had been made before my arrival on Imec, namely the Firefly MV camera, manufactured by Point Grey (see figure 3.6). This choice was motivated by the demands that, for our application, we need synchronizable, stable, FireWire-based color cameras in a low price category. In the following paragraphs we will

discuss the primary properties of the cameras, how they apply to our project and why, of all choices, this seemed the optimal case.

3.2.1 A team player

One of the key features that distinguish these cameras from a couple of ordinary cameras is their capability to *synchronize automatically*. Cameras that are on the same IEEE 1394 bus should synchronize instantly due to an on-board implementation. For cameras that are on different buses or on different systems (different PC), there is a software kit, delivered along with the cameras, that enables this kind of synchronization. In our case, this feature is of utmost importance, because when the cameras would not synchronize to each other, *'motion estimation'* would not be possible. Motion estimation is the principle that, when the experimental setup or stage is not static, we will assume that either the camera or the subject is on the move and out of the various frames captured on subsequent moments in time, we will try to derive and reconstruct the motion made by the camera or the subject.

This would not be possible without synchronization, because then you would risk that one of the frames is taken at a slightly different time, meaning that the camera or the subject has already moved compared to the frames captured by the other camera(s) and you would not have a chance to retrieve the motion the scene has undergone. Consider the following example (figure 3.7). We assume that both images were taken at the same time, but in reality camera 2 has captured the picture slightly later. From our point of view, for a time $t = 0.016$ seconds, the dropping ball is at two different heights (floor 2 and floor 1) at the same time so we will never be able to perform motion estimation, because we are not sure of the ball's exact position at every captured instance of time.

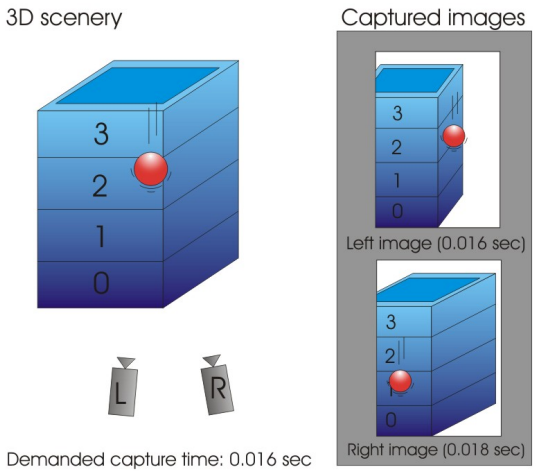

Figure 3.7: Unsynced images corrupt motion estimation

We experienced a minor setback ourselves when we started working with the cameras, because initially they would not synchronize. Thanks to a manufacturer's firmware release, the problem is far behind us. We are now guaranteed that frames on different cameras are captured within a timeframe of 125 microseconds, which makes it possible to work with

most of the normal movements. Thanks to the timestamp encapsulated in the FlyCapture API we are able to check and confirm that pictures remain synchronized.

3.2.2 400 Mbps IEEE 1394 (FireWire) digital interface

Transfer of data from the cameras to the controlling host, in our case the PC, happens through the FireWire interface, which was especially designed for time-critical applications such as audio and video applications. Its maximum data rate of 400 Mb/s obviously outperforms the old serial connectors such as RS/232 or PS/2 that go up to 20 Kb/s. These older connectors have already been replaced by a faster version that *theoretically* gets a data rate just as high as the FireWire interface, namely USB (Universal Serial Bus). Key differences between FireWire and USB are [15]:

- *Design goal*: USB was designed for simplicity and low cost, while FireWire aims for high performance and time-critical applications.
- USB uses a "*speak when spoken to*" protocol, which means that peripherals can not communicate with the host, unless the host specifically requests communication. The 1394 bus does not require a central controller or dedicated host computer for the data transfers, but instead operates peer-to-peer to allow any device on the bus to initiate transfers on its own.
- While USB devices cannot efficiently utilize all the available bandwidth because the communication is based on polling the devices, FireWire guarantees *fixed bandwidth with low overhead* for isochronous data transfer.

Not only does this standard outperform the other serial interfaces, it also provides the same services as existing IEEE-standard parallel buses at a potentially lower cost. Rather than transferring data via a parallel interface, such as EIDE and SCSI with expensive cables and connectors with as many as 68 pins, 1394 requires only four signal conductors in a low cost interconnecting cable. No need to argue that the FireWire interface is the perfect candidate for our time-critical video-based application with multiple cameras.

3.2.3 Color Space fundamentals

Because the perception of color is subjective and may differ from person to person, people have tried to come up with a method to help describe color either in human interaction or in machine communication, the result being color spaces. You could define a color space as follows [16]: "A model for representing color numerically in terms of three or more coordinates or parameters." These parameters do not tell us what the color is, that depends on what color space is used. E.g. the RGB color space represents colors in terms of red, green and blue coordinates.

The need for various color spaces can be easily explained with the following example (see figure 3.8). A computer monitor starts with black pixels that need to be lit, combining the three base colors of the human perception, red, green and blue. All together these three colors form white and that is why we call this an *additive color space*. Printers in contrary, start from a white paper and place filters on the surface to subtract or absorb

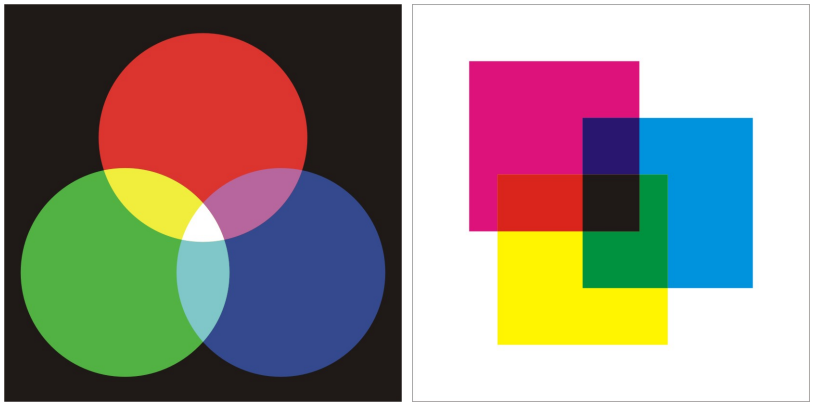

Figure 3.8: Additive colorspace vs. Subtractive colorspace

certain color ranges. Therefore the YCMK (Yellow Cyan Magenta Black) color space is used. We call this a *subtractive color space*. Of course there are also other color spaces used, e.g. intuitive color spaces such as HSL (Hue Saturation and Lightness), aimed to facilitate constructing a color of your choice.

In image processing however, there is a high need for color intensity, because various image processing algorithms use intensity images or grayscal images to perform their calculations on. The YUV color space solves this problem; it separates the *luminance* (intensity) component from the *chrominance (color)* information of colors [17, 18]. The Y is a combination of all three RGB components with different coefficients due to the sensitivity of the human eye. The U and V components are a combination of Y with respectively the blue and the red component. For the construction of grayscale images, only the Y component is needed because it represents the intensity of a pixel, so we will not make any further remarks on the U or V components. In our application, the camera transmits its captured information in Y8 format, which is an eight bit monochrome channel, thus only using the luminance component. This comes in handy, since we will be performing corner detection on grayscale images in a later stage.

3.2.4 Monochrome versus Color

The decision whether to use a monochrome camera or a color camera should be made entirely based on the application you are trying to build rather than on your preference of color or monochrome images. The Monochrome camera has an overall higher light sensitivity that makes its biggest difference in the region of the infrared spectrum (figure 3.9). Some online camera calibration methods are based on a technique using light pulses. To relieve the human eye, you might use infrared light pulses and that is where the monochrome camera comes in handy. Since we will be using a checker board for camera calibration, we are set with a couple of color cameras. Another argument for this cause is that we will be performing feature matching. It comes down to recognizing the same distinct points in two separate images of the same scene, so color images offer easier distinction than monochrome images where the only valuable test would be the intensity of the pixels.

This does not mean that we cannot use monochrome images anymore, because our

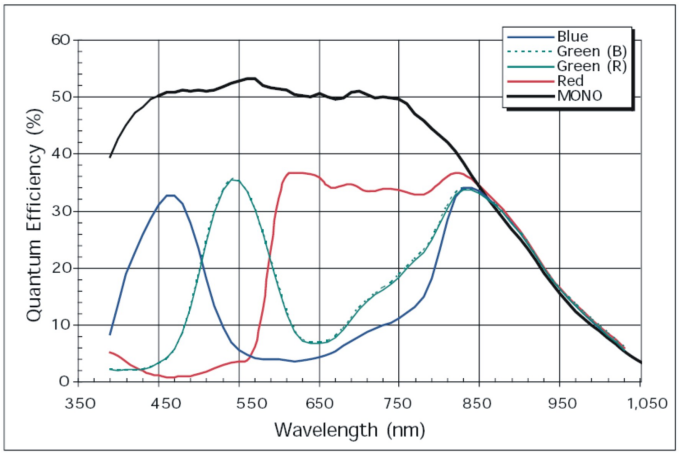

Figure 3.9: Spectral response of the Firefly MV

Firefly MV color cameras use a Bayer tiled image format but transmit their information in Y8 format [19, 20]. To convert this information to YUV or RGB or any other supported format, we use conversion functions that are found in the FlyCapture API. So we are free to choose the image format we need after capturing, in our application that would be monochrome or grayscale images for corner detection and color images for live streaming.

3.3 Camera steering

3.3.1 Point Grey Firefly API

To configure the cameras and to let our own customized functionality be implementable, the manufacturer has included a programming API with the product package. It is the Point Grey FlyCapture API and it provides a lot of functions and structures to implement the design we want. As we have already mentioned before, the most important issue in our camera application, is to get both cameras synchronized to a level that is acceptable for a live streaming video.

At a first glance, using some of the example programs that were included in the accompanying software package, we did not get the cameras to synchronize. And we did not really find any clues in the API to implement the synchronization. So we checked the information on the manufacturer's website, revealing that our the cameras would never synchronize with the present software. But as in so many cases, a little manufacturer help can come in very handy. We found a recently released firmware fix - we are talking August 2007 - that would solve the problem. And it did. Thanks to the firmware upgrade, cameras on the same IEEE 1394 bus synchronize automatically.

Of course, it would not suffice just to trust on the manufacturer's word, saying that they would synchronize, we would have to check this ourselves. That is where the programming API became of proper use. With every frame that is send across the FireWire cable, an accompanying timestamp travels along as you can see in listing 3.1.

Listing 3.1: The FlyCaptureImage structure.

```
struct FlyCaptureImage
{
    int iRows;
    int iCols;
    int iRowInc;
    FlyCaptureVideoMode videoMode;
    FlyCaptureTimestamp timeStamp;
    unsigned char* pData;
    bool bStippled;
    FlyCapturePixelFormat pixelFormat;
    unsigned long ulReserved[ 6 ];
}
```

This timestamping structure which is described in listing 3.2, has several components useful in different cases of synchronisation. With an application like ours, where the cameras are connected to the same 1394 bus, the ulSeconds and the ulMicroseconds promise to be the most useful. They represent the absolute *system time* when each frame was captured in seconds and microseconds.

Listing 3.2: The FlyCaptureTimeStamp structure.

```
struct FlyCaptureTimestamp
{
    unsigned long ulSeconds;
    unsigned long ulMicroSeconds;
    unsigned long ulCycleSeconds;
    unsigned long ulCycleCount;
    unsigned long ulCycleOffset;
}
```

In the case that you are synchronizing image grabs between two different systems or computers that share a common IEEE 1394 bus, you are advised to use the ulCycleSeconds and the ulCycleCount elements of the FlyCaptureTimestamp structure. Using the other elements would give incorrect measurements since the system timers will not be precisely syncronized.

3.3.2 Synchronized results

A small application is developed to allow the user to grab images from the cameras at a user-specific time. This means the image grabbing action has to be initiated with a keystroke. Pseudocode of how our program works is presented below.

Listing 3.3: The FlyCaptureImage structure.

```
// The number of images to grab.
#define _IMAGES_TO_GRAB 25

// The number of cameras on the bus.
#define ALL_CAMS        2


// What file format should we save the processed images as?
```

```
#define SAVE_FORMAT       FLYCAPTURE_FILEFORMAT_PPM
#define SAVE_FORMAT_RAW       FLYCAPTURE_FILEFORMAT_PGM

#define FILENAME_CONVERTED "images.ppm"
#define FILENAME_RAW         "raw_images.pgm"

// Initialize the camera.
printf( "Initializing camera %u.\n", CameraNumber );
error = flycaptureInitialize( context[CameraNumber], CameraNumber );
_HANDLE_ERROR( error, "flycaptureInitialize()" );

unsigned int AllCameras = ALL_CAMS;


for ( int iImage = 0; iImage < _IMAGES_TO_GRAB; iImage++ )
{

printf( "Grabbing image %d\n",iImage );

    for( CameraNumber = 0; CameraNumber < AllCameras; CameraNumber++ )
    {
        error = flycaptureGrabImage2( context[CameraNumber], &image[CameraNumber] );
        _HANDLE_ERROR( error, "flycaptureGrabImage2()" );
    }

printf( "Saving image%u%u.\n",CameraNumber,iImage );
error = flycaptureSaveImage(
    arcontext[CameraNumber],
    &image[CameraNumber],
    file,
    SAVE_FORMAT );
_HANDLE_ERROR( error, "flycaptureSaveImage()" );

printf( "Saving raw image%u%u.\n",CameraNumber,iImage );
error = flycaptureSaveImage( ... );
_HANDLE_ERROR( error, "flycaptureSaveImage()" );


printf( "(hit enter)\n" );
getchar();

}
```

It was of course necessary to give the user feedback and to give him proof that the processing was successful, that the images were stored correctly and that they were taken isochronously. To fulfil these demands there were two techniques available, an intuitive method and an intrusive, direct method.

- the *intuitive method* tries to provide visual control of the motion estimation problem. We came up with the following experiment: We put two reasonably well parallel aligned cameras in front of a grid. Then we dropped a bounce ball in front of the camera, giving us the possibility to check in both camera frames at what hight the moving ball has been captured. When synchronisation was successful, the ball should be floating at the same height in both images. Proof is given in figure 3.10,

where you can clearly see that the bouncing ball is at the exact same height in both captured frames, as far as you can read out the grid accurately.

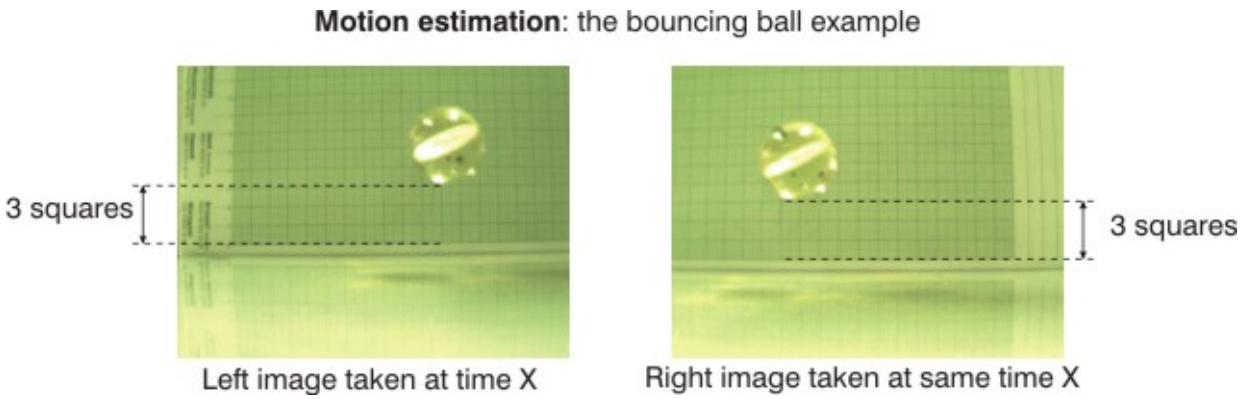

Figure 3.10: Without synchronization, the ball might be at two different heights at the "same" time.

So to the human eye, everything seems in order, but that does not mean that the synchronisation is accurate enough. A couple of still frames might look accurate at first sight, but when we take a series of images and combine them into a movie with our DIBR algorithm, the possible synchronization faults will easily disrupt the fluent motion of the bouncing ball.

- That is why there is also a more direct, *intrusive method* to check the accuracy of the camera sychronization, using the FlyCaptureTimestamp mechanism. With each sequence of images that is gathered at an instance of image grabbing from all cameras present on the 1394 bus, we print out the accompanying image timestamp to give the user feedback (see figure 3.11 and listing 3.4).

Listing 3.4: The timestamp output

```
for( CameraNumber = 0; CameraNumber < AllCameras; CameraNumber++ )
{
    printf("Camera %02u: time = %03u.%04u, seconds = %04u,
           microseconds = %04u\n",
           uiCamera,
           image[ uiCamera ].timeStamp.ulCycleSeconds,
           image[ uiCamera ].timeStamp.ulCycleCount,
           image[ uiCamera ].timeStamp.ulSeconds,
           image[ uiCamera ].timeStamp.ulMicroSeconds
           );
}
```

This guarantees us that the captured images are synchronized up to 125 microseconds, according to the manufacturers release note that came with the firmware update. This is quite logical since the ulCycleCount wraps around after 8000 values, what means that its smallest time notation is 1/8000th of a second = 125 microseconds.

With this problem out of the way, we can step on it and take our application to the next level. Now that images can be grabbed synchronously and on demand, it is time to

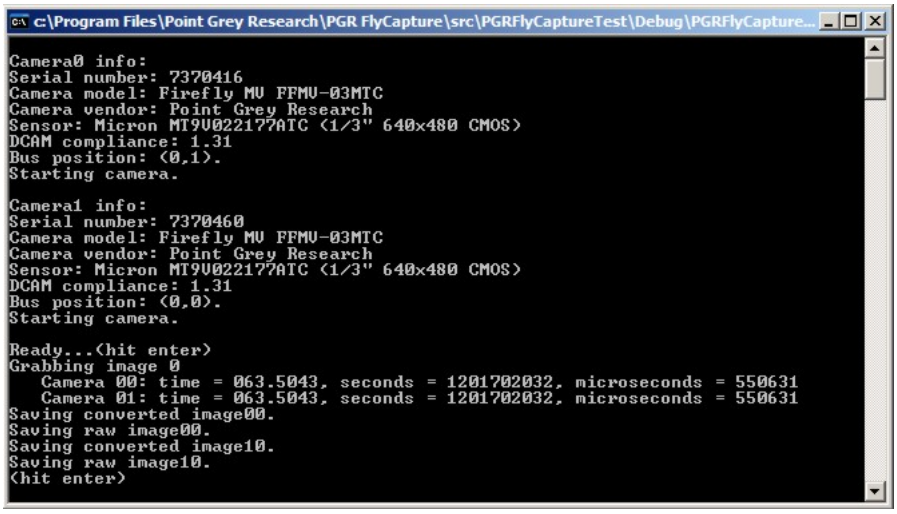

Figure 3.11: synchronization times are equal for both captured frames

implement feature detection on the captured images, which is the next step to full camera calibration.

Chapter 4

Feature point detection

When we want to relate several images to reconstruct the captured surroundings of the cameras, pixel-based matching would be too time-consuming and manually matching the images makes no sense at all. An easy yet thorough way to correlate images and objects in the images is necessary for our time-critical application. Therefore we use sets of interest points, points that are remarkable in both images. Since computational horsepower is not our number one concern, but our timeframe is, have we chosen for a computationally expensive but stable and accurate detector, the *Harris corner detector*. Using grayscale images to calculate intensity variations around the interest points, we are able to correlate objects or object features between images. The output of this feature point detection is the essential input for the upcoming chapter, because it will estimate the properties of the cameras, thus defining their behaviour and that of their surroundings, leading us to 3D reconstruction.

4.1 Requirements

The principle of feature point detection is used in a broad range of computer vision applications. This involves visual robotic servoing, object tracking, stitching panoramic view pictures and of course stereo matching. Most of the time, these applications require a way to relate two or more images in order to extract valuable information. In our line of work, we want to relate images that are taken at the same time, but from different point of views. The most straightforward way to implement some kind of image matching algorithm, is the brute force approach. Take one image as the reference image and determine the corresponding pixel in the image under test for every pixel of the original image. This would take an enormous amount of computational power and time since you would have to do a very expensive, full image search for every pixel, seriously downgrading your image processing application speed.

An intelligent approach suggests that you relate any pair of images, based only on some interesting or striking locations or objects present in the frames. Such locations are referred to as *points of interest* or *feature points* and can be detected using a feature point detector. Using just a small collection of significant points instead of every pixel in the image, will decrease your application's computation time dramatically. But using

too little feature points to relate images, can also result in a bad image correlation, that leads to failure of the application. The ideal amount of feature points to use is a trade-off, between avoiding false results and still keeping the computation time as short as possible, that has to be made with great caution. How such points of interest are detected, depends on the kind of interest point detector you are using. Some will define maxima of local curvature as interest points, other will use locations of highly varying texture. We are looking at corner points as interest points, because they are significant as an intersection point of two edges and thereby usually lie on the boundary between two objects or two parts of the same object.

To demonstrate what qualities a corner detector must harness to stand out, to separate itself from the rest of the pack, we will give a list of some of the most important properties. As you will find out, you cannot combine all of these qualities into one detector, since some of them are incompatible and others are application dependent. The application's developer will have to make the right decision at the start, depending on what criteria the application requires.

- Detecting *all true corners* and *no false corners* might seem an obvious demand, but it is not. It is simply application dependent, because, over all these years, there is still no strict definition of a corner in an image that covers all possibilities.

- The demand of *proper localization* is an appropriate one. Most of the time, you want the corner detector to point out the corners correctly instead of giving you an approximation of where the corner lies. Especially when you want to combine a series of images, for instance in a panoramic view. You want to be sure, when you cut and paste an image on a found edge, that it is actually cutting on the edge and not a couple of inches away. But in other applications, such as object detection, it is sufficient that you get a global approximation of the objects contours to perform recognition. In other words, good localization is desirable for all, but critical for some applications.

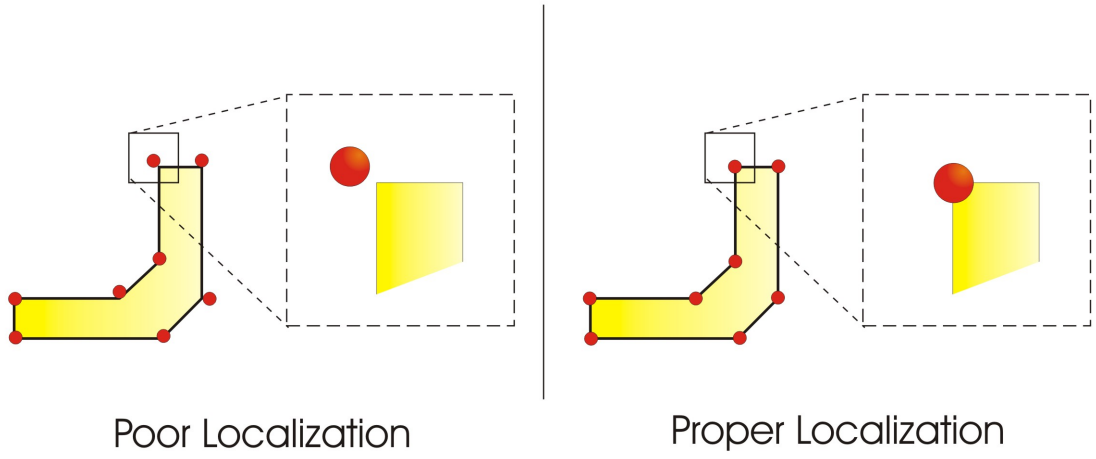

Figure 4.1: Example of pour versus proper localization

- When you are searching corners or interest points in several images that have all captured the same scene, you want your corner detector to detect the same interest points in all these images. That is why a *high repeatability rate* is a very desirable property for a corner detector (see figure 4.2). For instance, when using a visual servoing robotics application that moves through an environment, you need to find correspondences between consecutive frames. The same points have to be detected, even if there is a slight change in viewpoint or illumination. It assures you that you have a stable corner detector. We define the repeatability rate as the percentage of the total number of corner points which are repeated between two images.

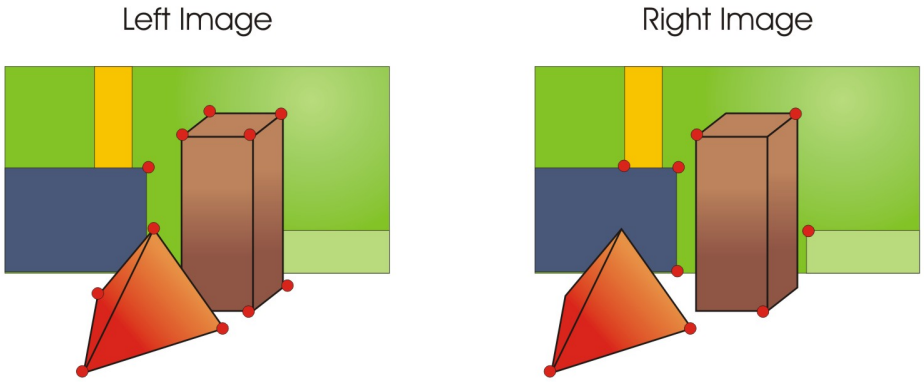

Figure 4.2: Illustration of corner detector with low repeatability rate

- Every real image is subjected to *noise*, which may lead to a false corner detection. That is why you might demand a detector that is robust against noise and whose localization will not suffer from it either.

- When building a real-time application, you might prefer a computationally efficient detector over a very robust and really accurate one. Again, this trade-off, is application dependent, but influences the working of your detector with great effect.

In the image processing application that we are building, a high detection rate and a high repeatability rate are of utmost importance. In a later processing stage, we will perform feature matching on a pair of images, which means a different number of corners in both images (low repeatability rate) or a low amount of corners detected (low detection rate), would lead to incorrect matching, and thereby to erroneous camera calibration.

4.2 Corner detector flowchart

Historically, corner detection algorithms have evolved since the late 1970's. Several models are not in use anymore today, while others are still popular, even after thirty years. But most remarkable is, that up till now, no 'universally good' real-time corner detector has been built. If you think about it, with the previous section in mind, it is not that unrealistic, since there are so many different applications that all have different demands

and even the definition of what a 'corner' exactly is, changes according to the subject it is applied to.

Over the years, three main trends were introduced in corner detection on grayscale images: edge-relation methods, topology methods and autocorrelation methods. Concerning edge-relation methods, topology methods or other alternative corner detection methods, you will hear harmonious names as Kitchen and Rosenfeld [21], SUSAN [22] or Curvature Scale Space (CSS) corner detector [23]. But we chose to implement an autocorrelation method that is widely used in stereo matching, *the Harris corner detector* [24]. This method has been around for nearly twenty years now, and in that period it has undergone a couple of revisions and modifications that have made it a suited candidate for our application. It is known for its capability of detecting almost all true corners, it has a good repeatability rate and over the years proper localization has been achieved as well. Since we will be implementing our corner detector on the GPU as well, it does not matter such a great deal that this detector is more computationally expensive than others.

The Harris corner detector uses *a measure of local autocorrelation* to calculate the corners in an image. It is the successor of the Moravec operator [25, 26], which introduced the concept of 'points of interest' and also resides on an autocorrelation method for corner detection. To perform autocorrelation, consider a local window in the image, shift this window in various directions and calculate the minimum change of intensity in which these shifts result. This operation is repeated for every pixel in the image, so the output image still has the same dimensions as the input image, but only now every pixel has an *interest value*. A point of interest is defined as a local maximum in this image of interest values. This means that corners, who have a large intensity variation in every direction, will be detected as well, making this a corner detector, but with a more relaxed definition of a corner.

Autocorrelation is the main principle where our corner detection algorithm is based on. But as you may have guessed, there is a bit more to it than simply performing autocorrelation calculations on the images. There are three main processing steps that are present in most of the corner detectors: applying the corner operator, thresholding the image and performing non-maximal suppression. These steps are schematized in a generic flowchart (figure 4.3) and a more into detail explanation follows in the subsequent sections.

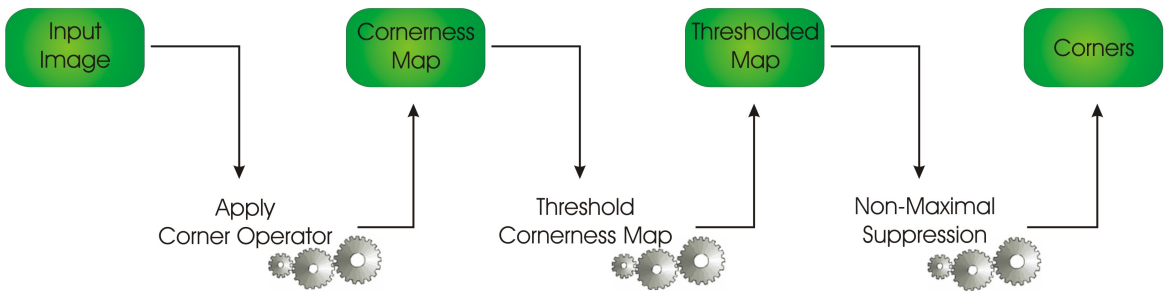

Figure 4.3: General flowchart of most corner detectors

4.2.1 Apply corner operator

The first step in detection algorithms is always applying the specific corner operator. This stage takes the original image as input and returns a processed version as output. For each pixel in the image, a *cornerness measure* is calculated, which indicates to what extent each pixel resembles a corner. As a consequence, we call the outcome of these calculations a *cornerness map*. How this cornerness measure is calculated varies according to which algorithm is used. We will summarize the Moravec interpretation of the cornerness measure concept here, as it is the basis for the Harris corner detector we will be implementing.

Moravec was the first to define interest points as points where there is a large intensity variation in every direction, which is the case at corners as well. But he was not so much trying to detect corners, as he was just trying to find distinct regions in an image that would allow him to match objects in consecutive image frames. This intensity variation became the key to constructing his cornerness measure. The calculation of the Moravec cornerness measure is built out of two steps:

1. Let $I(x, y)$ be the intensity of a pixel (x, y) and take a grayscale image and a window size as input parameters. For each pixel (x, y) in the image, calculate the *intensity variation* from a shift (u, v) compared to the reference pixel as

$$V_{(u,v)}(x, y) = \sum_{a=-\frac{N}{2}}^{a=+\frac{N}{2}} \sum_{b=-\frac{M}{2}}^{b=+\frac{M}{2}} [I(x + u + a, y + v + b) - I(x + a, y + b)]^2 \tag{4.1}$$

   where the shifts (u, v) are elements of Ψ, and represent the eight principle directions of figure 4.4

$$\Psi = [(-1, 1), (0, 1), (1, 1), (-1, 0), (1, 0), (-1, -1), (0, -1), (1, -1)] \tag{4.2}$$

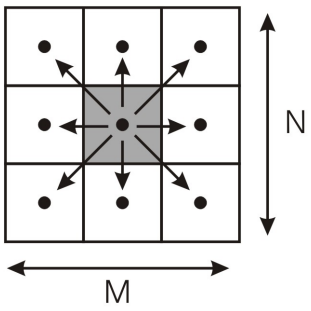

Figure 4.4: The eight shift directions for the Moravec intensity variation

2. Out of the resulting values of these calculations we construct *the cornerness measure* $C(x, y)$ for each pixel (x, y) as

$$C(x, y) = \min \left\{ V_{(u,v)}(x, y) \right\} \qquad with\ (u, v)\ \in\ \Psi \tag{4.3}$$

So for every pixel in the image we calculate the minimum intensity variation of shifts in the eight principle directions and assign this value as the cornerness measure to the pixel.

4.2.2 Threshold cornerness map

The output of the cornerness measure operation, the cornerness map, has to be thresholded. Remember that interest point detectors define corners or points of interest as local maxima in the cornerness map. You can easily imagine that there are a lot of local maxima in the cornerness map that are not at all true corners, so a threshold is applied.

This holds that every pixel in the cornerness map that has a value below the threshold, is set to zero. The value of the threshold has to be empirically chosen, not too high, so we prevent missing some true corners, and not too low, so no false corners are detected. Practice learns that there is rarely a threshold value that will exclude all false corners and include all true corners, so it will have to be chosen depending on the application's needs.

4.2.3 Perform non-maximal suppression

The final critical step in our corner detecion algorithm is to locate the local maxima in the thresholded cornerness map. In this map, thanks to the thresholding operation, there are only non-zero values in the surroundings of a corner that needs to be detected. In order to find a maximum that represents a real corner, we apply non-maximal suppression to eliminate high, but not the highest values, close to the real corner. So for every pixel, a check is performed whether it has the highest value within a certain range, for instance within a circular kernel with a five pixel radius. If this is not the case, the value of that pixel is set to zero. The search range that is used, the radius of the circle, is also a user-defined input variable that varies for every application. Corners are now easily found in the ouput image of this step, since they are the only pixels left with a non-zero value.

4.3 Implementing the Harris corner operator

In this section, a more detailed explanation of the Harris corner detector is given. It builds on the Moravec corner detector and uses the flowchart described earlier. We will demonstrate the difference between the Harris and the Moravec detector and explain why these changes are necessary and how we implemented them in our graphics pipeline.

In figure 4.5, a schematic overview is given of all the necessary steps that have to be implemented on the GPU to construct a full Harris corner detector. Although each of the steps of this storyboard is discussed separately in further sections, you can already recognize the global flowchart structure of section 4.2 in this figure. First, every important step in constructing the cornerness measure is given, and then we perform the thresholding and non-maximal suppression steps we are already acquainted with. The formula of autocorrelation to construct the cornerness measure is given as well, to clarify why such steps as Gaussian weighting and calculating partial derivatives are necessary, but the full derivation is given in section 4.3.4.

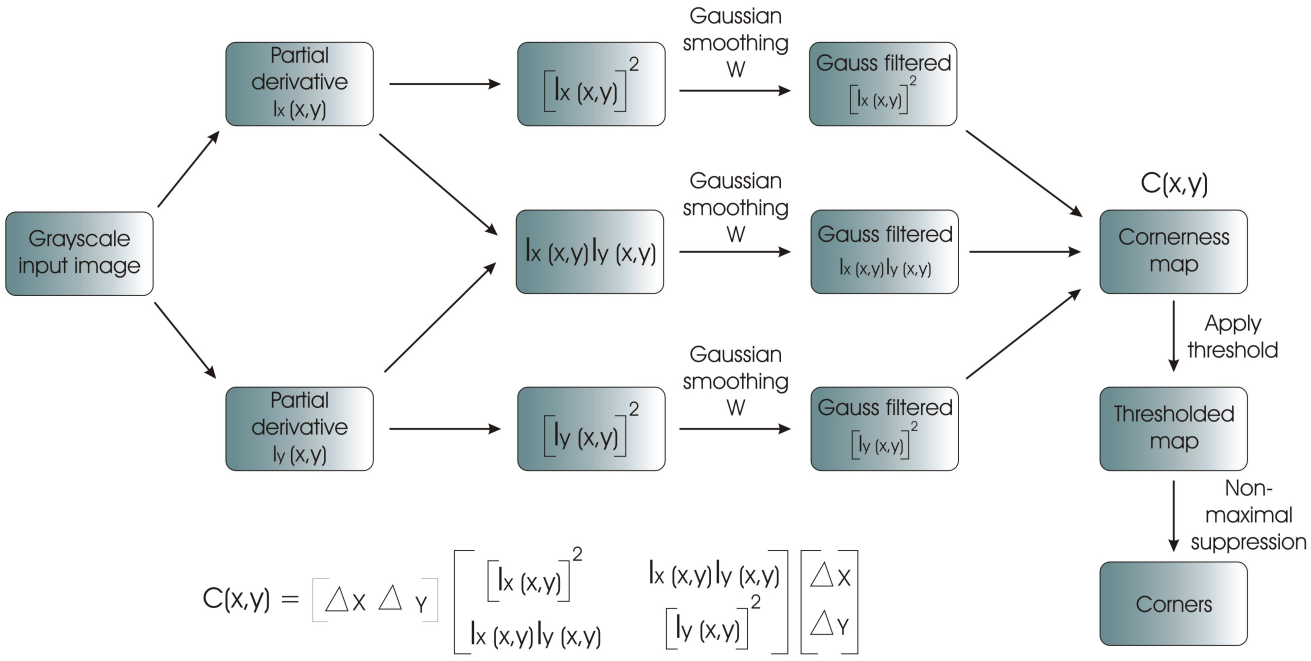

Figure 4.5: General flowchart of our Harris corner detector

4.3.1 Intensity based algorithm

We have mentioned it before, most interest point detection algorithms make use of some sort of intensity variation to perform edge or corner recognition, and so does the Harris corner detector. So any input image we want to process, has to be converted to an intensity image first. This implies that all RGB images have to be converted images that only use the Y component of the YUV format, what equals a conversion to grayscale images.

As we have seen in section 3.2.3 about color spaces, it is possible to create a YUV image from an RGB image and it has the important feature that it separates the luminance (intensity) and the chrominance (color) information of the image. Most important to remember about the YUV color space, is that the Y component contains the luminance information we need and that it is a combination of all three RGB components with different coefficients due to the sensitivity of the human eye. To make the conversion from an RGB image to a suitable grayscale corner detection input image, we use the simple one step formula [27]

$$Y = 0.299R + 0.587G + 0.114B \tag{4.4}$$

Implementing this step on the GPU really is a piece of cake, since it can make perfect use of the internal structures that accompany every pixel that arrives at the pixel shader. It should be clear why a pixel shader is used, since this is the stage of the graphics pipeline where texturing and coloring is performed.

Listing 4.1: The Grayscale conversion.

```
struct PS_OUTPUT_STD
{
    float4 computedValue: COLOR;
};


PS_OUTPUT_STD ps_greyConversion(VS_OUTPUT_STD vertexOutput)
{
    PS_OUTPUT_STD pixelOutput;

    float4 sampledTexel = tex2D(clampSampler0, vertexOutput.texelCoordinate);

    pixelOutput.computedValue.rgb = dot(sampledTexel.rgb, float3(0.299f, 0.587f, 0.114f));
    pixelOutput.computedValue.a = 1.0;

    return pixelOutput;
}
```

In listing 4.1 you can see that the output structure *pixelOutput* contains a 4 element vector, describing the red, green, blue and alpha (opacity) channel of every pixel. In order to get a grayscale image as output, we calculate the intensity out of the input RGB channels and give the output RGB channels all that same intensity value, so no color pops out.

4.3.2 Partial Derivatives

The Moravec operator we described in section 4.2, has a discrete isotropic response because the intensity variation is only calculated in the eight principle directions using two horizontal, two vertical and four diagonal shifts. Harris and Stephens performed an analytic expansion to this model, as you will see in section 4.3.4, resulting in a corner measurement that allows the variation of the autocorrelation to be calculated over any orientation. This analytical expansion uses partial derivatives to compute its autocorrelation function, but since the derivation is given in a subsequent section, we will illustrate here intuitively, that the use of image gradients will make it possible to calculate an intensity variation in all directions.

Very often the Prewitt operator [27] is used to approximate the first-order gradients of an image, because computing the real gradient is rather time consuming. Figure 4.6 shows an example of the gradient approximation for the horizontal and diagonal direction. Knowing this, we can rewrite the Moravec intensity variation measurement for a horizontal shift (figure 4.7) as follows

$$V_x = \sum_{i=1}^{9}(A_i - B_i)^2 = \sum_{i=1}^{9}(B_i - A_i)^2 \approx \sum_{i=1}^{9}(\frac{\partial I_i}{\partial x})^2 \tag{4.5}$$

where

$$\frac{\partial I_i}{\partial x} \equiv I_i \otimes (-1, 0, 1) \approx B_i - A_i \tag{4.6}$$

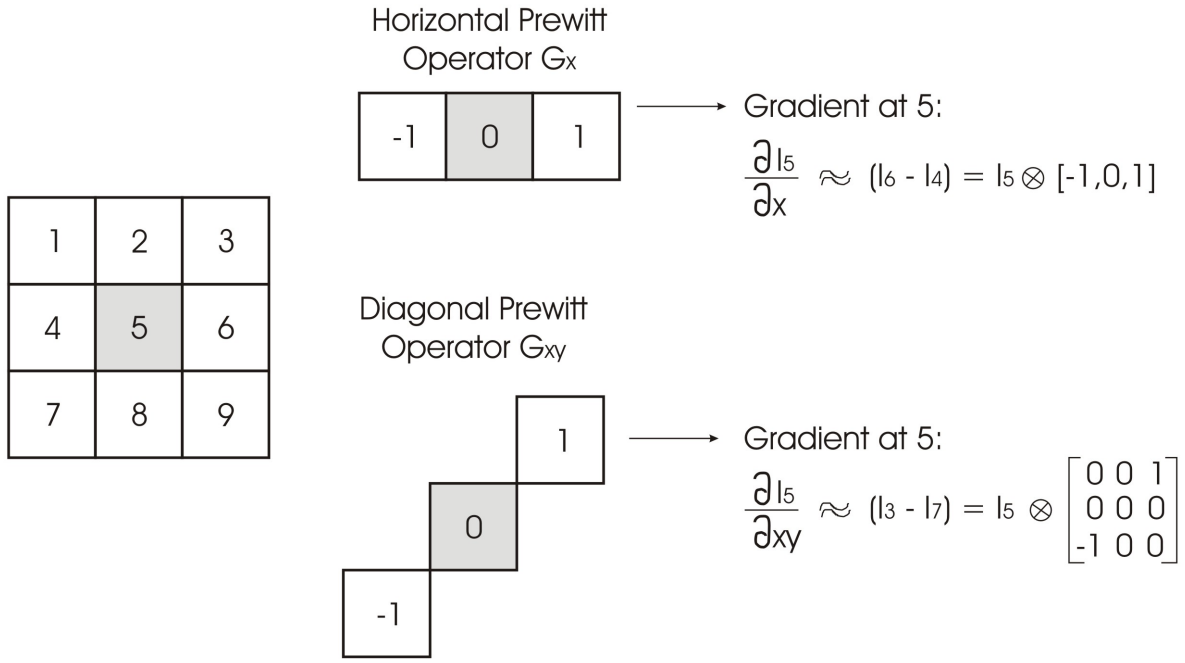

Figure 4.6: Sample use of the Prewitt operator

The same derivation can be made for a shift in the vertical, diagonal or any other direction. This analysis indicates that intensity variation can be written as a function of the gradient of the image. So for any shift (u, v) the intensity variation becomes

$$V_{(u,v)}(x,y) = \sum_{j=-\frac{N}{2}}^{j=+\frac{N}{2}} \sum_{k=-\frac{M}{2}}^{k=+\frac{M}{2}} \left(u\frac{\partial I_j}{\partial x} + v\frac{\partial I_k}{\partial y} \right) \qquad with\ (u,v)\ \in\ \Psi \tag{4.7}$$

where $\frac{\partial I_j}{\partial x}$ and $\frac{\partial I_k}{\partial y}$ are computed as ilustrated in equation (4.6)

Thanks to this new measurement, we obtain an rotationally invariant description of the intensity variation, since proper choice of u and v allows us to calculate the autocorrelation for any direction.

According to the formula in figure 4.5, we will need partial derivatives in both x- and y-directions as well as a multiplication of both. So we have to send our GPU the assignment to calculate the necessary partial derivatives out of the grayscale input image and store them in temporary textures, using the render-to-texture function (see 2.3.3), to reuse this data in our next processing step, calulating the cornerness measure.

We have two options when we want to implement this on the GPU: either we stick to the Prewitt approximation of the gradients and build a shader to execute this functionality, or we try to use some of the additional specialized hardware and let the GPU calculate the image gradients by itself. Because we were still in a testing phase, and for starters we just wanted a working version of the Harris corner detector, we chose to stick to the Prewitt approximation. Programming the GPU to use specialized hardware would consume too much of our development time, so initially we left this part open for optimization.

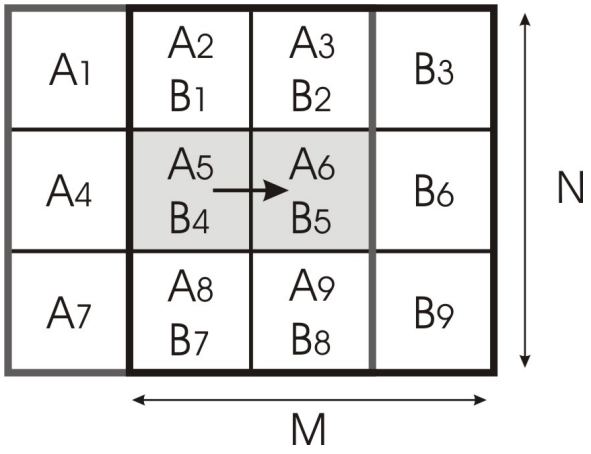

Figure 4.7: Window shift in the horizontal x-direction

4.3.3 Gauss convolution

In image processing, convolution is the multiplication sum of the convolution kernel with the covered image region of the kernel. The convolution kernel is actually the window that is used as an overlay on the image and every element of that kernel window has a certain weight. In figure 4.8 is shown how convolution works and, as you can see, it returns an output image with the same dimensions as the input image. The convolution operation is executed for every pixel and the pixel value is now the result of this calculation.

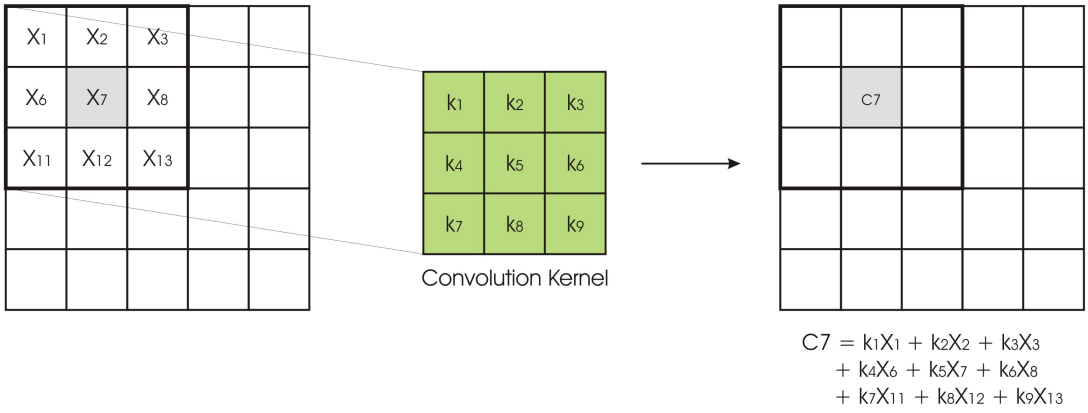

Figure 4.8: The convolution operation

The Moravec detector uses such a convolution kernel as well, but it does not catch the eye as much as the convolution kernel of the Harris detector will. The kernel used by Moravec has binary values, so only pixels within the window are multiplied with a weight of 1.0, and all others are not part of the convolution. The use of binary values and the fact that it uses a square window, makes the Moravec detector to be considered noisy.

Intuitively, variable weights should be assigned to measurements made closer to the center of the window, but up untill now an equal emphasis is put on every measurement, disregarding their distance from the center pixel. A Gaussian weight distribution would

stress the importance of the center pixel more, and lower the influence of the measurements on the border of the convolution kernel. But using a square window disturbs our measurements as well, since we would like to take pixels into account that lie on an equal distance from the kernel center. Because of the square window the Euclidian distance from the center pixel to the kernel edge varies for different directions. A circular window, or a square window simulating circular window values, would help solve this problem.

1	2	1
2	5	2
1	2	1

0.004	0.015	0.026	0.015	0.004
0.015	0.059	0.095	0.059	0.015
0.026	0.095	0.15	0.095	0.026
0.015	0.059	0.095	0.059	0.015
0.004	0.015	0.026	0.015	0.004

Figure 4.9: A simple and more complex example of a 2D Gaussian weighted window

Both these demands call for a *2D Gaussian weighted convolution kernel* that is inherently circular isotropic(see figure 4.9). Although it is still a square, values at the corners are really low, to cause, as the name says, a distribution that applies equal Gaussian weighting in every direction. When using a simple Gaussian kernel as in figure 4.9, the convolution equation out of figure 4.8 becomes

$$C_7 = 1 \cdot x_1 + 2 \cdot x_2 + 1 \cdot x_3 + 2 \cdot x_6 + 5 \cdot x_7 + 2 \cdot x_8 + 1 \cdot x_1 1 + 2 \cdot x_1 2 + 1 \cdot x_1 3 \quad (4.8)$$

and you can see that this computation takes 9 operations. This means that executing such a 2D convolution on a window with dimensions $M \times N$ leads to an algorithmic complexity of $O(N \times M)$.

To implement this step on the GPU, we used the most important property of a 2D circular isotropic kernel. Mathematics learns us that we can split a 2D circularly isotropic kernel into a horizontal 1D convolution followed by a vertical 1D convolution. This will reduce the algorithmic complexity, and thus the number of computations and the processing time, to $O(N+M)$. In GPU terms this means that we will divide the convolution in a horizontal pipeline pass that we use as the input for the vertical pass, as you can see in listing 4.2.

Listing 4.2: Two 1D convolution kernels.

```
technique aLive_HorizontalConvolution
{
    pass P0
    {
        VertexShader = compile vs_3_0 vs_horConvolution(horConv_horTexelStride);
        PixelShader = compile ps_3_0 ps_horConvolution( horConv_weightDistribution,
                                                        horConv_normalisationFactor);
    }
```

```
}

technique aLive_VerticalConvolution
{
    pass P0
    {
        VertexShader = compile vs_3_0 vs_verConvolution(verConv_verTexelStride);
        PixelShader = compile ps_3_0 ps_verConvolution( verConv_weightDistribution,
                                                        verConv_normalisationFactor);
    }
}
```

Besides dividing the 2D kernel in two 1D kernels, we also adopted some speedups from the aLive framework. The latter uses rasterizer exploitation (functions vs_horConvoluion and vs_verConvolution in listing 4.2), texture coordinate interpolation and further division of the kernel to increase processing speed by leveraging the utilization. An in-depth discussion can be found in Lu [28].

4.3.4 Cornerness measure

The cornerness measure for the Harris corner detector starts by performing an analytical expansion of the autocorrelation function used in the Moravec operator [29]. To provide better insight, a short version of the derivation is given. For a shift $(\Delta x, \Delta y)$ and a reference pixel (x, y) the Moravec autocorrelation function is

$$c(x,y) = \sum_{j=-\frac{N}{2}}^{j=+\frac{N}{2}} \sum_{k=-\frac{M}{2}}^{k=+\frac{M}{2}} W\ [I(x_j, y_k) - I(x_j + \Delta x, y_k + \Delta y)]^2 \tag{4.9}$$

with (x_i, y_i) as the points in the Gaussian window centered on (x, y), with the Gauss-function $W = e^{-\frac{x_j^2 + y_k^2}{2\sigma^2}}$ omitted for clarity. The shifted image is approximated by a Taylor expansion truncated to the first order terms

$$I(x_j + \Delta x, y_k + \Delta y) \approx I(x_j, y_k) + \begin{bmatrix} I_x(x_j, y_k) & I_y(x_j, y_k) \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} \tag{4.10}$$

where I_x and I_y are the partial derivatives in x- and y-direction. Now substitution of equation (4.10) in equation (4.9) and further derivation leads us to the matrix notation of equation (4.7) in the section about partial derivatives

$$\begin{aligned} c(x,y) &= \sum_{j=-\frac{N}{2}}^{j=+\frac{N}{2}} \sum_{k=-\frac{M}{2}}^{k=+\frac{M}{2}} W \left(\begin{bmatrix} I_x(x_j, y_k) & I_y(x_j, y_k) \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} \right)^2 \\ &= \begin{bmatrix} \Delta x & \Delta y \end{bmatrix} \begin{bmatrix} \sum\sum_W (I_x(x_j, y_k))^2 & \sum\sum_W I_x(x_j, y_k) I_y(x_j, y_k) \\ \sum\sum_W I_x(x_j, y_k) I_y(x_j, y_k) & \sum\sum_W (I_y(x_j, y_k))^2 \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} \\ &= \begin{bmatrix} \Delta x & \Delta y \end{bmatrix} C(x,y)) \begin{bmatrix} \Delta x & \Delta y \end{bmatrix} \end{aligned} \tag{4.11}$$

and $C(x, y)$ is now the matrix that captures the new local autocorrelation that is as good as rotationally invariant. With this matrix and the matrix' eigenvalues, a cornerness measure is constructed. Harris and Stephens [24] proposed following cornerness measure so the eigenvalue of the matrix $C(x, y)$ could be left out

$$C(x, y) = \begin{bmatrix} D & E \\ E & F \end{bmatrix} \tag{4.12}$$

with a cornerness measure

$$c(x, y) = Det(C) - k \cdots Trace(C)^2 \tag{4.13}$$

where the size of the parameter k has already been studied by Z. Zhang et al. [30], demonstrating that an optimal value for k lies around $k = 0.04$. The other elements of the equation are

$$Det(C) = D + E \tag{4.14}$$
$$Trace(M) = DE - F^2 \tag{4.15}$$

Thanks to all our previous work, calculating the partial derivatives and performing Gaussian weighting on the output images, the only thing that is left to do, is implement equation (4.13) in the graphics pipeline. So a simple pixel shader is written where we use the output of the previous stages (D, E, F) that is stored in temporary textures to combine all of that into an output image that holds a cornerness measure for every pixel. This step is visualized in a slightly altered picture of the corner detector flowchart (figure 4.10).

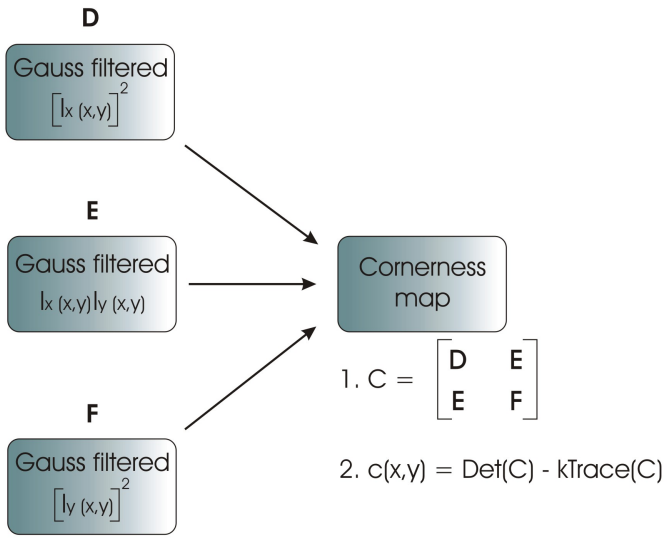

Figure 4.10: Combining all outputs to construct the cornerness measure

4.3.5 Thresholding and non-maximal suppression

There is not that much left to say about the last two steps because they are identical and the Harris corner detector does not need to change them, compared to the Moravec corner detector. So a user-defined threshold is applied on the cornerness map to exclude false corners and then non-maximal suppression is performed to properly localize the true corners. The non-maximal suppression gets the best results when you use a circular perimeter to search for other local maxima, but since we were still building a test version we have not implemented a circular search range, but an easy rectangular one. It will not take long for an experienced GPU programmer to add this functionality to the shader.

4.4 Experimental results

We present the experimental results we got from the Harris corner implementation in a pipelined fashion according to the flowchart model on which we have based our implementation. For every step a didactic image is used that illustrates the functionality of this step to the fullest, because simply showing you dark pictures with a couple of dots, that have a change throughout these steps that is almost unnoticeable with the human eye, would be unsatisfactory. We worked with a model image, that is built out of a couple of entirely black rectangles against a white background, and with a real scene setup, that is also frequently used for stereo correspondence testing, the Tsukuba scene [31] (figure 4.11).

Figure 4.11: The images used for testing the Harris corner detector

As you may have noticed, this corner detector that we have implemented is *a* working version, but it is not yet *the best* working version. But that was not really needed, as the development of this corner detector was an explorative step towards camera calibration and was meant to estimate the amount of work and development time it would take to write our own camera calibration code on graphics hardware. Now that we have already implemented a first step of the camera calibration method, we are closer to determining the approximate time frame that would be needed to construct a full camera calibration application on graphics hardware.

But as we are working with a *fixed camera setup*, we are not strictly compelled to build this entire step. What we really want at the end of the ride, is to obtain preprocessed images that can be used as input for the aLive framework, so we can replace the implemen-

tation of our own graphics hardware accelerated camera calibration with a tool available for free use (Camera calibration tool by Thomas Svoboda [32]). As a consequence we can focus on the further preprocessing of the images, namely distortion correction (chapter 6) and rectification (chapter 7). Yet we have built a corner detector that can be the basis for futher optimization and construction of a very accurate corner detector or even for full camera calibration on the GPU in the years to come. To be complete, we will still summarize the steps that need to be executed to obtain camera calibration, which has its core in matrix computations, in the following chapter.

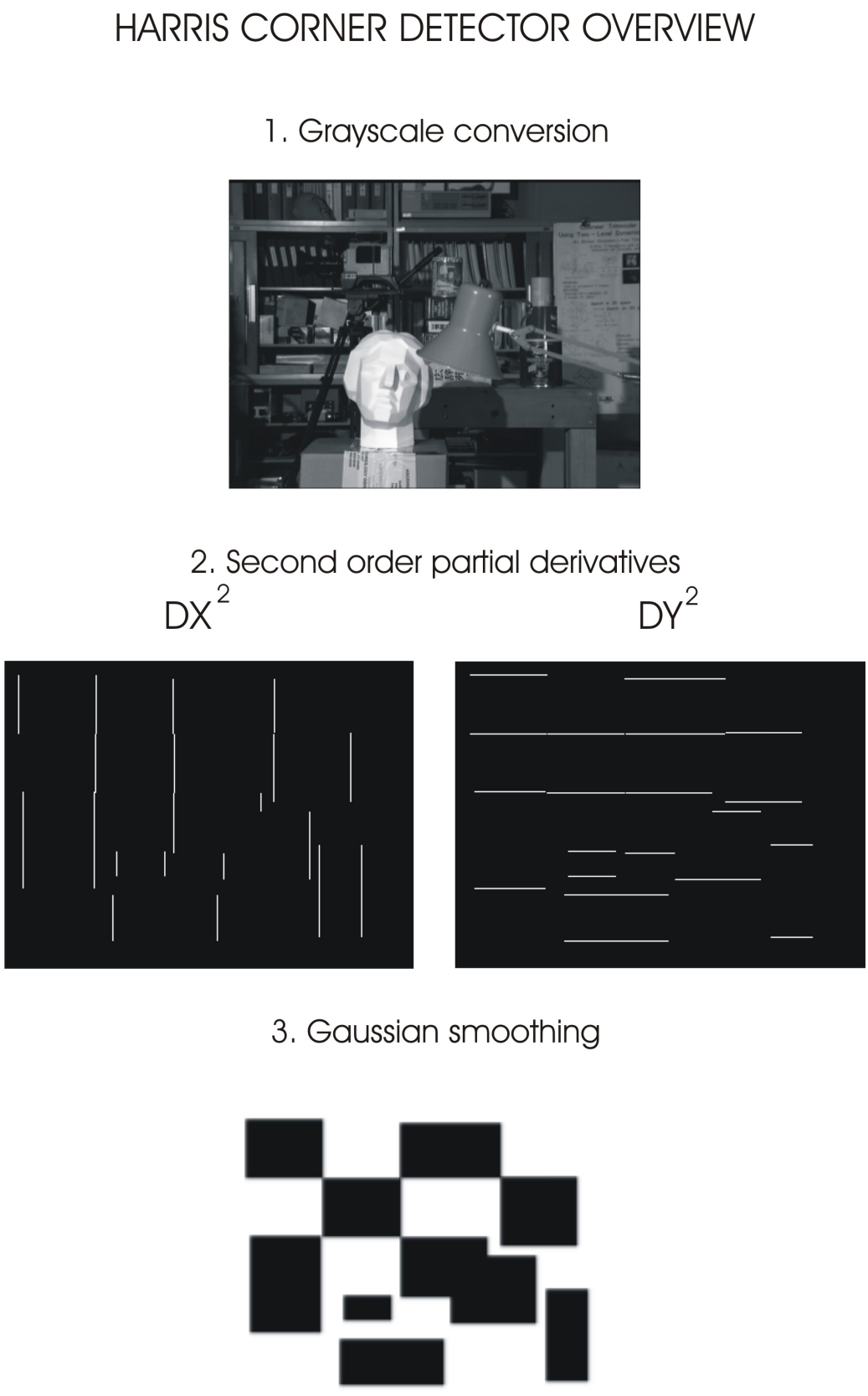

Figure 4.12: First part of the Harris corner detector overview

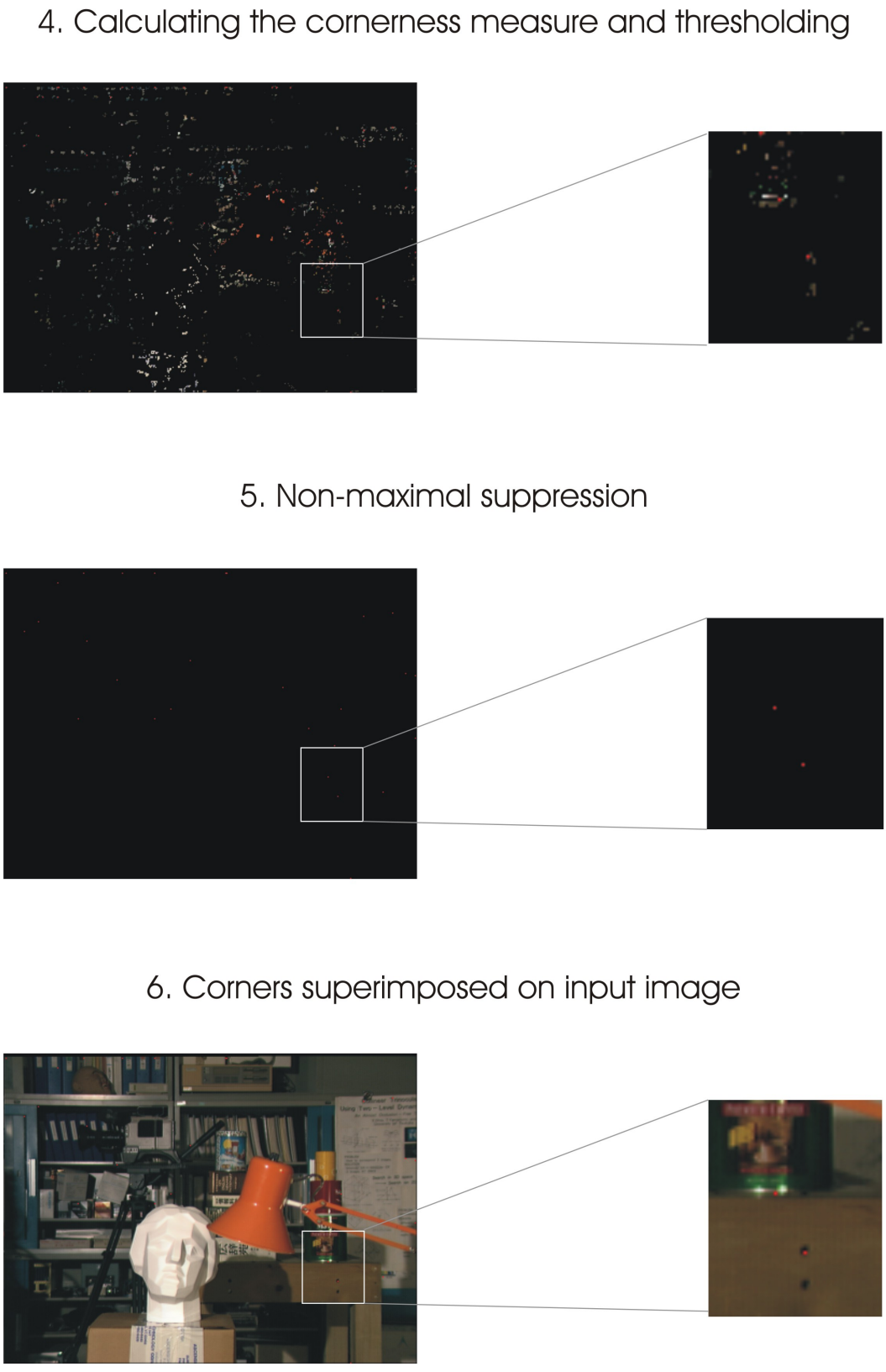

Figure 4.13: Second part of the Harris corner detector overview

Chapter 5

Camera calibration

We speak of a *calibrated camera pair* if the relative rotation and translation of the cameras is known and if the camera properties are known as well, what means that we have to determine the intrinsic and extrinsic camera parameters. Remember from chapter 3 that the camera parameters describe the transformation from 3D space coordinates to 2D image points coordinates. More specific, the extrinsic parameters define the location and orientation of the cameras' reference frames with respect to the world reference frame, and thus their mutual orientation. The intrinsic parameters on the other hand, link the pixel coordinates of an image point to their coordinates in the camera reference frame. Retrieving these parameters out of a given pair of images from an uncalibrated camera pair is our objective in this chapter.

5.1 Calibration model

Their are two ways in achieving camera calibration, either using *photogrammetric* calibration, or using *self*-calibration. Photogrammetric calibration observes a calibration object who's geometry is known with very high precision and the object is usually built out of a couple orthogonal planes. This kind of approach requires a very precise camera setup and expensive calibration equipment [33]. Using self-calibration, often no calibration object is used and the camera calibration is performed using a pair of images from either one moving camera or from a pair of frozen cameras. The first case, with the moving camera is a very flexible approach, but also one of the hardest to implement. In this work, we have chosen an approach that is some sort of a special case of self-calibration, because we planned on using a fixed camera setup in combination with a checkerboard. The latter makes it much easier to detect feasible points of interest, needed in a later calibration stage. This technique is also called *photometric self-calibration* because it uses easily recognizable points, e.g. a red led or a checkerboard. Although we are not using the calibration with a checkerboard, we still use photometric self-calibration, because the Svoboda tool uses a green led as easy interest point for calibration purposes.

As for the camera model that we will be using throughout our explanation, we would like to remind you that we use the finite pinhole camera model, discussed in section 3.1.1. Most important to remember is that the perspective projection matrix P is a

representation of the 3D-2D mapping the camera induces. So for a point Q in 3D space and its corresponding image point q this gives

$$q = PQ \tag{5.1}$$

and P is constructed out of the two matrices we want to construct in this chapter, the intrinsic matrix (K) and the extrinsic matrix ($[R \mid t]$).

$$P = K \left[R \mid t\right] \tag{5.2}$$

5.2 Epipolar geometry

Epipolar geometry describes the intrinsic projective relationship between two different views and is independent of scene structure. It only depends on the cameras' internal parameters and relative pose. This geometry is usually considered when trying to locate corresponding points in different views in a stereo matching application, so we will demonstrate why epipolar geometry plays such a big part in this point correspondence as well as in scene reconstruction.

5.2.1 Terminology

Consider a point Q in world coordinates and its corresponding image points q and q', located on the image planes of the cameras with camera centers C and C' respectively. The line that connects the camera centers is called the *baseline*. When we backproject the rays from image points q and q' into the world coordinate system, they will intersect at point Q. Now both of these rays and the baseline, or the points Q, C and C' form a plane, that will be further referred to as the *epipolar plane* π (see figure 5.1).

The points of intersection of the baseline with the image planes are called *the epipoles* and they are also the projecion of one camera center onto the image plane of the other camera. The epipolar plane π, that we can construct with QCC', intersects the image planes of both cameras as well. These intersections form lines in the image planes and they are called *epipolar lines*. Since the epipoles e and e' lie on the baseline and are thus also a part of plane π, we can also construct the epipolar line l' using the projection of the other camera center C (epipole e' in this camera's image plane), and the projection of the 3D-space point Q (the image point q'). The same works for the epipolar line l.

Varying the position of the world point Q, will change the orientation of the epipolar plane π and the epipolar lines l and l'. But since every epipolar plane will always include the baseline, every epipolar line will go through the image plane's epipole. So the epipole can also be described as the intersection of all existing epipolar lines. Since the epipoles are also the projection of the camera centers onto the opposing image plane, varying the position and orientation of the cameras and there image planes will change the position of the epipoles.

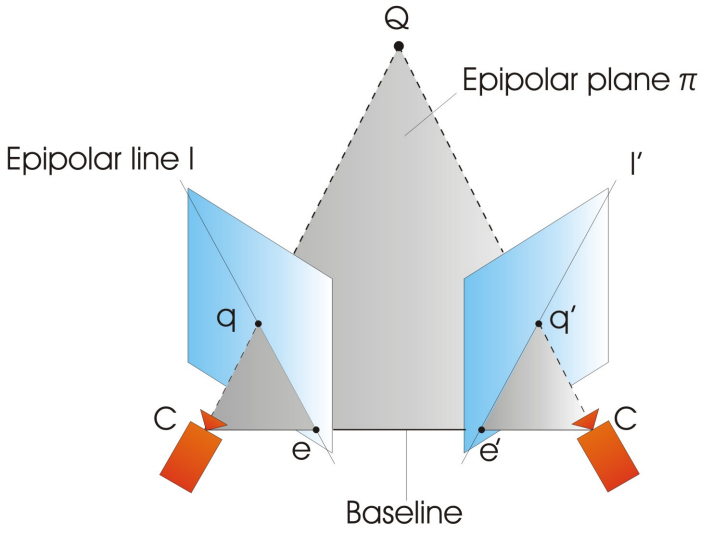

Figure 5.1: The epipolar plane π

5.2.2 The epipolar constraint

Now imagine that only the image point q and the camera centers C and C' are given, and we want to determine the position of the corresponding image point q' (see figure 5.2). We can backproject the ray from the known image point into world coordinate space, using the line that connects camera center C and image point q. The world point Q must be on this line, since it was a ray from Q going through C that constructed the image point q in the first place. Projecting this ray on the image plane of the other camera, gives us the line l', which is an epipolar line and goes through the epipole e'. Because l' is the projection of the ray that contains Q, we know for sure that the corresponding image point q' lies somewhere on that line.

Approaching this from a different point of view, returns the same result. With C, C' and q known, we can construct the baseline and the epipolar plane. Since the corresponding image point q' is always a part of the epipolar plane, we know that q' must lie somewhere on the intersection line l' of the epipolar plane and the image plane of the other camera. This mapping can be described as

$$q \rightarrow l' \tag{5.3}$$

and is a point-to-line mapping. The fact that we can only determine the line that contains the corresponding image point using epipolar geometry, but not the precise location, is called the *epipolar constraint*. Although 'constraint' might have a negative connotation, try not to see this as a bad thing, because thanks to this correlation, we can narrow the search for corresponding points in a stereo matching application to a single line l' instead of the entire image. Because this mapping can only be calculated for a pair of calibrated cameras, the need for camera calibration is clear.

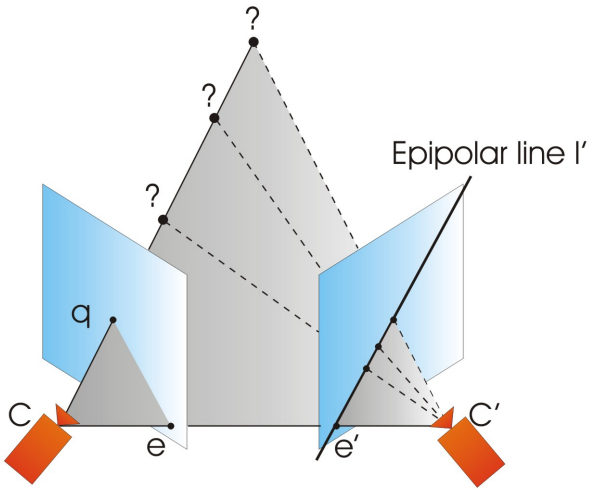

Figure 5.2: The epipolar constraint

5.3 Fundamental matrix

The fundamental matrix is used to describe the mapping from a point to a line in a stereo image pair, which we have discussed in the previous section. The matrix holds intrinsic and extrinsic information of the cameras [34]. You could also say that the fundamental matrix is the algebraic representation of epipolar geometry. This leads to the transformation of equation (5.3) into

$$l' = Fq \tag{5.4}$$

with F a 3×3 matrix

$$F = \begin{bmatrix} f_{11} & f_{12} & f_{13} \\ f_{21} & f_{22} & f_{23} \\ f_{31} & f_{32} & f_{33} \end{bmatrix} \tag{5.5}$$

and since the mapping from the corresponding image point q' back to the original image plane, results in a transfer to the epipolar line as well, we can write

$$l = F^T q' \tag{5.6}$$

We know that the image point q' is a point on the epipolar line l', which results in

$$q'^T l' = 0 \tag{5.7}$$

So when we combine equation (5.4) and (5.7) we get

$$q'^T l' = q'^T F q = 0 \tag{5.8}$$

for any pair of matching points q and q' in two image planes. The result of equation (5.8) is important for the determination of the fundamental matrix, since this gives us a

way of characterizing the F-matrix in terms of image points, without the need of camera matrices. So equation (5.8) proves that we can compute the fundamental matrix out of image points alone.

Now, if we define q as $\begin{bmatrix} x & y & 1 \end{bmatrix}^T$ and q' as $\begin{bmatrix} x' & y' & 1 \end{bmatrix}^T$, each corresponding image pair will give us one linear equation expressed in the unknown terms of the fundamental matrix F. For q and q' this gives us the linear equation

$$x'xf_{11} + x'yf_{12} + x'f_{13} + y'xf_{21} + y'yf_{22} + y'f_{23} + xf_{31} + yf_{32} + f_{33} = 0 \tag{5.9}$$

So for a set of n corresponding image points we can expand equation (5.9) to a set of linear equations

$$\begin{bmatrix} x'_1x_1 & x'_1y_1 & x'_1 & y'_1x_1 & y'_1y_1 & y'_1 & x_1 & y_1 & 1 \\ x'_2x_2 & x'_2y_2 & x'_2 & y'_2x_2 & y'_2y_2 & y'_2 & x_2 & y_2 & 1 \\ x'_3x_3 & x'_3y_3 & x'_3 & y'_3x_3 & y'_3y_3 & y'_3 & x_3 & y_3 & 1 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ x'_nx_n & x'_ny_n & x'_n & y'_nx_n & y'_ny_n & y'_n & x_n & y_n & 1 \end{bmatrix}_{(n\times 9)} \begin{bmatrix} f_{11} \\ f_{12} \\ f_{13} \\ \vdots \\ f_{33} \end{bmatrix}_{(9\times 1)} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}_{(n\times 1)} \tag{5.10}$$

with f the 9-vector made up of the entries of F in row-major order.

To calculate the elements of the fundamental matrix F, we would need at least nine linear equations, since F has nine unknown variables. But their are two other restrictions that reduces the minimal number of linear equations to seven. These restrictions come down to the fact that F is scalable and that F has to be of rank 2 [34]. So a minimum number of seven corresponding images points is essential to be able to calculate a fundamental matrix.

5.4 Image points correspondence

To be able to perform further calculations that will lead us to the extraction of the necessary camera parameters, or even just to compute the fundamental matrix, we need sets of corresponding image points, as mentioned in the previous section. That means that the interest points that were originally detected by our Harris corner detection algorithm in each image separately, now have to be matched or found across a pair of given images, assumed that the cameras are capturing the same scene and point matching is possible. To illustrate this, let us take a look at the glass example of figure 5.3.

When the edge of the glass is detected as an interest points q_0 and q'_0 in the left and the right image respectively, we want these two interest points, one in each image, to be written down as a pair of corresponding image points (q_0, q'_0), because they originate from a light ray coming from the same position on the 3D object.

The method to determine these pairs of corresponding image points, looks a lot like the correlation method of our Harris corner detector, explained in chapter 4. That is why we will not go into much detail on this topic, as there are many other concepts, necessary

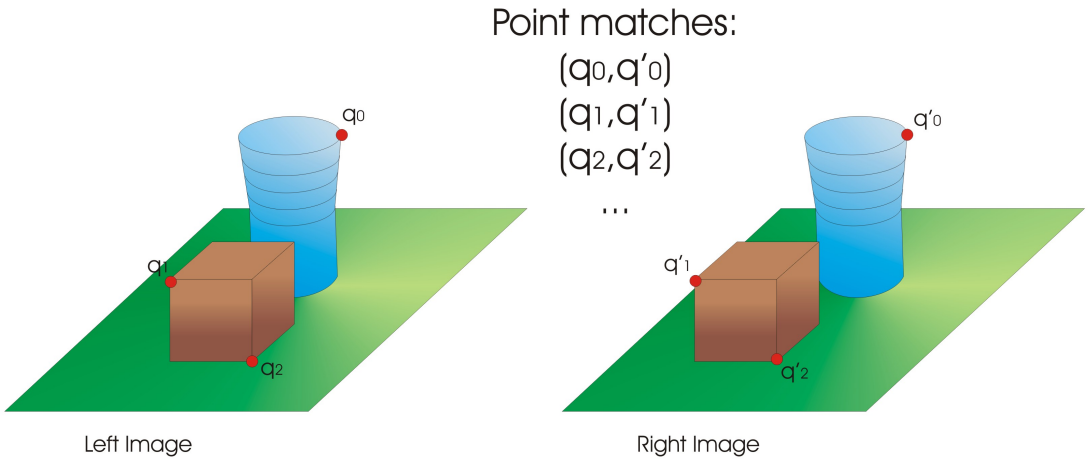

Figure 5.3: Example of corresponding image points

to get the cameras calibrated, that are yet left untouched. It basically comes down to the following; In the reference image, an interest point is selected and a local window is placed on top of it, with the interest pixel located in the center. Now, a measurement is made to have a reference that gives us an idea of the intensity variation in that local window or the intensity of the different pixels that are part of this window.

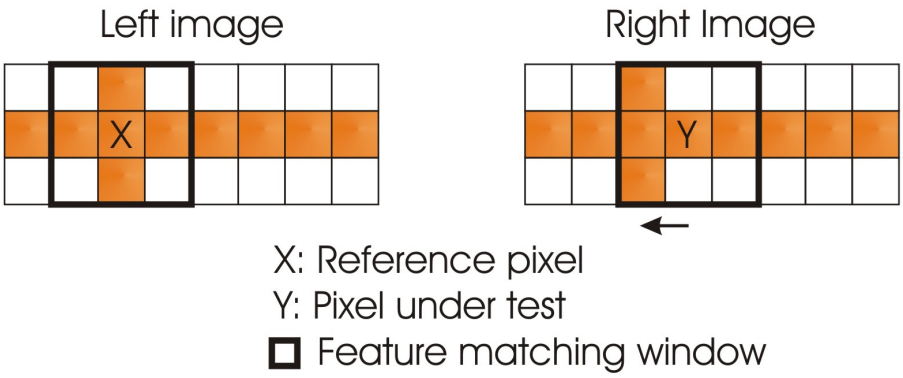

Figure 5.4: An example of point matching

Then, we place the same window and perform the same measurement on each and every interest point in the other image. Every measurement is compared to the reference measurement, and only when the correlation between the reference point and a point in the other image is the highest of all interest points in that image, we can accept it as *a match*. These two points are now almost considered a corresponding image pair. But first we reverse the process and apply the same steps starting in the other image, and only if the interest point, that was detected as a match by the first step, detects the original reference point as a good match, we agree that a pair of corresponding image points is found. In figure 5.4, you can see an example of a successful match and a mismatch. Obviously the algorithm is a little more advanced as is stated here, but with this general concept in your mind and the knowledge you have gained frome the Harris corner detector, you should

have no trouble reading some more advanced literature on this topic if necessary.

5.5 Calculating the fundamental matrix

5.5.1 The eight-point algorithm

Calculation of the fundamental matrix can be done in several ways. You could use a geometric approach, like the Gold Standard method or the Sampson distance [34], but we are going to use a relatively simple way of calculating the F-matrix, the algebraic eight-point algorithm [35]. It is a linear way of estimating a fundamental matrix, given a set of point correspondences. It takes off by rewriting equation (5.10) into

$$\begin{bmatrix} x'_1x_1 & x'_1y_1 & x'_1 & y'_1x_1 & y'_1y_1 & y'_1 & x_1 & y_1 & 1 \\ x'_2x_2 & x'_2y_2 & x'_2 & y'_2x_2 & y'_2y_2 & y'_2 & x_2 & y_2 & 1 \\ x'_3x_3 & x'_3y_3 & x'_3 & y'_3x_3 & y'_3y_3 & y'_3 & x_3 & y_3 & 1 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ x'_nx_n & x'_ny_n & x'_n & y'_nx_n & y'_ny_n & y'_n & x_n & y_n & 1 \end{bmatrix} \begin{bmatrix} f_{11} \\ f_{12} \\ f_{13} \\ f_{21} \\ f_{22} \\ f_{23} \\ f_{31} \\ f_{32} \\ f_{33} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \tag{5.11}$$

or even shorter

$$Af = 0 \tag{5.12}$$

So the matrix A holds the set of linear equations necessary to compute the elements of the 9-vector f, that are also the elements of the fundamental matrix. It is a homogenous set of equations, and since f can only be determined up to a scale factor, matrix A has to be at least of rank 8. In practice, this means that at least eight pair of correspondent points have to be found to form a matrix A of rank 8 and make it possible to compute f.

In the previous section, we have explained that there is a second constraint that would allow us to compute the F-matrix with only seven pairs of corresponding image points. This method will prove to be more complex than the eight-point algorithm and would even lead to 3 different fundamental matrices in some cases. But right now we are not looking for a way to compute the F-matrix with the minimum amount of corresponding image pairs. As you will see in section 5.6, we rather want an easy way to compute a fundamental matrix that can be optimized further and we will use a whole bunch of corresponding points, so there is no need to limit this number to seven. Thanks to the simple start and possible further optimization using an algorithm called *Ransac*, we can construct a fundamental matrix that is not dependant of a small number of point matches. Through optimization, the constructed matrix is the best fit for all feature points, so there is room for a little error in the feature matching computations as well.

After determining the set of eight linear equations, the computation of the F-matrix is quite straightforward. The solution for f out of equation (5.12) is found using the Singular Value Decomposition (SVD) of A

$$A = UDV^T \tag{5.13}$$

and f is now the singular vector corresponding to the smallest singular value of A, that is the last column of V if the diagonal elements of D are ranked in descending order. Simply reorganizing the elements of f in a 3×3 matrix gives the desired F-matrix.

These simple steps, setting up the linear equations and performing singular value decomposition, form the basis of the entire eight-point algorithm. In sections 5.5.2 and 5.5.3, we will make some slight adjustments to this method in order to obtain a robust and trustworthy algorithm.

5.5.2 Constraint enforcement

One of the key properties of the fundamental matrix is that it has to be of rank 2. A lot of applications rely on this property, for instance if the calculated F-matrix would not be of rank 2, this would cause for the epipolar lines not to go through the same point, in which case our epipolar geometry is seriously disturbed because all epipolar lines should cross the same point, the epipole. And the problem is, that solving the set of linear equations from (5.10), will generally not cause the fundamental matrix to be of rank 2, so we should undertake action to enforce this constraint on the calculated F-matrix. The matrix F will be replaced by another matrix F' that is satisfactory to the rank 2 constraint. Mathematically this problem is described as

$$\textit{Minimize the Frobenius norm } \|F - F'\| \textit{ subject to } rank(F') = 2 \tag{5.14}$$

Using singular value decomposition, this problem is not too hard to solve. The SVD of F is

$$F = UDV^T \tag{5.15}$$

with $d_1 \geq d_2 \geq d_3$ elements of

$$D = \begin{bmatrix} d_1 & & \\ & d_2 & \\ & & d_3 \end{bmatrix} \tag{5.16}$$

Then the F' we are looking for is

$$F' = U \begin{bmatrix} d_1 & & \\ & d_2 & \\ & & 0 \end{bmatrix} V^T \tag{5.17}$$

F' is now a fundamental matrix that is still a solution of the set of linear equations and is of rank 2 as well, so the constraint has been properly enforced.

5.5.3 The normalized eight-point algorithm

Up till now we have described the main principles and computations of the eight-point algorithm. But the key to success, to a robust and noise resisting algorithm, is actually proper normalization of the input data [35], even before we construct the linear equations. In the case of our algorithm, the proposed normalization, represented by T for the reference image and T' for the corresponding image, performs two simple steps:

- The image points are *translated* in such a way that the center of the collection is at the origin of the coordinate system.
- The points are *scaled* as well, to make the RMS distance from the reference points to the origin equal $\sqrt{2}$.

Knowing all this, we can summarize the entire eight-point algorithm in a couple of simple steps. The goal remains to calculate a matrix F out of a set of at least 8 pair of corresponding points, so $x_i'^T F x_i = 0$. The steps we have to take to accomplish this goal are

1. *Normalization*: with T and T' the normalization transformations described result in $\hat{x}_i = Tx_i$ and $\hat{x'}_i = Tx'_i$. Out of these sets of normalized corresponding points $(\hat{x}_i, \hat{x'}_i)$, we will calculate the fundamental matrix.
2. *Linear solution*: calulate a matrix $\hat{F}$ out of the linear equations of $\hat{A}$ as described in equation (5.13).
3. *Constraint enforcement*: compute a new $\hat{F}'$ to fulfill the demand $\text{rank}(\hat{F}') = 2$ using the SVD of the original $\hat{F}$ as in equation (5.15)
4. *Denormalize* the solution using $F = T'\hat{F}'T$. The resulting F is the fundamental matrix that corresponds to the set of linear equations formed with the original pairs of corresponding points (x_i, x'_i).

5.6 RANSAC: RANdom SAmple Consensus

Up to this point, we have assumed that the set of corresponding points that was presented was only susceptible to errors in the measurement of these points' position, induced by the interest point detection, the Harris corner detector in our case. But in reality, there will be some point mismatches as well, so some points in one image will be linked to false correspondence points in the other image as you can see in figure 5.5, where the arrows show the shift from one image to another. In an ideal situation, the arrows should far more parallel to each other.

We call these points *outliers*, as they are data points that do not fit the model of image correspondence. You could also say that they are measurements following a different, unmodelled error distribution. Now we would like an algorithm that can estimate for instance the homography between the sets of corresponding points using only the *inliers*,

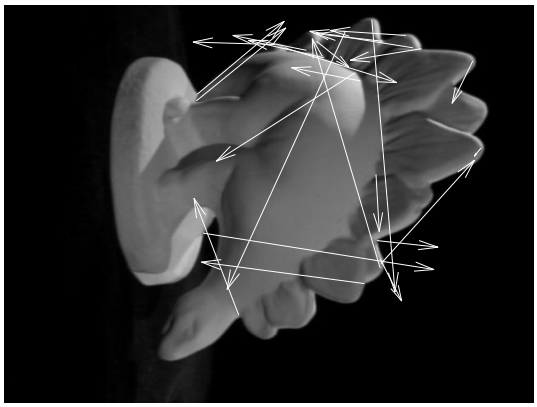

Figure 5.5: Example of erronous point matching

or correct point matches in this case, and thus perform a calculation that is robust against outliers. Inliers can be generally described as data points which can be explained by the model or data points that fit the model.

5.6.1 The RANSAC algorithm

The RANSAC algorithm is such a robust way of fitting data to a model in the optimal way, despite of some outliers. There are simpler algorithms like case deletion to calulate a robust model fit, if the amount of outliers is really low. But one of the special abilities of RANSAC is that it is able to cope with a very large proportion of outliers [36]. The main difference with conventional smoothing techniques, is that instead of trying to use as much data as possible to construct an initial solution and gradually eliminate false data, RANSAC uses a small initial data set and enlarges it with consistent data when possible. Why conventional smoothing techniques cannot cope with outliers is illustrated in figure 5.6.

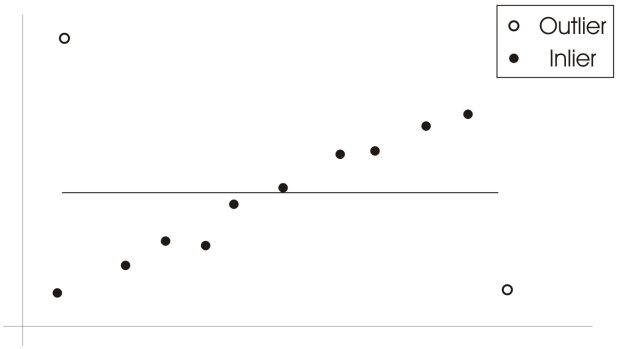

Figure 5.6: The outliers influence the least squares fit a great deal

First, RANSAC selects a random sample v out of the given set of data V with outliers present, so $v \subset V$. This random sample is the minimal subset needed to determine the model (for instance, to draw a line, you only need two data points). The points of this sample v are now called hypothetical inliers and a model is fitted to these points, what

means that all free parameters of the model are constructed using the data points of the sample set v. After the initial solution to the model is determined, we determine which other data points, that were not already in the subset, lie within a distance threshold d_T from the model. So for every point in the remaining data set, RANSAC tests how well the point fits to the constructed model. If it fits well, so it lies within the threshold d_T, then this point is classified as a hypothetical inlier as well. All these points together, the sample data and the ones within the threshold, form the consensus set S and define the inliers of the given data set V. If we have a high number of inliers, we can say that we have a reasonably good model. If the number of inliers is too low, relative to a threshold I_T, the model is discarded.

However, the constructed model was estimated using only the sample data set v, so now we reestimate the model using all the hypothetical inliers of the consensus set S and at the same time, we estimate the error of the inliers relative to the model. These steps are iterated a fixed number of times, each time producing either a model that is rejected, because it has a number of inliers below a threshold I_T, or a model which is then refined using the consensus set S together with an error measure. If this error measure is lower than our previous *best* model, the new model becomes the best model and the old one is discarded.

The distance threshold d_T, the threshold for the number of inliers I_T and the number of iterations necessary to find the optimal solution, either have to be determined empirically or are subject to some statistical formulas [34], which we will not further explore in this work. Most important to remember is that RANSAC is able to do robust estimation of model parameters and that it can estimate the parameters with a high degree of accuracy even when outliers are present in the data set.

There are some alternatives to RANSAC as well, and they give an alternative score to the model constructed from a number of data points. Instead of defining the score on the number of data points within a certain threshold distance, we can score the model by the median of distances from the origin to all points in the data. The model with the lowest median is then selected and the method that uses this is called the Least Median Squares (LMS). The advantage of LMS is that it needs no knowledge of the error statistics or any input thresholds, but it has as disadvantage that the number of outliers cannot be greater than 50%, otherwise the median itself will be and outlier. We will not discuss this method any further as it is treated in the Master's thesis of Gabriels [37] to a much greater extent.

5.6.2 A simple RANSAC illustration

To illustrate the theoretical approach that was given in the previous section, we will apply RANSAC to find the best approximation of a line through a set of points. The collection of points is given in figure 5.7, where black dots represent inliers and the hollow dots are the outliers. Now, choosing a minimal random sample (only 2 data points needed), you can immediately see that drawing a line through (a, b) will get much more support than drawing a line through (c, d). So the latter will never be chosen as the optimal case.

Once two points have been chosen and a line is drawn, for instance (a, f) as in figure 5.8a, we check how many other points fit this model, or in other words, how many points

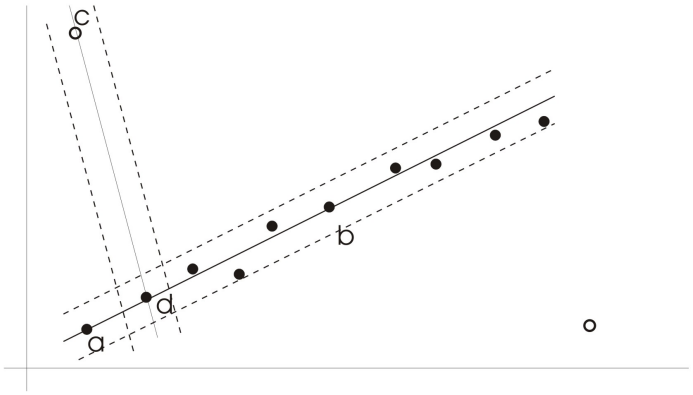

Figure 5.7: the 2D RANSAC example

lie within a distance threshold from this line. In this case, four other points can be defined as inliers. The next step is now to refine our model using all the inliers, which leads to the line in figure 5.8b. Depending on how big the estimated error measurement is compared to our previous best line, we will keep this line or discard it. These steps will be reiterated for a predefined number of times to produce the optimal line or the line with the most inliers or points within the threshold distance.

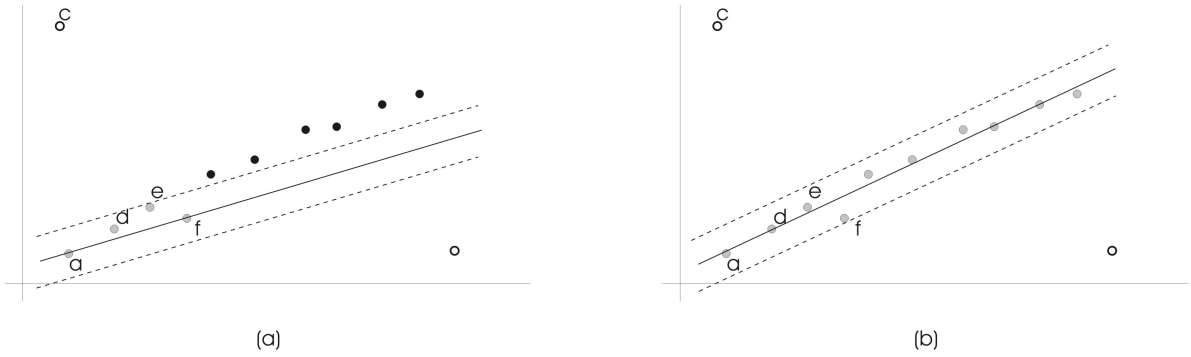

Figure 5.8: The initial sample (a) and the refined sample set (b)

5.6.3 Computing an optimal F-matrix with RANSAC

As we have mentioned earlier, using RANSAC to calculate a fundamental matrix is necessary because we cannot suppose that no mistake was made during point matching. Certain pairs of corresponding points are falsely linked and are therefore considered as outliers of our model. RANSAC fits neatly in the entire approach of the fundamental matrix as you can see in figure 5.9. The algorithm to compute the F-matrix takes two images and the accompanying set of corresponding image points as input, and outputs the fundamental matrix. The distance measure used in figure 5.9, calculates how closely a matched pair of points satisfies the epipolar geometry, given a current estimate of F.

Remember that, in section 5.3 we derived

$$x_i'^T F x_i = 0 \tag{5.18}$$

so it can be easily measured how closely the estimated F-matrix fits this relationship. When the error in the result is too big, and thereby exceeding a predefined threshold, the pair of corresponding points is considered an outlier, otherwise an inlier. The other steps in the flowchart should be clear, considering the previous sections that we have discussed.

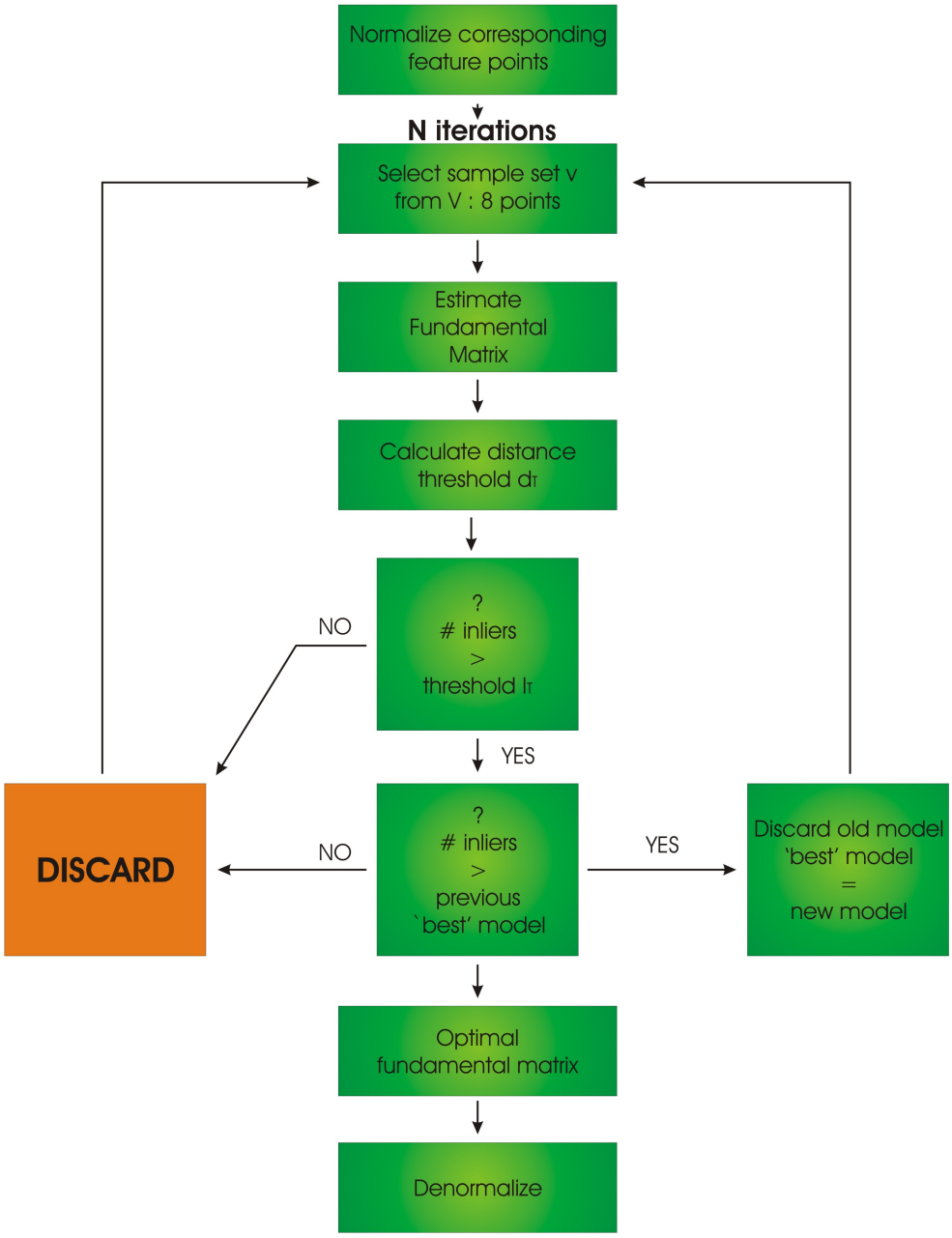

Figure 5.9: The Ransac flowchart

5.7 Extraction of the camera parameters

The calculation of the camera matrices has to be performed in two steps, in cooperation with the fundamental matrix we computed in the previous section. First we calculate the intrinsic camera matrix K and then, using K and the fundamental matrix, we can extract the extrinsic camera matrix.

5.7.1 The intrinsic matrix

The matrix K contains the intrinsic parameters, such as the focal length, the position of the principal point and others (see chapter 3).

$$K = \begin{bmatrix} f & & p_x \\ & f & p_y \\ & & 1 \end{bmatrix} \tag{5.19}$$

For the computation of this matrix out of the fundamental matrix and a given pair of images, we have to make a couple of assumptions, otherwise it would not be possible [38]:

1. The camera plane has to be rectangular, which is the case in most modern cameras.
2. The principal point has to lie at the center of the camera plane. So both p_x and p_y lie at the half of the number of pixels that make up the camera plane in the principal directions.

Thanks to these assumptions, the problem of estimating the intrinsic parameters is now reduced to finding the focal length f. In [39], a formula for f is given

$$f = \sqrt{-\frac{p'^T[e']_x IFpp^T F^T p'}{p'^T[e']_x IFIF^T p'}} \tag{5.20}$$

where F is the fundamental matrix, I is the identical matrix, p and p' are the principal points in both images, constructed like this

$$I = \begin{bmatrix} 1 & & \\ & 1 & \\ & & 1 \end{bmatrix} \qquad p = \begin{bmatrix} p_x \\ p_y \\ 1 \end{bmatrix} \qquad p' = \begin{bmatrix} p'_x \\ p'_y \\ 1 \end{bmatrix} \tag{5.21}$$

and with the epipole e' as $[e_1\ e_2\ e_3]^T$ that can be written as

$$[e']_x = \begin{bmatrix} 0 & -e'_3 & e'_2 \\ e'_3 & 0 & -e'_1 \\ -e'_2 & e'_1 & 0 \end{bmatrix} \tag{5.22}$$

The epipoles e and e' can be found with the help of the fundamental matrix. Since e is on the epipolare line for which $l' = Fq$ for every point q in an image, we can state

that $Fe = 0$. So e can now be computed by applying singular value decomposition on the fundamental matrix. When $F = UDV^T$ is calculated, e is the column of V that represents the smallest singular value decomposition. The computation of e' is analogue to this one, only now we calculate the SVD of the matrix F^T.

Thus, we have found a way to compute the focal length f of the intrinsic matrix K and as a consequence, we know all intrinsic camera parameters. The computation of the focal length of the second camera is performed with almost the same formula, but the principal points and epipoles in the formula that used to refer to one camera, now refer to the other and vice versa.

5.7.2 The extrinsic matrix

Given the fundamental matrix F and the intrinsic matrix K, it is now possible to compute the extrinsic camera parameters. As you may or may not remember, these parameters define the rotation and translation of the cameras in reference to the world coordinate frame. Quite obviously, they exist of a 3×3 rotation matrix R and a 3×1 translation vector t.

To calculate the extrinsic parameters, we need the *essential matrix* E, which is in fact a specialized version of the fundamental matrix. But, opposite to the fundamental matrix, it only includes information about the rotation and the translation of the image planes, so it is not influenced by the intrinsic matrix. If the intrinsic matrix K is known, and let q be an image point, then we can obtain the point $\hat{q} = K^{-1}q$. We can construct that same point $\hat{q}$ like this: $\hat{q} = [R \mid t]Q$ and we call this a point expressed in *normalized coordinates*. Using normalized coordinates we can state, just like for the fundamental matrix (equation (5.8)), for normalized corresponding image points $(\hat{q}_i, \hat{q'}_i)$

$$\hat{q'}_i^T E \hat{q}_i = 0 \tag{5.23}$$

proving that the essential matrix is in fact very related to the fundamental matrix, but it does not take the camera properties into account. Computing the essential matrix can be done, using the fundamental matrix and the intrinsic matrix that we have calculated in the previous sections. The relationship between these three goes like this

$$E = K'^T F K \tag{5.24}$$

Once we have the essential matrix, the extrinsic parameters can be extracted. We can choose to have one of the camera reference frames coincide with the world reference, so the normalized perspective projection matrices become

$$P = [I \mid 0] \tag{5.25}$$

and

$$P' = [R \mid t] \tag{5.26}$$

Using the singular value decomposition of the essential matrix, we can compute the projection matrix P'. When the SVD of E equals

$$E = U \begin{bmatrix} 1 & & \\ & 1 & \\ & & 0 \end{bmatrix} V^T \tag{5.27}$$

there are four possible solutions for P'. For the rotation matrix, two solutions are available

1. $R = UWV^T$
2. $R = UW^TV^T$

and there are two options for the translation matrix as well

1. $t = u_3$
2. $t = -u_3$

with u_3 as the last column of the matrix U and

$$W = \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{5.28}$$

That leaves us with four possible combinations for $P' = [R \mid t]$, namely

1. $P' = [UWV^T \mid +u_3]$
2. $P' = [UWV^T \mid -u_3]$
3. $P' = [UW^TV^T \mid -u_3]$
4. $P' = [UW^TV^T \mid -u_3]$

As these four propositions are the algebraic representation of four different geometric camera setups, we have to determine the one that reflects reality. You can see in figure 5.10 that there is only one case where the object is in front of both cameras, thus this is the correct case and the rotation and translation parameters that are linked with this situation are the ones we are looking for. You can find the right geometric situation by selecting the case for which the reconstructed 3D point Q has a positive depth for both cameras. Reconstructing point Q is no longer an obstacle since we have obtained the intrinsic and extrinsic parameters and image points are related to their corresponding world points through $q = K[R \mid t]Q$.

Even though it required some serious wanderings to achieve our goal, we have obtained a method to compute the extrinsic and intrinsic camera parameters starting from a pair of images. So thanks to the epipolar geometry, feature point correspondences, the fundamental matrix, the RANSAC optimization and the essential matrix, we can call our pair of cameras 'calibrated'.

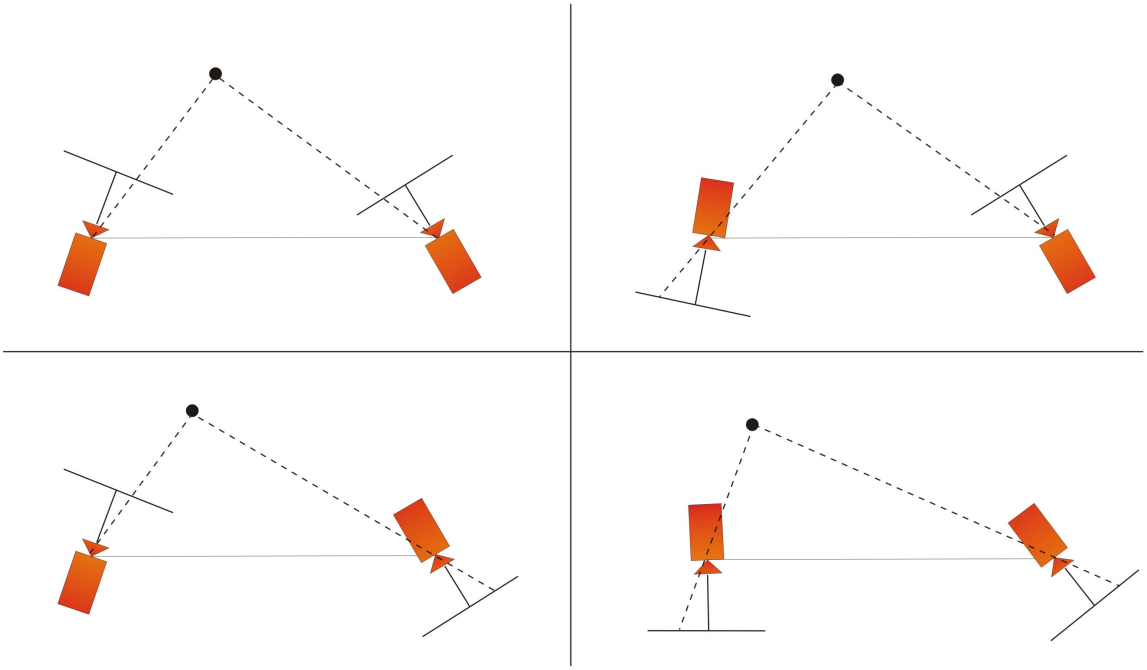

Figure 5.10: The four possible camera orientations for calibrated reconstruction from E

Chapter 6

Distortion correction

Distortion of images occurs through the use of real lenses instead of the ideal situation we are used to work with. They cause radial and tangential distortions, that have to be filtered out, because if the images are not a good representation of reality, further image processing becomes rather difficult. Solving this problem goes over two different paths, either the straightforward way or the reverse approach. Our job is to choose the option that is best suited for a GPU implementation. The GPU pipeline has a different approach on working with input images, so we have to adapt our algorithms accordingly. On this lower level of programming, some pitfalls like erroneous texture sampling need to be avoided. Without the use of any filtering technique like bilinear interpolation, the images will not return the demanded quality expected from a distortion correction algorithm.

6.1 Lens effects

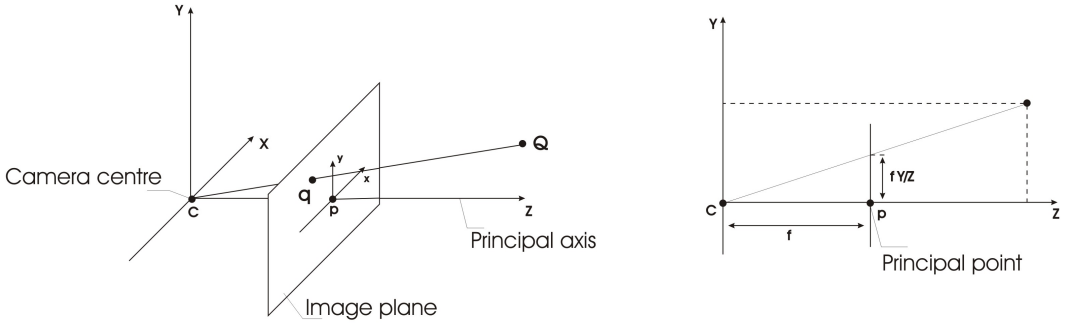

Figure 6.1: The pinhole camera model

Throughout the previous sections we have always assumed that the camera model we use, is an ideal representation of the image capturing process. The pinhole model is a linear model and therefore it can only model linear effects, so it assumes that the transformation from world coordinates to image coordinates is perfectly linear. Considering the geometric situation we have used in section 3.1.1 (repeated in figure 6.1), this model expects the world point, the image point and the camera center to be collinear and it also expects that

lines in world coordinates remain lines in the image coordinate system. A real camera differs from the pinhole model in several ways.

A real camera does not use a simple hole as the point of convergence of incoming light rays, instead it uses real lenses. Due to various constraints in the lens manufacturing process, we experience non-linear deformations or distortions that make the pinhole model inaccurate during image grabbing with real cameras (figure 6.2).

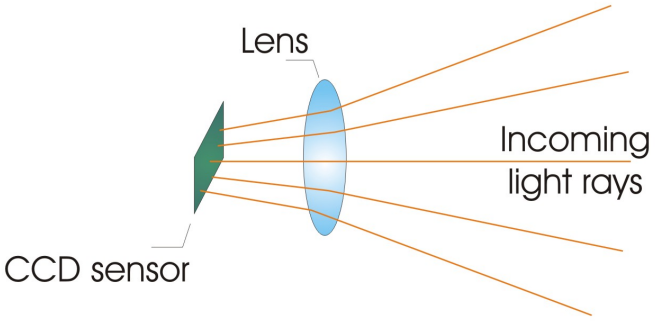

Figure 6.2: Deformation of the incoming light rays through real lenses

Among these lens effects are tangential distortions and radial distortions, both part of the *Brown-Conrady Model* [40], but the most important and well-known non-linear distortion is *radial distortion*. It is called *radial* because it is radially symmetric and this lens effect causes straight lines in the real world to be depicted as curves after the image transformation. In general, radial distortion can be defined as an alteration in magnification from the center of distortion to any point in the image, measured in radial direction from that center. Usually, there is no tangential distortion present, thus the center of distortion is located at the image center, where, most of the time, the principal point lies as well. The magnitude of distortion is proportioned to the distance from the image center. That is exactly why otherwise straight edges are curved the most near the edge of the image, as you can see in figure 6.3.

Figure 6.3: An obvious example of radial distortion

Depending on the kind of magnification we are experiencing, there are two categories of radial distortion [41]. If your view on the world is enlarged near the center of the image,

we call it *barrel distortion*. If, on the other hand, the image has shrunk near the image center, we speak of a lens effect called *pincushion distortion*. As pincushion distortion is very rare for our type of cameras, we will be handling barrel distortion. Both cases and the corrected image are illustrated in figure 6.4. We do not like a wrong representation of reality and neither do the DIBR algorithms since they will not produce correct output images that relate to the real world, if their input image do not give a true representation of reality. That is why these lens effects, that are inavoidable, have to be filtered and corrected before any stereo matching image processing is possible.

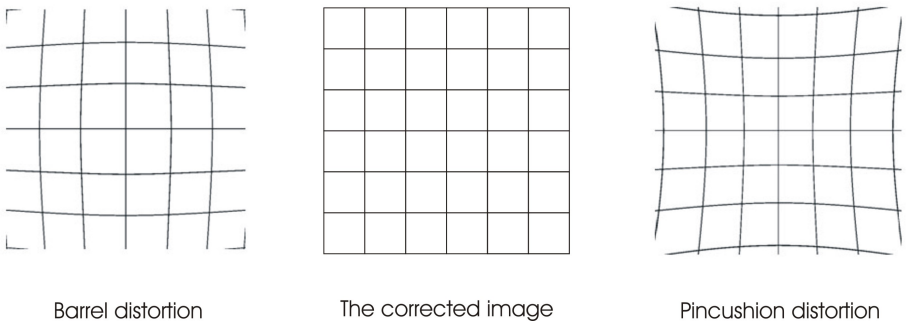

Figure 6.4: Barrel and pincushion distortion

To solve this problem, we will correct the captured images to make it seem, for other units that take the images as their input, as if the images were taken with a linear device. The result we are aiming for has already been illustrated in figure 6.4.

6.2 Distortion correction

Before we think about correcting the distortion effects, we first have to consider when and where in the imaging process we have to correct them. Since the effect takes place in the lens, mathematically we should perform distortion correction just after the initial projection from world coordinates to image coordinates. So the influence of the intrinsic camera parameters should not be taken into account. From equation (3.9) and (3.10), we can write for a 3D space point Q, that it is related to its corresponding (non-distorted) image point q_u as

$$q_u = \begin{bmatrix} x_u \\ y_u \\ 1 \end{bmatrix} = [R \mid t]Q \tag{6.1}$$

where x_u and y_u are the notation of the ideal,undistorted image coordinates without the influence of the intrinsic K-matrix. The actual projected image coordinates are related to the ideal image coordinates by a radial deformation, so the radial distortion can be modelled as

$$\begin{bmatrix} x_d \\ y_d \end{bmatrix} = L(r) \begin{bmatrix} x_u \\ y_u \end{bmatrix} \tag{6.2}$$

In this equation, is r the radial distance, measured out of $r^2 = x_u^2 + y_u^2$ from the center of radial distortion. $L(r)$ is a function that models the radial distortion, and, as this effect is non-linear, it is typically modeled using a Taylor expansion. It can be shown that only the even terms play a role of importance in this expansion [33], so the distortion approximation becomes

$$L(r) = 1 + \kappa_1 r^2 + \kappa_2 r^4 + \kappa_3 r^6 + \dots \tag{6.3}$$

It is obvious that only the low-order terms will make a difference in the distortion model, so that is why most of the time only $\kappa_1, \kappa_2, \kappa_3$ are calculated. These three κ's are called the *distortion parameters* and are computed at the time of camera calibration. They are essential input parameters for distortion correction, but they are also themselves output parameters from a previous preprocessing step. Finally, radial distortion can be modeled as the relation between the ideal image points and the actual image points

$$q_d = \begin{bmatrix} x_d \\ y_d \end{bmatrix} = \begin{bmatrix} (1 + \kappa_1 r^2 + \kappa_2 r^4 + \kappa_3 r^6 + \dots) x_u \\ (1 + \kappa_1 r^2 + \kappa_2 r^4 + \kappa_3 r^6 + \dots) y_u \end{bmatrix} \tag{6.4}$$

But this is not yet the equation that will correct the images, because this equation only models what the distorted pixel coodinates will be, when the undistorted coordinates are given. To calculate the distortion correction [42], we use equation (6.5)

$$\begin{aligned} x_u &= x_c + L(r)(x_d - x_c) \\ y_u &= y_c + L(r)(y_d - y_c) \end{aligned} \tag{6.5}$$

and here are (x_d, y_d) the measured/actual coordinates, (x_u, y_u) the corrected/ideal image coordinates and (x_c, y_c) the center of radial distortion. Now we have an equation that allows us to calculate the corrected coordinates from a given distorted image, or in other words, we can calculate where every pixel of the distorted image should be positioned in the corrected image, in order to remove the non-linear radial distortion.

6.3 GPU implementation

There are two ways to construct the undistorted, corrected image and which one you choose, should depend on the kind of computational resources you have at your disposal. First, we can determine the pixel locations in the undistorted image using equation (6.5). This means that we construct a new image from the ground up, where the pixels are mapped from the distorted image on the right locations in the undistorted image according to equation (6.5), as you can see in figure 6.5. Or, we can use the reverse approach, where the pixels' color in the new image is defined through the transformation of equation (6.4), that defines the corresponding distorted pixel location and thus the right color. The general concept is to apply image distortion to a perfect image, just like a camera lens would, to find out which corrected pixel maps to which distorted pixel and vice versa.

The reverse approach, is extremely well-suited for implementation on the GPU, as we will explain on the basis of figure 6.6. To invoke a program on the graphics card, we

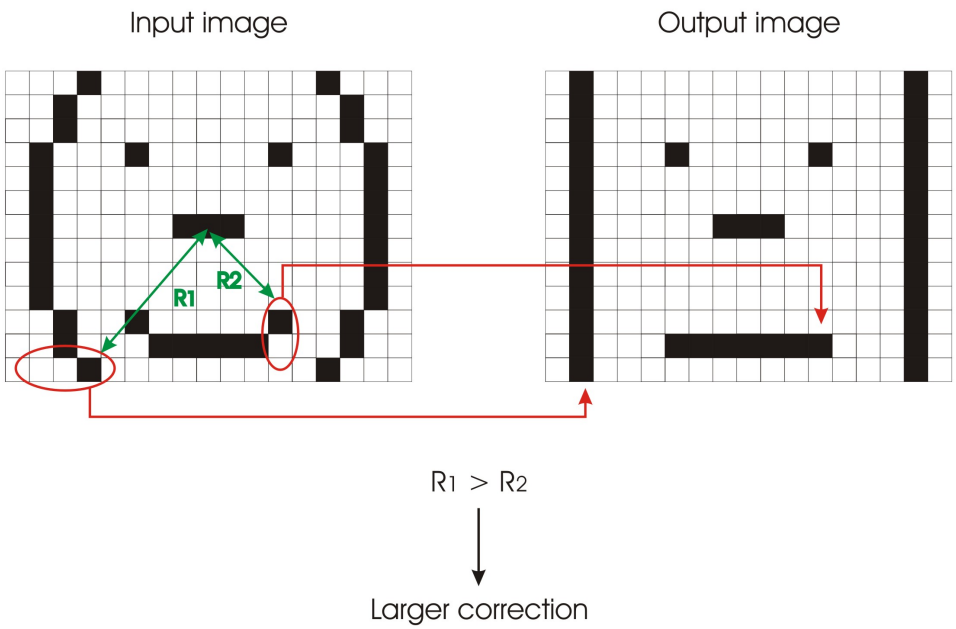

Figure 6.5: Mapping the distorted image pixels onto the corrected pixel locations

always have to send at least a geometrical entity (triangle, quadrilateral) to the start of the pipeline, the vertex processor. Most of the time we use a quadriliteral with the same dimensions as the image we want to process. The image we are going to process, in this case the distorted image, is loaded in a texture, so it sits in GPU memory and can easily be accessed from the programmable shaders.

Once the quad has passed the vertex shader and the rasterizer has interpolated the vertex coordinates to create fragments, we have a quad as input for the fragment shader that has the same dimensions and the same amount of pixels as the texture image, so we can speak of a one-to-one texel-to-fragment mapping (texel = texture pixel). These fragments have the coordinates of pixels in an ideal, undistorted image, so we will use a texture access to map the distorted image pixels onto the corrected fragments. This operation is performed using equation (6.4): we calculate the texture coordinates of the distorted image we want to sample, out of the input coordinates that accompany the fragments created in the rasterizer. As you can see in listing 6.1, once the right texture pixel is found, we apply this color to the fragment in the corrected image, thus creating an undistorted image as output of our processing pipeline.

Listing 6.1: The fragment program the computes distortion correction

```
PS_OUTPUT_STD ps_distortion(VS_OUTPUT_STD vertexOutput, uniform float k1,
                                                        uniform float k2,
                                                        uniform float k3,
                                                        uniform float k4,
                                                        )
{
    PS_OUTPUT_STD pixelOutput;
    float3 m_undist, m_dist;

    r_square = pow(m_undist,2)+ pow(m_undist, 2);
```

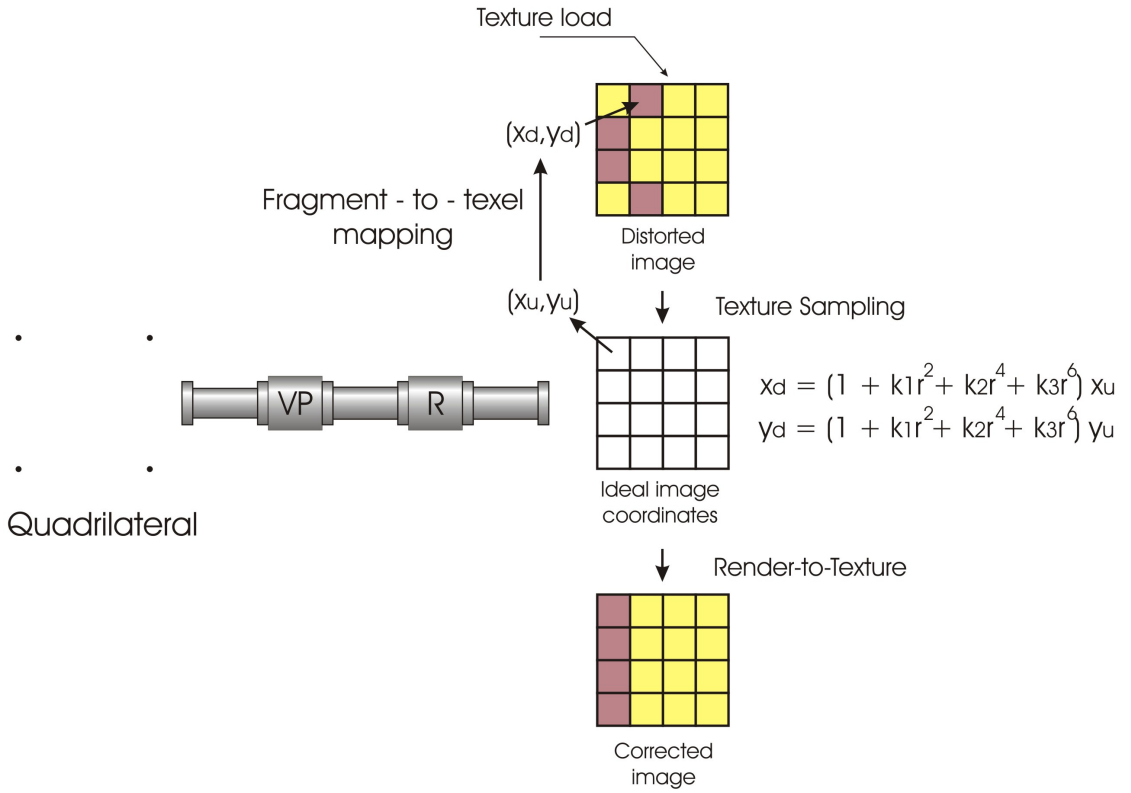

Figure 6.6: Using the reverse approach in the GPU pipeline

```
    m_distort.x = (1 + k1*r_square + k2*(pow(r_square, 2))
                + k3*(pow(r_square, 3)))*m_undist.x;
    m_distort.y = (1 + k1*r_square + k2*(pow(r_square, 2))
                + k3*(pow(r_square, 3)))*m_undist.y;
    m_distort.z = m_undist.z;

    //perform texture sampling at calculated locations
    float4 sampledTexel = tex2D(inputSampler, float2(m_distort.x, m_distort.y));

    //color the fragment with the calculated distorted pixel's color
    pixelOutput.computedValue.rgb = sampledTexel.rgb;


    return pixelOutput;
}
```

This is only the general approach that we have used in the graphics pipeline, but in reality it takes a series of steps to compute the right coordinates, so you will see more calculations in the actual shader. We summarize the necessary steps for an ideal image pixel m, related to its 3D point M through $m = K[R \mid t]M$:

1. Calculate the undistorted pixel $m_u = K^{-1}m$ to eliminate the influence of the intrinsic matrix K;

2. Compute the radial distance from the center of image distortion $r^2 = m_{ux}^2 + m_{uy}^2$;

3. Determine the coordinates of the distorted pixel m_d with Taylor expansion of equation (6.4) and the given $\kappa_1, \kappa_2, \kappa_3$;

4. Compute the texture coordinates of the distorted image $m_t = Km_d$. Multiplication with K is necessary because the distorted image is obtained with the influence of the intrisic camera parameters;

5. determine the right pixel to sample in the texture using bilinear interpolation;

6. Sample the texture at the calculated location and apply the color of that pixel to the fragment m, which we started the calulations with;

Now there is one step in particular that we have not mentioned so far, namely bilinear interpolation. While the distortion correction calculates the location of the pixel in the distorted image that needs to be displayed, it does not necessarily mean that this will result in perfect integer coordinates [43]. As a consequence, the actual location lies 'between' the pixels in the original image. The principle of bilinear interpolation is there to solve that problem.

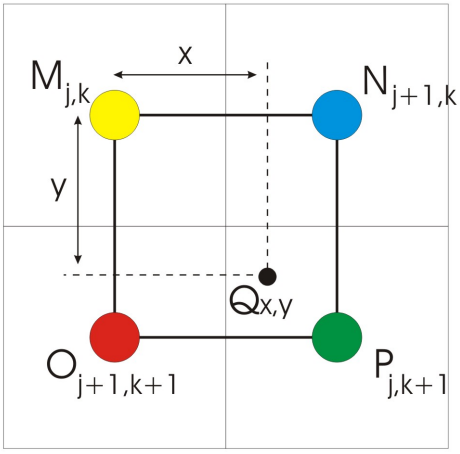

Figure 6.7: Bilinear interpolation neighborhood

The solution to our problem is stated in figure 6.7. Bilinear interpolation is used instead of simply truncating results that have a fractional component, because truncation, or alternatively rounding, causes errors in pixel location. These errors in turn, cause the distortion correction to produce shattered lines with a sawtooth profile, as you will see in our experimental results. This algorithm obtains the color of the calculated location by taking a weighted sum of the pixel values of the four closest neighbors. It computes the pixel value at the location Q in the figure as follows

$$Q_{x,y} = mM + nN + oO + pP \tag{6.6}$$

where

$$m = (1-x)(1-y) \qquad n = (1-y)x$$
$$o = (1-x)y \qquad p = xy$$

Implementing this on the GPU is standard, because it has been foreseen as one of the environment settings in shader programming that you can choose the kind of interpolation you want when calculated texture coordinates do not result in integer values. This principle is called *texture filtering*. The possible options and an example of the environment shader settings are listed in 6.2. LINEAR points to bilinear interpolation, and for the other options, that may or may not be hardware supported, we refer to the literature [44].

Listing 6.2: The fragment program the computes distortion correction

```
-The possible options:

typedef enum D3DTEXTUREFILTERTYPE
{
    D3DTEXF_NONE = 0,
    D3DTEXF_POINT = 1,
    D3DTEXF_LINEAR = 2,
    D3DTEXF_ANISOTROPIC = 3,
    D3DTEXF_PYRAMIDALQUAD = 6,
    D3DTEXF_GAUSSIANQUAD = 7,
    D3DTEXF_FORCE_DWORD = 0x7fffffff,
} D3DTEXTUREFILTERTYPE, *LPD3DTEXTUREFILTERTYPE;

-Used in a pipeline environment setup:

texture inputImage;
sampler inputSampler = sampler_state
{
    Texture     = <inputImage>;
    MipFilter   = NONE;
    MinFilter   = LINEAR;
    MagFilter   = LINEAR;
    AddressU    = BORDER;
    AddressV    = BORDER;
    AddressW    = BORDER;
};
```

6.4 Experimental results

The datasets used to test the functionality of our distortion correction algorithm are obtained from the "Expertisecentrum voor Digitale Media" (EDM) in Hasselt. The data consist of a couple of images accompanied by the proper distortion parameters for each camera, all capturing more or less the same scene. Even though the distortion parameters were obtained from a self-calibrated camera setup and not from a recognized website or database to assure data accuracy, they proved to be accurate enough to demonstrate the

workings of distortion correction. Even though data was not always as accurate as it should be, there was enough information at hand to test the basic functionalities.

To demonstrate that working with the lower-order terms of the Taylor expansion is good enough for accurate results, figure 6.8 is used. In the original picture (on the left), the iron beam behind the green curtain is bent too much to represent the actual situation. Distortion correction was performed with only one of the κ parameters, κ_1, wich has the greatest influence of all coefficients.

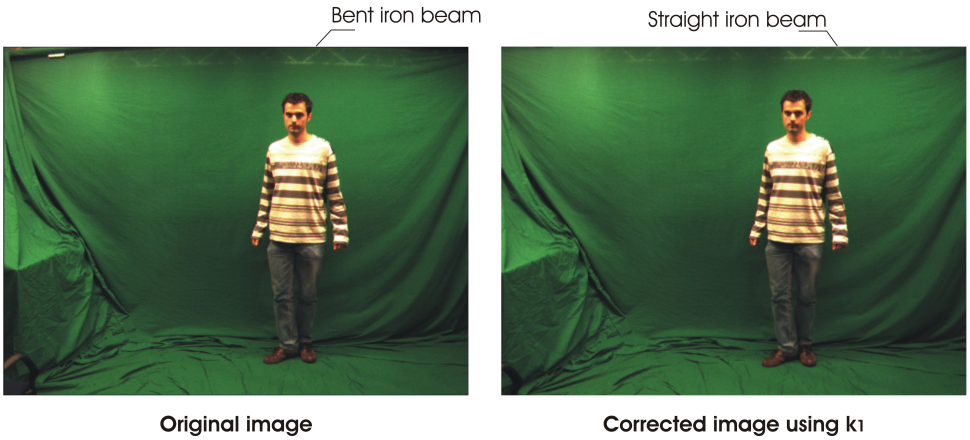

Figure 6.8: Distortion using only the first distortion parameter

Clearly, for this image, it suffices to use just one parameter, to convert the image back to a situation that approaches reality. Using more parameters will increase accuracy mostly in the corners, but will never have the same inpact as κ_1. They are less important in the Taylor expansion, so only small, but often essential adjustments can be made with the less important terms.

Our second test was used to test the influence of the bilinear interpolation when sampling textures. In the left image of figure 6.9, no interpolation method whatsoever was used, resulting in a sawtooth pattern near the edges of the iron poles. In contrary, take a look at the right image, where no crude imperfections are visible because for constructing this image, the bilinear texture interpolation method was used.

So, in order to create smooth edges and avoid image imperfections that are too easy noticable for the human eye, bilinear interpolation is needed.

Finally, in figure 6.10, the distortion correction is executed to its full extent. The first image is the original image, the second is the corrected image with κ_1 only, the third one uses both κ's, κ_1 and κ_2, given as input data. And in the fourth image we saved the image in a larger texture, to be able to see the correction transformation take place. The image now looks as if it was shot with pincushion distortion, but since it was actually captured with barrel distortion, applying pincushion distortion to the image reverses the process and gives an undistorted image as output. So this test harnesses all the properties of our distortion correction algorithm.

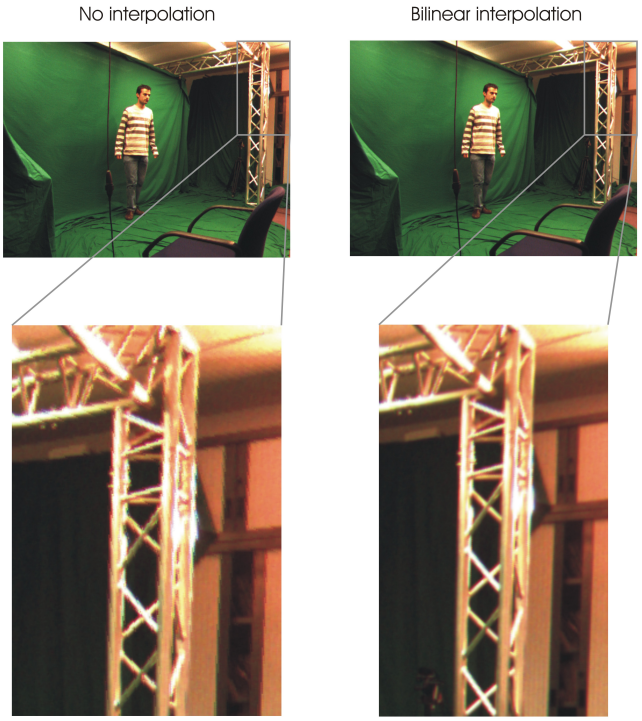

Figure 6.9: Demonstration of the usefulness of bilinear interpolation

Figure 6.10: Bilinear interpolation neighborhood

Chapter 7

Rectification

The process of rectification is essential for subsequent stages as it assures that all corresponding object features wind up on the same horizontal scan line, thus reducing the stereo matching cost and the algorithmic complexity. Only out of a rectified image pair, a stereo matching algorithm is able to construct a depth map, to calculate the requested intermediate viewpoints. Rectification is based on epipolar geometry and camera matrices, so there is no rectification without camera calibration. Through computation of a new pair of camera matrices, any uncalibrated camera pair can be transformed to a calibrated one. Since all this eventually comes down to a per-pixel matrix transformation, it is easily implemented using a GPU shader. To prove that rectification was successful, epipolar reconstruction of rectified images of specialized datasets can show us that epipoles will now reside at infinity, making the transformation a success.

7.1 Stereo matching fundamentals

7.1.1 Usefulness and approach

In a couple of words, stereo matching makes it possible to aqcuire depth information out of two images. These images are either viewing the same scene under a slightly different angle or they have at least a lot of similarities. This concept is based on the workings of the human eye, where depth perception is possible because each eye captures the surrounding reality from a slightly different viewpoint. Just like we have a left and a right eye, we have a left and a right picture, as you can see in figure 7.1. Information out of both views is used to construct a depth image, necessary to compute intermediate images in a later stage, called *image warping*.

Producing such a depth image, is the most time-consuming part in a stereo matching application. Basically, stereo matching will approach this problem by taking one pixel out of a reference image (mostly the left image), and performing a two-dimensional search in the other image (the right image) for the corresponding pixel. This search is executed using a technique called *intensity-based matching*. Again, as with feature correspondence, a window overlay is used on the reference pixel and an intensity function is calculated. The same action is performed on the pixel under test in the right image and both results

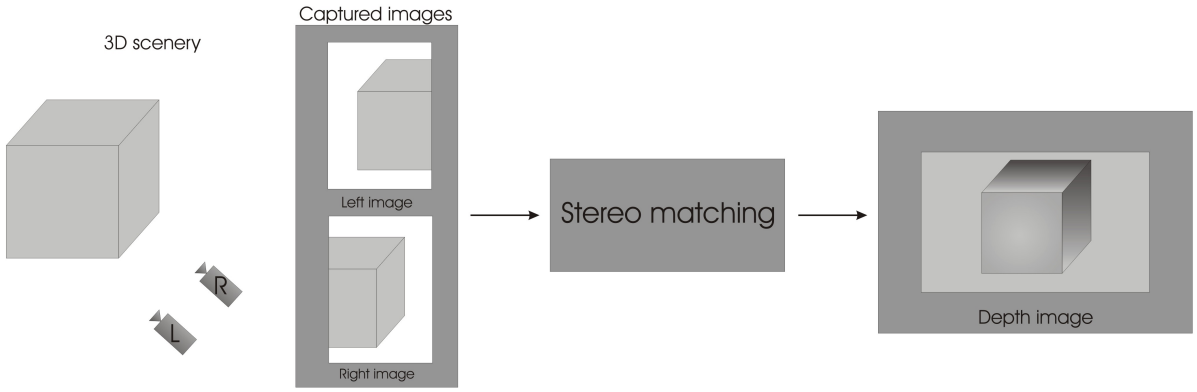

Figure 7.1: Stereo matching flowchart

are compared to produce a matching cost. The pixel in the right image that results in the lowest matching cost is defined as a *match*.

But, since such a 2D search would require a lot of computational power and could seriously downgrade the processing time of our application, stereo matching applications assume that images are captured in a parallel setup. That means both optical axes of the left and right camera are parallel aligned, so the image planes are coplanar. This limits the search range to a single horizontal line (*scan line*) on which the intensity matching has to be performed. The stride on this line, starting from the coordinates of the reference pixel, to get to the pixel with the lowest matching cost, is called *the disparity* and is expressed in pixel units. For instance, if we find the minimal matching cost in the right image, and it corresponds to a pixel that is shifted five pixels to the left, the disparity is 5. Calculating the minimal matching cost for every pixel in the reference image, returns a *disparity map*, which you can see in figure 7.2. The larger the disparity, the whiter the pixel values, and the pixel color gradually darkens as the disparity decreases.

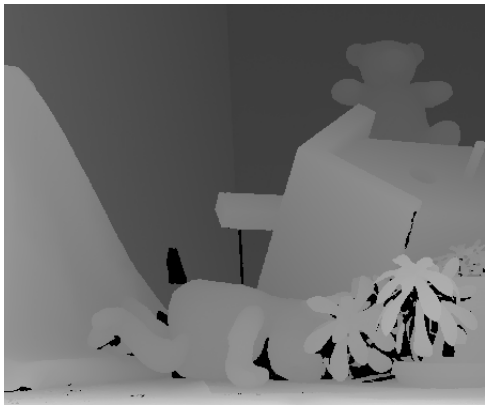

Figure 7.2: A disparity map of the Teddy scene

A disparity map is also called a depth map. The reason is simple and demonstrated in figure 7.3. Obviously, the closer objects are to the camera, the larger their shift, when comparing their position in the left and the right image. The further away from the camera they are, the smaller their movements. This principle is called *the motion parallax*.

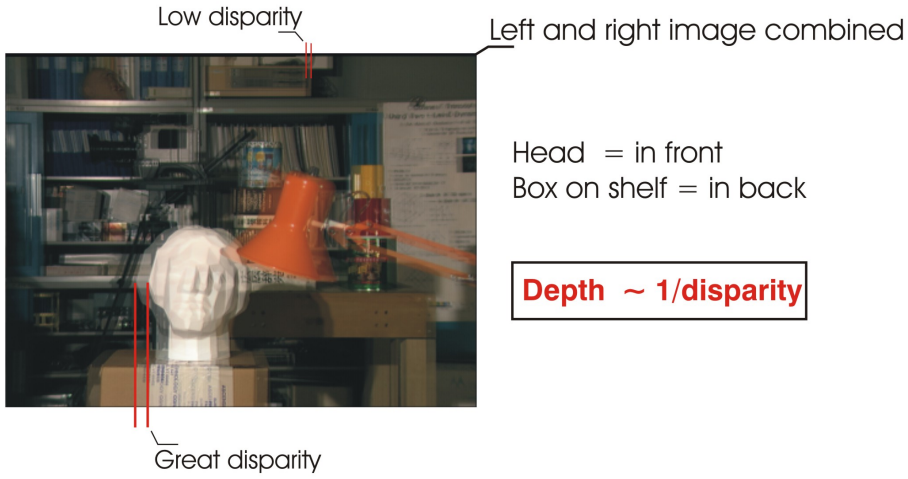

Figure 7.3: The closer to the camera, the greater the disparity

So larger depth gives a lower disparity and less depth results in high diparity value. Since a disparity map proves to be enough to describe the depth information encapsulated in a scene, there is really no need to transform this disparity image to an actual depth map.

Image interpolation calculations are now done with information embedded in the depth map. Suppose we want to construct an image that is exactly halfway between the left and the right image (figure 7.4). That means that certain objects, that were visible in the left image, will now get occluded because others slide in front of it. How big this shift will be and how much objects will get occluded, is extracted from the depth map. Since it gives us the disparity values, transferring from the left to the right image, the transfer from the left to halfway will only yield half the disparity a full transfer would. Dividing the entire depth map in half gives every object the right amount of shift to construct a intermediate image that reflects the real world view from that viewpoint.

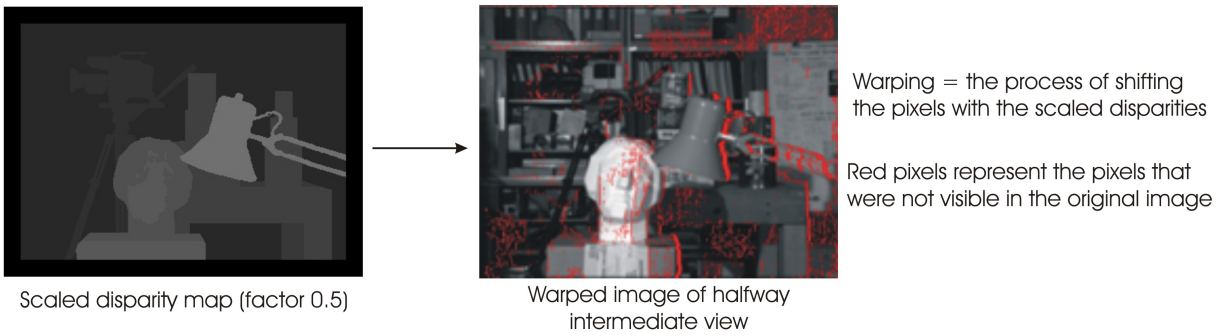

Figure 7.4: Shifting reveals unvisible pixels and occludes others

This is a simplified disposition of what stereo matching is, but it provides sufficient information to comprehend the general concepts and understand why rectification of input images is required. For an into depth essay of stereo matching, image warping and occlusion handling, Rogmans [3, 28] is advised.

7.1.2 Image rectification

Thanks to image rectification, stereo matching can be limited to a scan line instead of an entire image search. But this is a consequence of the rectifying transformation, it does not define what it is. In fact, rectification is also referred to as epipolar rectification since it causes a specific change in the epipolar geometry of a camera setup. Or, more specific, it defines a transformation of each image plane such that pairs of corresponding epipolar lines become collinear and parallel to one of the image axes [45]. The situation before rectification is shown in figure 7.5.

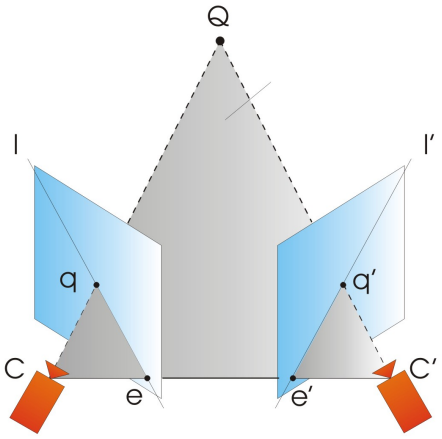

Figure 7.5: Normal epipolar geometry

This is the same epipolar geometry from section 5.2. Given a point in the left image, to find the corresponding point in the right image, we can enforce the epipolar constraint. This determines that the corresponding image point is on the epipolar line in the right image plane formed by the intersection of the image plane and the plane CQC'. Figure 7.6 shows that, the more the image planes are aligned, the further away the epipoles lie.

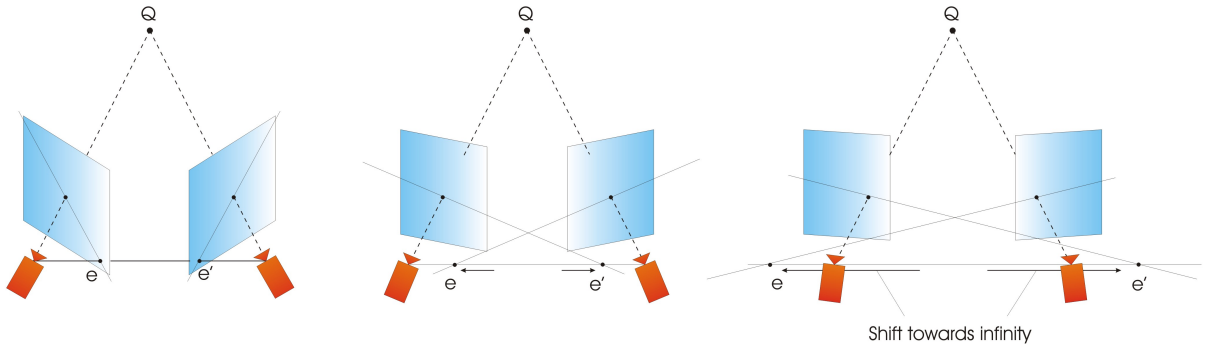

Figure 7.6: Aligned image planes cause the epipoles to move towards infinity.

That means, that if we consider the special case where the image planes are coplanar, both epipoles will lie at infinity, and the focal (image) planes are parallel to the baseline as well. Now, since the epipoles are located at infinity, the epipolar lines form a bundle of parallel lines in the image planes that converge at infinity (see figure 7.7).

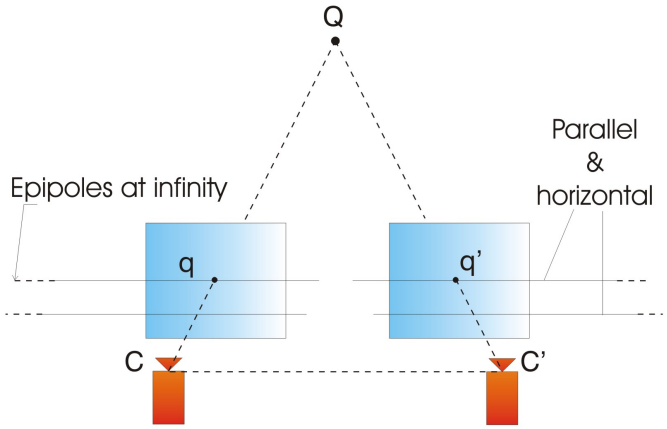

Figure 7.7: parallel epipolar lines intersect at infinity

In conclusion, using the epipolar constraint, we can limit the otherwise 2D search to a one-dimensional search on an epipolar line to find the corresponding image point. In the special case that both image planes are coplanar, the epipoles lie at infinity and the epipolar lines are now horizontal, so our search is limited to the same horizontal line as in the reference image. The transformation that moves the epipoles towards a point at infinity is called rectification, and any pair of images can be transformed so the epipolar lines become parallel and horizontal in each image.

7.2 The rectification process

7.2.1 Rectifying the camera matrices

For image rectification, we assume that camera calibration has already been performed, so we are talking about a calibrated camera pair and the perspective projection matrices P_{o1} and P_{o2} are known. The basis of image rectification is to define two new PPMs P_{n1} and P_{n2} that are obtained by rotating the old ones around their optical centers. This leads to parallel epipolar lines, but that does not yet mean that they are horizontal. For epipolar lines to be horizontal as well, the baseline must be parallel to the new X-axis of the cameras. The corresponding image points must have the same vertical coordinate as well, so that is why the new cameras are required to have the same intrinsic matrix, to make them use the same scales on their vertical axis. The new rectified images can be seen as if they were captured with a new pair of cameras that differ with the old cameras by a rotation around their optical centers.

From section 3.1.3, we know that we can write the new PPMs as

$$P_{n1} = A[R \mid -Rc_1] \tag{7.1}$$
$$P_{n2} = A[R \mid -Rc_2] \tag{7.2}$$

where c_1, c_2 are the old optical centers, as they are the same for the new PPM because no translational movement is involved in image rectification. The intrinsic matrix A is equal for both PPMs and can be chosen arbitrarily, so either A_{o1} or A_{o2}. The rotation matrix R , which gives the camera's orientation relative to the world frame, is the same for both cameras as well, because now they are parallel aligned. The matrix R can be written as

$$R = \begin{bmatrix} r_1^T \\ r_2^T \\ r_3^T \end{bmatrix} \tag{7.3}$$

with r_1, r_2, r_3 as its row vectors that define the X, Y, Z axes of the cameras relative to the world reference frame. Following the previous assumptions we have made, these axes are determined as

1. The new X-axis is parallel to the baseline: $r_1 = \frac{c_1 - c_2}{\|c_1 - c_2|}$.
2. The new Y-axis is orthogonal to the new X-axis and to k: $r_2 = k \wedge r_1$ where k is an arbitrary unit vector, but is often chosen as the unit vector of the old Z-axis, so Y has to be orthogonal to the new X and the old Z.
3. The new Z-axis is othogonal to XY: $r_3 = r_1 \wedge r_3$.

Although this algorithm fails when it has to deal with a pure forward motion, it is relatively simple and we can make good use of it because stereo correspondence will never work on a pair of images that have undergone a pure forward motion anyway, because no depth information can be extracted. For more general solutions of this problem, [34] is suggested.

7.2.2 The rectifying transformation

Now that we know what the PPMs of the new, rectified camera pair look like, the last step is finding the transformation that transforms the old image planes to the new ones. In fact, this means that we are not touching the cone of rays that intersects the image planes, but we are simply rotating the image planes. Consider the left image, then we are looking for a transformation T_1 that maps $\tilde{P}_{o1}$ onto $\tilde{P}_{n1}$. Now, if we write the PPMs as

$$\tilde{P}_{o1} = [L_{o1} \mid l_{o1}] \tag{7.4}$$
$$\tilde{P}_{o2} = [L_{o2} \mid l_{o2}] \tag{7.5}$$

and for any 3D point $\tilde{Q}$ we can describe the relation to its image points $\tilde{q}_{o1}$ and $\tilde{q}_{n1}$ in the old and new image planes respectively,

$$\tilde{q}_{o1} = \tilde{P}_{o1}\tilde{Q} \tag{7.6}$$
$$\tilde{q}_{n1} = \tilde{P}_{n1}\tilde{Q} \tag{7.7}$$

The world coordinates of $\tilde{Q}$ can also be described in terms of the equations of the optical rays (rays from $\tilde{Q}$ to c_1)

$$\tilde{Q} = c_1 + \lambda_o L_{o1}^{-1} \tilde{q}_{o1} \tag{7.8}$$
$$\tilde{Q} = c_1 + \lambda_n L_{n1}^{-1} \tilde{q}_{n1} \tag{7.9}$$

and both λ's of the old and new projections are arbitrary scale factors. Out of the equations (7.8) and (7.9) we can conclude

$$\tilde{q}_{n1} = \lambda L_{n1} L_{o1}^{-1} \tilde{q}_{o1} \tag{7.10}$$

This is the equation that describes the direct mapping from the old image plane to the new one, so the sought transformation $T_1 = L_{n1} L_{o1}^{-1}$. For the right image, the derivation of T_2, and thus the rectifying transformation, is analog. The only matter that still requires a little concern, is that the produced results will not be integer values, so bilinear interpolation will be necessary again. But we have already discussed in section 6.3 that this feature is part of the standard functionality of our graphics card, so the problem is solved relatively easy.

7.3 GPU implementation

As we have tried throughout this entire thesis to map everything to the graphics card and perform the least calculations possible on the CPU, this approach is used as well when implementing the rectification. Considering what is explained in the previous paragraph, we can subdivide the rectification process in two steps for a fixed camera setup:

1. First we have to recalculate the new perspective projection matrices for the rectified camera pair. Combining the new and old PPMs as in equation (7.10), we can construct the mapping transformation T_1 from the old left to new left image, and calculate T_2 for the mapping from the old right to the new right image.

2. Once we have the transformation matrices, and if we assume that our stereo setup is fixed, no additional calculations are needed, because the matrices T_1, T_2 work for all images captured by either the left or the right camera. So we program the per-pixel transformations in the graphics pipeline to produce the rectified versions of the input images.

Because step one only requires a one-time calculation, there is no need to program this on the GPU, since most of our computation time will be lost on overhead to set up the calculations. Most image processing caculations involve matrix multiplications and the graphics pipeline has hardware optimized for matrix calulations, so that is why the DirectX 9 API is equipped with a lot of matrix functions that lower the complexity of matrix computations almost to the level of regular algebra. Examples of these very useful functions are D3DXMatrixIdentity(), D3DXMatrixMultiply(), D3DXMatrixInverse(), ... [44] that make the use of library calls a simple but very profitable solution.

Once we have the transformation matrices, implementing the transformation is not a tricky business. It is an operation that has to be executed on every pixel to transform the entire image plane from the old to the new, rectified orientation. Thus, we use a pixel shader to compute the transformation and output the rectified image. We keep the transformation matrices in memory, then, pairs of images with the same timeframes are sent in, matrix T_1 is applied to the left image, matrix T_2 is applied to the right image and a pair of rectified images is outputted, ready for use as input for a stereo matching algorithm.

7.4 Experimental results

7.4.1 Middlebury datasets

Datasets used to evaluate the image rectification, are datasets presented by the College of Middlebury [46]. The Dinoset and the Templeset, both multiview examples are ideal for testing our rectification process, because the baseline can be varied according to the developer's desire and the cameras are definitely not parallel aligned. These sets are all captured with one camera that is rotated around the object of interest as you can see in figure 7.8. The positions that are included in the final datasets are highlighted in the picture as the red and blue cameras.

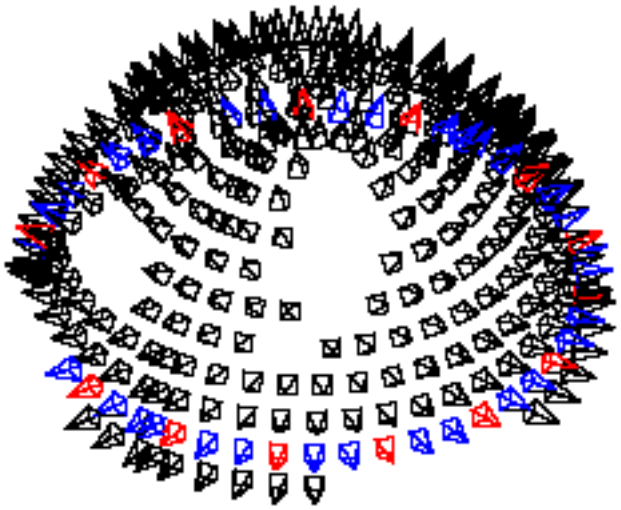

Figure 7.8: Captured camera positions in the Dino and Temple sets

Most important about these datasets is the accompanying calibration data. The calibration data is guaranteed to be very accurate, so we do not have to worry about wrong input data. When we would use some of our own calibration data, we risk starting off on the wrong foot because our calibration calculations may contain errors or still be inaccurate. Using these datasets as input, we can create an objective evaluation and measurement of the accuracy of our image rectification algorithm.

7.4.2 Results and validation

We have two validation methods to check the accuracy of the produced results after rectification. On one hand, we can manually check the translation of corresponding image points between the two output images. This is not a waterproof method, since we are visually matching image points instead of reconstructing the epipolar geometry. But in some cases, as you will see in the results, the objects in the images offer not enough feature points to allow proper calulations. So that is why we check the results by hand as well, as a first indication that the images might be rectified. On the other hand, if we can find enough corresponding points, using Matlab code from the work of Gabriels [37], we can reconstruct the epipolar geometry and have the application draw the epipolar lines on the output images. If the epipolar lines turn out to be parallel and horizontal, rectification was successful.

In the results that are presented, we have the dataset with a picture of Sammy Rogmans that was obtained at the EDM, and thus the calibration parameters to these images were calculated there as well. Although there was the possibility that the calibration data would not be accurate enough to perform proper rectification, figure 7.9 proves us wrong. Even though no epipolar reconstruction was possible due to a lack of feature points, caused by the homogenous clothing and background, you can still see that rectification was successful according to the manual check. Further we also have the Dinoset from Middlebury, where no reconstruction was possible either because of untextured surface, so in figure 7.10, again a manual check was our only way of validating the result. The last dataset is give in figure 7.11 and is the Templeset from Middlebury, which gives the best results, because epipolar reconstruction is possible on these images and the epipolar lines are all parallel and horizontal. So proof is given in several ways that our rectification transformation works like it should.

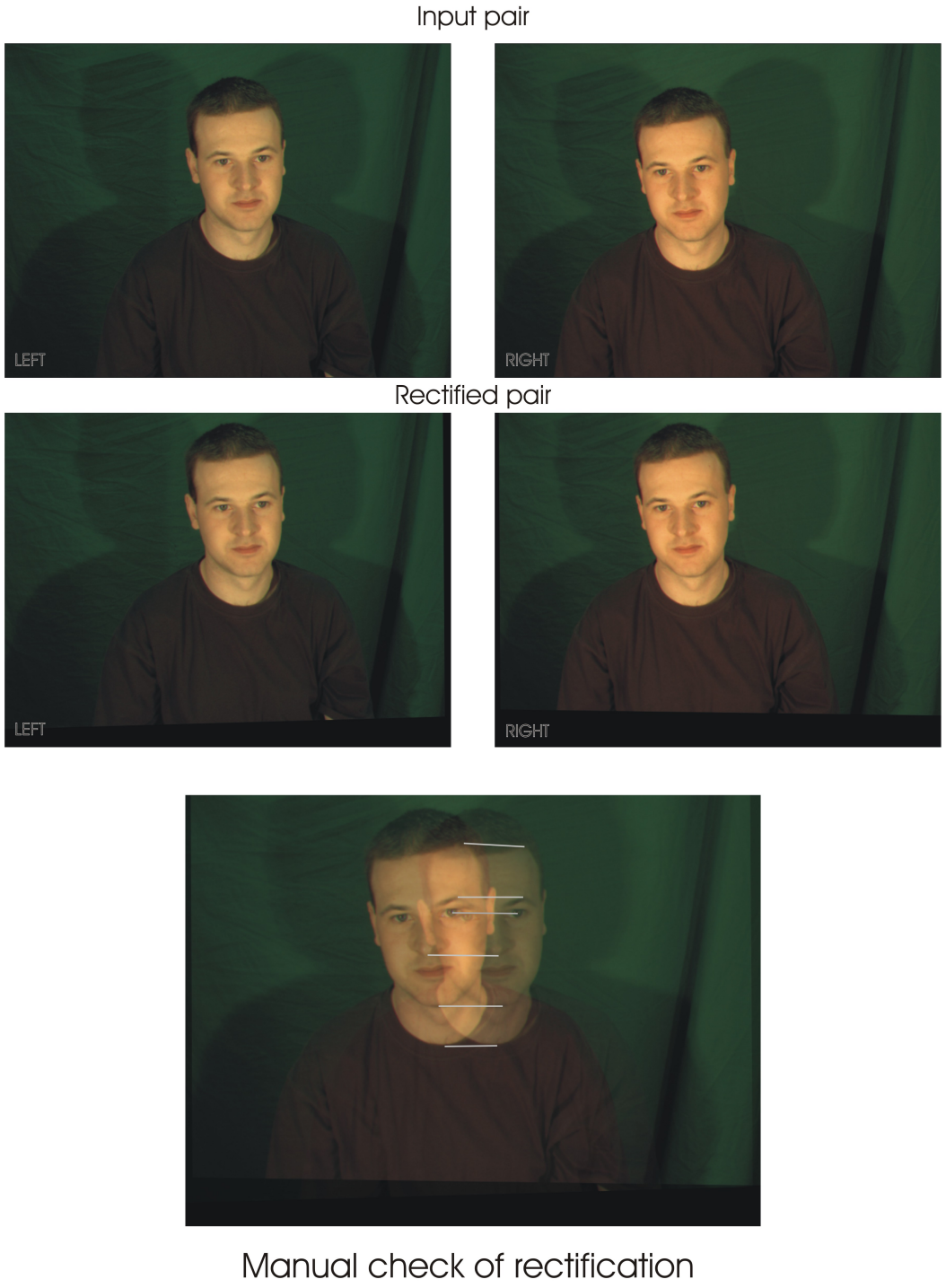

Figure 7.9: Using a manual check, the Sammy dataset looks to be properly rectified

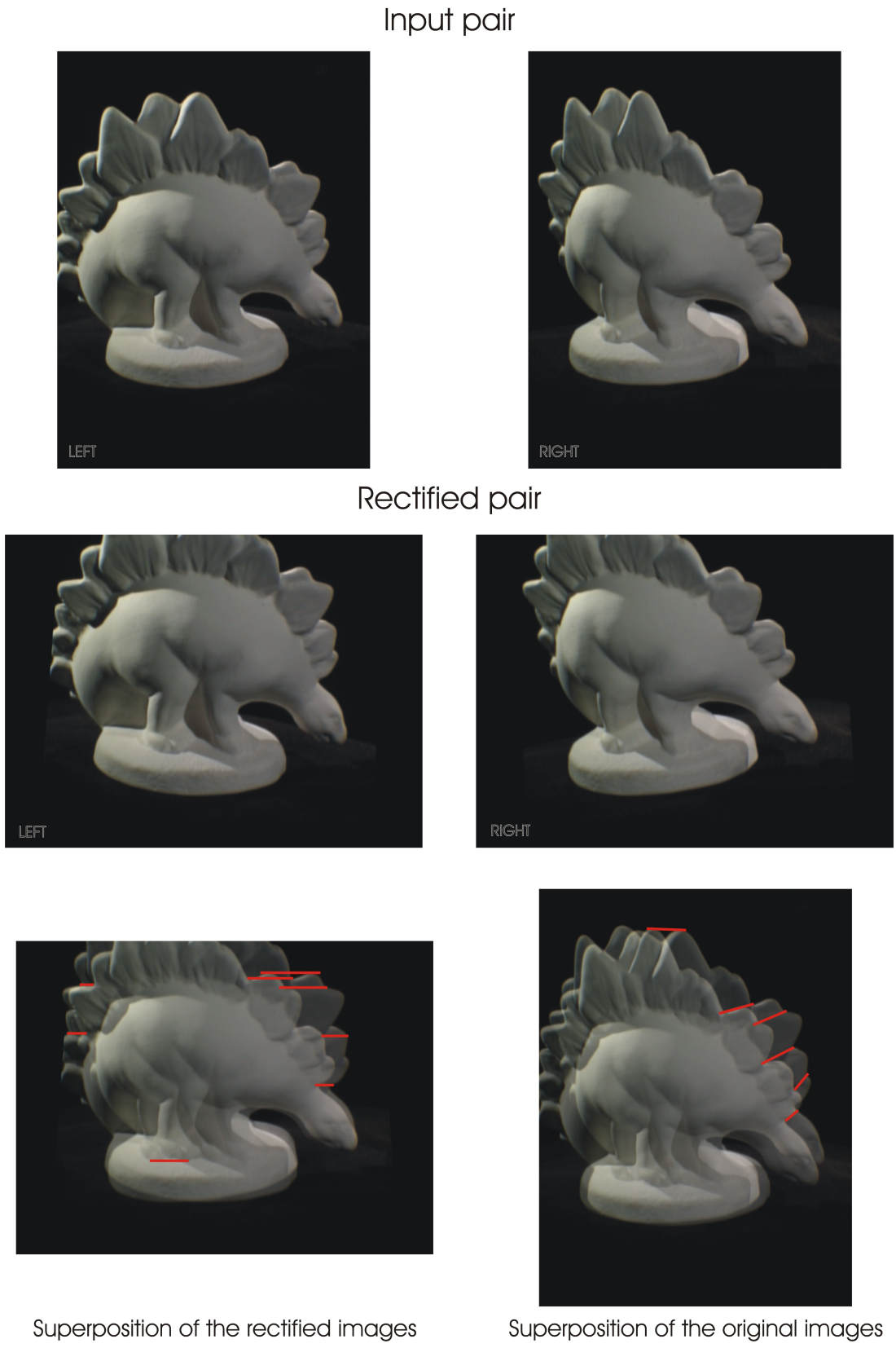

Figure 7.10: The shift of feature points is only parallel and horizontal in the rectified pair

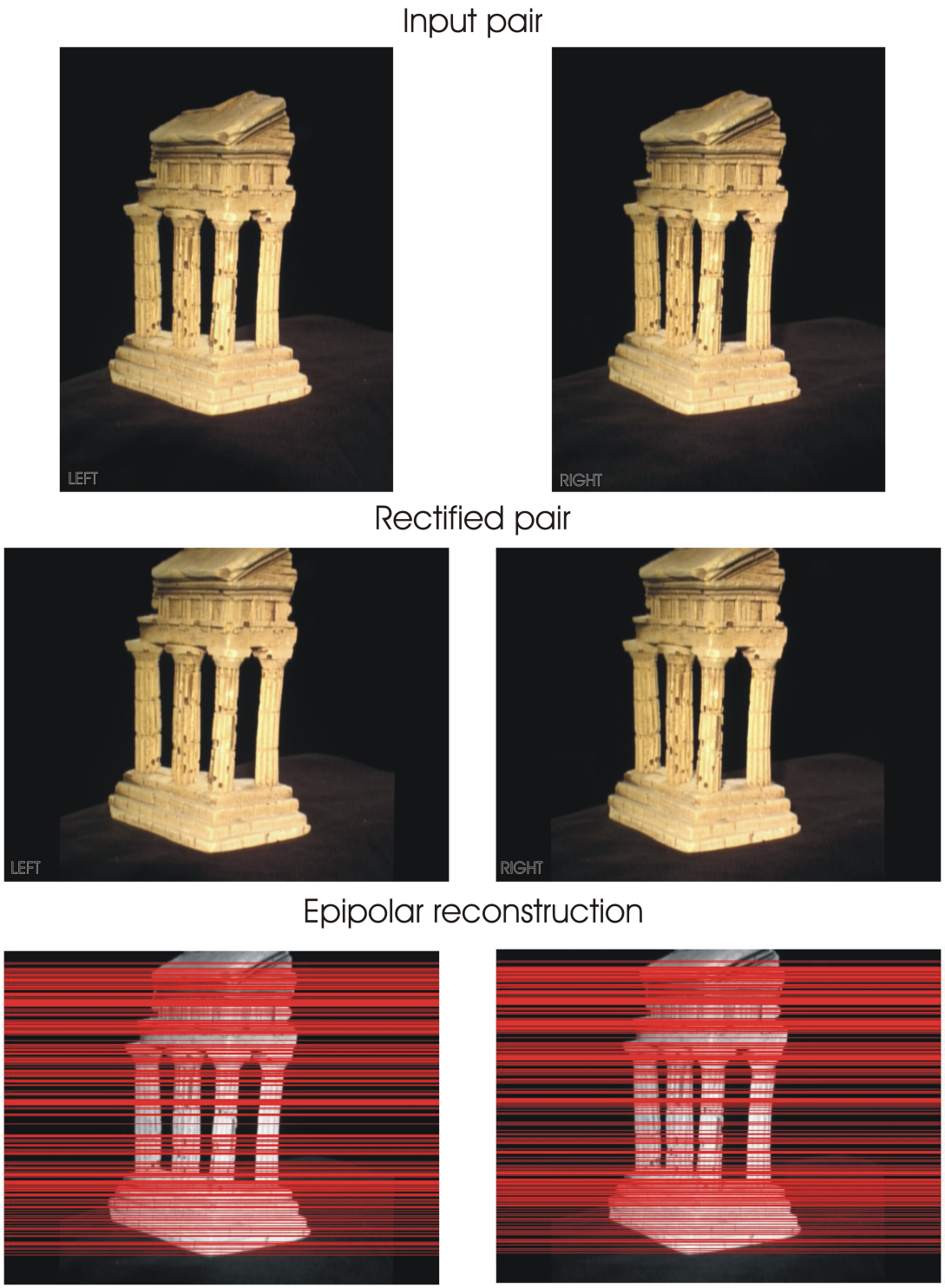

Figure 7.11: Epipolar reconstruction offers a certified proof of proper rectification

Chapter 8

Conclusions and Future work

8.1 Conclusions

A revolutionary new technique, called Free Viewpoint Video, allows a spectator to freely choose his custom viewpoint. That viewpoint is then synthesized with the information gathered from a couple of fixed, physical cameras. To allow this concept to apply on real live video feeds as well, the captured images have to be adequately preprocessed because the image rendering algorithms that calculate the requested viewpoints, take ideal images as input, without considering possible deformations in real images. This mainly involves correcting radial distortion, caused by the use of real camera lenses, and rectifying the input images in order to align the object features on the same scan line. Then a stereo matching application can use the preprocessed images as input and compute the requested intermediate viewpoint out of a practical stereo camera setup. To keep the application within real-time limits, we implement this vast amount of computations on the GPU.

Performing rectification or distortion correction is only possible if the intrinsic and extrinsic camera parameters of the stereo camera setup are known. That is why the cameras are subjected to a method of self-calibration using a recognizable pattern (checkerboard or LED-light). Such self-calibration methods relate images with the special pattern, so no really intensive feature matching is required as long as there are enough images provided. We developed a Harris corner detector on a PC's graphics hardware that can be used to detect points of interest on checkerboards or detecting LEDs in dark images. Out of these point correspondences, the mapping between the different image planes can be derivated. And with the mapping between corresponding image planes, the properties of the cameras and their relative position can be determined. Once the intrinsic and extrinsic camera parameters are known, we have a calibrated camera pair and we can continue to the next image processing step. Since we are developing a static camera setup application, there is no real need for continuous recalibration, so the calibration can be executed offline once for the entire setup. That is one of the reasons why we used a tool for the camera calibration, so we could devote our development time on the subsequent processing stages, distortion correction and rectification.

Radial distortion is the most common form of distortion and is modeled by a Taylor expansion chopped off behind the low order terms. We approach this problem from the

reverse way, when implementing it on the GPU. Instead of calculating a mapping from the given distorted image to a new corrected image, we do it the other way around. We send in a quadrilateral at the start of the graphics pipeline and, after rasterization, the mapping from the corrected image to the distorted image that is loaded in a texture, is calculated. This is of course because a GPU always has to sample a texture to give colors to the fragments in the pipe, and because the mapping from ideal coordinates to distorted coordinates is straightforward, we make good use of this property.

When radial distortion is removed, the corresponding images pairs need to be rectified, so they look like a couple of parallel aligned cameras. Together with the camera calibration parameters, a new perspective projection matrix can be determined for every camera. Transforming the old PPMs into the new PPMs causes the epipolar lines to converge at infinity. That means that the epipoles are at infinity and the epipolar lines are parallel aligned, what was one of the first demands for further image processing. Implementing this on the GPU can be condensed in a pixel-per-pixel matrix multiplication, making it the suited candidate for a simple fragment shader.

Thanks to the highly parallel nature of the GPU, these two image preprocessing steps, distortion correction and rectification, do not downgrade the stereo matching application a lot. The image distortion correction and rectification achieves a frame rate of 3691 fps, for 450×375 image resolutions, so the entire applicaton only suffers from a minor setback of 1.32%. The highest achievable frame rate for the entire image processing application results in 43.9 fps.

8.2 Future work

Our work is far from done here, as there are several optimizations and possibilities that need our attention because of their great potential. An entire self-calibration framework can be developed, so we do not have to rely on someone else's tool. This will also give the researchers a good experimental knowledge, so we can build our own calibration setup as well. Working with the cameras themselves is actually also a possible thesis subject. Every camera has a different white balance, the cameras need to be synchronized over different IEEE 1394 buses, the correct Region of Interest (ROI) should be transmitted and many more. Further, there is a spot to develop a very accurate, intrusive benchmarking mechanism, to perform precise timings, based on the system clock of the GPU instead of the CPU. The distortion and correction operations need to be optimized and combined on a lower level, so the rest of the application suffers as little as possible from the image preprocessing. There are more options than those mentioned here and some of them might take more time than just a couple of weeks, but that should not stop us, there are definitely still a lot of challenges in this domain.

Bibliography

[1] "De nayer instituut." http://www.denayer.wenk.be.

[2] "Inter-university micro electronics center." http://www.imec.be.

[3] S. Rogmans, *A Generic Framework for Implementing Real-Time Stereo Matching Algorithms on the Graphics Processing Unit*. 2007.

[4] S. Stroeykens, "Speelgoed-supercomputer," *de Standaard Wetenschap nr 169*, p. 2, 2007.

[5] J. Gemmel, K. Toyama, C. Zitnick, T. Kang, and S. Seitz, "Gaze awareness for video-conferencing: a software approach," *Multimedia, IEEE, Volume 7*, pp. 26–35, October 2000.

[6] "The nvidia corporation." http://www.nvidia.com/page/products.html.

[7] R. Fernando and M. J. Kilgard, *The Cg tutorial, the definitive guide to programmable real-time graphics*, ch. MATLAB Interface to Generic DLLs. Addison Wesley, 2003.

[8] "The microsoft developer's network library." http://msdn.microsoft.com.

[9] R. Fernando, M. Harris, M. Wloka, and C. Zeller, "Programming graphics hardware," *Eurographics, Tutorial*, 2004.

[10] J. D. Owens, D. Luebke, N. Govindaraju, and M. H. a.o., "A survey of general-purpose computation on graphics hardware," *The Eurographics Association, volume 26*, pp. 80–113, 2007.

[11] "The nvidia geforce 9 series." http://www.nvidia.com/object/geforce9.html.

[12] R. Fernando and M. Pharr, *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Addison Wesley, 2005.

[13] J. Loviscach, "Shader programming: An introduction using the effect framework," *RITA, Volume IX*, 2004.

[14] "General purpose computations on the graphical processing unit." http://www.gpgpu.org.

[15] "1394 serial bus interface: Convergence bus promises to unite pcs and digital consumer equipment." http://www.wdc.com/en/library/firewire/2579-001014.pdf.

[16] M. Stokes, M. Anderson, S. Chandrasekar, and R. Motta, "A standard default color space for the internet - srgb," *http://www.w3.org/Graphics/Color/sRGB*, November 1996.

[17] A. Ford and A. Roberts, "Color space conversions." http://www.poynton.com/PDFs/coloureq.pdf, August 1998.

[18] T. S. Group, "Digital expert tech talk - color and color space." http://dx.sheridan.com/tech/main.html.

[19] Point Grey Research Inc., *Firefly MV Getting Started Manual*, april 2007.

[20] "Point grey research inc.." http://www.ptgrey.com.

[21] L. Kitchen and A. Rosenfeld, "Gray level corner detection," *Pattern recognition letters*, pp. 95–102, 1982.

[22] S. Smith and J. Brady, "A new approach to low-level image processing," *International Journal of Computer Vision*, pp. 45–78, 1997.

[23] F. Mokhtarian and R. Suomela, "Robust image corner detection through curvature scale space," *IEEE Trans on Pattern Analysis and Machine Intelligence*, pp. 1376–1381, 1998.

[24] C. Harris and M. Stephens, "A combined corner and edge detector," *Proc. Alvey Vision Conf., Univ. Manchester*, pp. 147–151, 1988.

[25] H. P. Moravec, "Towards automatic visual obstacle avoidance," *Proc. 5th International Joint Conference on Artificial Intelligence*, p. 584, 1977.

[26] H. P. Moravec, "Visual mapping by a robot rover," *International Joint Conference on Artificial Intelligence*, pp. 598–600, 1979.

[27] "Wikipedia, the free encyclopedia." http://www.wikipedia.org.

[28] S. Rogmans, "Parallelization of stereo to an mpsoc platform with gpu specific optimization." http://www.phusix.be/teaching.html, June 15th, 2007.

[29] K. G. Derpanis, "The harris corner detector," October 27, 2004.

[30] Z. Zhang, R. Deriche, O. D. Faugeras, and Q. Luong, "A robust technique for matching two uncalibrated images through the recovery of the unknown epipolar geometry," *Artificial intelligence, vol. 78, no. 1-2*, pp. 87–119, 1995.

[31] "Middlebury stereo correspondence datasets." http://vision.middlebury.edu/stereo/.

[32] T. Svoboda and T. Pajdla, "Multi-camera self-calibration." http://cmp.felk.cvut.cz/~svoboda/SelfCal/.

[33] Z. Zhang, "A flexible new technique for camera calibration," *Technical report MSR-TR-98-71*, December, 1998.

[34] R. Hartley and A. Zisserman, *Multiple View Geometry in Computer Vision second edition*. Cambridge University Press, 2003.

[35] H. Longuet-Higgins, "A computer algorithm for reconstructing a scene from two projections," *Nature, 293*, pp. 133–135, September 1981.

[36] M. F. adn R.C. Bolles, "Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography," *Comm. Assoc. Comp. Mach.*, pp. 391–395, 1981.

[37] N. Gabriels, *Robust Self-Calibration in Multi-Camera setup*. 2008.

[38] M. K. Chandraker, "Two-view focal length estimation for all camera motions using priors,"

[39] S. Bougnoux, "From projective to euclidean space under any practical situation, a criticism of self-calibration," *Proceedings of the Sixth International Conference on Computer Vision, IEEE Computer Society*, p. 790, 1998.

[40] D.C.Brown, "Decentering distortion of lenses," *Photometric Engineering, Vol. 32, No. 3*, pp. 444–462, 1966.

[41] L. Ma, Y. Chen, and K. L.Moore, "Camera calibration: a usu implementation," *Center fo Self-Organizing and Intelligent Systems, Department of Electrical and Computer Engineering, Utah State University*, 2007.

[42] E. Tola, *Multiview 3d reconstruction of a scene containing independently moving objects*. 2005.

[43] K. Gribbon and C. J. an D.G. Bailey, "A real-time fpga implementation of a barrel distortion correction algorithm with bilinear interpolation," *Image and VIsion Computing New Zealand*, pp. 408–413, 2003.

[44] M. Corporation, "Directx sdk documentation."

[45] A. Fusiello, "Epipolar rectification," *Department of Science and technology, Univerity of Verona*, 2000.

[46] "Middlebury multiview datasets." http://vision.middlebury.edu/mview/.