Hannes Plank, BSc

Design and Implementation of a Sensor Fusion System featuring a Time-of-Flight and Color Camera

MASTER'S THESIS

to achieve the university degree of

Diplom-Ingenieur

Master's degree programme: Telematics

submitted to

Graz University of Technology

Supervisor

Assessor: Ass.Prof. Dipl.-Ing. Dr.techn. Christian Steger

Institute for Technical Informatics

Advisor: Ass.Prof. Dipl.-Ing. Dr.techn. Christian Steger
Dipl.-Ing. Dr.techn. Norbert Druml (Infineon Technologies Austria AG)

Graz, September 2015

AFFIDAVIT

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master‘s thesis dissertation.

Date

Signature

Kurzfassung

In den vergangenen Jahren ist das Interesse an Tiefenkameras stark gestiegen, und es sind einige Systeme auf dem Endkundenmarkt erschienen. Time-of-Flight Kameras sind dabei das System mit dem kleinsten Formfaktor. Bald werden Tiefenkameras in mobile Geräte wie Tablets und Smartphones integriert werden. Time-of-Flight Kameras messen die Tiefe für jeden Pixel, indem Licht ausgestrahlt wird und die Laufzeit gemessen wird. Ein Bildsensor ist mit einem Photonic Mixture Device für jeden Pixel ausgestattet, mit dem es möglich ist die Phase zwischen ausgestrahltem und gemessenem Licht zu erfassen und in eine Spannung umzuwandeln.

Wie alle anderen Tiefenkameras sind die Daten eines Time-of-Flight Systems nicht perfekt. Durch das relativ große Photonic Mixture Device auf jedem Pixel ist die Auflösung beschränkt. Dadurch, dass nur wenig ausgestrahltes Licht zum Sensor zurückreflektiert wird, sind die Tiefenwerte leicht verrauscht.

Durch die Kombination von einer Time-of-Flight mit einer Farbkamera, lässt sich die Qualität der Tiefendaten stark verbessern. Wenn das Farbbild als Hilfe zum Aufskalieren verwendet wird, kann man dabei die Auflösung verbessern und gleichzeitig das Rauschen verlustfrei unterdrücken. Künftige mobile Geräte mit Time-of-Flight Kameras werden eine geeignete Farbkamera haben. Wenn man für jeden Farbwert auch Tiefeninformation hat, öffnet das neue Möglichkeiten in den Bereichen Computer Vision, Augmented Reality und Computational Photography.

Diese Masterarbeit erforscht die Durchführbarkeit der Farb- und Tiefendaten Fusion auf mobilen Geräten mit hohen Bildraten. Dazu werden zuerst State-of-the-Art Algorithmen vorgestellt und evaluiert. Ein neuer Ansatz wird vorgestellt, welcher Qualität und Geschwindigkeit kombiniert.

Mit der High-End Qualcomm Snapdragon 810 Plattform und einer Farb- und Tiefenkamera wird ein Prototyp gebaut. Die Bildverarbeitungspipeline wird auf dem mobilen Grafikprozessor implementiert. Das Ergebnis ist ein System, welches die Fusion von Farb- und Tiefendaten mit interaktiven Bildraten durchführt. Eine ausführliche Evaluierung zeigt, dass die Tiefendaten eine deutlich verbesserte Auflösung und geringeres Rauschen aufweisen.

Abstract

In recent years, depth sensing systems have gained popularity and have begun to appear on the consumer market. Of these systems, Time-of-Flight (ToF) are the smallest available and will soon be be integrated into mobile devices such as smartphones and tablets. Time-of-Flight cameras measure the depth of a pixel by emitting pulsed light, an image sensor then measures the time it takes the light to travel to the scene and back. This works with a photonic mixture device on each pixel, which is able to convert the phase shift of the incoming light into a voltage.

Like all other available depth sensing systems, Time-of-Flight cameras do not produce perfect depth data. This imperfection is due to the large photonic mixture device on each pixel and therefore the resolution is limited. Time of Flight data is noisy, because the large quantity of emitted light does not travel back to the image sensor.

Combining the data of a Time-of-Flight and a color camera can vastly improve depth image quality. By using the color image as guidance, the depth image can be upscaled with resolution gain and noise reduction. Future mobile devices with ToF cameras will feature a suitable high-resolution color camera. When there is depth information available for every color pixel, new possibilities in computer vision, augmented reality and computational photography arise.

This thesis explores the feasibility of fusing color and depth data on mobile devices with high frame rates. State-of-the-art algorithms are reviewed and evaluated. A novel depth upscaling algorithm is introduced, combining the creation of high quality depth data with fast execution.

The high-end Qualcomm Snapdragon 810 platform, a color and ToF camera are used to create a sensor fusion prototype. The complete processing pipeline is implemented on the mobile GPU to maximize performance. The result is a system capable of fusing color and depth data at interactive frame-rates. As an extensive evaluation reveals, the depth data demonstrates improvement in resolution and reduced noise.

Danksagung

Diese Diplomarbeit wurde im Jahr 2015 am Institut für Technische Informatik an der Technischen Universität Graz durchgeführt.

Ich möchte mich hiermit die Gelegenheit ergreifen, um mich bei den Menschen zu bedanken, die mich während des Studiums und dieser Masterarbeit gefördert und unterstützt haben.

Herrn Ass.Prof. Dipl.-Ing. Dr.techn. Steger und dem Institut für Technische Informatik gilt großer Dank wegen der ausgezeichneten Betreuung der Masterarbeit. Bei Infineon geht oberster Dank an Dipl.-Ing. Dr.techn. Norbert Druml. Er hat mich perfekt betreut und ausgezeichnet unterstützt. Außerdem möchte ich meiner Abteilung der CRE bedanken, in deren angenehmer Arbeitsatmosphäre es möglich war tief in das faszinierende Thema dieser Masterarbeit einzutauchen.

Besonderer Dank gilt meinen Eltern, welche mich während des Studium außerordentlich unterstützt haben und immer an mich glaubten. Ohne sie hätte ich diesen Meilenstein vielleicht nicht erreicht. Bei meiner Freundin Bianca möchte ich mich für die eifrige Unterstützung und schönen Momente bedanken, die mir sehr geholfen haben.

Graz, September 2015 Hannes Plank

Contents

1 Introduction **14**
- 1.1 Motivation . . . 15
- 1.2 Objectives . . . 16
- 1.3 Outline . . . 16

2 Related Work **17**
- 2.1 Time-of-Flight Cameras . . . 17
  - 2.1.1 Principle . . . 17
  - 2.1.2 Problems . . . 18
  - 2.1.3 Post Processing Enhancements . . . 20
  - 2.1.4 Advantages to other Depth Sensing Systems . . . 21
- 2.2 Color and Depth Sensor Fusion . . . 21
  - 2.2.1 Sensor Fusion Rigs . . . 22
  - 2.2.2 Depth Super-Resolution . . . 24
  - 2.2.3 Depth Map Hole Filling . . . 27
  - 2.2.4 Calibration . . . 28
  - 2.2.5 Refining RGB-D Data with Additional Geometric Data Models . . . 29
  - 2.2.6 Multi-Camera Fusion . . . 30
- 2.3 Computer Vision on Graphics Hardware . . . 31
  - 2.3.1 Khronos OpenCL . . . 32
  - 2.3.2 Khronos Vulkan . . . 32
  - 2.3.3 Renderscript . . . 32
  - 2.3.4 Halide . . . 32
  - 2.3.5 OpenGL Compute Shaders . . . 32
- 2.4 Applications of RGB-D Images . . . 33
  - 2.4.1 Image Segmentation . . . 33
  - 2.4.2 Computational Photography . . . 34
  - 2.4.3 Augmented Reality . . . 34
- 2.5 Related Commercial Projects . . . 37

3 Design **38**
- 3.1 Requirements of the Sensor Fusion System . . . 38
- 3.2 The Sensor Fusion Platform . . . 38
  - 3.2.1 Snapdragon 810 . . . 39
  - 3.2.2 High Performance Computing with the Adreno 430 GPU . . . 39

3.2.3 OpenGL ES 3.1 . . . 44
3.2.4 OpenCL and OpenGL Interoperability . . . 44
3.2.5 The Selection of the Color Camera . . . 45
3.3 ToF and Color Sensor Fusion Procedure . . . 46
3.3.1 Calibration . . . 47
3.3.2 Depth to Color Mapping . . . 49
3.3.3 Image-Guided Depth Upsampling . . . 50
3.4 Design of a Depth Upscaling Algorithm . . . 51
3.4.1 Evaluation of the State-of-the-Art Upsampling Algorithm . . . 51
3.4.2 Design of the Guided Depth Diffusion Upscaling Algorithm . . . 52
3.5 Software Design . . . 56
3.5.1 Interface to Depth Camera . . . 57
3.5.2 Interface to Color Camera . . . 57
3.5.3 Data Flow . . . 57

4 Implementation **59**
4.1 The Sensor Fusion Platform . . . 59
4.1.1 Dual Camera Setup . . . 59
4.1.2 Camera Calibration . . . 61
4.1.3 Processing Hardware . . . 62
4.2 Development . . . 62
4.2.1 Development Environment . . . 62
4.2.2 Workflow . . . 64
4.3 Sensor Fusion Framework . . . 65
4.3.1 Android Java Implementation . . . 65
4.3.2 Native C Implementation . . . 67
4.4 Implementation of a State-of-the-Art Upscaling Algorithm for Evaluation . 70
4.5 GPU Implementation of the Sensor Fusion Processing Algorithm . . . 71
4.5.1 Mapping . . . 71
4.5.2 Computation of the Guidance Image . . . 74
4.5.3 Guided Diffusion Depth Upscaling . . . 74

5 Results **80**
5.1 Camera Calibration . . . 81
5.2 Exploration of the Parameter Space . . . 82
5.2.1 Interpolation Parameter Sigma . . . 82
5.2.2 Kernel Size . . . 82
5.2.3 Guidance Image . . . 83
5.3 Performance . . . 86
5.3.1 GPU Parameters . . . 86
5.3.2 Kernel Size . . . 87
5.3.3 Computational Complexity . . . 87
5.3.4 Local Memory Cache . . . 88
5.4 Robustness . . . 89
5.4.1 Holes . . . 89
5.4.2 Depth Resolution Reduction . . . 89

5.4.3 Concurrency Evaluation 90
5.5 Evaluation of the Sensor Fusion Result 92
5.5.1 Comparison with the Joint Bilateral Filter 95
5.5.2 Comparison with the Fast Minimax Path-Based Joint Depth Interpolation Method 97
5.5.3 Comparison with the Anisotropic Total Generalized Variation Method 98

6 Conclusion and Future Work **99**
6.1 Conclusion 99
6.2 Future Work 100
6.2.1 Future Applications 100

List of Abbreviations

AR	Augmented Reality
ARM	Advanced RISC Machines
API	Application Programming Interface
C99	C programming language standard, established in 1999
CGI	Computer Generated Imagery
CUDA	Compute Unified Device Architecture
DOF	Degree Of Freedom
DSP	Digital Signal Processor
DSLR	Digital Single Lens Reflex
DDR	Double Data Rate
GPU	Graphics Processing Unit
GPGPU	General Purpose computation on Graphics Processing Unit
HDMI	High Definition Multimedia Interface
IR	Infra Red
JNI	Java Native Interface
LED	Light Emitting Diode
MST	Minimum Spanning Tree
NIR	Near Infra Red
RANSAC	RANdom SAmple Consensus
RGB	Red Green Blue
RGB-D	Red Green Blue - Depth
RMS	Root Mean Squared
SDK	Software Development Kit
SIMD	Single Instruction Multiple Data
SLAM	Simultaneous Localization And Mapping
SOC	System On a Chip
TOF	Time-of-Flight
UVC	USB Video Class
V4L	Video For Linux

List of Figures

1.1 The principle of a Time-of-Flight depth sensing system [Dru+15] 14

2.1 The princple of a ToF depth sensing system [Han+13] 17
2.2 The photonic mixture device. The difference between $+u_m$ and $-u_m$ is proportional to the depth [RH07] 18
2.3 Depth bias on a checkerboard target. Left: Intensity image; Right: The depth data visualized in 3D [Zhu+11] 19
2.4 Various ToF depth post-processing techniques, applied to data of the PMD PicoS ToF camera with the LightVis [Tec14] software. The first image is unprocessed raw depth data. 20
2.5 Means of RGB-D sensor fusion, from top to bottom: Mapping color to depth; Mapping depth to color; Creating a virtual viewport 22
2.6 Left: Sensor Fusion Rig; Right: Calibration target with holes [Par+11] . . 23
2.7 Simple local depth upsampling methods [TM98] 25
2.8 Depth upsampling with cost volumes [Yan+07] 25
2.9 From left to right: Low resolution depth input: High resolution intensity input; 3D result, Depth image result [Fer+13] 26
2.10 Minimax based upsampling from left to right: The minimax Path and the geodesic path on an image; Seed depth values in image graph; Minimum spanning tree; Subseeds; Each subtree is traversed [Dai+15] 27
2.11 The concept on holes caused by shadows in a structured light camera system [DBC12] 28
2.12 Calibration images of Herrera's method [HCKH11] 29
2.13 The result of the Kinect Fusion algorithm demonstrated with an increasing number of captured frames [Iza+11] 30
2.14 Left: Scheme of the OmniKinect system; Right: 3D visualization of the setup [Kai+12] 31
2.15 Graph Cut by Yu et al [YZ12]: Top: Input color and depth image; Bottom: The result without and with depth influence 33
2.16 Left: 3D model of a real object (toy car); Right: Virtual object, placed behind toy car. [Qua15] 36
2.17 The sensor architecture of the RGB-Z sensor [Kim+12] 37

3.1 The memory model of OpenCL [K.11] 40
3.2 The OpenCL work-groups [K.11] 42
3.3 OpenCL in Android [Int15] 44

3.4 Areas of the depth image covered by the Logitec color camera. Left: VGA(640x480); Right: QHD (640x360) . . . 46
3.5 The RGB-D sensor fusion processing pipeline, demonstrated with the 2005 middlebury dataset [HS07] . . . 46
3.6 Left: Color coded depth image, Right: Amplitude image . . . 47
3.7 Depth camera C_L to color camera C_H data projection from Ferstl [Fer+13] 49
3.8 The blue line marks the edge of the aloe leaf. It is also a border in the depth image. The red edge on the wallpaper does not have a corresponding depth discontinuity. . . . 50
3.9 Errors of the upscaling algorithm of Dai [Dai+15] . . . 52
3.10 The input for the upscaling algorithm . . . 53
3.11 The influence path of depth value (7/5) to pixel (3/6). . . . 53
3.12 The guidance image filter kernel . . . 54
3.13 Left: The color image; Right: The guidance image . . . 55
3.14 The interface diagram between the Java and C part of the implementation. 56
3.15 The data flow of the sensor fusion system . . . 58

4.1 The components of the sensor fusion implementation . . . 59
4.2 The sensor fusion camera system . . . 60
4.3 Left: Visualization of the camera configuration and calibration targets by Bouguet Matlab toolbox [Bou15]; Right: Calibration Image with mapped corners . . . 61
4.4 The Dragonboard APQ 8094, a development kit for Snapdragon 810 . . . 62
4.5 Modules of the dual-plattform implementation. Red: Module used by Windows; Blue: Module used by Android; Red/Blue: Module used on both platforms . . . 64
4.6 The development cycle for developing the sensor fusion algorithms . . . 65
4.7 Class diagram of the sensor fusion framework . . . 66
4.8 Module diagram of the C implementation of the GPU processing handling . 68
4.9 Results of the re-implementation of the upscaling algorithm of [Dai+15] on the Middlebury [HS07] datasets. From top to bottom: Color image; Ground truth depth image; Upsampled depth image. The original depth image resolution: 80x60 pixel . . . 70
4.10 The color and ToF sensor fusion procedure . . . 72
4.11 The memory buffers of the depth upscaling GPU computation . . . 75
4.12 The influence of a depth value with kernel size of 3, and an example pixel path . . . 76
4.13 Diffusion parsing. All operations depend on each previous step. . . . 78
4.14 The concept of local memory caching . . . 79

5.1 Left: Fused color and depth image; Right: 3D surface mesh . . . 80
5.2 The re-projection error of the color image in pixels [Bou15] . . . 82
5.3 Evaluation of the parameter σ . . . 83
5.4 Evaluation of the kernel size . . . 83
5.5 Saturation restriction; Left: The guidance images; Right: The resulting upscaled depth images . . . 84

5.6 Evaluation of edge detectors for guidance image computation. Left: The guidance images; Right: The resulting upscaled depth images 85
5.7 The execution times of the GPU processing pipeline stages 86
5.8 The memory transfer volume and execution time of the depth upscaling stage 87
5.9 X-Axis: The top 20 fastest local work-group size configurations; Y-Axis: The execution time advantage compared to no local memory caching 89
5.10 Interpolation of artificially created depth holes 90
5.11 The sensor fusion result in comparison with a reduced original depth data resolution 91
5.12 The color images of the two test-sets 92
5.13 The sensor fusion result for the noisy test-set. Top: Depth images; Bottom: 3D surface meshes 93
5.14 The sensor fusion result for the test-set 2. Top: Depth images; Middle: 3D surface meshes re-lighted; Bottom: 3D surface meshes without additional lightning 94
5.15 The joint bilateral filter [Kop+07] in comparison with the guided diffusion filter. From top to bottom: Depth image; Zoomed depth image; 3D surface mesh; Sensor fusion result with reduced original depth resolution 95
5.16 The bilateral filter can improve the image guided diffusion 96
5.17 The results of the minimax path-based depth interpolation [Dai+15], compared to the original depth image and the proposed guided diffusion upscaling method 97
5.18 Comparison of guided diffusion and anisotropic total generalized variation upscaling methods 98

List of Tables

3.1 The processing units on the Qualcomm Snapdragon 810 platform 39
3.2 The OpenCL memory model characteristics of the Adreno 430 GPU 39

5.1 The intrinsic parameters of the color camera 81
5.2 The intrinsic parameters of the ToF camera 81
5.3 The execution times of the top 5 local work-group size configurations 86
5.4 The memory operations and bandwidth of the depth upscaling algorithm for different kernel sizes 88
5.5 Corrupted Pixels due to GPU thread conflicts 90
5.6 The number of operations per color pixel compared to the joint bilateral filter (1 iteration) [Kop+07] 97

Chapter 1

Introduction

In recent years, there have been huge advancements in depth sensors. The first mass-produces device on the consumer market was the Microsoft Kinect Sensor, which had a massive impact on research, products and applications in the field.

Currently, depth sensors are on the edge of becoming available on mobile devices. Depth sensing based on the Time-of-Flight principle, is the smallest system available. The Time-of-Flight method is based on measuring the time a photon takes to travel from the camera to the scene. This time is proportional to the depth of each pixel. The principle is illustrated in Figure 1.1.

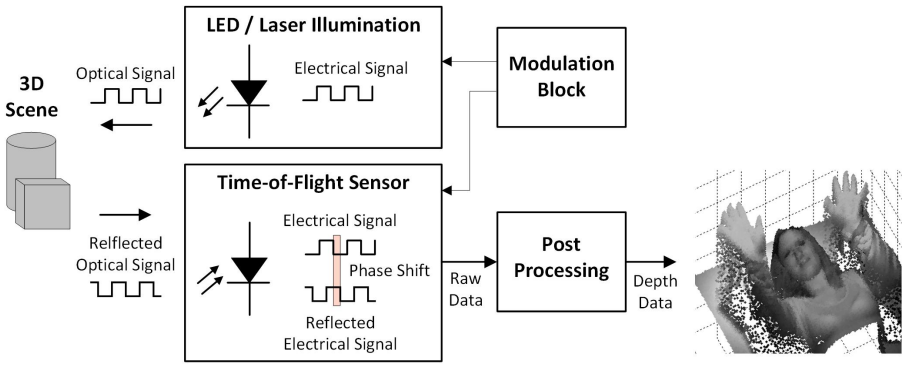

Figure 1.1: The principle of a Time-of-Flight depth sensing system [Dru+15]

An active illumination unit emits a modulated light signal. The light signal is reflected by the scene captured by the ToF sensor. The ToF sensor measures the phase difference of the incoming to the outgoing light. A photonic mixture device on each pixel enables the conversion of the phase to a voltage, which is proportional to the depth. All available systems have flaws and disadvantages. Currently, there is no superior depth sensing system on the market. Some weaknesses of the Time-of-Flight technology include:

- **Noise**

  ToF sensors often produce noisy data. The amount of sensed light decreases dramatically over distance.

- **Resolution**

  A ToF camera measures the phase difference directly per pixel on the sensor chip with the photonic mixture device. The device makes the pixels relatively large which limits the resolution.

- **Measurement Errors**

  Various errors are introduced during the measurement process, causing not all pixels to contain valid depth values. The most common is the low-signal error, when sufficient light is not reflected.

There are numerous countermeasures for these errors, but using additional information is the best way to counter these flaws. While a combination of several different depth sensing systems makes sense and has been thoroughly researched, practical applications are limited due to the increased complexity. Fusing depth and color can enhance depth resolution and reduce noise.

1.1 Motivation

The availability of Time-of-Flight based depth sensing systems in smartphones and tablets soon is approaching. Their resolution is limited and their applications are currently focused on mapping and indoor-navigation. High-resolution color cameras are ubiquitous on mobile devices like smartphones and tablets. Research has shown, that depth image quality can be improved immensely by using color information. There is a strong correlation between depth and color discontinuities. These discontinuities can be preserved, when upscaling the depth image, causing the depth image to improve resolution. When every color image pixel is associated with a depth value, new applications for depth sensors arise. Fields like augmented reality, computational photography and computer vision in general can benefit of a unified color and depth sensor.

This master's thesis aims to show the feasibility of color and depth sensor fusion on mobile devices. State-of-the-art sensor fusion methods are reviewed and tested. A novel algorithm, capable of improving Time-of-Flight depth images in terms of resolution and lossless noise reduction is proposed. The key difference between state-of-the-art approaches is the speed and feasibility for an efficient implementation on a mobile platform's GPU. A prototype is developed to demonstrate the proposed algorithm and to verify the feasibility by reaching interactive frame-rates.

1.2 Objectives

The focus is on the implementation of a sensor fusion system on a mobile platform. Literature research is conducted in the fields of image sensors, depth sensor fusion and GPU computing. State-of-the-art fusion algorithms are evaluated in terms of performance and quality. The target platform is examined and an efficient GPU-assisted implementation is designed and developed. These activities can be categorized into:

- Research on fundamentals and similar state-of-the-art projects
- Design of a suitable sensor fusion algorithm
- Component selection and setup of the sensor fusion platform
- Implementation of the algorithm and a demonstration application

The outcome of this thesis is a mobile platform with a color and ToF camera, producing a fused color and depth image.

1.3 Outline

The way this thesis is structured resembles the process of the project. Chapter 2 reviews the related research and is the result of extensive literature research. After introducing the various topics of this thesis, the state-of-the-art publications are listed and reviewed in this chapter. Chapter 3 specifies the design requirements for the sensor fusion hardware components and how they are met. It shows the evaluation of the most recent color and depth fusion algorithm, and how the final algorithm is designed. Aspects of software development and GPU computing on the mobile platform are mentioned as well. Chapter 4 describes the implementation of the software on the CPU and GPU and shows the final sensor fusion platform. The results are discussed in detail in Chapter 5. The performance, quality and parameters of the sensor fusion algorithm and its intermediate steps are evaluated and discussed. Chapter 5 contains the conclusion and possible future work.

Chapter 2

Related Work

2.1 Time-of-Flight Cameras

Time-of-Flight (ToF) cameras are range cameras that sense depth. Each pixel encodes the distance between the camera and a point in the scene. This works by illuminating the scene with pulsed light and measuring how much time the light takes to travel from the scene, back to the camera.

2.1.1 Principle

Figure 2.1 shows the principle of a ToF camera system. An active illumination unit emits pulsed infra-red light. The light is reflected and captured by the image sensor. With the use of optical filters and additional signal processing measures, other light-sources do not influence the measurement.

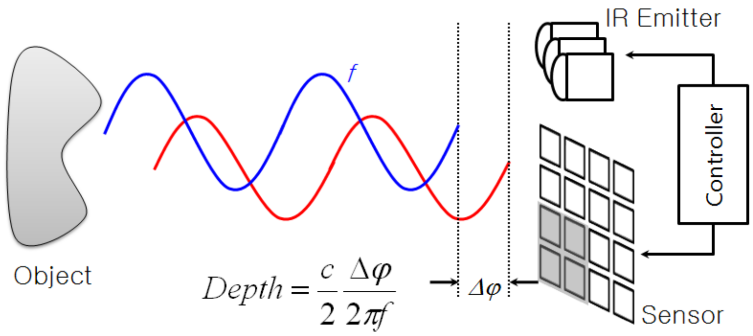

Figure 2.1: The princple of a ToF depth sensing system [Han+13]

The incoming pulsed light signal has a phase shift in comparision to outgoing light. The phase shift is caused by the travel time of the light and is thus proportional to the distance. There is a photonic mixture device for each pixel on the sensor, as shown in Figure 2.2. The purpose is to convert the phase difference into a voltage. Electron holes are created by the photons arriving on the sensor's surface. They are directed into either of two buckets. Depending on the modulation, an electrical field regulates which bucket is filled. After

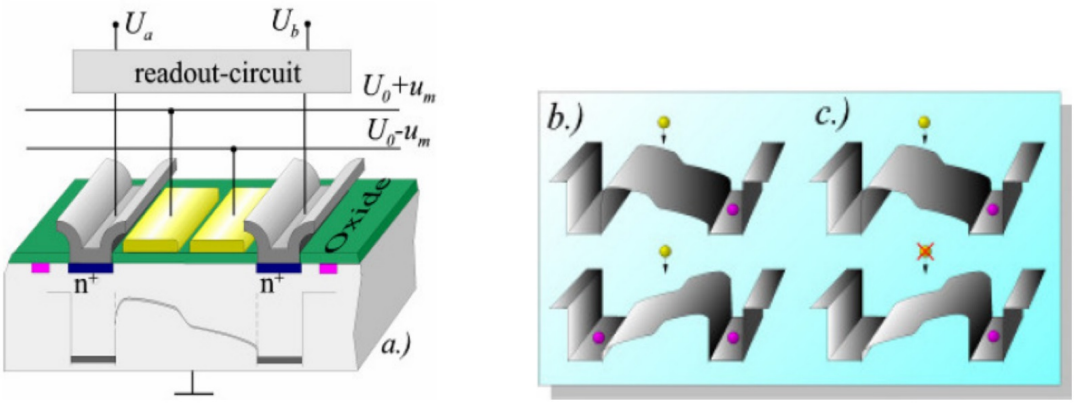

Figure 2.2: The photonic mixture device. The difference between $+u_m$ and $-u_m$ is proportional to the depth [RH07]

several cycles, the content of the buckets is read out. The phase shift of the light depends on the voltage between the buckets. The measured depth is proportional to this voltage. The voltage is digitized and converted to a distance in a further processing step.

2.1.2 Problems

One of the goals in this project is to improve the depth image quality. This section provides an overview of the imperfections of ToF sensing systems and their countermeasures.

Multi-Path Interference

A light pulse illuminates the complete scene. Since some light is diffusely reflected, some areas are indirectly illuminated. Consequently, this indirectly reflected light is also captured by the ToF sensor and can cause erroneous measurements. Stefan Fuchs [Fuc10] introduced a multi-path interference compensation method, based on simulating directional interference on captured depth images.

Background Light

Other light sources, like daylight, can influence the measurement. The active illumination unit therefore emits infrared light. An optical filter eliminates most of the background illumination. There exist on-chip per-pixel signal processing circuitry to counteract the influence of the background light. The patented PMD SBI [Mö+05] method is implemented on the ToF camera used in this master's thesis.

Transparent/Reflective Surfaces

Surfaces that do not reflect light diffusely are a challenge for most active illumination based depth sensing systems. If a surface does not sufficiently reflect light from the illumination unit, there is no measurement possible.

Resolution

The current generation of ToF sensors has a relatively low image resolution. This is due to the the relatively large photonic mixing device for every pixel. One of the goals of this thesis is to improve the low resolution depth images by using high-resolution edge information from an additional color image sensor. Various state-of-the-art methods of image guided depth super-resolution are introduced in section 2.2.2.

Power Consumption

The small form-factor of ToF sensors enables usage in mobile devices. Since the scene is actively illuminated ToF systems have a relatively high power-consumption. The power of the active illumination unit can be reduced, if the pixels on the image sensor are more functional to incoming light. Due to the photonic mixture device, only a limited area of each pixel is sensitive to light. An array of micro-lenses can be placed over the sensor. Each pixel has its own micro-lens, projecting the incoming light to the relevant photosensitive area.

Depth Bias

If a surface is dark, only a small amount of light is reflected, causing erroneous depth measurements. Figure 2.3 shows the depth bias on a checkerboard. When dark surfaces reflect a specified portion of light, the depth bias can be compensated using calibration. Ferstl et al. [Fer+13] derive a function of the depth bias and compensate the bias to a certain degree.

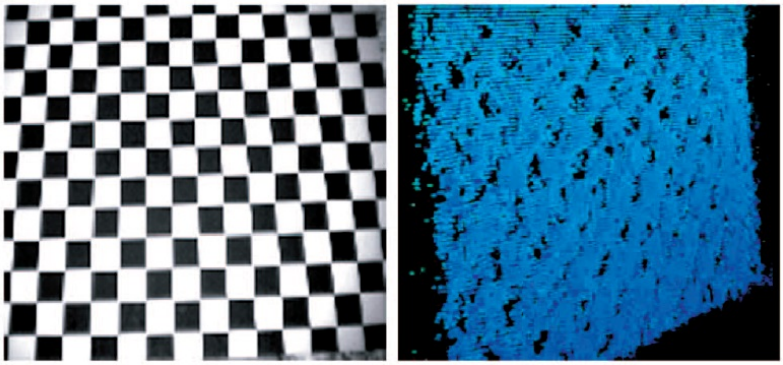

Figure 2.3: Depth bias on a checkerboard target. Left: Intensity image; Right: The depth data visualized in 3D [Zhu+11]

Motion Artifacts

Each depth value is measured by the phase difference of several incoming light pulses. If there is movement between these pulses, the depth image suffers in quality. Due to the high frame rates of ToF systems, this effect is less inferior than motion blur during color image acquisition.

2.1.3 Post Processing Enhancements

This section introduces post-processing techniques, which refine the measured depth data. The sensor fusion implementation of this masters project can also be seen as post-processing technique since it improves depth data quality. The techniques in this section however are applied only on the depth data and do not use a color image as guidance. Figure 2.4 shows three established post-processing techniques. These techniques are used by PMD SDK [Tec14], a software development kit for PMD ToF cameras, which is used in this project.

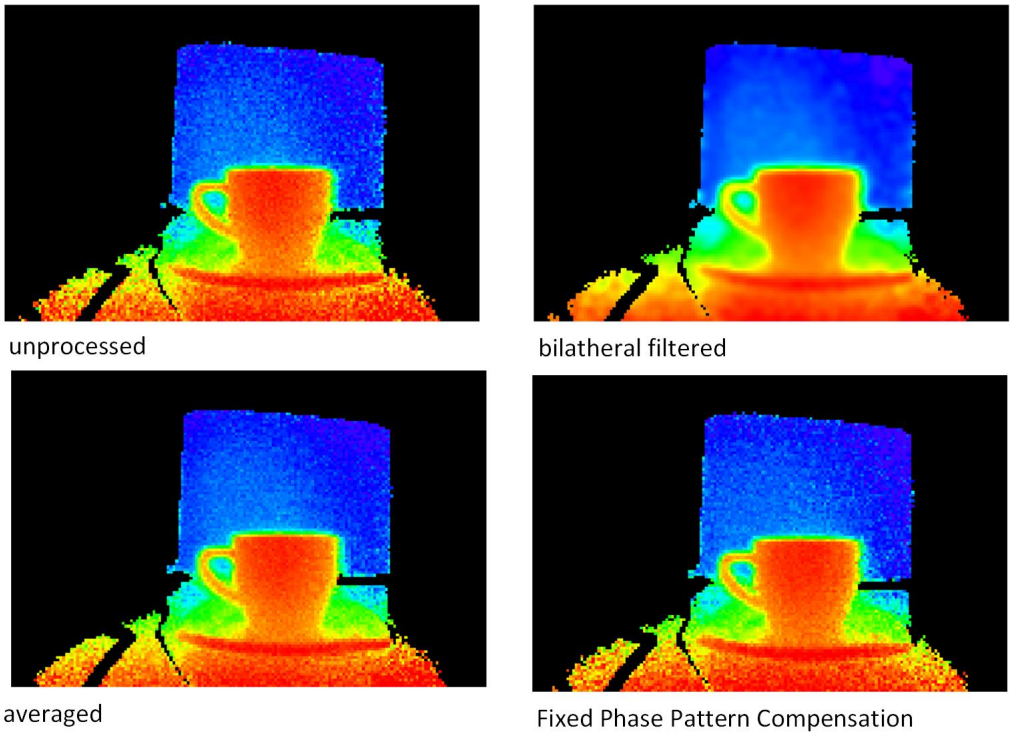

Figure 2.4: Various ToF depth post-processing techniques, applied to data of the PMD PicoS ToF camera with the LightVis [Tec14] software. The first image is unprocessed raw depth data.

Pixel Flagging

Time-of-Flight cameras are also able to measure the amplitudes of reflected light. The higher the amplitude, the more light is reflected. This provides a reliability measure for each pixel. Pixels with too low amplitude can be flagged by the sensing system or processing software. These pixels might be disregarded from further use.

Bilateral Filtering

The bilateral filter [TM98] is introduced in Section 2.2.2. It can be used to filter noisy raw depth data directly. The bilateral filter is edge-preserving and thus useful in smoothing noisy Time-of-Flight depth data. The filter can also be modified to use the amplitudes image as guidance [Tec14].

Frame Averaging

If a fixed camera captures a static scene, the depth frames can be averaged to minimize noise. This is a good way to receive a reference measurements for testing post-processing methods.

2.1.4 Advantages to other Depth Sensing Systems

Structured Light

In a structured light system, a pattern is projected. The depth is estimated by capturing the pattern and calculating the disparity for each pixel. Since the projector and the camera need a baseline of several centimeters, the form factor is rather big. Some areas of the depth image may not hold any depth information due to occlusions. Means of filling these undefined holes in depth images were published in [Liu+13] and [DBC12]. Since holes can also appear in ToF cameras due to invalid measurements, hole-filling methods are discussed in Section 2.2.3.

Stereo

When depth is retrieved from stereo matching, the algorithms are usually computationally complex. The image resolution is good, the depth calculation however fails for areas without distinctive texture. There are very promising attempts to combine a ToF sensor with a stereo camera system to combine the advantages of each system [Nai+13].

2.2 Color and Depth Sensor Fusion

RGB-D sensor fusion is the combination of data from a color (RGB) and a depth camera (D). Associating a color image with depth information provides a better approximation of reality.

The human brain understands the depth of a scene by interpreting the two images from the eyes. Computers can also use two cameras to recognize depth, but have to run a stereo matching algorithm. ToF systems provide a direct measurement of depth. Every depth imaging system has its flaws and currently, no superior system has been established. Imperfect depth data can be improved however, by fusing depth with color images. Numerous approaches are reviewed in Section 2.2.2. As shown in Figure 2.5, there are three categories of fusing depth and color data:

- **Mapping color to depth**

  Each depth value is assigned a color value. The result is a point-cloud with color information for each point. This the can be useful for 3D reconstruction [Kai+12] and

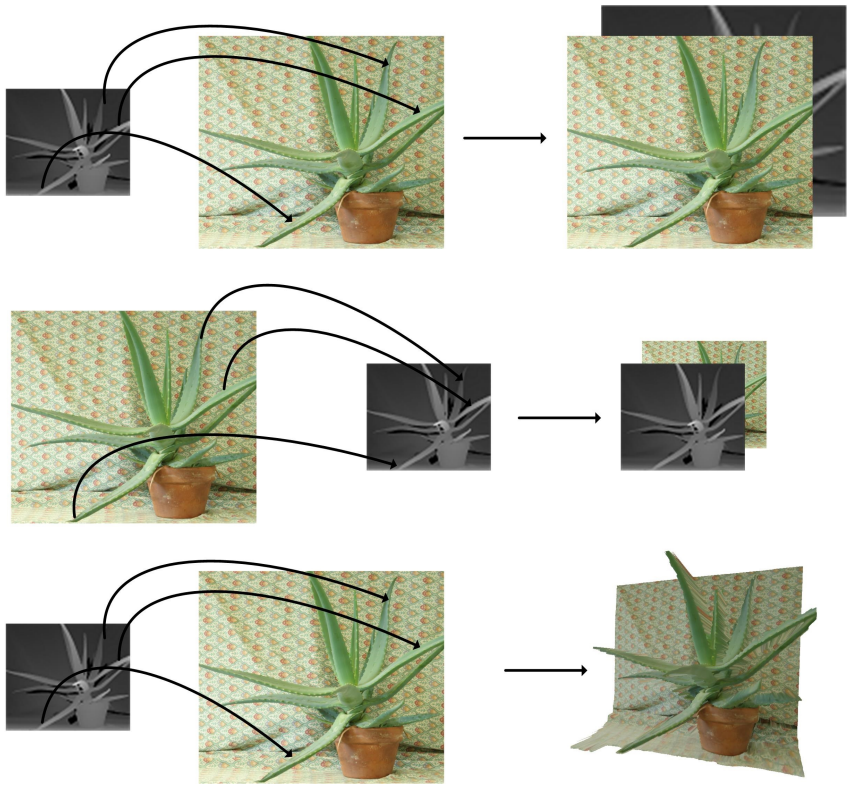

Figure 2.5: Means of RGB-D sensor fusion, from top to bottom: Mapping color to depth; Mapping depth to color; Creating a virtual viewport

feature matching in simultaneous mapping and localization (SLAM) based systems [Goo14].

- **Mapping depth to color**

  Every pixel in an RGB image holds an approximated depth value. Since the RGB images in most camera systems have a higher resolution, there is a lot of research on image-guided upscaling methods, see Section 2.2.2. The main goal of this project is to implement this way of color and depth sensor fusion on a mobile device.

- **Virtual Viewport**

  At first every color value is associated with an approximate depth value. With this mapping, the image can be warped to an arbitrary viewport. This is known as view synthesis and has applications in CGI, movie production [FBK10] and mixed/augmented reality.

2.2.1 Sensor Fusion Rigs

This section shows camera setups used for depth and color sensor fusion. Ferstl et al. [Fer+13] use a ToF and a high-resolution color camera for sensor fusion. The algorithm they developed is introduced in Section 2.2.2. After deriving the intrinsic and extrinsic

camera parameters, they define the intensity camera position as the world coordinate center. Depth values are projected to world coordinates and re-projected to the image space. Depth values are associated just with the nearest RGB pixel, resulting in an image frame with sparse depth values. The missing depth values are assigned during their super resolution upsampling algorithm. The algorithm uses information from the RGB image to create a super resolution depth image and is described briefly in Section 2.2.2. The cameras are mounted vertically in this setup. Since the dimensions of the cameras have a larger width than height, stacking the cameras on top of each other brings the optical centers closer together. A depth and RGB camera pair can be arranged arbitrarily, as long as both face the scene. In such a dual camera setup, the disparity between the cameras is ideally as small as possible. Otherwise, there can be huge areas without depth information after warping. Various approaches to interpolate these holes are shown in section 2.2.3.

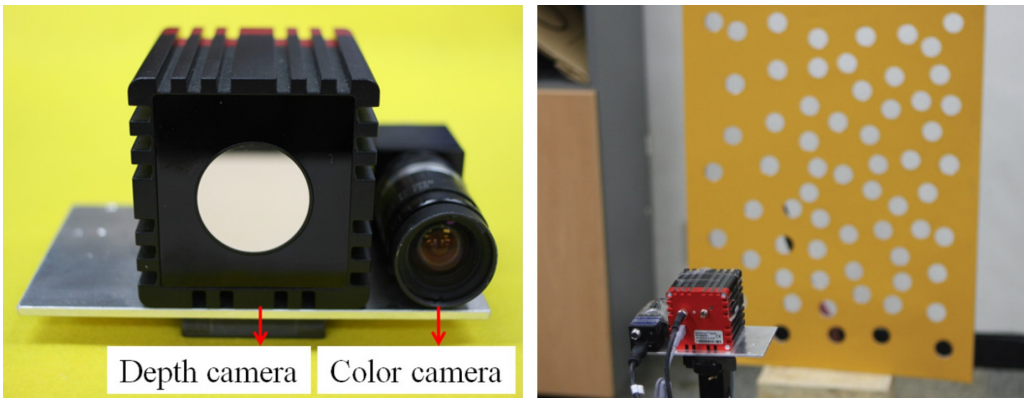

Figure 2.6: Left: Sensor Fusion Rig; Right: Calibration target with holes [Par+11]

The rig used by Park et al. [Par+11], shown in Figure 2.6, is used to demonstrate the depth image upsampling algorithm that was used. A SwissRanger ToF camera is mounted next to a Point Grey Research RGB camera. As there is no intensity image available with this ToF camera, the method of calibration is different. A planar calibration pattern with circular holes is used. The holes in this pattern, can be located by the depth and the RGB camera.

Gudmundsson et al. [GAL08] built a system to fuse a ToF camera with a stereo depth sensing system. The images of the RGB cameras are cropped to get the same field of view as the ToF camera. Stereo-calibration is used to calibrate the ToF camera with each color camera. The high resolution image is down-scaled to the resolution of the ToF camera. The stereo cameras are then calibrated using the full resolution.

Another stereo and ToF fusion system was set up by Zhu et al. [Zhu+11]. Besides geometric calibration, they also perform photometric calibration of the ToF sensor. A planar checkerboard target is captured with 16 different distances. Their experiments conclude that the depth bias can be described as a per pixel linear function. From this observation, per pixel look-up tables are generated, and the depth values are refined.

2.2.2 Depth Super-Resolution

Since depth cameras usually offer lower image resolutions than RGB cameras, improving the resolution of the depth sensor is a well researched topic. Langmann et al. [LHL11] review a number of approaches.

To be able to compare the results of a depth upsampling algorithm, there needs to be a metric to quantify the quality of the new depth image. Scharstein et al. [SS02] compare the results of various stereo matching algorithms with groundtruth data. They use the RMS error and the percentage of bad matching pixels to compare the generated depth image d with groundtruth data g. The RMS error R is the mean quadratic distance between the depth values. N is the number of pixels.

$$R = (\frac{1}{N} \sum_{(x,y)} |d(x,y) - g(x,y)|^2)^{\frac{1}{2}}$$

The percentage of bad pixels B, is a measurement on how many pixels differ more than a defined tolerance δ from the groundtruth.

$$B = \frac{1}{N} \sum_{(x,y)} (|d(x,y) - g(x,y)| > \delta)$$

Color image guided depth upsampling methods can be classified by local and global methods. Local depth image upsampling means that mapped depth values have a limited local influence on the output. These methods are sometimes simple, however when applied iteratively may be complex. The global methods generally yield to better results, but are more computationally intensive.

Local Methods

The premise of these upsampling methods is that depth values are mapped to the high-resolution color image. There are many undefined depth values in the resulting sparse depth image. The local methods aim to interpolate these undefined values. Some methods, like the one introduced in this thesis, overwrite the mapped depth values for noise reduction.

Figure 2.7 shows the simplest methods. The color images in the figure demonstrate how simple viewport wrapping appears in each upsampled depth image. The most primitive method is nearest neighbors upsampling. Each pixel is assigned the closest depth value. Bicubic upsampling uses bicubic interpolation of the surrounding depth values for each pixel. Gaussian upsampling weight the distance to the surrounding depth values with a Gaussian kernel.

The last upsampling method shown in Figure 2.7 however, is a method which takes the color image as guide. It is a variation of the bilateral filter by Kopf et al. [Kop+07]. Bilateral filtering of color images is first introduced by Tomasi [TM98]. It was intended to smooth images while preserving edges. It works by calculating an average neighborhood for each pixel. The neighboring pixels are weighted by color similarity and the euclidean distance. Homogenous areas in the image are smoothed, while sharp edges are preserved. Camera pairs consisting of a depth and RGB camera can improve their depth resolution

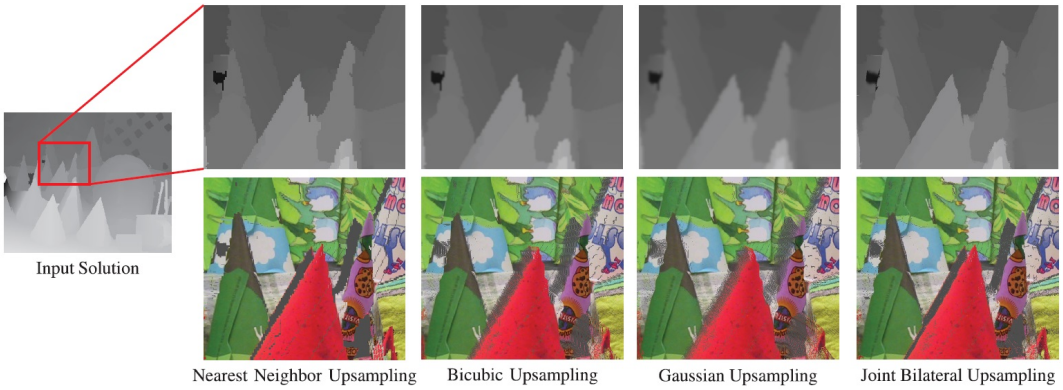

Figure 2.7: Simple local depth upsampling methods [TM98]

by bilateral filtering using the color image. Kopf et al. [Kop+07] introduced this idea which works by weighting the image kernel by the color distance of the guidance image. It enhances the information of the depth image, because depth discontinuities correlate with edges of corresponding color images [TF03]. The joint bilateral filter is one of the inspirations of the design of the filter used in this project.

There have been many attempts at using bilateral filtering to enhance depth maps so far. An advanced approach was published by Yang et al. [Yan+07]. Figure 2.8 sketches their idea: First they construct a cost volume and apply a bilateral filter to each slice of the volume. The best cost is then selected and iteratively applied to the depth map. The refined depth map is updated and a new cost volume is created. This process is applied iteratively until a final depth map is generated.

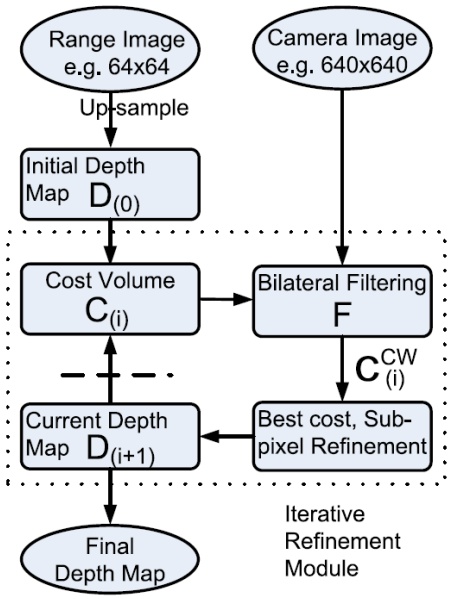

Figure 2.8: Depth upsampling with cost volumes [Yan+07]

Lui et al. [LTT13] propose a depth upsampling algorithm utilizing geodesic distances. The approach is also based on the premise of having a high resolution color image and a 1-1 mapping between the depth and color pixels. A geodesic curve in this context means the shortest path on an 8-connected image graph. The weights of the graph are the differences of the intensity values. For each pixel, the geodesic distance to the location of all projected depth values is calculated. By weighting the depth values with the geodesic distance, the value of the high resolution depth-map image is estimated. With this algorithm, known depths are propagated on the high resolution image, while depth discontinuities are preserved. Since this method has a complexity of $O(n^2 \log n)$, the authors introduce restrictions to reduce the complexity. They assume that only K geodesic closest depth pixels are needed for a good approximation. It is further assumed that pixels with a low geodesic distance are spatially close. With these restrictions, an algorithm with the complexity of $O(K \cdot n)$ is proposed. Since this upsampling method starts with a high resolution sparse depth image, it is robust against missing depth information. If large areas are affected, the parameters of the algorithm need to be adapted.

Global Methods

Ferstl et al. [Fer+13] improve the lateral depth resolution by formulating a convex optimization problem. The data term forces the solution to be similar to the initial depth map. A second order regularization term preserves sharp edges according to the color image. The optimization problem is convex, but not smooth and is minimized by primal-dual optimization. The computation can be parallelized, however the optimization is computationally complex, and not realtime capable. A sample of the algorithm is shown in Figure 2.9.

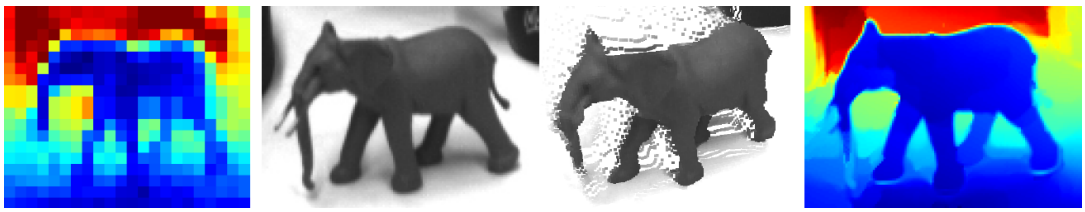

Figure 2.9: From left to right: Low resolution depth input: High resolution intensity input; 3D result, Depth image result [Fer+13]

Dai et al. [Dai+15] follow a similar approach. They claim that their algorithm improves quality and performance of the previously mentioned solution by Lui et al. [LTT13]. The algorithm also interprets the color image as a graph. The depth values are sparsely mapped to the RGB image and are denoted as seed pixels. Instead of using the geodesic distance to determine the filtering weights of the depth seed pixels, the method uses the minimax path. The minimax path is the path between two nodes in a weighted bidirectional graph which minimizes the maximum edge weight. Using the minimax path distance as a measure to distribute depth information along an image has the advantage that depth information barely spills across color discontinuities, resulting in a sharp depth image. Since all minimax paths of the high resolution color image need to be calculated, the problem can be formulated as the minimum spanning tree problem. The spanning

tree, as seen in Figure 2.10, is then cut into many smaller trees on the depth seeding point nodes. This is possible because it is assumed that one depth seed pixel is the most reliable source of depth information hence no further depth information has to be spread on this path. The authors introduce a number of optimizations, to reduce the computational complexity to O(n). They propose a node updating algorithm where every pixel on the image graph has to be visited twice. The benchmarks of this method are comparable to global methods, however the authors do not publish high quality results for the evaluation. Since this algorithm appears to be ideal for the sensor fusion prototype, it was implemented and evaluated in Chapter 3.

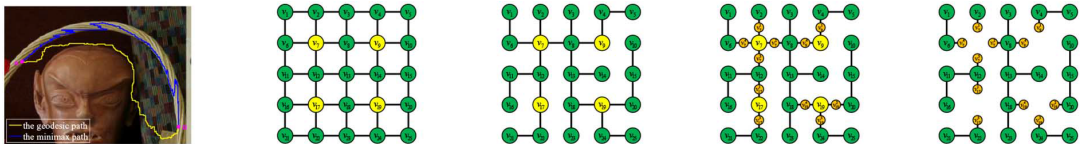

Figure 2.10: Minimax based upsampling from left to right: The minimax Path and the geodesic path on an image; Seed depth values in image graph; Minimum spanning tree; Subseeds; Each subtree is traversed [Dai+15]

2.2.3 Depth Map Hole Filling

When a depth image is warped to a new viewport, some areas of the new image might not be assigned with any depth. In this project, the new viewport is the color camera, which is located next to the depth camera, both facing in the same direction. The holes are regions which are occluded in the original image. The same artifacts occur in structured light systems, when objects occlude the projector. Figure 2.11 shows the geometric cause of holes. They also occur in RGB-D sensor fusion, when the depth image is mapped to the coordinate space of the RGB image. The sensor fusion prototype does not suffer from holes since the cameras are mounted very closely together. However, holes in the form of invalid measurements can appear and are evaluated in Section 5.4.1.

Another reason for undefined depth pixels are invalid measurements. This is mostly caused by depth pixels out of range of the sensor or in the case of the active illumination systems transparent or reflective surfaces. Since the Microsoft Kinect camera system is prone to delivering incomplete depth frames, a lot of publications use this system to demonstrate hole filling methods.

Danicu et al. [DBC12] propose a fast method of filling holes in Kinect depth images based on morphing operations. The method uses an algorithm to processes the horizontal lines of the depth image. The depth value profile of each line is scanned for holes and filled with the neighboring depth value which is further away. This exploits the fact that holes in the depth frame can be seen as shadow cast on the background of the scene. No information from the color image is used for interpolation. It also applies to holes caused by depth image warping, since this scenario is comparable.

Liu et al. [Liu+13] formulate an energy minimization problem to gather the missing depth information. They use the RGB camera of the Kinect to exploit the correlation between color and depth values as described in Section 2.2.2. The idea is to use an

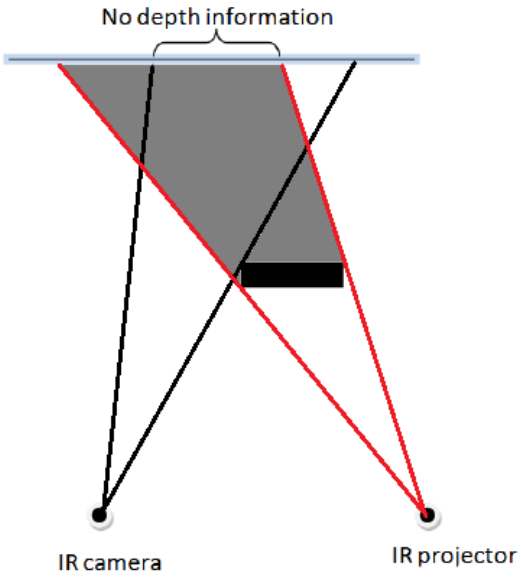

Figure 2.11: The concept on holes caused by shadows in a structured light camera system [DBC12]

energy function based on the correlation between depth pixels and a small RGB image neighborhood. To preserve sharp edges, a TV_{21} prior is incorporated into the energy term.

It is also possible to use the depth upsampling method of Ferst et al. [Fer+13] as described in Section 2.2.2. As previously mentioned, this method improves the depth image resolution by using RGB information and claims to also fill holes. As larger holes do net seem to be perfectly filled, using a different method to fill holes as a pre-processing step might be an advantage. The method of Lui et al. [Dai+15] introduced in Section 2.2.2, interpolates holes automatically.

2.2.4 Calibration

Camera calibration is an essential step in many fields of computer vision. Calibration makes the difference between measuring and sensing. Calibration of camera systems can be classified into intrinsic and extrinsic calibration. The intrinsic parameters of a camera describe the internal workings.

The extrinsic camera parameters enable a transformation between 3D world coordinates to 3D camera coordinates. Depending on the calibration method, the origin of the world parameters can either be a calibration target or an arbitrary point. In a multi-camera setup, it might be convenient to define the origin of the coordinate system in the center of one of the cameras. To describe a camera in world coordinates, the simple pin-hole camera model is a sufficient approximation. Calibrating a depth and a color camera is not straight forward, since it requires a precise depth target.

Herrera et al. [HCKH11] describe a practical process to calibrate a depth and color camera pair. They come to the conclusion, that calibrating cameras individually and then calculating the pose introduces error. They propose an unified approach, where color and

depth features simultaneously improve the calibration of the camera pair. The calibration can be performed using a checker board pattern for the RGB camera and a planar surface like a table for the depth camera.

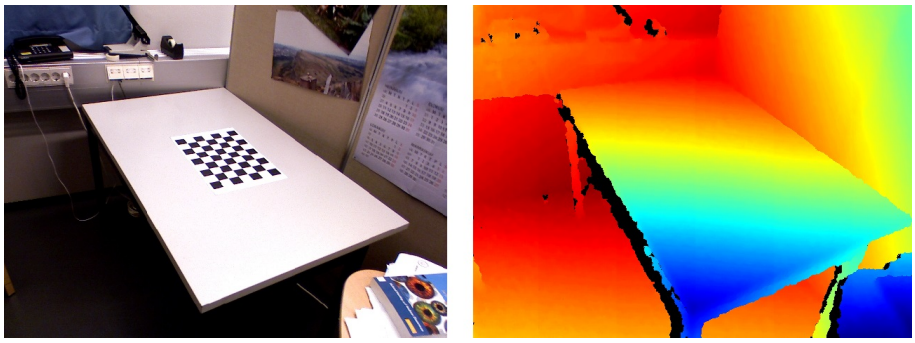

Figure 2.12: Calibration images of Herrera's method [HCKH11]

A sample calibration image pair is shown in Figure 2.12. The benefit is that no high cost calibration equipment, like depth targets or robot arms are necessary. An advantage of ToF depth cameras are the availability of an intensity image. This image is a gray-scale representation of the scene, illuminated by the ToF light source. It is gathered by measuring the amplitude of the emitted light impulses. Calibrating a system of two 2D cameras is simpler and more practical. The classic method of calibrating a 2D camera system involves capturing a planar visual pattern. The 3D coordinate system of the checkerboard is known and a projection matrix from 2D to 3D can be estimated. There are well-established methods and tools like OpenCV or Bouguet Camera Calibration Toolbox for Matlab [Bou15]. The later is used for calibration in this project.

It is also possible to do camera self-calibration as described by Cui et al. [FLM92]. A self calibration only requires multiple images from the same static scene, captured by the same camera from multiple positions. Self calibration is useful, if the geometric relation between the cameras is not fixed. While the calibration procedure is simple and flexible, self calibration methods prove to be inferior compared to calibration with a planar pattern according to [CN10].

2.2.5 Refining RGB-D Data with Additional Geometric Data Models

It is possible to use depth image sequences to create a 3D map of the captured objects or scene. While these techniques are not the subject of this thesis, these maps can be reprojected and be used to refine depth images. A very popular and well working algorithm is Kinect Fusion [Iza+11]. It reconstructs a dense surface model by integrating 2D pointcloud data over time. Due to 6 degrees of freedom (DoF) pose tracking, it is possible to move the depth camera and gather depth images from multiple viewports. As seen in Figure 2.13, the captured 3D representation of the scene increases its detail during the capturing process. Errors introduced from the depth sensing system (in this case the Microsoft Kinect) decrease over time as seen Figure 2.13.

Since the camera position and orientation are tracked, it is possible to convert the gathered surface model into a high quality RGB-D image. Knowing the camera position

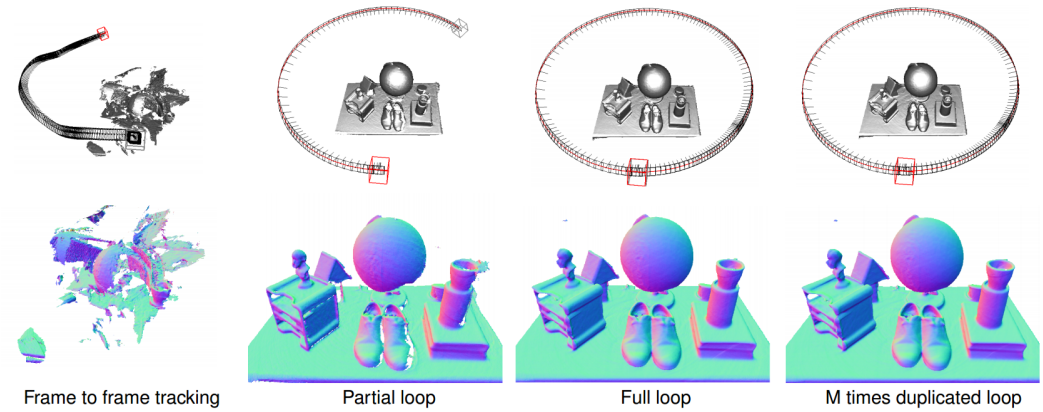

Figure 2.13: The result of the Kinect Fusion algorithm demonstrated with an increasing number of captured frames [Iza+11]

and having a high quality surface model, also opens possibilities for high quality augmented reality. It is possible to place virtual objects precisely into an existing scene. Ray-tracing the surface model gives a good depth map and enables high quality occlusions. The drawback on this algorithm is that modern PC hardware is required to archive interactive frame rates. Fast camera movements cause the tracking to fail and re-initializtion takes some time. It also does not solve the problem of detectable surfaces like glass.

2.2.6 Multi-Camera Fusion

Fusing depth and RGB data is not limited to a camera pair. There are a variety of approaches to fuse data from multiple cameras. Stereo cameras can also sense depth by matching patches of the images and calculating disparities. This can lead to high resolution depth images, however there are serious drawbacks. Some areas often do not have depth information. This is due to missing texture or repetitive patterns. The calculation of the depth map is also computationally complex. Using a second low resolution depth camera can compensate for these flaws. The stereo depth computation can be sped up, since knowing the approximate depth reduces the search space during feature matching. The topic of stereo and ToF fusion is well researched and Grzegorzek et al. [Nai+13] provide an overview in their survey.

Zhu's [Zhu+11] approach fuses a ToF camera with an RGB stereo camera system based on reliability. First they apply a photometric calibration which is described in Section 2.2.1. They use the different characteristics of the cameras to calculate a reliability map for each depth sensing system. The reliability of the stereo system is retrieved from the costs of the stereo matching algorithm. As previously mentioned, the accuracy of ToF depth acquisition suffers from dark areas. Since the active illumination unit and the lens are imperfect, the reliability decreases towards the edges of the images. These sources of unreliability are incorporated into the ToF reliability model. The depth image from the two sensing systems is ultimately refined by a local and a maximum a posteriori (MAP) based Markov Random Field energy-minimization approach [DT05].

Creating content for 3D television requires conservatively two synchronized cameras. The stereoscopic 3D effect depends on the baseline of these cameras. To acquire close up images, the baseline needs to be so small, that the cameras can not be physically moved together. This problem can be avoided when 3D content is generated by using depth information. Frick et al. [FBK10] introduce a combination of two Time-of-Flight cameras and five color cameras. Reconstructing virtual viewpoints often suffers from the problem that some areas in the new viewpoint are undefined. This is caused by occlusions, as discussed in Section 2.2.3. Color images from additional cameras are used in this approach to fill the occluded areas.

The OmniKinect project [Kai+12] uses multiple Kinect RGB-D cameras in a circular setup, which is shown in Figure 2.14. The object in the center is captured from all sides simultaneously. The Kinect cameras produce an RGB-D data stream from all sides with interactive frame-rates. To fuse the data, the Kinect Fusion algorithm [Iza+11] is modified for multiple cameras. Kinect Fusion, introduced in Section 2.2.5, is a method for fusing multiple depth images captured from one camera to a 3D model. The difference of OmniKinect is that these frames are captured simultaneously. The modified algorithm of OmniKinect replaces the live position tracking to one-time initial pose estimation. Since structured light sensors interfere with each other, small vibrating motors are attached to each Kinect camera. The vibrations blur the projected IR pattern of the other cameras. The cameras are not affected from their own vibrations because the IR projector and sensor are connected rigidly.

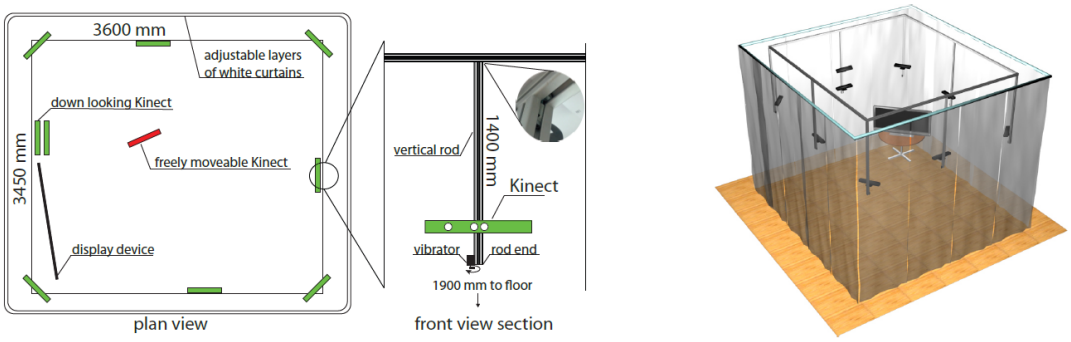

Figure 2.14: Left: Scheme of the OmniKinect system; Right: 3D visualization of the setup [Kai+12]

2.3 Computer Vision on Graphics Hardware

Computer vision in general faces a lot of ill-posed and inverse problems. Increasing sensor resolution, more complex algorithms and higher quality expectations, feed the ever lasting demand for high performance computer vision systems. Programmable shader units on graphic cards initiated the area of general purpose computing on graphics hardware (GPGPU). Recently, graphics programming API's like OpenGL, Direct X and recent Vulcan started to support general purpose computing. Many computer vision algorithms are prone to be parallelized and thus suitable to run on graphics hardware.

2.3.1 Khronos OpenCL

OpenCL is an open high performance computing standard, maintained by the Khronos consortium [Con15]. This API is used for the sensor fusion processing in this project. It was chosen because it is an open standard and the only GPGPU API available on the target platform. A detailed description and usability in the context of this project is found in the Design chapter of this thesis.

2.3.2 Khronos Vulkan

During the process of this thesis, the Khronos consortium unveiled the new open graphics API Vulkan [Con15]. Vulkan aims to unify computing and graphics on modern devices. It is designed to run on any device, on any platform. These features make it a very interesting option for the sensor fusion system developed in this project. However, it has not been released yet.

2.3.3 Renderscript

Google is developing a new mean of utilizing the GPU on a large scale of Android devices. This API aimes to simplify the development process as much as possible. The language is C99-derived and was designed to scale well, across a large variety of Android devices.

2.3.4 Halide

Developing a high performance software for multiple devices often requires device specific fine tuning. Scheduling synchronization, memory and data transfer are traditionally managed by the developer in the computation code. This usually takes a lot more effort and limits the complexity of the algorithms. The recently introduced Halide compiler [RK14] introduces a new paradigm. Instead of developing highly optimized code for specific hardware, the algorithm and the schedule is separated. The processing algorithm is defined as straight forward without regard of the hardware. The scheduling and the necessary parameters are defined in a separate scheduling part. This enables adaption of high parallel code to different architectures and devices. Different parameters can be explored without the need to modify the algorithm. The Halide compiler is currently (2015) suitable for multi-core CPUs and CUDA enabled GPUs.

2.3.5 OpenGL Compute Shaders

With OpenGL ES 3.0, it is also possible to use the OpenGL rendering API for computation [Bai15]. The Compute Shaders work like other modern OpenGL shaders, but are not part of the graphics rendering pipeline. Computer Shaders are usually used together with OpenGL rendering and are invoked by the application before the rendering. The shaders are small programs written in the GLSL shader language and are executed on the GPU in parallel. The reason why this method of GPGPU computing was not chosen for this project, was that zero-copy memory transfers between GPU and application memory space seemed to be easier with OpenCL.

2.4 Applications of RGB-D Images

This section introduces ways to use the result of the sensor fusion process. Many of these applications benefit greatly of the quality gain, introduced during the depth image upscaling. This section also lists fields of computer vision which profit from the use of RGB-D sensors.

2.4.1 Image Segmentation

Image segmentation and object recognition tasks immensely benefit from depth information. For instance, segmenting an object with the same color texture as the background is only possible via depth information. Well established segmentation algorithms, like Graph Cut [BFL06] can be adapted to be used with additional depth data. Graph Cut based computer vision algorithms see the image as a graph where each pixel is a node and is connected to either 4 or 8 neighbor pixels. An energy function is formulated for each edge weight. The image segmentation line follows a path with minimal energy, which can be seen as minimum flow problem.

The approach from Yu et al. [YZ12] performs image segmentation of RGB-D images, captured by the Kinect sensor. It takes both color and depth information into account. As seen in Figure 2.15, it enables the segmentation of objects that are the same color as the background.

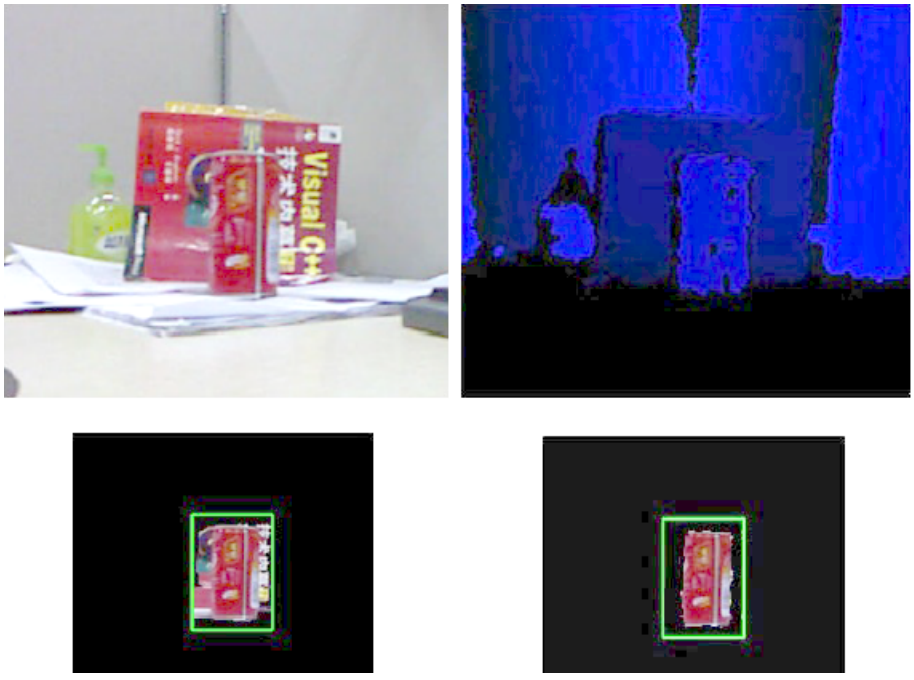

Figure 2.15: Graph Cut by Yu et al [YZ12]: Top: Input color and depth image; Bottom: The result without and with depth influence

To incorporate depth information, they modify the iterative Grab Cut approach of Rother et al. [CR04]. This Graph Cut based approach uses a Gaussian mixture model that

incorporates two distributions to model foreground and background pixels. Yu models the depth information as a single Gaussian model and multiplies it to the color term of the foreground possibility model. For the background probability model, the depth is modeled as a uniform distribution. The reason for the uniform distribution is that the background is assumed to be complex with a lot of variation in depth values. Yu also uses the statistical advantage of available depth data to speed up the traditional Graph Cut algorithm.

2.4.2 Computational Photography

Computational Photography involves techniques of capturing, post processing and modifying digital images. It involves popular applications like the computation of high dynamic range images and panorama stitching. The use of depth cameras in computational photography is a relatively new area and not wide spread. Depthkit [JGM15] offers an opportunity for photographers to combine a Microsoft Kinect depth camera with a DSLR camera. These homemade rigs are mostly used to generate 3D images as an art form.

Depth cameras can be used for rapid auto-focus or enable selective focusing. This can benefit smaller cameras on mobile devices, which are bound to contrast auto-focus and a depth of field. Refocusing means to take an image with a wide depth of field and blur arbitrary areas like the background. This can add simulated bouquet to an image captured by a small camera. The Bouquet effect is traditionally only possible to capture with cameras with a big sensor and a large aperture. Bouquet is generally accepted as aesthetic since it can provide a better depth perception on images.

Refocusing a scene is usually associated with light field cameras, however RGB-D frames can be refocused too. With a narrow depth of field, the sharpness of a region depends on the distance to the camera. With high quality depth information, it is possible to blur certain regions which have similar depth values. The depth based Graph cut segmentation, introduced in Section 2.4.1, can assist to split an image into fore and background. However if an RGB-D image has perfect depth information, the image can be refocused solely by the depth information.

2.4.3 Augmented Reality

One of the key applications of a fused RGB and ToF data is augmented reality. This section introduces various means of augmented reality (AR), and shows how this field benefits of the unified RGB-D sensor. Then Qualcomms Vuforia SDK [Qua15] is introduced and its relation to this project is discussed.

Augmented reality in general, is about fusing the environment with computer generated content. Fused color and depth information aids augmented reality in terms of reduced computational power and quality of occlusions. The goal in AR is to enhance or modify reality by embedding virtual information. This creates a number of new ways to interact with computer systems. For instance, additional information can be pinned to specific objects in a scene. It could increase efficiency in maintenance and in general work-flows, where a lot of information must be looked up.

Not only is Augmented reality about embedding information, it has also various entertainment purposes. Being able to embed virtual objects naturally into a scene has a lot of

potential. Menki et al. [ML] provide an overview of the various applications. Augmented reality is a very active research area and will benefit from emerging mobile depth cameras. Applications for mobile AR are wide spread and range from commercial catalogs to virtual furniture shopping.

The following steps are necessary to render virtual objects into a 2D image:

- **Geometry acquisition**

  Assuming all sensors are calibrated, the position and orientation of the camera in the environment needs to be calculated. This is a challenge, since the quality of the augmented image largely depends on correct localization. Visual markers are the most simple way to obtain a scene coordinate system. Markers like QR codes can provide the AR system with additional information.

  Simultaneous Localization and Mapping (SLAM) [Chr15] means to build a local model of the environment and use it for localization at the same time. It works by capturing multiple frames from different locations. Systems with a single camera use triangulation of frame to frame correspondences to calculate the distance to salient feature points.

  A depth camera can speed up the geometry acquisition process and works better in general.

- **Localization**

  The device is localized by comparing the seen 3D points with a model of the scene. A popular algorithm to retrieve the homography between these two pointclouds is RANSAC [FB81].

- **Rendering the Virtual Scene**

  Virtual objects are rendered using the previously acquired scene coordinate system. If the virtual objects are supposed to blend into the scene, the camera characteristics need to be simulated during or after the rendering. The immersion of the virtual objects may also improve by realistic shading. Information about light sources may be estimated by using information from the color camera.

  If there is information embedded, like info-boxes, the scene coordinate system might only be used for locating the information.

- **Fusing the virtual scene with the image from the camera**

  Abstract items can be blended into the 2D image directly. Since cameras are imperfect, virtual objects might be visually adapted to match the camera quality. This involves re-sampling to simulate the Bayer pattern of the sensor or adding noise and motion blur [FBS06].

  Occlusions are an important step in integrating an object into a scene. An occlusion occurs if a real object is located in front of a virtual object. To be able to render the virtual object behind the real object, it is necessary to know the distance from the camera to each pixel. RGB-D camera systems provide this information which is very convenient in handling occlusions. The resolution and precision of the depth to RGB mapping is crucial for high quality occlusions.

Qualcomm Vuforia

Qualcomm Vuforia [Qua15] is a software development kit for mobile platforms. It provides the augmented reality functionality in the form of a software library. The default platform are smartphones and tablets with color cameras. The most recent release (04/2015) also supports virtual reality platforms. There is no support for depth cameras yet.

Vuforia tracks objects or markers and estimates the necessary parameters to render virtual objects into a video stream. With the help of depth cameras, the SDK could lose the need for specific objects for registration of the coordinate system. It is connected with the graphics engine Unity [Tec15], which allows relatively simple high-end rendering.

As mentioned in Section 2.4.3, one of the problems of augmented reality is occlusions. With just a color camera, it is a hard task to tell whether a virtual object is behind a real one. The current state of Vuforia is object based occlusions (05/2015). For registered real objects, a 3D model is created. This 3D model is rendered into the depth buffer. All virtual objects with bigger depth values are occluded and thus not rendered.

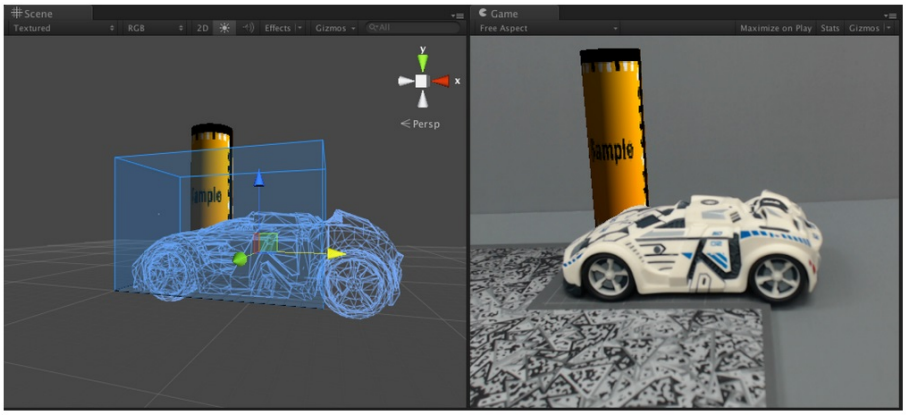

Figure 2.16: Left: 3D model of a real object (toy car); Right: Virtual object, placed behind toy car. [Qua15]

A color and depth sensor fusion result, containing depth information for each color pixel, makes it easy to occlude virtual objects. There is no need for predefined objects, since the occlusion check can be done just by comparing virtual and sensed depth.

Stereoscopic 3D Content

To create a stereoscopic 3D effect, two images with a shifted point of view are displayed to each eye of the observer. It is possible to generate two virtual views from a RGB-D image. As mentioned in Section 2.2.3 however, some areas will not contain information and need to be interpolated. In computer generated graphics, a perfect depth map is usually available. Generating and displaying a stereoscopic view in computer graphics systems thus works very well. Since the perfect depth map is used, the scene is only rendered once, and a second virtual viewport is created. Hence this does not cause a huge performance loss. This creates another application for fused RGB-D data. Using a ToF camera for generating 3D television content and movies is explored and tested by Frick et al. [FBK10] and Huhle et al. [HFS07].

2.5 Related Commercial Projects

There is a large spectrum of academic publications on depth and color image data fusion. The most relevant work was mentioned in Section 2.2.2. However there are no academic projects to our knowledge (2015), researching color and depth sensor fusion on a mobile platform with limited resources.

In 2014, Intel announced their RealSense Technology [Inc15], which aims to make 3D sensors a part of peoples daily life. They released the Dell Venue 8 tablet, which incorporates the RealSense system. The technology and algorithms of RealSense are not publicized. It is however known that the sensor of the Dell tablet consists of a system of two color cameras, where depth is calculated from stereo matching. The applications of Intel RealSense are computational photography and gesture control. It is possible to take pictures with a RealSense device and perform selective focusing,thus applying artificial bokeh or depth sensitive filters to an image. This post-processing technique was mentioned in Section 2.4.2 before.

Google Tango [Goo14] is a pioneer project, combining a motion tracking camera (gray-scale, fish-eye), a color color and a depth camera on a mobile platform. The upcoming prototype (07/2015) features the same Infineon/PMD ToF depth sensor as used in this thesis.

A completely different way to retrieve RGB-D data is to use a hybrid sensor. Samsung developed an image sensor in 2012 [Kim+12], which combines color and depth pixels. Since the depth pixels need to compare incoming light with the modulation frequency, as mentioned in Section 2.1, they need more space than simple CMOS pixels. As seen in Figure 2.17, they place square depth pixels between the rows of the color pixels. The problem with this sensor is that surface area of the depth pixels is too small to sense depth properly. They are about 1-2 % of the size of recent (2015) Time-of-Flight depth cameras. If there are color and depth image pixels on a sensor, it is not possible to filter unwanted light from other light sources with an optical near infrared (NIR) filter on the lens. In their approach, pixel-wise NIR filters were used which were not satisfactory according to the authors.

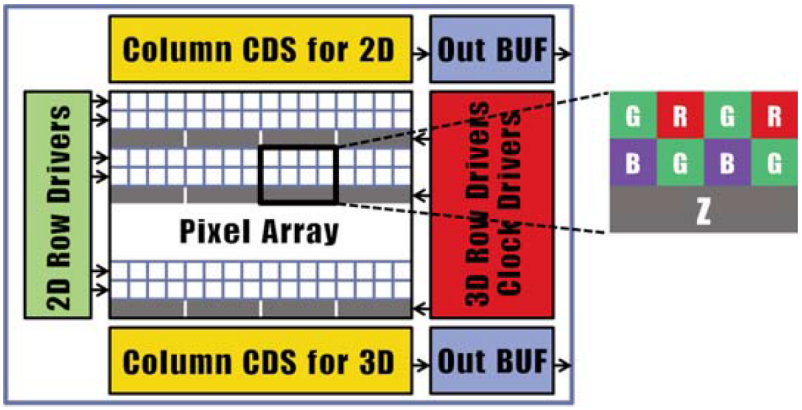

Figure 2.17: The sensor architecture of the RGB-Z sensor [Kim+12]

Chapter 3

Design

This chapter introduces the selected hardware platform, explains the fusion algorithms and discusses the software design.

3.1 Requirements of the Sensor Fusion System

The goal of this project is to create a system capable of fusing data produced by a color and a ToF sensor. The design goals for the sensor fusion demonstrator are specified as:

- **Precise Depth to Color Mapping**

  The cameras in the system are precisely calibrated, enabling a transformation of depth values to the coordinate system of the color image. Several calibration methods for depth sensing systems are discussed in Section 2.2.4.

- **Depth Quality Enhancement**

  Using the higher resolution image of a color sensor, the depth image can be enhanced in terms of noise-reduction and resolution. Various approaches are introduced in Section 2.2.2.

- **Performance**

  The goal is to prove the feasibility of depth and color sensor fusion on mobile devices. The algorithms are therefore designed to be executed on the GPU in order to reach interactive frame-rates.

3.2 The Sensor Fusion Platform

This section introduces the hardware platform, which is used for the color and depth sensor fusion demonstrator. It lists the hardware specifications and how the GPU is interfaced for computation.

The system consists of two cameras (ToF and color) and a Snapdragon 810 development board. The cameras face the same direction, are rigidly mounted and calibrated. An Android application uses native libraries to receive data from the sensors. The GPU of the Snapdragon 810 platform is used for the sensor fusion computation.

3.2.1 Snapdragon 810

The Qualcomm Snapdragon 810 is a System on a Chip (SoC). It unifies 8 CPU cores, a GPU, a digital signal processor (DSP) and communication modules onto one chip. It is the fastest smartphone SoC available (2015), and was chosen to prove the feasibility of ToF and color sensor fusion on the upcoming hardware generations.

CPU	Quad-core ARM Cortex A57 & quad-core A53
GPU	Adreno 430 GPU
DSP	HexagonV56 DSP

Table 3.1: The processing units on the Qualcomm Snapdragon 810 platform

For the sensor fusion implementation, the development kit Dragonboard APQ8094 is used. The board features the Snapdragon 810 with 4 GB LP-DDR4 RAM. There are various I/O ports, including 2x USB 3.0 and 2x MIPI. An on-board display is only optionally available, so an external 1080p display is connected via HDMI.

3.2.2 High Performance Computing with the Adreno 430 GPU

Since performance is critical in this project, most of the processing algorithms are implemented on the graphics processing unit (GPU). The embedded GPU on the Qualcomm Snapdragon 810 platform is intended for rendering graphics, but can be utilized as a computing device. The GPU driver therefore supports the OpenCL standard 1.2 which is introduced in Section 2.3.1.

OpenCL is an open API for high performance computing on parallel architectures like GPUs. It provides a framework to execute code on these architectures. Qualcomm provides good developer support for their implementation of OpenCL, including a 60 page programming guide, debugging and profiling tools.

The Adreno 430 GPU shares the same memory as the application processors, which accelerates memory transfers. In a traditional PC system with a separate graphic card, memory transfers are a bottleneck. The nature of the sensor fusion system requires a high volume of memory data transfer. With shared memory, it is possible to retrieve a pointer from OpenCL and directly write data to the video memory.

For GPGPU computing, the memory characteristics of a device are important. Since the architecture allows massive parallel execution, the internal GPU memory transfers affect the performance dramatically. Table 3.2 shows the memory specifications of the OpenCL computing model.

Global Memory Size [MByte]	1811
Local Memory Size [KByte]	32
Maximum Work-Group Size [1]	1024
Global Memory Cacheline Size [Byte]	64
Global Memory Cache Size [KByte]	128

Table 3.2: The OpenCL memory model characteristics of the Adreno 430 GPU

Developing with OpenCL

A program executed on the computing device is called a kernel. The code is compiled at run-time by the CPU and is executed on the GPU by calling an OpenCL function. Each kernel is executed multiple times with different parameters. A decrease in processing time can only be achieved, if the computation problem is able to be split into several thousand kernels that are executed at the same time. Otherwise, the kernel execution overhead is too high and a CPU implementation may be faster [RT12].

The OpenCL C language is based on C99 with extensions and restrictions to fit the device model. It does not permit certain C features such as function pointers or recursion. To use the instruction parallelism, there are built-in high precision vector data types.

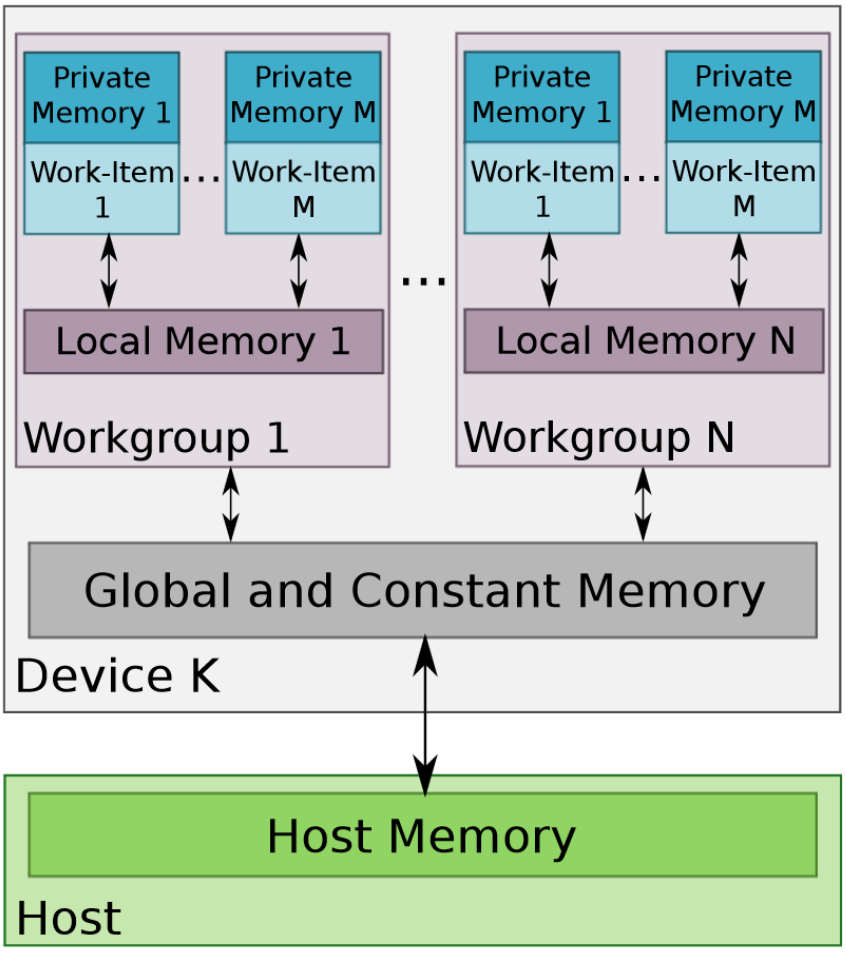

Figure 3.1: The memory model of OpenCL [K.11]

When developing with OpenCL, it is important to regard the memory model shown in Figure 3.1. Unlike CPU based systems, there are multiple kinds of memory, which have different purposes and access patterns. The OpenCL memory model is the same for all devices, supporting OpenCL. It is an abstract model and does not represent the actual hardware. It consists of:

- **Host Memory**

  This is the main memory (RAM) of the device. The GPU does not have access, therefore the RAM and GPU memory are not shared. This issue is resolved by using RAM-GPU memory transfers when required. OpenCL helps to either map the global GPU memory to host memory or to copy complete buffers. RAM-GPU data transfer is commonly a bottleneck however SoCs like the Snapdragon 810 share the memory with the GPU.

- **Global and Constant Memory**

  This is the main memory of the computing device. It is simultaneously accessible for all work-items. All input data for the computation needs to be transferred to this memory before computation. An exception to this are kernel arguments, which are parameters for execution and usually do not contain input data themselves. The kernels access this memory by pointers, which are provided by the kernel arguments. The access is slow compared to the other types of memory. It is coalesced, which means that single requests from work-items are bundled together and handled simultaneously.

  On the Snapdragon 810 platform, this memory is a part of the system memory. Hence it is possible for the CPU to directly access GPU memory. This is an advantage since RAM and GPU memory transfers are often a bottleneck on separated CPU and GPU platforms. For the sensor fusion application, it means that the device drivers of the cameras can directly write their data to this memory. This avoids any memory copies which cause a noticeable delay.

- **Local Memory**

  The local memory is shared with all items in one work-group. The access latency is much lower than global, however it is higher than private memory latency. When two threads try to access an address located at the same memory bank simultaneously, there is a bank conflict. Bank conflicts can slow down the execution immensely and are best avoided during the algorithm design.

- **Private Memory**

  This is the fastest memory in the OpenCL memory model. Each work item has its own memory and it cannot be accessed from outside. This memory is typically not managed by the developer. Usually the OpenCL compiler takes care to utilize this memory. It is possible to allocate static arrays in private memory, but when too much is allocated, global memory is used instead automatically. This can cause a tremendous slow-down.

When designing and implementing algorithms for GPUs, this memory model needs to be regarded. The basic principle is to design an algorithm, which can be executed in parallel and requires as less memory access operations as possible. The parallel threads should be as independent of each other as possible. Synchronization methods exist, but they cause performance drawbacks. It is often better to split the computation into several different GPU programs, and execute them subsequently. The different kernels of this project are discussed in Section 4.5.

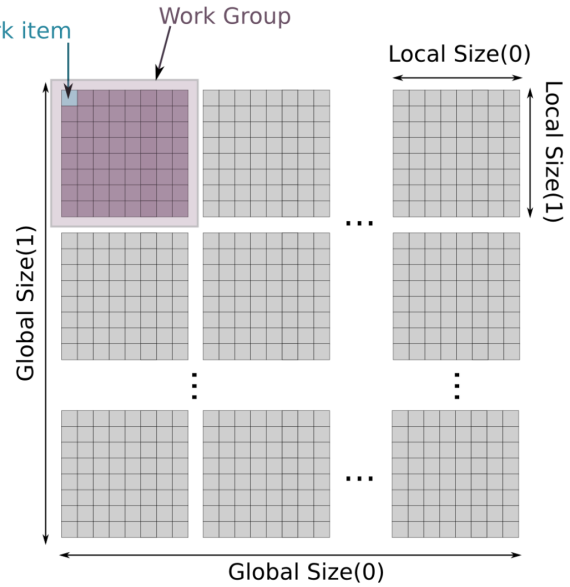

Figure 3.2: The OpenCL work-groups [K.11]

A fundamental concept is the grouping of work-items as shown in Figure 3.2. Each work-item performs a certain task on the input data. The work-items are grouped into work-groups and executed simultaneously. Each work-group has its own local memory. It is possible to arrange the work-groups in up to three dimensions, depending on the device. The dimensionality of the work-items are dependent on the dimensionality of the input data. Since all GPU computing in this thesis uses images as input data, all work-groups are arranged in a two-dimensional grid. The global position and the local position inside a work-group can be requested by every work-item. This is how each thread knows its location and thus the spatial parameter for the computation. The maximum size of the work-groups (local work-group size) is determined by the GPU hardware and depends on the individual device. The local work-group size can have the same number of dimensions as the global size, but it is not required to use them. Due to the memory cache, it is not trivial to find the optimal local work-group size. Qualcomm recommends a brute force performance evaluation in their OpenCL developing guide.

Before a kernel is executed on the computing device, kernel arguments are specified. These arguments are a way for the host program to propagate information to all GPU work-items. The arguments usually contain computation parameters and pointers to data in the global memory. They can be set by the OpenCL API, and are treated in the kernel code as function parameters.

Conditional statements on GPUs are sometimes problematic. GPU programming APIs like OpenCL or CUDA, have a C-like syntax with some extensions and limitations. The code is however not executed like on a CPU. Conditional statements can cause massive computation overhead, depending on the implementation of the vendor.

Vectorization of data is another important performance aspect. There are vectorized datatypes provided by OpenCL, containing up to four elements of integers or floats. Using vectorized data types does not only increase computation performance, it also helps to accelerate memory transfers. Copying a data vector is nearly always as fast as copying a single element.

To show parallel GPGPU development with OpenCL in this project, it is a good approach to use an image as an example. A GPU program (kernel) is executed for every pixel. This pattern of GPGPU computing is used in the sensor fusion implementation of this project. The work-items are arranged into a 2D grid. The global work-group size is the pixel dimensions of the image. The image is zero-padded, so that the image dimensions can be evenly divided by 32. The local work-group size for each dimension is a number, evenly dividing the global work-group dimensions. The sum of work-items per work-group is limited by the GPU device and can be requested by the OpenCL API. Each kernel acquires the position on the image by requesting the global work-item ID. A pointer to the image in global GPU memory is provided as kernel argument. The work-item can now address the image and copy the value of its pixel to private memory, then perform the necessary processing operations. If work-items need to access the neighboring pixels (for example to do Gaussian filtering), it is faster to copy these values to local memory first. Each work-item copies its pixel into shared memory. Each kernel can now access the neighboring pixels by local memory access. After the work-group is done with the processing, these values are copied back to global memory. This technique is used in this project to maximize computation speed.

An important performance aspect is the usage of local memory. Local memory access is significantly faster than global memory access. The threads (work-items) are executed in packages (work-groups). Each work-group has a certain amount of local memory available. The size of the available local memory depends on the device and can be requested with the OpenCL API. The input data for the work-items is usually placed in global memory. If the kernel accesses and manipulates this memory, it is a good idea to transfer it to local memory. A good access pattern divides the memory transfer tasks evenly among the threads.

OpenCL on Android

Applications for Android are usually developed in Java. The OpenCL libraries can only be used in C or C++, so a native library for an Android application needs to be created. The native implementation uses the Java Native Interface(JNI) to interface with the Java application. The *libOpenCL.so* library, containing the OpenCL implementation is usually provided by the device vendor. It can often be retrieved directly from the device with the Android Debug Bridge (ADB) and then be integrated into the application. The OpenCL programs are usually text-files and are attached to the Android application as asset file.

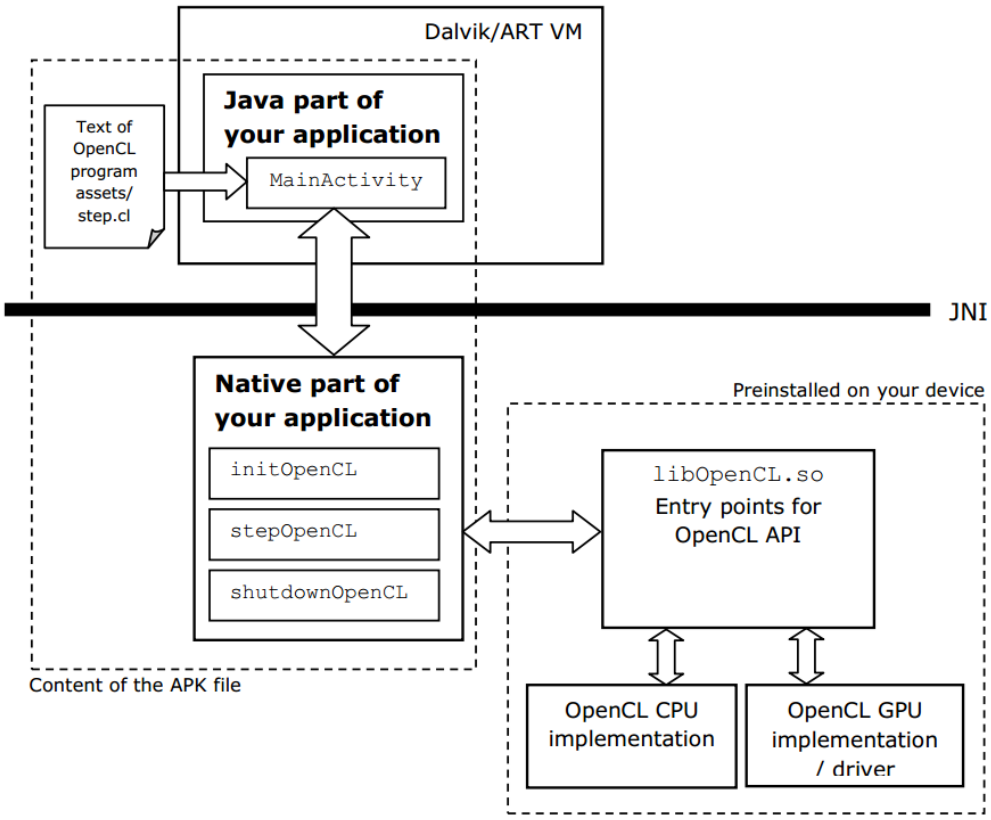

Figure 3.3: OpenCL in Android [Int15]

3.2.3 OpenGL ES 3.1

The Adreno 430 GPU supports the OpenGL ES 3.1 standard. OpenGL is an open graphics API and enables rendering on the GPU. OpenGL ES 3.1 is very similar to standard OpenGL, however not all datatypes or geometry shaders are available. Vertex and fragment shaders are supported. Vertex shaders allow dynamic geometry manipulation and generation on the GPU. Fragment shaders enable the manipulation of every rendered pixel directly. This is important for an application that wants to use the sensor fusion product.

3.2.4 OpenCL and OpenGL Interoperability

It is possible to exchange pointers to memory buffers between the APIs OpenCL and OpenGL. In this project, the upscaled depth and color images are located in GPU memory during complete processing with OpenCL. When an application wants fast GPU access of the sensor fusion results, OpenCL-OpenGL interoperability is the fastest way.

To use this feature of OpenCL, the extension *cl_khr_gl_sharing* is used. At first the OpenGL context is created and then shared with OpenCL. It is possible to share OpenGL textures or pixel buffer objects (PBOs). The memory access needs to be managed by the application, so that there are no access conflicts. OpenCL for computing and OpenGL for rendering can be done on the GPU simultaneously. GPU computing execution might

be halted in favor for rendering tasks. Depending on the vendor, it may be important to keep the kernel execution times low to increase the rendering rate.

3.2.5 The Selection of the Color Camera

The camera system consists of a depth ToF camera and a color camera. While the depth camera was provided by Infineon (Infineon Time-of-Flight eval. kit), the choice of the color camera was arbitrary. The project's hardware platform, the desired application and the specifications of the depth camera, were considered to compile a list of color camera requirements:

- **Wide Viewing Angle**

  The measured horizontal viewing angle of the ToF camera is 88 degrees. The ideal color camera would have the same viewing angle, so no depth information is lost.

- **USB Interface**

  A USB interface is ideal since the Dragonboard development kit does not offer a compatible MIPI color camera (01/2015).

- **UVC-Compliant**

  In order to use the V4L (Video for Linux) drivers, which are provided by Android, the camera needs to be UVC compliant.

- **Image Quality**

  The image of the color camera is a guidance for the depth image upscaling and thus has an impact on depth image quality. During upscaling, color saturation differences and RGB color space distances are regarded to detect discontinuities. Ideally, the image is not distorted, so the distortion correction would not occupy resources on the sensor fusion platform.

- **Configurability**

  V4L defines a rich set of camera setup commands. Most cameras however do not support all commands. Important for this project is the ability to disable the auto-focus. While focusing on lower end camera systems, the image size changes and makes calibration a difficult task. Direct image adjustments like saturation, brightness, sharpening or contrast is interesting for experimentation with the depth upscaling algorithm, but are not required.

These requirements led to the choice of the Logitec B910 Webcam. It meets all requirements, however the horizontal viewing angle of 78 degrees could be wider. An assessment of other available cameras possessed an even narrower angle of 60 degrees.
Figure 3.4 shows how much of the depth camera image can be mapped to the color camera image. An unfortunate characteristic of the camera is the limited viewing angle on resolutions with 4:3 aspect ratio. Setting the camera in a resolution mode like VGA, the raw sensor data is cropped on the sides of the image, limiting the horizontal viewing angle. Since both aspect ratio modes (4:3 and 16:9) cover about the same amount of depth pixels,

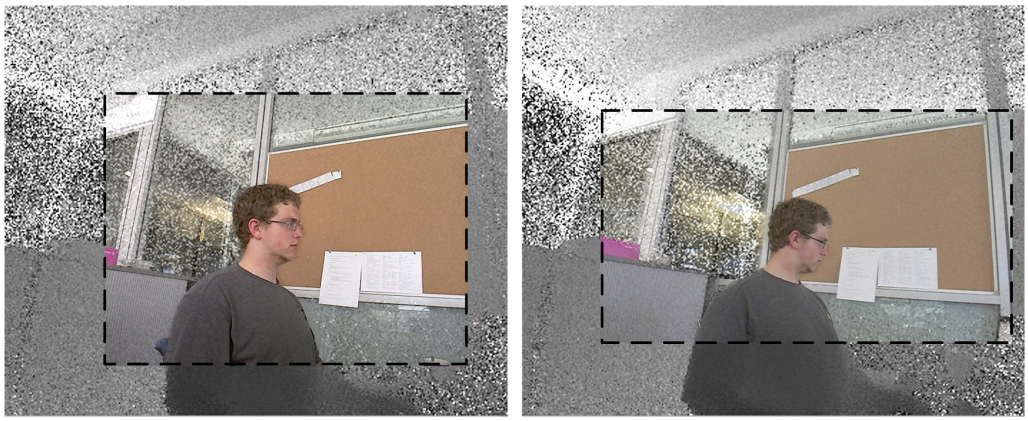

Figure 3.4: Areas of the depth image covered by the Logitec color camera. Left: VGA(640x480); Right: QHD (640x360)

the VGA(640x480) resolution with 4:3 aspect ration was chosen. VGA has more pixels than the similar, but wider QHD mode. A higher resolution mode like HD(1280x720) would exhaust the workload for the mobile platform. It would also require on-camera compression, since the USB bandwidth is too low to transfer uncompressed HD video data at feasible frame-rates.

3.3 ToF and Color Sensor Fusion Procedure

Fusing ToF and color image sensor data in this project means to map depth to color data. The final result is a unified color and depth sensor, delivering a high-resolution color image with a depth data association for each pixel. The motivation for this category of sensor fusion is applications in augmented reality, computational photography and refinement of the depth data itself. These and several other ways of depth and color fusion are listed in Section 2.2.

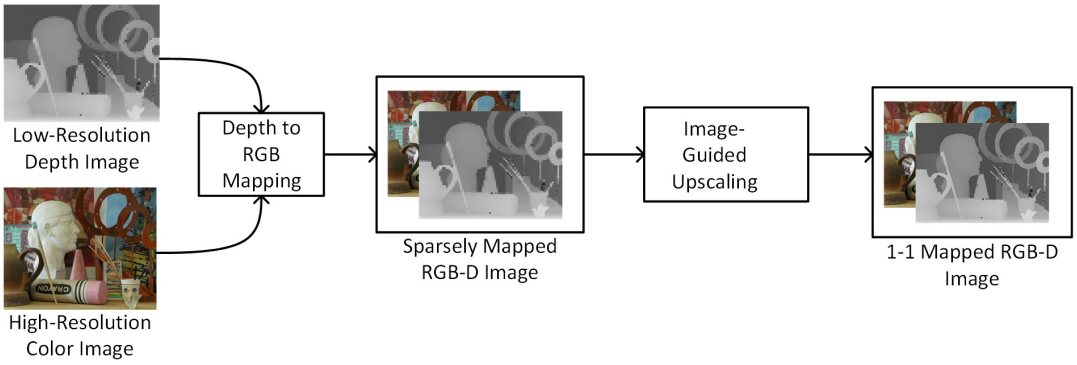

Figure 3.5: The RGB-D sensor fusion processing pipeline, demonstrated with the 2005 middlebury dataset [HS07]

Figure 3.5 shows the outline of the sensor fusion procedure. At first, an RGB and depth image are gathered from the sensors. The depth image has a much smaller resolution than the RGB image. The depth values are mapped to the image space of the RGB image. The upscaling algorithm interpolates this sparsely mapped depth image. Depth discontinuities are preserved by using the discontinuities of the color image as guidance. The result is a high-resolution depth image with a 1-1 mapping to the RGB image. This is referred to as RGB-D frame in this thesis.

3.3.1 Calibration

Like in many computer vision systems, camera calibration is an important preparation task. Camera calibration is the process of retrieving knowledge of the camera parameters. Since two cameras are used, an intrinsic and extrinsic calibration is necessary.

Intrinsic camera calibration in this context means to resect the camera and get the internal camera parameters. These parameters describe the projection between 3D and 2D camera image space. It must not be confused with photometric calibration, which is not necessary in this context. A pinhole camera model is sufficient to describe the cameras projective behavior.

The extrinsic parameters describe the spatial relation between the cameras. They are necessary to map pixels from one camera coordinate system to another. Since the extrinsic camera parameters of a rigid camera system never change after the assembly, an initial calibration is required.

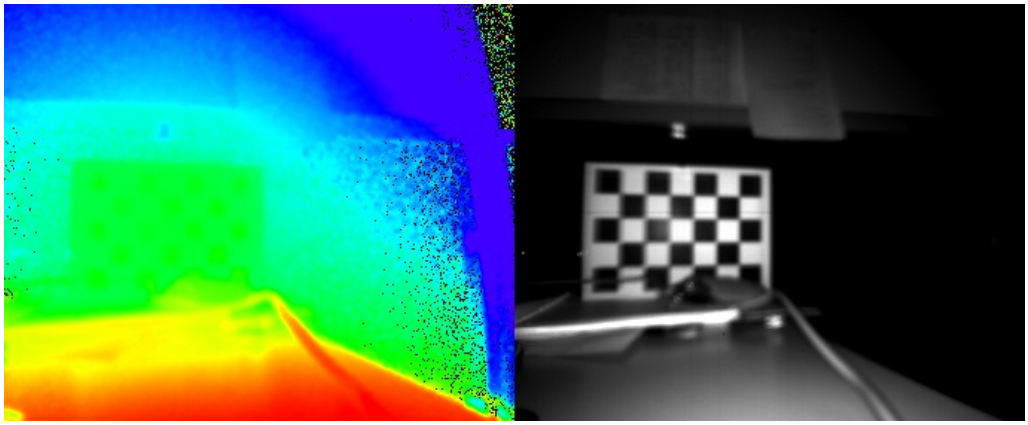

Figure 3.6: Left: Color coded depth image, Right: Amplitude image

Calibration methods for depth cameras are reviewed in Section 2.2.4. Standard 2D calibration methods are much more common and well-established toolboxes exist. A method is available to reduce the calibration procedure of a ToF camera to a simple 2D stereo calibration:

Time-of-Flight sensors are capable of delivering an amplitude image, as seen in Figure 3.6. The amplitude image contains the measured amplitudes of modulated incoming light. The amplitude is proportional to the amount of light reflected by the scene. The image can be interpreted as a gray-scale 2D image, thus simplifying the calibration process. Planar visual calibration patterns can be used and the extrinsic calibration procedure is the same

as with a stereo camera pair. This is well researched and an implementation in the form of a Matlab Toolbox from Bouguet [Bou15] is widely used.

To calibrate the camera system, several images of a checkerboard target are necessary. The board is differently positioned and rotated in each image. It is possible to detect the checkerboard corners with sub pixel accuracy. The camera parameters are estimated by minimizing the re-projection error of these feature points. The calibration process retrieves the following camera parameters:

- **Distortion**

  Lenses are usually imperfect and cause barrel distortions. Calibrating the lens distortion is about finding the parameters of a function, which corrects distortion by remapping pixels.

  Distortion compensation is split into radial and tangential distortion. The radial distortion is approximately radially symmetric. Wide-angle lenses are usually designed to have this kind of distortion. Radial distortion is corrected by scaling the normalized x and y coordinates by a factor. The factor depends on the distance from the center r and is modelled as a polynomial:

$$x_{corrected} = x \cdot (1 + k_1 r^2 + k_2 r^4 + k_i r^{i \cdot 2} + ...)$$
$$y_{corrected} = y \cdot (1 + k_1 r^2 + k_2 r^4 + k_i r^{i \cdot 2} + ...)$$

  The radial distortion factors are denoted as k_i. It is sufficient to only take the first three factors k_{1-3} into account for a good approximation.

  Tangential distortion occurs when lenses are not perfectly parallel to the sensor. The pixel position are corrected with the following formula:

$$x_{corrected} = x + [2p_1 xy + p_2(r^2 + 2x^2)]$$
$$y_{corrected} = y + [p_1(r^2 + 2y^2) + 2p_2 xy]$$

- **Intrinsic Matrix**

  The intrinsic matrix A contains parameters, which approximate the camera's projection behavior in a pinhole camera model. With this model, it is possible to project a 3D point $\begin{bmatrix} x_w & y_w & z_w & 1 \end{bmatrix}^T$ in homogenous world coordinates to homogenous 2D camera image coordinates $\begin{bmatrix} x_c & y_c & 1 \end{bmatrix}^T$.

  The projection formula for homogenous coordinates is:

$$\begin{bmatrix} x_c \\ y_c \\ z_c \end{bmatrix} = A \begin{bmatrix} x_w \\ y_w \\ z_w \\ 1 \end{bmatrix}$$

The intrinsic matrix A is composed the following way:

$$A = \begin{bmatrix} \alpha_x & 0 & p_x & 0 \\ 0 & \alpha_y & p_y & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The parameters α_x and α_y are the focal lengths. α_x and α_y are different, when the sensor pixels are not quadratic or the lens is not perfectly radial symmetric. p_x and p_y are the offsets of the principal point from the image center.

- **Extrinsic Parameters**

  The extrinsic parameters describe the spatial relation between the cameras. It is common to express them as transformation matrix T and rotation matrix R. They enable to map a 3D point from one 3D camera space to another.

3.3.2 Depth to Color Mapping

Mapping depth vales to color images is commonly described in literature. The method of this thesis is inspired by approach of Ferstl [Fer+13]. Since it is difficult to mount cameras in a way, that the optical axes are exactly parallel, this project also takes camera rotation into account. Otherwise, this thesis uses Ferstl's formulas and terminology.

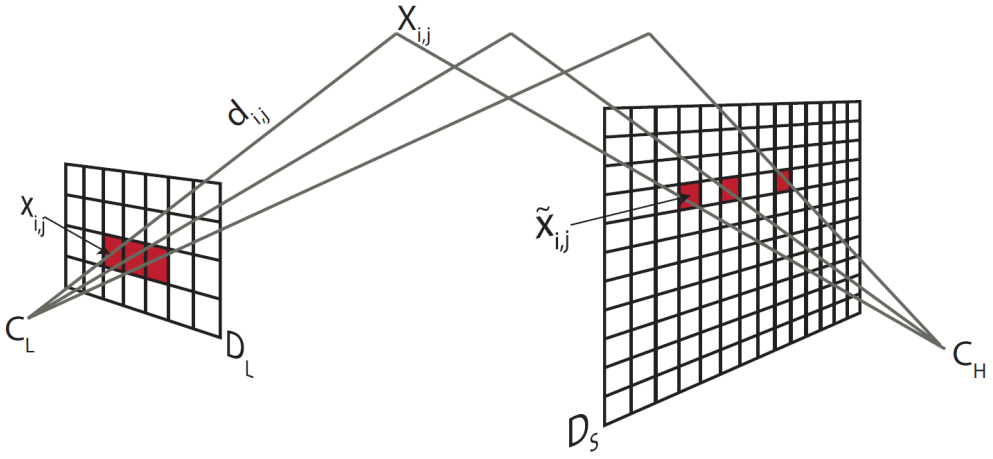

Figure 3.7: Depth camera C_L to color camera C_H data projection from Ferstl [Fer+13]

The procedure of mapping a depth image to the image space of a color camera, as visualized in Figure 3.7, is the following: The depth values with pixel position $x_{i,j}$ and depth $d_{i,j}$ are converted into 3D points $X_{i,j}$.

$$X_{i,j} = T + Rd_{i,j}\frac{\tilde{P}_D x_{i,j}}{\|\tilde{P}_D x_{i,j}\|}$$

$\tilde{P}_D$ is the pseudo-inverse of the projection matrix of the depth camera. The image center of the depth camera (pinhole model) is the origin of the 3D space. Hence, the extrinsic parameters are applied as rotation R and translation T. The rotation matrix might be ignored if the cameras are aligned perfectly parallel. The depth values $X_{i,j}$ are now in the 3D color camera coordinate system.

$$\tilde{x}_{i,j} = P_C X_{i,j}$$

The projection matrix P_C is the intrinsic matrix of the color camera. The result of the projection is a list of depth pixel positions $\tilde{x}_{i,j}$ in the image space of the color camera D_H. By rounding the new depth image coordinates $\tilde{x}_{i,j}$, the depth-values are associated with the nearest neighboring pixel. The result is a sparsely mapped RGB-D frame. This can be interpreted as RGB image, where some pixels have depth information associated.

3.3.3 Image-Guided Depth Upsampling

As mentioned in Section 2.2.2, there are a lot of potential quality improvements by using color information to upsample depth images. Upsampling in this context, refers to scaling the depth image from the ToF sensor to the same size of the image of the RGB camera.

A property that can be exploited when a depth image is upsampled and a color image is used as a guide is that the edges in a color image are often edges in a depth image. Differently formulated, it means that depth discontinuities correlate with color discontinuities

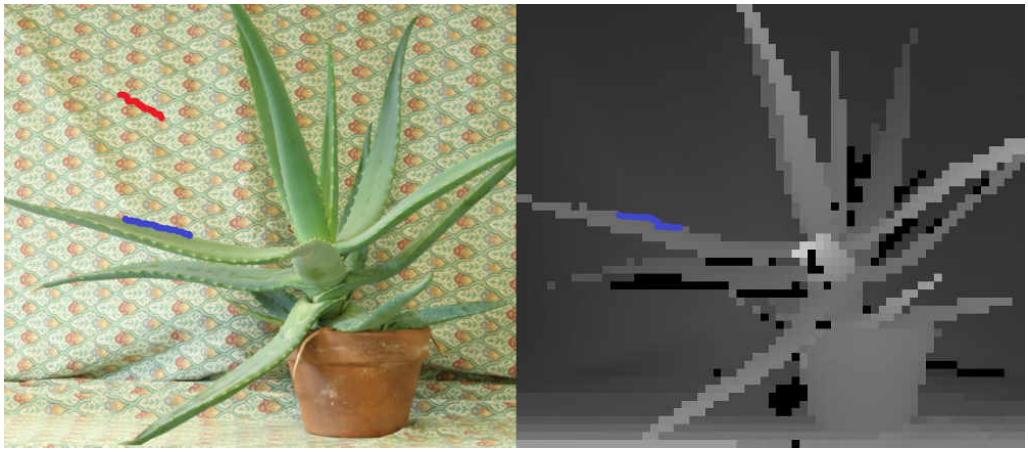

Figure 3.8: The blue line marks the edge of the aloe leaf. It is also a border in the depth image. The red edge on the wallpaper does not have a corresponding depth discontinuity.

Figure 3.8 shows an example: The blue line can be extracted by an edge detection algorithm. It is possible for a depth upscaling algorithm to refine the resolution of the depth image in the area of the blue line. The red line marks one of the many edges without a corresponding depth discontinuity. It is dependent on the algorithm whether there are problems with edges such as the red one in Figure 3.8.

The input for the upsampling method in this project are depth values, mapped to the image space of the RGB image. In Section 3.3.2 it is described how this sparse RGB-D

frame is obtained. There is a large variety of image guided depth upsampling algorithms. In Section 2.2.2 a selection of state-of-the-art algorithms is introduced. The criteria for an ideal upsampling algorithm in this project includes depth noise reduction, resolution gain, speed, and parallelism for an efficient GPU implementation.

3.4 Design of a Depth Upscaling Algorithm

The most sophisticated algorithm in the sensor fusion processing pipeline is the depth data upscaling. In this step, the depth information is vastly improved in terms of noise reduction and resolution. Edges of the color image are used for depth value interpolation. In Section 2.2.2, literature on image-guided upscaling is introduced and discussed. This section provides insight on the alleged best suitable approach (2015) and shows how it was re-implemented for evaluation. A novel algorithm is then introduced, which eliminates the disadvantages of many methods, performs well on noisy Time-of-Flight data and is designed to reach high performance on a GPU.

3.4.1 Evaluation of the State-of-the-Art Upsampling Algorithm

A large number of existing algorithms are not suitable for the purpose of this project because they are too computationally complex. Even the faster local methods such as joint bilateral filtering, are often applied iteratively, which also yields to slow computation. There are graph-based methods however, like the one from Dai et al. [Dai+15], which have an $O(n)$ complexity under certain circumstances. It is the most recent graph-based method and claims to be superior to their predecessors. A description of the algorithm is located in Section 2.2.2 along with other State-of-the Art image-guided depth upscaling algorithms. This algorithm had to be re-implemented since the authors do not provide an implementation nor were they contactable. Details on the re-implementation can be found in Section 4.4 in the Implementation chapter.

The results were retrieved by downsampling ground truth data to the low resolution of 80x60 pixel. This was motivated, due to the sensor resolution of Time-of-Flight cameras in general being low. As ToF data tends to be noisy decreasing the resolution down to 80x60 reduces noise. Dai's method relays on perfect depth data and performs no noise reduction. The re-implementation of Dai's method, as described in Section 2.2.2 was applied to the test data. It clearly shows a massive improvement of depth resolution however, there are also numerous erroneous regions.

Figure 3.9 marks some of these regions. These errors are caused by the nature of the algorithm: The minimum spanning tree (MST) minimizes the edge weights of the image graph. In an image graph, every pixel is a node and edge weights are the color differences of the pixels. Since the MST connects the complete image, it must cross edges at some point. When these edges are crossed, depth values lose influence due to the interpolation formula. The influence loss however is not enough. Numerous countermeasures have been implemented to avoid these artifacts, but it was only possible to decrease their effect. The difficult implementation on a GPU, the uncertainty to ever reach interactive frame rates on the target platform and the artifacts lead to the decision to design and implement a different algorithm. The ideas that depth values spreading influence across the image and passing edges leads to less influence, inspired the design of a new upscaling algorithm.

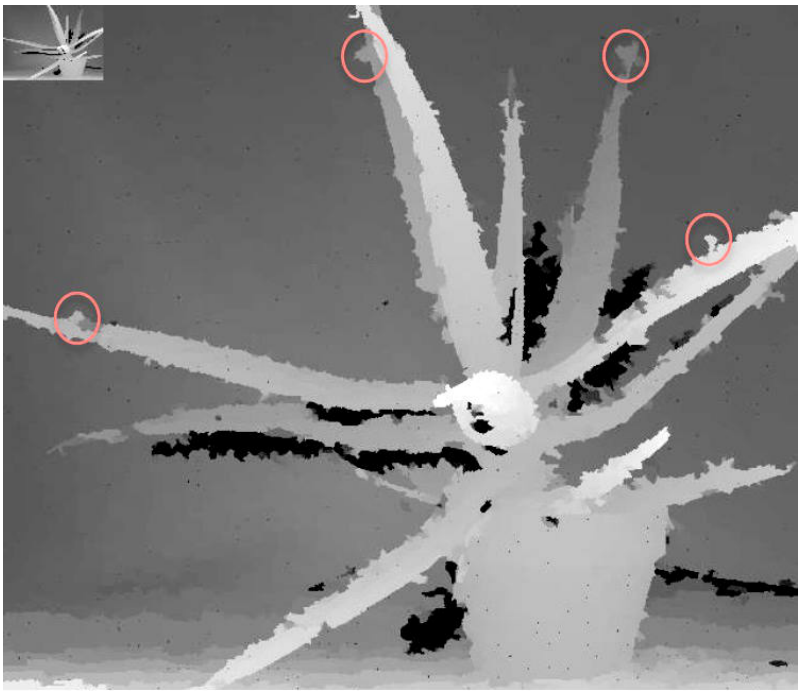

Figure 3.9: Errors of the upscaling algorithm of Dai [Dai+15]

3.4.2 Design of the Guided Depth Diffusion Upscaling Algorithm

Upscaling depth images with color images as guidance is well researched, as shown in Section 2.2.2. However, most algorithms are designed with disregard to computation performance and are not suitable for the requirements of this project. The requirements of a depth upscaling algorithms in this project are:

- **Speed**

  One of the goals of this master's project is to prove the feasibility of depth and color sensor fusion on mobile devices. The algorithm should be able to upscale depth images on a mobile platform with at least 5 frames per second. The 5 frames per second is a rather arbitrary choice, but it is sometimes considered the minimum frame-rate for humans to experience motion. Besides the upscaling computation, the platform should be able to run an application using the fused data simultaneously.

- **GPU Computation**

  An efficient GPU implementation should be possible. Parallel computation and the ability of local memory usage is beneficent for a successful GPU implementation.

- **Depth Information Gain**

  The benefits of having high resolution color information available should be demonstrated by retrieving a depth image of higher quality. Ideally, the resulting depth image contains more information due to the higher resolution, but also has less noise.

The Principle

The upsampling is a crucial part of the sensor fusion processing pipeline. It is executed after low resolution depth values are mapped to the RGB image. The input, as seen in Figure 3.10, consists of a buffer with some depth values (sparsely mapped depth frame) and the color guidance image.

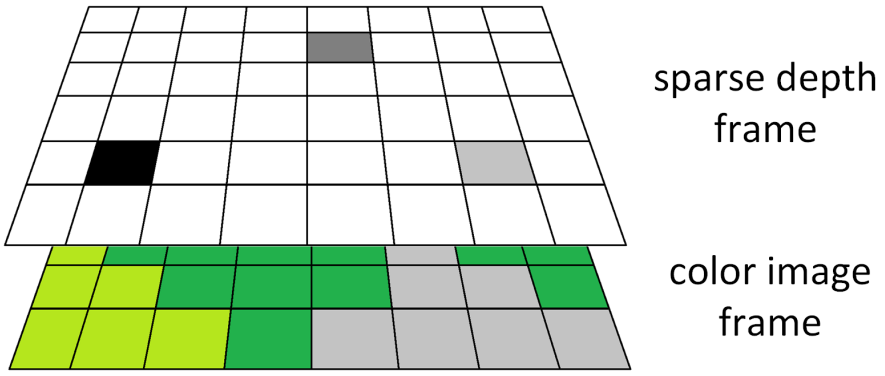

Figure 3.10: The input for the upscaling algorithm

The principle of the algorithm can be interpreted as depth values spreading their influence over a local area of the image. The influence decreases when edges of the color image are passed. Ideally, every pixel of the color image is reached by several depth values. The influence for each pixel is saved in a separate buffer and updated every time a depth value influences a pixel. In Section 3.4.2, the interpolation and updating algorithm are explained in detail.

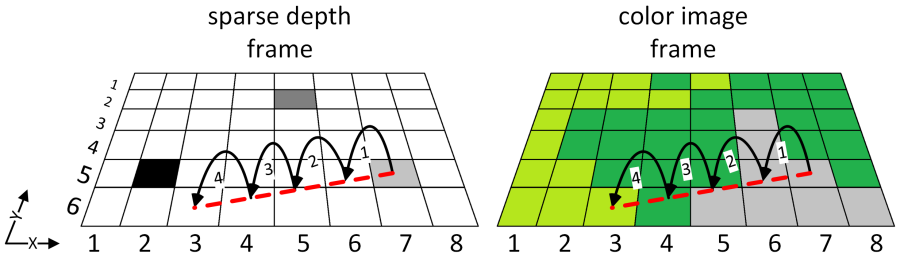

Figure 3.11: The influence path of depth value (7/5) to pixel (3/6).

Figure 3.11 shows the path of influence of a depth value. The dotted red line marks the traveled path to the destination pixel. For every pixel on the path, the influence is calculated and updated. Step 3 from pixel (5/1) to (4/1) passes a different colored region. The influence decreases after this step. During the next step (4), another edge is passed and the influence decreases again. In contrast to the joint bilateral filter, the spatial distance between each pixel and the depth value has no influence.

A Measure for Color Discontinuities

To be able to quantify how much influence each depth value loses on the output image, a measure for color image discontinuities needs to be defined. These differences are the edges of an image.

Dedicated edge detection techniques such as Canny, Sobel, Laplacian of Gaussians and Prewitt were evaluated. The results of the evaluation are discussed in Section 5.2.3. This evaluation lead to the conclusion that a binary edge detection is not the best way to create an upscaling guidance image. After experimenting with numerous edge detection filters, a simple filtering technique yielded the best results. The design of this filter was influenced by the goal to create a fast GPU implementation. Since memory access consumes time on a GPU, the filter only considers the values of the 4 surrounding pixels. The filter kernel is shown in Figure 3.12. The numbers are the weight for each pixel. The guidance value is the sum of these weighted pixels.

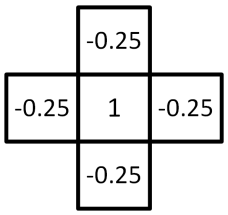

Figure 3.12: The guidance image filter kernel

The resulting guidance image can also be seen as a gradient image, since each pixel stores the sum of gradients magnitudes to its neighbors. An experimental evaluation showed that this filter yields the best results if applied to the color saturation and sum of RGB values R of an image. The sum of the red, green and blue channels is a simple approximation of brightness.

The saturation is a parameter of the HSV color model. It is used to detect edges between objects which differ in color but not in brightness. The saturation S and RGB sum R, are combined by calculating a guidance value for each of them (G_R, G_S) and are applied to the larger value of the final guidance image G. It is wise to define an RGB sum threshold T_R, where no saturation gradient is able to influence the guidance image. This is caused by saturation noise in darker areas and its compensation is evaluated in Section 5.2.3. The computation of the guidance image G is expressed as:

$$G_R(x,y) = |R(x,y) - 0.25 \cdot R(x-1,y) - 0.25 \cdot R(x,y-1) - 0.25 \cdot R(x+1,y) - 0.25 \cdot R(x,y+1)|$$
$$G_S(x,y) = |S(x,y) - 0.25 \cdot S(x-1,y) - 0.25 \cdot S(x,y-1) - 0.25 \cdot S(x+1,y) - 0.25 \cdot S(x,y+1)|$$

$$G(x,y) = \begin{Bmatrix} G_R(x,y) & if\big(G_R(x,y) > G_S(x,y)\big) \\ G_S(x,y) & if\big(G_S(x,y) > G_R(x,y) + T_R\big) \end{Bmatrix}$$

The fast GPU implementation is explained in detail in Section 4.5.2. In Figure 3.13, an example color image with detected edges is shown. Since the edges are linearly mapped to intensity values, not all edges on the figure are visible to the human eye.

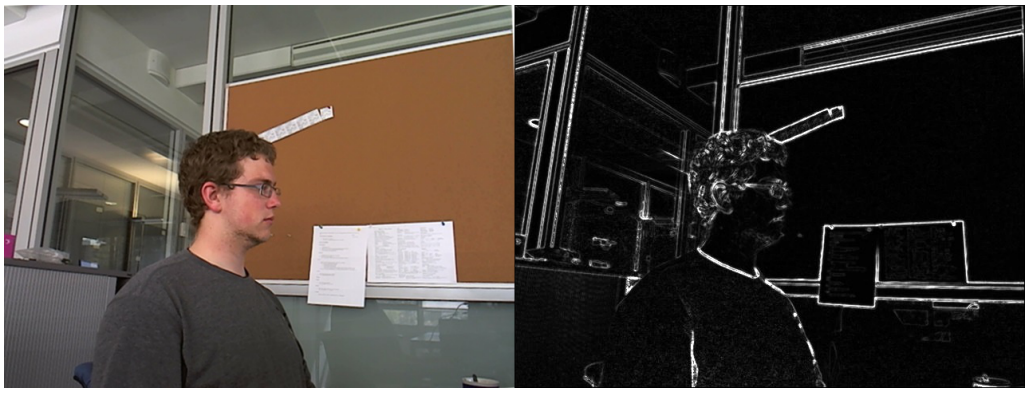

Figure 3.13: Left: The color image; Right: The guidance image

Influence Interpolation

The influence each depth value has on the final depth image is a scalar value w, which is used during interpolation. When there are edges between the depth value d and the current pixel $p_{(x/y)}$, w should to be low. For these properties, w is stated as function of the sum of edge weights between the depth value d and $p_{(x/y)}$. The sum of edge weights S_E is called path length because of the previous approach to interpret the image as a graph. The path length is calculated by looking up the guidance value of each pixel position between d and $p_{(x/y)}$. The weight for interpolation w, is a function of the path length S_E:

$$w = e^{\frac{-S_E}{\sigma}}$$

The reason for using the exponential function e^x is due to the vast decrease of influence w with small changes of S_E. The parameter σ regulates the decay of influence. In Section 5.2.1, various values for σ are evaluated. The interpolation formula to calculate the interpolated depth d_i for n depth values d with n edge weights w is:

$$d_i = \frac{\sum d_n \cdot w_n}{\sum w_n}$$

Ideally, four or more depth values influence the depth of one output pixel. The number depends on how far each depth value spreads its influence. The covered area is a circle with a radius of k pixel. Due to the dual camera setup, the position of each mapped depth pixel depends on the depth value. This is the reason, why depth values cannot be arranged in a homogenous grid on the color image. As the mapped positions of the depth values change every frame, it is impossible to exactly predict how many depth values will influence an output pixel. The interpolation formula is robust, if only one depth value influences the pixel, the pixel value will be identical with the depth value. If there are no depth values influencing a pixel, there is no interpolation and the value remains undefined.

3.5 Software Design

The software for fusing the color and ToF sensor data was developed for the Snapdragon 810 platform, which is introduced in Section 3.2. The software runs in the Android ecosystem, and the front-end part is thus developed in Java. The sensor fusion processing algorithms are executed on the GPU. OpenCL is used as API for GPU computation. The Java Native Interface (JNI) is used to load and interface with C libraries, which handle the GPU computation with OpenCL.

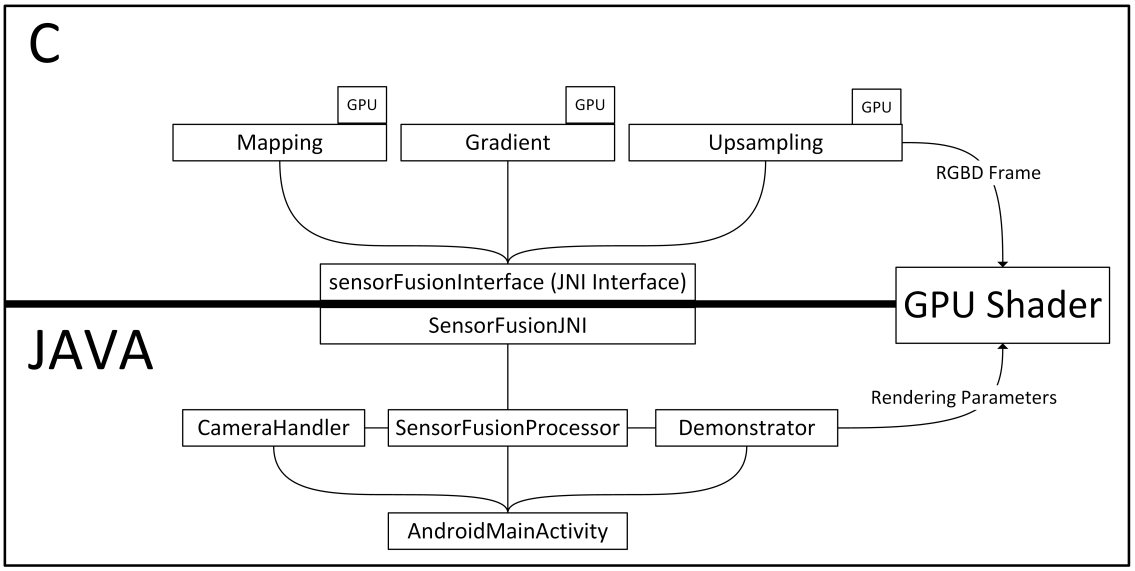

Figure 3.14: The interface diagram between the Java and C part of the implementation.

As shown in the interface diagram 3.14, the application is divided into a Java and C part and are connected with the JNI. The Java application acts as the controller of the sensor-fusion system. Its tasks include:

- **Camera Data Gathering**

  The Java class *CameraHandler* initializes the depth and color camera, by calling functions in native libraries. The description how the cameras are accessed can be found in Section 3.5.1 and 3.5.2. The class provides methods to copy camera data directly to the GPU memory.

- **Initialization and Control of the GPU Processing System**

  The implementation of the OpenCL initialization and processing is implemented in C, and is accessed by Java via JNI. The Java application executes the functions of the C libraries enabling the GPU computation, and adapts certain parameters for performance optimization.

3.5.1 Interface to Depth Camera

The depth camera (Infineon ToF Evaluation Kit) is accessed by using the PMD SDK from PMD technologies. It is a C library that can be included and used with the JNI. There is a plugin for depth data processing that is also a shared library. The processing and capturing of data can be controlled with the SDK. The unprocessed depth data is first copied to a private array on the main memory. The SDK provides a method to copy the depth image to an arbitrary memory location. In the case of this implementation, it is a location on the main memory dedicated for the GPU. The pointer is provided by OpenCL after executing the command *mapBuffer*. The limited data transfer bandwidth of the USB 2.0 connection and the simultaneous use of the USB color camera limits the frame-rate to about 15 fps.

3.5.2 Interface to Color Camera

The color camera, introduced in Section 3.2.5 is a Logitec webcam, connected by USB. The Video for Linux 2 (V4L2) drivers supporting UVC are used with the native Android Webcam Library [CP15]. The library is modified for this project, adding control of the auto-focus. The interface is also rewritten, so that a custom pointer can be specified as target for the image data. This enables the video driver to copy the image data to the GPU memory.

3.5.3 Data Flow

The data flow is a crucial aspect of image processing systems. The aim of the software design is to minimize data transfer and design memory access for optimal cache usage. The way to avoid unnecessary memory transfer starts at the depth and color camera interfaces, which talk to the drivers. The interfaces are described in the previous Sections 3.5.1 and 3.5.2. The Java application provides these interfaces with pointers to GPU buffers. These GPU buffers are located in a certain sections of RAM, since the Qualcomm Snapdragon 810 platform has a shared memory architecture.

After the data is copied to the GPU memory, all further processing calculations are done on the GPU. The depth image is mapped to the color image space, resulting in a list of mapped depth data. This procedure was introduced in Section 3.3.2. The color image is converted to a guidance image for depth image upscaling.

Finally an application, receives the upscaled depth and color data by sharing pointers between OpenGL and OpenCL. The shaders are controlled by the Java OpenGL implementation and produce the desired visual output for the sensor fusion demonstration. Alternatively, the data can also be copied into RAM.

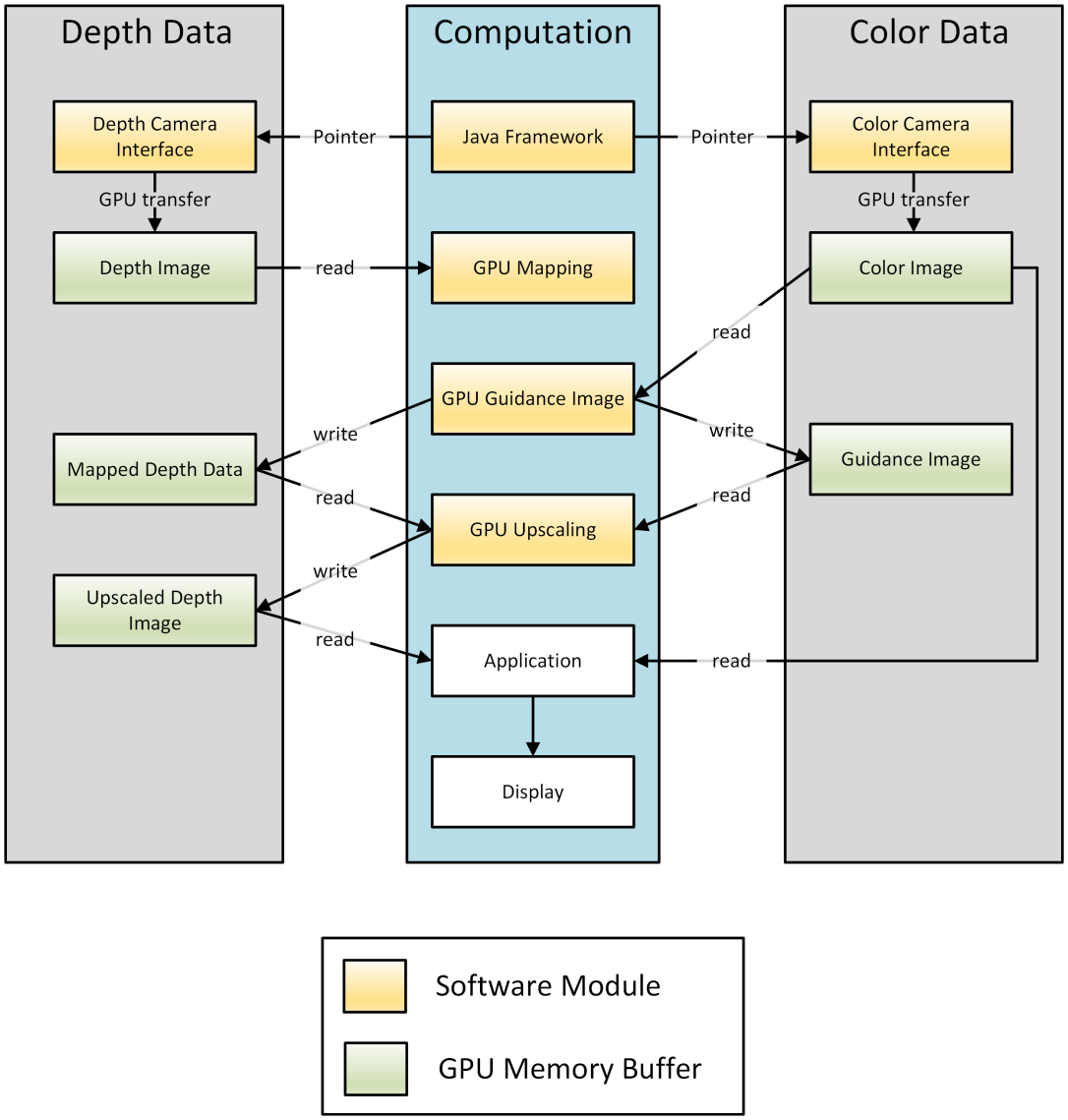

Figure 3.15: The data flow of the sensor fusion system

Chapter 4

Implementation

This chapter gives detailed insight into the sensor fusion system prototype. It begins by examining the hardware platform and the setup, the configuration and calibration are outlined. The next focus is on the work-flow and methods for efficient software development.

The implementation is an Android application, developed for the Qualcomm Snapdragon 810 platform. The application consists of a sensor fusion framework, a native library for GPU computing and three GPU programs, as shown in Figure 4.1. The framework is developed in Java and can be integrated into any Android application. The OpenCL 1.2 API is used to enable fast computation on the graphics device. The native library manages the GPU processing, and is developed in C and the Android Native Development Kit (NDK). The GPU programs are developed in OpenCL C and form the sensor fusion processing pipeline.

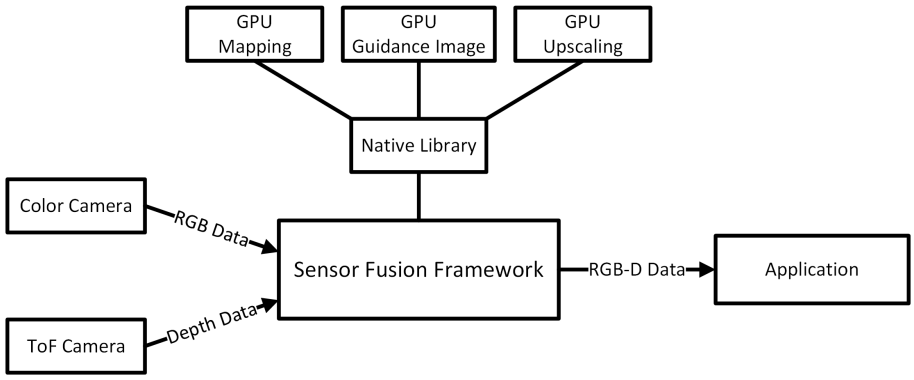

Figure 4.1: The components of the sensor fusion implementation

4.1 The Sensor Fusion Platform

4.1.1 Dual Camera Setup

The camera system consists of an Infineon Time-of-Flight evaluation kit depth camera and a Logitec B910 color camera. The camera specifications were introduced in Section 3.2.5. For optimal sensor fusion results, the cameras are mounted as close as possible and ideally

face exactly the same direction. Like the setup of Ferstl et al. [Fer+13], the cameras are mounted vertically. The final setup is shown in figure 4.2.

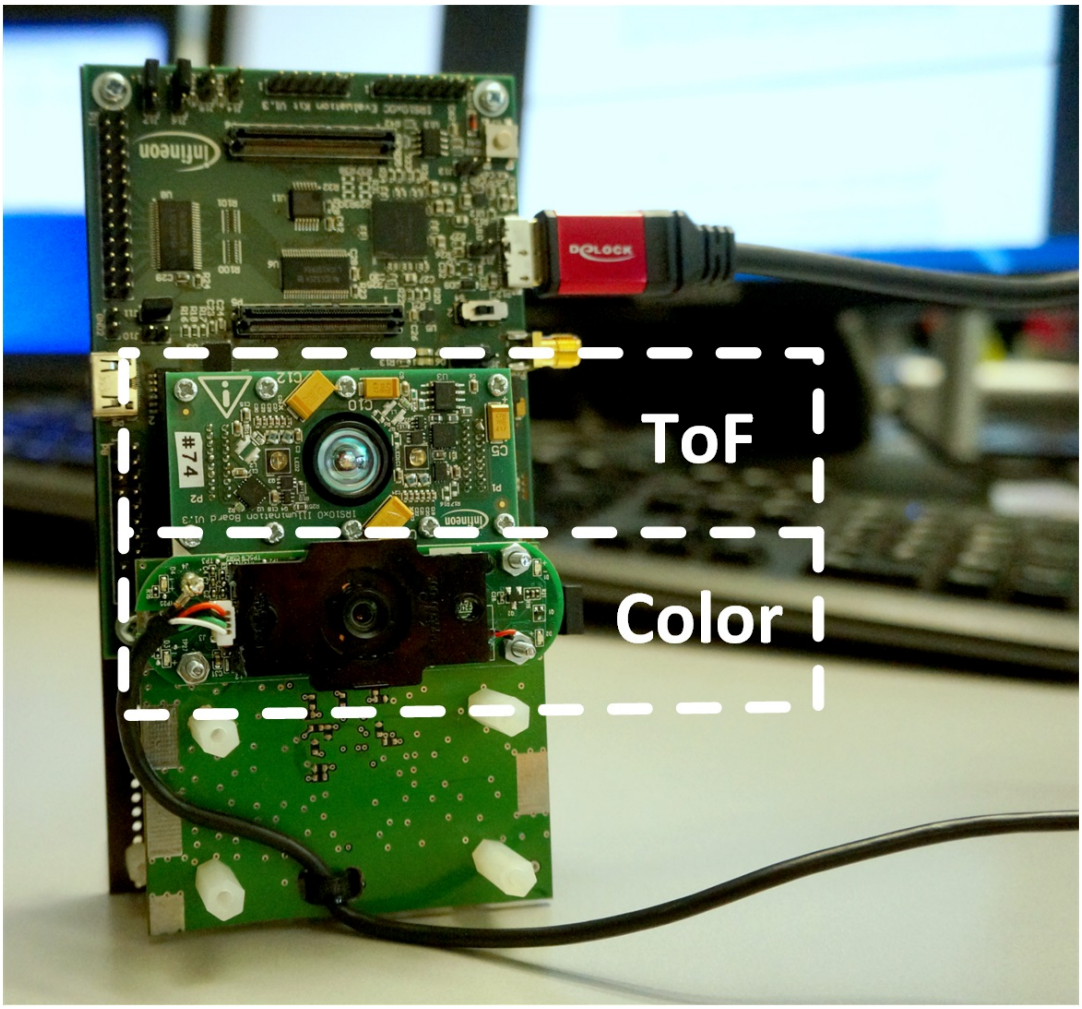

Figure 4.2: The sensor fusion camera system

As mentioned in Section 3.2.5, disabling the auto-focus of the color camera is crucial for a precise depth data mapping. The Logitec B910 camera is able to do this by a UVC command. To send this command, the Android webcam library [CP15] is modified. The following code is inserted into the startup module:

```
        set_autofocus( fd, 0);
        set_focus(fd, -1);
```

The first command disables the auto-focus, the second one sets the focus to infinity. The Logitec B910 webcam does not support the second command, however the camera always sets the focus infinity on startup. Executing the command anyway helps with compatibility to other UVC color cameras.

Another modification of the color camera is direct memory copy. The implementation of the Android Webcam Library [CP15] copies the color data to a Java array. Because the

Qualcomm Snapdragon has a shared GPU memory, it is possible to copy data directly to the GPU memory. Therefore, the method *loadNextFrameDirectly* is added to the webcam interface. It forwards the GPU buffer pointer directly to the webcam library.

4.1.2 Camera Calibration

Camera calibration is necessary to transform the depth values to the color image. This section describes how the cameras of the sensor fusion platform are calibrated. Camera calibration is explained in Section 3.3.1. Literature about camera calibration in a depth sensing system is reviewed in Section 2.2.4.

The calibration of the camera system can be reduced to the simpler, well established stereo camera calibration. This is possible because the ToF camera features an amplitude image, which captures the incoming light for each depth pixel. Initially, the intrinsic parameters are retrieved for each camera, then the extrinsic parameters of the system are estimated by evaluating point to point correspondences.

The two most popular options for calibration are either OpenCV [Gar15] or Bouguet Matlab toolbox [Bou15]. Both calibration toolsets yield to the same results, since they use the same methods. They both take multiple images of checkerboard patterns as input. For this project, the Bouguet Matlab toolbox was selected, as it is possible to mark the checkerboard corners manually. The OpenCV calibration relies on automatic corner detection, which does not always work with the low resolution intensity image of the ToF camera. Both tools use the same algorithm and offer sub-pixel refinement of the checkerboard corners.

At first, several images are captured with the sensor fusion system. The ideal calibration dataset consists of images with a rich variety of angles and distances. Figure 4.3 shows the camera configuration and the positions of the checkerboard patterns. The right side of the Figure shows a sample calibration image of the color camera with mapped checkerboard corners of the depth image. The calibration implementation minimizes the error of this mapping.

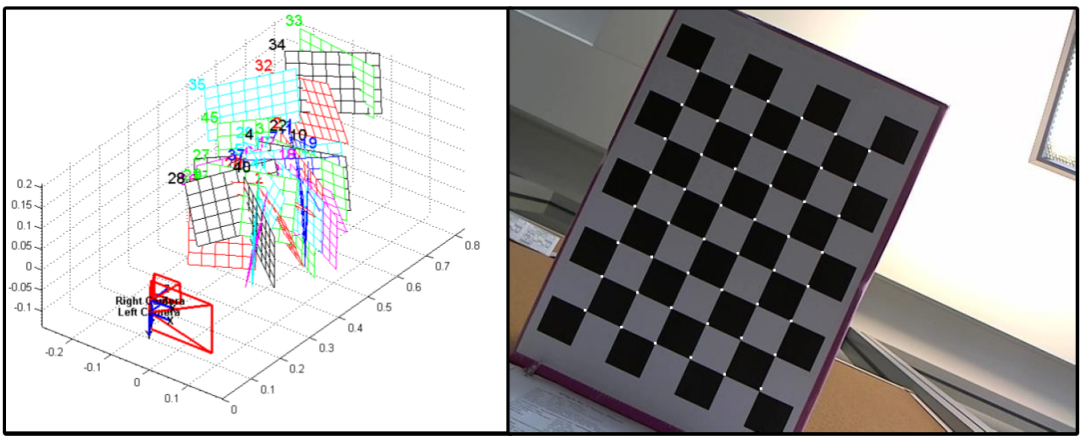

Figure 4.3: Left: Visualization of the camera configuration and calibration targets by Bouguet Matlab toolbox [Bou15]; Right: Calibration Image with mapped corners

4.1.3 Processing Hardware

The sensor fusion processing platform, as shown in Figure 4.4, is the Dragonboard APQ 8094 with the Qualcomm Snapdragon 810 SoC. The hardware platform was introduced in Section 3.2. All developed software is able run on smartphones with the same operating system an similar hardware. The GPU implementation is self-optimizing, which means, that it can optimize its computation parameters in a few seconds. The development kit is rooted and the application needs read/write rights on the camera hardware. The access is enabled by setting the privileges for *dev/bus/usb* (ToF camera) and */dev/video2* (color camera).

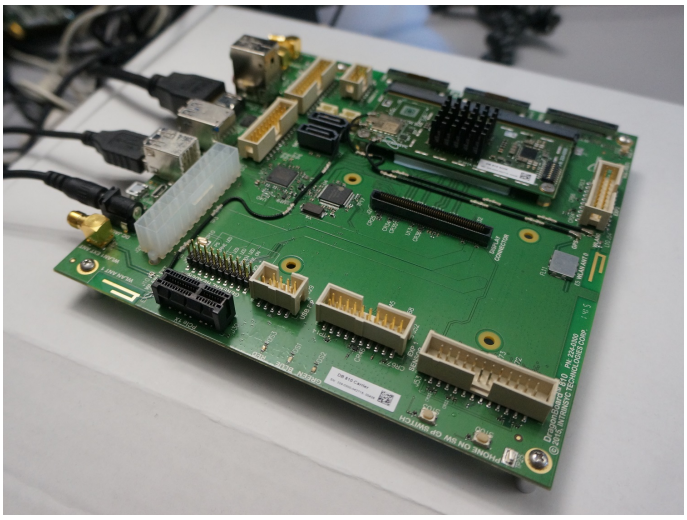

Figure 4.4: The Dragonboard APQ 8094, a development kit for Snapdragon 810

4.2 Development

The development of the software implementation is not trivial. The main application is developed in Java, but for color and ToF camera access and GPU computing, native libraries are developed in C.

4.2.1 Development Environment

The sensor fusion Framework is developed with Android Studio. The C libraries and GPU programs are developed and tested in Visual Studio on a PC, and then compiled with Android NDK for the mobile platform.

Android Studio - Java

The Android Studio currently (2015) supports just Java development and is the main platform for Android application development. Google recommends using Java as primary language and to use the Java Native Interface (JNI) to execute code from pre-compiled C or C++ libraries. Gradle is used as build system. The file *build.gradle* contains the

parameters of the build. For this project the following lines need to be added to the *Android* Section:

```
aaptOptions \{
noCompress '.cl', 'cl'
\}
```

These lines ensure, that the OpenCL kernel source code files are not compressed during the packaging process. This might be necessary to prevent file corruption, since the GPU programs are included in the assets directory of the *.apk* file.

Including native C libraries is not straight forward (2015). With the GCC compiler, included in the NDK, the C source is compiled into binary *.so* files. These *.so* files need to packed into a *.jar* archive file. The exact procedure is described in Section 4.2.1. To tell the Gradle build system to regard the content in these *.jar* file, this command is added to the *dependencies* section of the *build.gradle* file:

```
compile fileTree(dir: 'libs', include: ['*.jar'])
```

JNI - C

The Java Native Interface (JNI) enables Java applications to run C or C++ code. To use it in Android the Native Development Kit (NDK) has to be installed. All native code can only be executed by certain interface methods that contain certain datatypes and calling conventions. While there might be functional code in these files, the usual way is to just cast all datatypes and call the actual methods of the implementation.

To tell Android about the locations and properties of these files, there need to be two files in the same folder as the source code for the interface methods: *Android.mk* and *Application.mk*: In *Android.mk*, the source files for the Android application are specified. To keep the Android configuration simple, only files with the JNI interface methods are specified here. The source files are directly included into the interface file with the *include* command. The *Application.mk* file contains build parameters and compiler options.

To compile the native C source, the *NDKBUILD.cmd* file is called upon. A simple script (*build.bat*) is in the same directory and does everything automatically. It compiles the source by executing *NDK-BUILD.cmd*. Then it calls on another script called *build_jar.bat*. The compiler (GCC) generates *.so* files, which are shared binary libraries. These libraries are packed to a *.jar* file with this script, so Gradle can find the libraries and connect them with the Java implementation.

Android Studio offers the option to execute shell scripts before building the project. It additionally checks if a script fails, and cancels the build. These scripts are executed automatically in Android Studio before the build process.

OpenCL

The GPU programs are called kernels and are developed in OpenCL C. They have the file extension *.cl* file are copied into the asset folder by the script *copy_computekernels.bat*.

Cross-Compilation

The Android tool-chain is not the best environment to develop experimental GPU code. It takes minutes to compile and execute GPU code. A test framework for Windows therefore was developed and is located in the */jni/sensorFusionInterface/sensorFusionInterfaceImplementation*. Most files, except *oclLoadKernel_android.c*, *sensorFusionTesting.c* and *oclLoadKernel_windows.c*, are used among both platforms. This test environment loads test images from the hard drive, using OpenCV, and mimics the computation like on the Android implementation. All operation system specific code is guarded by preprocessor macros. The test-framework is written in C++ to use features like file streams. The code for Android however is developed in strict C99. Figure 4.5 shows the modules of this cross-platform implementation.

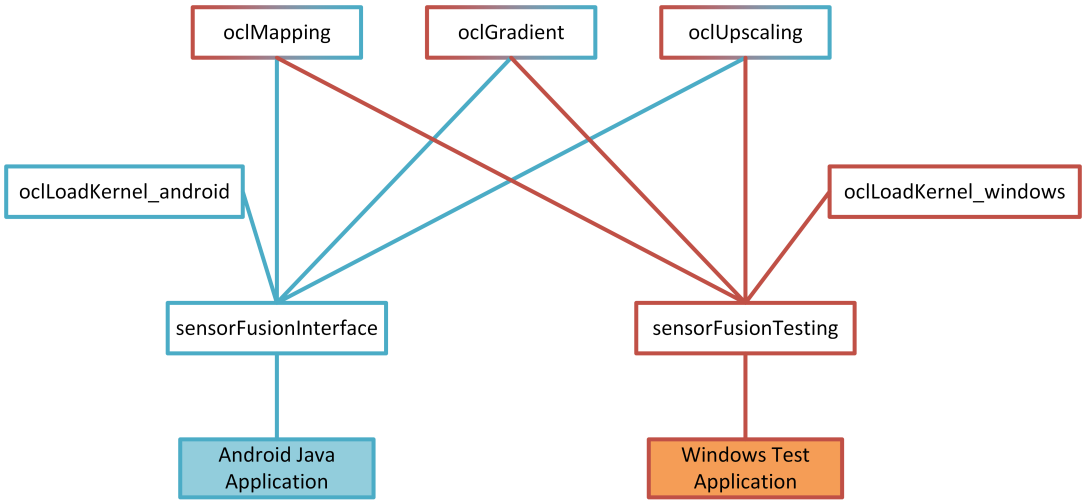

Figure 4.5: Modules of the dual-plattform implementation. Red: Module used by Windows; Blue: Module used by Android; Red/Blue: Module used on both platforms

4.2.2 Workflow

The sensor fusion algorithms are developed in a work-flow shown in Figure 4.6. At first a proof-of-concept prototype is developed with Matlab. All features and principles are experimentally implemented. Before actual data was retrieved from the ToF and color cameras, the Middlebury datasets [HS07] were used for evaluation. When each algorithm was sufficiently developed, the Matlab Code was refactored. Since Matlab is ideally used with vectorized data, all operations using vector data were replaced by scalar data operations. The resulting Matlab code was a closer approximation of how the desired GPU code appears. This step is also necessary to be able to compare intermediate results for bug tracking and helps code translation.

A test framework in C++ is developed for Windows. Section 4.2.1 contains more information about the dual-platform implementations. The Matlab implementation of each algorithm was re-implemented in OpenCL C, using the test-framework for verification. The reason for this step was the faster development environment on a Windows PC. It

takes several minutes for a project to compile, build and execute on a mobile device with Android Studio and Gradle.

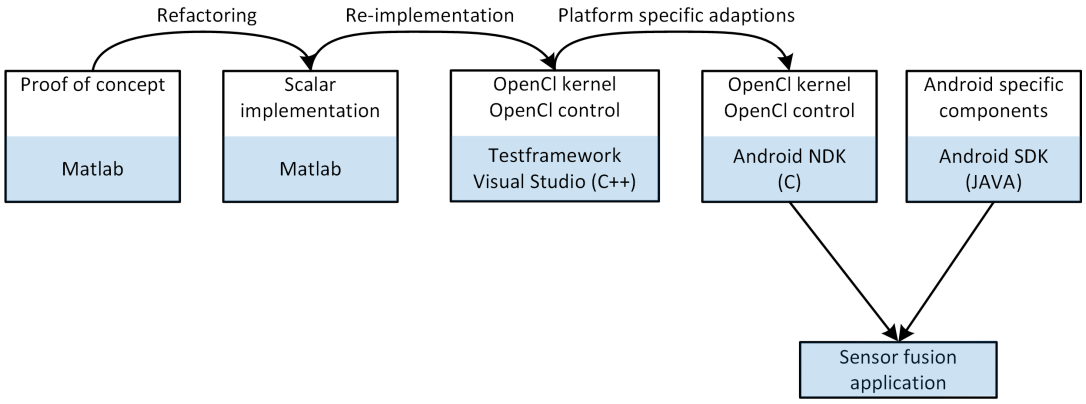

Figure 4.6: The development cycle for developing the sensor fusion algorithms

4.3 Sensor Fusion Framework

The sensor fusion framework is a crucial part of the final sensor fusion implementation. It handles the interaction with the cameras, executes the GPU programs (OpenCL kernels), and provides fused sensor data to other applications. The design choice to develop the central part of the sensor fusion demonstrator as a framework was motivated to enhance re-usability. As shown in Section 3.5, the framework consists of two parts: The Java application, developed in the Android ecosystem, and the native libraries, developed in C. The active C implementation is necessary to use the OpenCL API.

4.3.1 Android Java Implementation

The Java implementation of the framework is the heart of the sensor fusion processing system. It connects the hardware with the sensor fusion computation on the GPU. It unifies the color (RGB) and the depth (D) sensor into a unified RGB-D sensor. Other applications can integrate the framework to gain access to the RGB-D data.

The classes of this framework are shown in Figure 4.7. The Android Activity class is an Android application accessing the framework. The framework is provided by the classes **SensorFusionProcessor** and **CameraHandler**. These classes handle the camera system and the GPU sensor fusion system. The application can access various parameters and data of the sensor fusion system, and can start the camera capturing and processing system.

Class: CameraHandler

This class uses the native depth (*PMD SDK*) and color camera interface (*NativeWebcam*) to access the camera hardware. *PMD SDK* and *NativeWebcam* are interfaces to external implementations. These are C libraries using the Linux camera drivers. The purpose of the

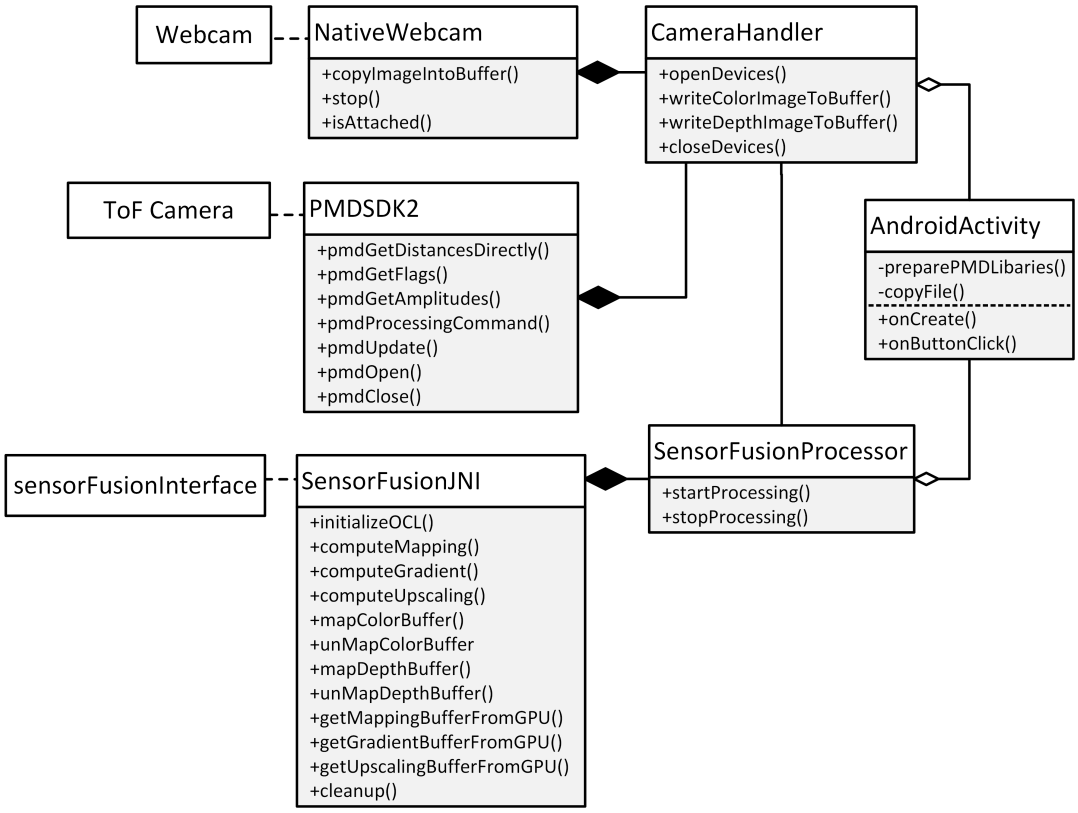

Figure 4.7: Class diagram of the sensor fusion framework

CameraHandler class is to unify both cameras and simplifying access. The camera system is started by calling the method *openDevices()*. The class *SensorFusionProcessor* uses the methods *writeDepthImageToBuffer()* and *writeColorImageToBuffer()* to copy data from the cameras hardware directly to GPU memory.

Class: SensorFusionProcessor

This class regulates the sensor fusion processing at a high level. It uses the interface class *SensorFusionJNI* to access the native implementation *sensorFusionInterface*. This native C implementation enables the execution of GPU programs and is described in detail in Section 4.3.2. Since this class executes all steps of the GPU sensor fusion pipeline, this class can evaluate the computing performance. The main loop does the following steps per frame to calculate the sensor fusion product:

- **Color and Depth Data Acquisition** The depth image is transferred from the camera to a memory buffer on the GPU. This works by requesting a pointer to the GPU memory from OpenCL, by calling the method *mapDepthBuffer()*. This pointer is provided to the camera system by calling *writeDepthImageToBuffer()*. This method requests the current image from the depth sensor via the PMD SDK [Tec14] and copies the processed depth image directly the GPU buffer. The method *sensorFusionJNI.unMapDepthBuffer()* closes access to the buffer and enables the

GPU to use it. The color data is acquired the same way as the depth data, using the adequately named methods.

- **GPU program execution** The GPU programs are executed by simply calling the methods *computeMapping()*, *computeGradient()* and *computeUpscaling()* of the *sensorFusionInterface* class.
- **Sensor fusion data retrieval** The sensor fusion result is in GPU memory. It can be copied into the main memory, or used directly. With OpenCL and OpenGL inter-operation, it is possible to access the sensor fusion result without any copy.

The application using the sensor fusion framework, can start and stop the computation with the public methods of this class. An experimental implementation executes all GPU programs with different local work-group size parameters, determining the optimal parameters within seconds. This automatically optimizes the GPU performance on different platforms.

4.3.2 Native C Implementation

The purpose of the C implementation is to enable GPU computing with OpenCL. It initializes the OpenCL API, creates GPU memory buffers and executes kernels. It is also able to map data from the GPU memory to the main memory, and enables fast and direct memory transfers. For developing purposes, it is possible to retrieve the hardware specifications of the GPU to ensure optimal computing parameters. It is also possible to read out intermediate results of the sensor fusion computation. This part of the framework is implemented as C library, and interfaced with Java by JNI. The library's characteristics forbid the storage of data in the stack memory, and thus the complete data is saved in a struct in heap memory. This data contains mainly OpenCL handles, but also memory addresses for the buffers on GPU memory and computation parameters. For a description of the parameters, see Section 3.2.2. Figure 4.8 shows the modules of the implementation. The connections between the modules represent function calls.

Module: sensorfusionFusionInterface

This module is the interface to the Java implementation. It uses the JNI, and is compiled with GCC. It also abstracts the GPU computation to a simple API with a view function calls.

- **initializeOCL()**
  Creates the OpenCL context. It loads and compiles the GPU programs and creates and allocates buffers on GPU memory
- **computeMapping()**
  Starts the mapping computation on the GPU.
- **computeGradient()**
  Starts the computation of the guidance image on the GPU .

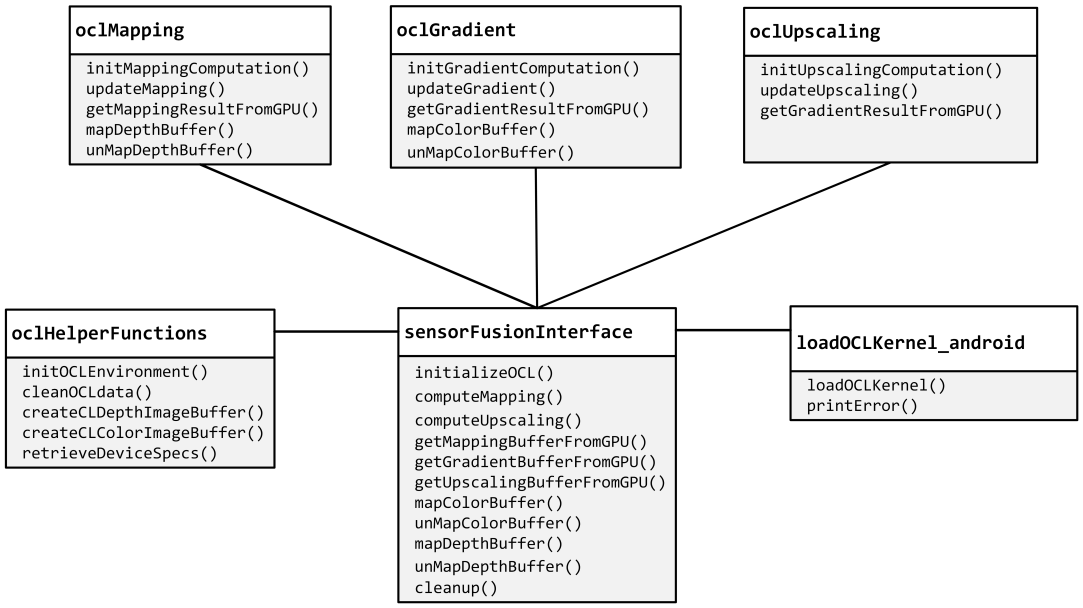

Figure 4.8: Module diagram of the C implementation of the GPU processing handling

- **computeUpscaling()**
  Starts the upscaling computation on the GPU.
- **getMappingBufferFromGPU()**
  Copies the mapping computation result to the main memory.
- **getGradientBufferFromGPU()**
  Copies the result of the guidance image computation to the main memory.
- **getUpscalingBufferFromGPU()**
  Copies the upscaled depth image to the main memory.
- **mapColorBuffer()**
  Requests a pointer to a GPU buffer, which is located in a GPU specific part of the RAM. Access to this GPU memory needs to be requested with this call. The color image is written to this buffer. The GPU might not be able to access this buffer as long it is mapped.
- **unMapColorBuffer()**
  Closes access to the color image buffer. After this function is executed, the GPU can access this buffer again. The GPU might be able access this buffer anyway.
- **mapDepthBuffer()**
  The same procedure as with *mapColorBuffer()*, but with the depth image.
- **unMapDepthBuffer()**
  The same procedure as with *unMapColorBuffer()*, but with the depth image.

- **cleanup()**
  Releases the GPU buffers, closes the OpenCL system and frees all heap data.

This interface is used the following way by the Android application: At first *initializeOCL()* is called. If this function is executed successfully, the GPU device is ready and all GPU programs are compiled correctly. The next step is to transfer color and depth data to the GPU memory. Since the Snapdragon 810 platform has a shared memory architecture, it is possible to request a pointer from the GPU directly. This works by calling *mapDepthBuffer()* and *mapColorBuffer()* and copying data directly from the camera drivers to this memory location. After the data is copied to GPU memory, *unMapDepthBuffer()* and *unMapColorBuffer()* are called to close access to the buffers. After the data is in the GPU memory, *computeMapping()*, *computeGradient()* and *computeUpscaling()* are called. The GPU programs are started with separate methods, because this enables to schedule the computation steps. An example, for applied GPU scheduling is a system, which pushes camera data to GPU memory, while simultaneously executing GPU programms. Separately executable GPU programs also allow to use *getMappingBufferFromGPU()*, *getGradientBufferFromGPU()* and *getUpscalingBufferFromGPU()* to obtain intermediate results for evaluation purposes.

Module: oclHelperFunctions

This module is the last layer of abstraction of the OpenCL API. It provides simple functions for creating the OpenCL environment and creating GPU memory buffers. It can also request the device specific parameters of the GPU.

Module: loadOCLKernel_android

This module does the OpenCL C GPU programs loading in Android. There is also a Windows version of this module with the *_windows* post-fix.

Modules: oclMapping, oclGradient, oclUpscaling

These modules execute the GPU programs (kernels). Each of them is dedicated to a step of the sensor fusion processing pipeline. Each module provides the following functionality:

- **Initialization**

  The initialization function is typically called once. It creates all the GPU memory buffers and connects their pointers with kernel arguments.

- **Update**

  This enqueues the computation on the waiting queue on the GPU. It is optional to also execute the OpenCL command *clFinish()*, which blocks until the computation on the GPU is finished. It is possible to change the local work-group size at each update, by setting the parameters. For the *upscalingKernel()*, it is also possible to only schedule a part of the input data. This enables the application to stay responsive.

- **Buffer Mapping**

  If the GPU program receives external data before execution (i.e. an RGB frame from the color camera), OpenCL can provide a pointer for direct access to the GPU memory. This is possible, because the GPU and main memory of the Snapdragon 810 is located on the same memory hardware.

4.4 Implementation of a State-of-the-Art Upscaling Algorithm for Evaluation

The minimax path-based depth interpolation method of Dai [Dai+15] promises best results for realtime applications (12/2015). Because there was no reference implementation and the authors were not contactable, the algorithm was re-implemented. At first, a simple prototype was implemented in Matlab, but not all concepts of Dai could be implemented in Matlab. The minimum spanning tree (MST) was calculated with the Boost BGL library, and it was not possible to modified the implementation to our needs. The algorithm was then implemented in C to do the computations during the construction of the MST, the authors intended. Figure 4.9 shows the results of the re-implementation after an exhaustive evaluation of the parameter space.

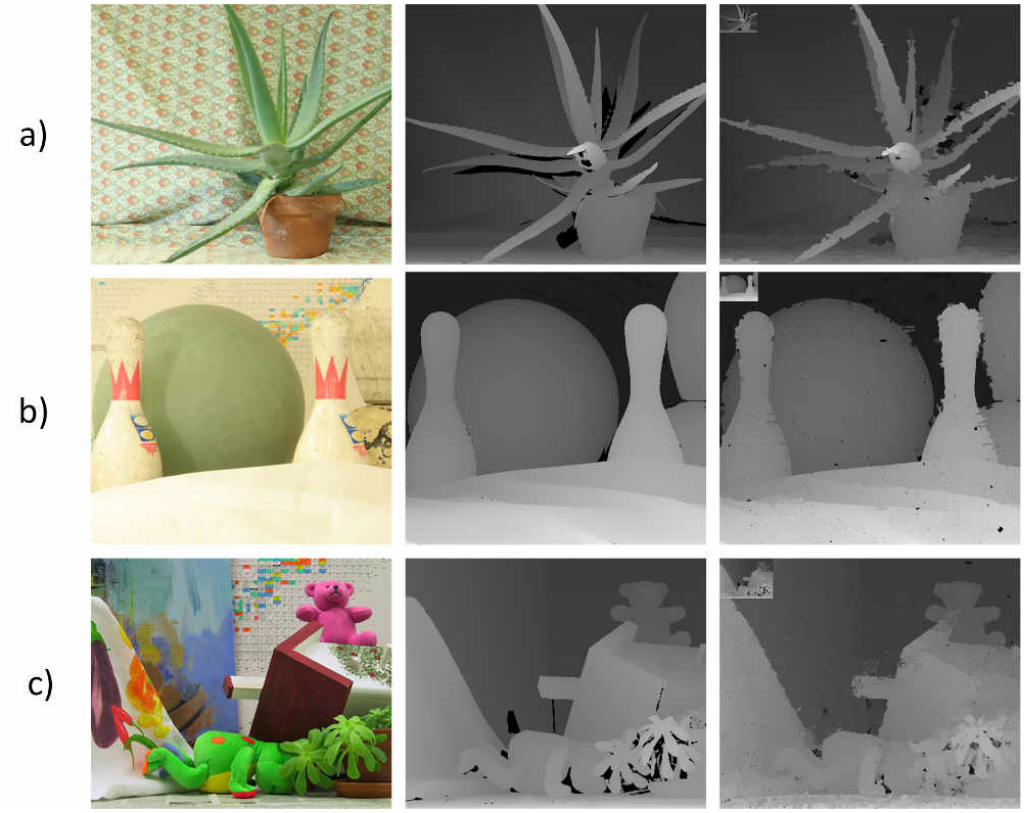

Figure 4.9: Results of the re-implementation of the upscaling algorithm of [Dai+15] on the Middlebury [HS07] datasets. From top to bottom: Color image; Ground truth depth image; Upsampled depth image. The original depth image resolution: 80x60 pixel

Before creating the C implementation, a lot of effort was put into the design of the MST algorithm. Calculating the minimum spanning tree is by far the most computationally complex part of the algorithm. There exist methods to run this algorithm on the GPU [MGG12], but the speedup is limited. The MST computation does not map well on GPU architectures. Every parallelizable step needs synchronization, and there is a massive amount of memory access which is a bottleneck.

One way to optimize MST computation is to take properties of the graph into account. In the case of an image graph, image pixels are nodes and the euclidean color distance between the pixel are the edge weights. This means, that the graph is planar and each node has only has 4 edges, which only connect to neighbors. There exist algorithms, which are able to calculate the MST for a planar graph in linear time [Mat94]. The C implementation was used to evaluate the results to see if it was suitable for this project. As discussed in Section 3.4.1, the quality of the results was adequate, but the performance was not sufficient. The implementation and study of Dai's method lead to the invention of a new approach, which is described in Section 4.5.3.

4.5 GPU Implementation of the Sensor Fusion Processing Algorithm

The sensor fusion processing pipeline is developed for the GPU of the Snapdragon 810 platform. The purpose of this implementation is to explore the capability of mobile GPUs for color and depth sensor fusion. The complete sensor fusion procedure is executed on the GPU and split into 3 stages (mapping, guidance image calculation and upscaling), as described in Section 3.3. Figure 4.10 shows the GPU programs in the context of the system. The arrows represent memory access, and enables to see the origins of the in- and outputs of the GPU programs. The OpenGL shader programs represent an application, using the sensor fusion result. Alternatively, the data can be copied to a part of the shared memory with CPU access. The upscaling kernel depends on the output of the other kernels and is the last stage in the sensor fusion pipeline. The execution order of the other kernels is arbitrary.

4.5.1 Mapping

The mapping is the transfer of depth data to the coordinate space of the color image. The input for this algorithm is the depth image, which is copied directly to the GPU memory. The intrinsic and extrinsic parameters of the cameras influence the mapping transformation. They were retrieved during the initial calibration process, described in Section 4.1.2. The precision of the camera calibration is discussed in Section 5.1. The camera parameters are included in the kernel in form of OpenCL pre-processor constants, and are located in either the fast OpenCL private memory, constant memory or even in the registers. OpenCL determines the actual memory location during compile time.

The output of this GPU kernel is a buffer, containing a list of depth values and their x and y position in the image space of the color image. This list contains the same information as a *sparsely mapped depth frame*, which is commonly found in literature. If mapped to the color image, not all positions hold a depth value. These positions

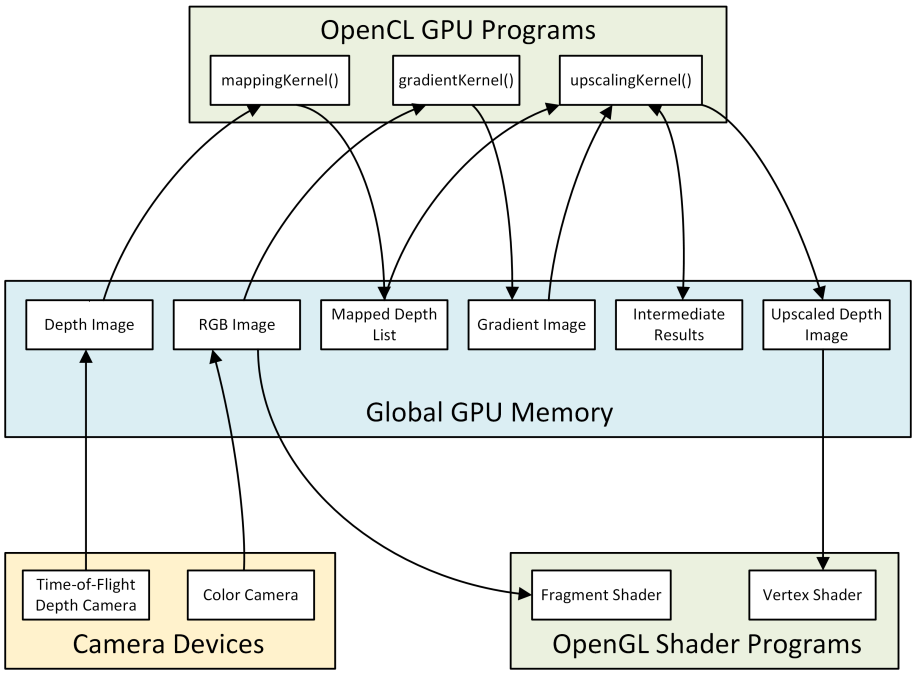

Figure 4.10: The color and ToF sensor fusion procedure

are interpolated during the upscaling stage. Before the depth values are transformed to the image space of the color camera, the lens distortion of the depth camera needs to be compensated. While it is possible to distort pixels by simply applying the formula introduced in section 3.3.1, undistorting pixels is an iterative process The algorithm for the iterative solution according to [Pet15] is the following:

Normalize the pixel coordinates:
$x = C^{-1} \cdot x_{img}$
Repeat until x converges:
$r = \|x\|^2$
$m = k_1 r^2 + k_2 r^4$
$x = \dfrac{C^{-1} \cdot x_{img}}{1 + m}$
denormalize:
$x_{undist} = C^{-1} \cdot x$

C is the camera matrix of the ToF camera. There are only a few iterations necessary. The GPU implementation does 10 iterations. The undistortion is so fast on the GPU that it can't be measured. The mapping kernel then transforms the depth values as described in Section 3.3.2:

$$X_{i,j} = T + Rd_{i,j}\frac{\tilde{P}_D x_{i,j}}{\|\tilde{P}_D x_{i,j}\|}$$
$$\tilde{x}_{i,j} = P_C X_{i,j}$$

The inputs are the depth image pixel positions $x_{i,j}$, and the depth values $d_{i,j}$. The output $\tilde{x}_{i,j}$ are pixel coordinates on the color image. T and R are the extrinsic camera parameters and P_D and P_C the intrinsic camera matrices. Since the depth values are mapped to the original color image, they need to be distorted to model the imperfect color camera lens.

$$\tilde{x}_{dist} = \tilde{x} \cdot (1 + k_1 r^2 + k_2 r^4)$$

For the parallel computation, the work-items are arranged in a 2D grid. Each work-item computes the mapping of one depth value. The grid has the same dimensions as the mapped depth image. Each thread can request their global position with the simple function call *get_global_id()*. The global position is the depth image pixel position $x_{i,j}$.

OpenCL C supports vector data types and operations. The computation is vectorized, and the matrix operations are split into rows. The GPU implementation was stripped from all pre-processor statements, memory transfers and declarations and is listed here:

```
// Projection from 2D image space to the 3D image camera plane
v.x = PROJ_INV_FCX * depthImagePosition.x + PROJ_INV_CCX;
v.y = PROJ_INV_FCY * depthImagePosition.y + PROJ_INV_CCY;
vn.z = 1;

// From 3D camera image plane to 3D depth camera space
vn = normalize(vn) * depthValue;

// Transformation to color camera 3D space, using the extrinsic
   parameters
pos3D.x = ROT_11 * vn.x + ROT_12 * vn.y + ROT_13 * vn.z + TRANS_1;
pos3D.y = ROT_21 * vn.x + ROT_22 * vn.y + ROT_23 * vn.z + TRANS_2;
pos3D.z = ROT_31 * vn.x + ROT_32 * vn.y + ROT_33 * vn.z + TRANS_3;

// Projection to color 2D image space
pos2Dh.x = COLOR_FCX * pos3D.x + COLOR_CCX * pos3D.z;
pos2Dh.y = COLOR_FCY * pos3D.y + COLOR_CCY * pos3D.z;
pos2Dh.z = pos3D.z;

// Conversion to inhomogeneous coordinates
colorImagePosition.x = pos2Dh.x / pos2Dh.z;
colorImagePosition.y = pos2Dh.y / pos2Dh.z;
```

Listing 4.1: GPU mapping computation

All work-items are bundled to work-groups and executed at the same time. This can cause concurrency problems when the same threads access the same memory. Since all algorithms in this projection are designed to be executed in parallel, this problem is minimal. An evaluation of faulty data due to concurrency can be found in Section 5.4.3. The vectorized datatypes speed up computation, since the GPU architecture is SIMD. After computation, the kernels write their result in an array in global memory. Since global memory access is very slow, multiple elements are written simultaneously, which takes as much time as one element. The type of the output buffer is *ushort4*, which contains 4 elements with 16 bit resulting in 64 bit data. The vectorized data is written to the array in global memory the following way:

```
nodeListEntry.x = (ushort) floor(colorImagePosition.x+0.5f);
nodeListEntry.y = (ushort) floor(colorImagePosition.y+0.5f);
nodeListEntry.z = (ushort) floor(depthValue*DEPTH_FACTOR+0.5);
nodeListEntry.w = 1; // Flag, can be set to 0 for invalid values
nodeList[outputIndex] = nodeListEntry;
```

Listing 4.2: GPU mapping result transfer

Qualcomm recommends 128 bit data transfer in each memory write access, so a future optimization can process several depth values per thread and write them simultaneously. The mapping kernel however is the fastest stage in the processing pipeline, and thus not a subject for performance optimization.

4.5.2 Computation of the Guidance Image

The computation of the guidance image is done according to the procedure defined in Section 3.4.2. The GPU implementation works by dispatching one kernel for every guidance image pixel. Like the other kernels, the work-items are mapped to a 2D grid. The input for this kernel is a buffer in global memory, containing the color image. The computation of the final guidance value is done according to the formulas in Section 3.4.2. The saturation value S of each RGB pixel is calculated, using the standard formula:

$$S = \begin{cases} 0 & if\big(MAX(RGB) = 0\big) \\ \dfrac{MAX(RGB) - MIN(RGB)}{MAX(RGB)} & else \end{cases}$$

The euclidean RGB distance D_{RGB} is calculated by applying the L2 norm to the difference of the RGB values. The guidance value is the higher value of D_{RGB} or S and is written into a buffer on global GPU memory. It is used in the next processing step, the depth image upscaling. Section 5.2.3 contains an evaluation of parameters and edge detection methods for guidance images.

4.5.3 Guided Diffusion Depth Upscaling

This is the last and most significant step in the sensor fusion processing pipeline. It is the part of the implementation, which adds resolution to the depth data, and reduces noise. This section is written in a top-down approach: It starts with a simplified, straightforward implementation of the algorithm and then introduces advanced concepts. It finally ends with GPU performance optimizations. The complete depth upscaling is done in one OpenCL GPU program and requires the most time in the pipeline. Figure 4.10 shows the upscaling kernel in the context of the system. The in- and outputs and temporal buffers of this kernel are visualized in Figure 4.11.

The *gradientImage* is a global memory buffer containing the guidance image. The *depthList* is a buffer in global memory containing the pixel position and depth values. The local memory buffers *gradientImageCache* and *weightCache* are used for performance optimizations which are explained in Section 4.5.3. The *upscalingResult* is a global memory buffer that contains the senor fusion product at the end of the computation. During the processing, this buffer is also used to store interpolation information. Each work-item

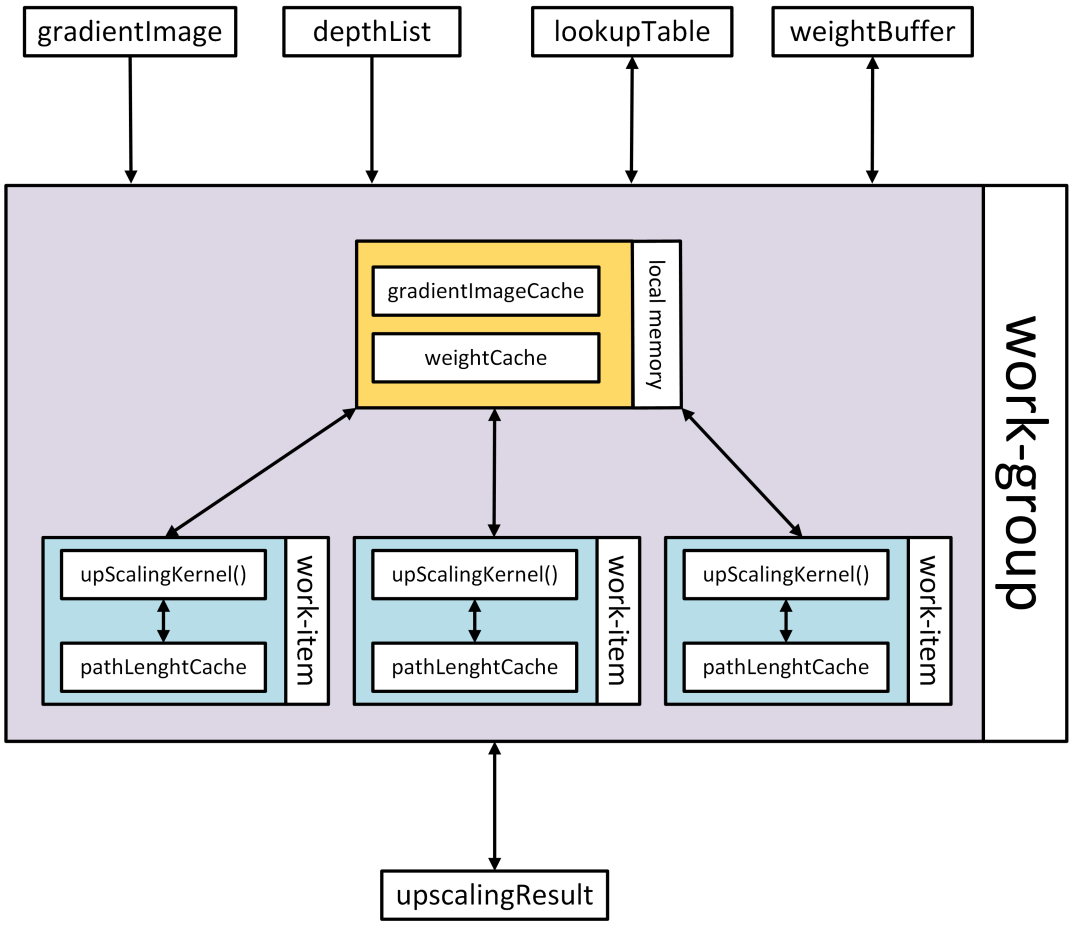

Figure 4.11: The memory buffers of the depth upscaling GPU computation

holds an instance of the private memory array *pathLenghtCache*, which buffers a measure of depth influence for each pixel.

The upscaling algorithm is introduced in Section 3.4.2. The implementation of this algorithm works in the following way: Each depth pixel spreads its influence over a pixel path on the color image. The pixel-paths are the pixels which are on a line between the depth pixel and the currently processed pixel. Figure 4.12 illustrates a pixel path. The area which that is influenced by a depth value is called kernel. A kernel size of 3 processes all pixels with an euclidean distance of 3 pixels to the depth value. The path length p is the sum of guidance image pixel values, which are on the path between the pixel location of the depth value d_n and pixel i. It is a measure on how many edges are between the current pixel and the depth value. The higher the number, the less influence a depth value has on the pixel.

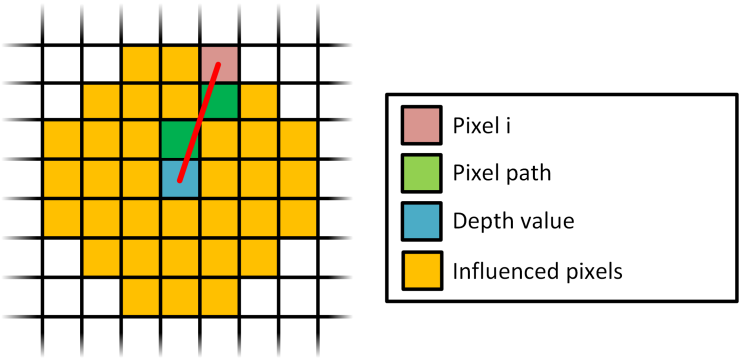

Figure 4.12: The influence of a depth value with kernel size of 3, and an example pixel path

The processing of each pixel is a weighted interpolation of the influencing depth values d_n. This is the interpolation formula from Section 3.4.2:

$$w_n = e^{\frac{-p}{\sigma}}$$
$$d_i = \frac{\sum d_n \cdot w_n}{\sum w_n}$$

There are n depth values which influence the pixel i. The depth values are weighted with the weights w_n and each weight is a function of the path length p. The goal is to decrease the influence as much as possible, if depth spreads over the edges. The weight is therefore a negative exponential function of p with the scalar parameter σ. For the implementation, the sums in this equation are reformulated as an update function. Instead of calculating the sum to get the final value d_i, d_{sum} and w_{sum} are introduced, and updated each time, a depth value d_n influences pixel i:

$$d_{sum} = d_{sum} + d_n \cdot w_{new}$$
$$w_{sum} = w_{sum} + w_{new}$$
$$d_i = \frac{d_{sum}}{w_{sum}}$$

Every pixel has its own d_i, d_{sum} and w_{sum} value, which results in three image-sized buffers on the GPU. For a VGA guidance image, this means 3 floating point buffers with (640x480) entries. Due to their size, these buffers need to be in global GPU memory, where access is very slow. To avoid at least one buffer, d_{sum} can be re-calculated every time instead of saved. d_{sum} is recalculated by reshaping the update formula:

$$d_{sum} = d_i * w_{sum}$$

w_{sum} and d_i are updated after the calculation of d_{sum}. The w_{sum} and d_i values in the formula are a result of a previous pixel update calculation.

If depth value d_n is the first depth value, then w_{sum} and d_i are zero. This leads to the final update formula, which is used in the implementation:

$$w_n = e^{\frac{p}{\sigma}}$$
$$w_{sum} = w_{sum} + w_n$$
$$d_i = \frac{d_i \cdot (w_{sum} - w_n) + d_n \cdot w_n}{w_{sum}}$$

This update formula is expressed in pseudo-code, using the buffer terminology, introduced in Figure 4.11:

```
xy = retrieveCurrentPixelPosition()
depthValue = retrievecurrentDepth()
pathLength = pathLength + gradientImage(xy)
pathWeight = exp(-pathLength/sigma)
weightedDepth = upscalingResult(xy) * weightBuffer(xy)
weightBuffer(xy) += pathWeight
weightedDepth += weightBuffer(xy) * depthValue
upscalingResult(xy)  = weightedDepth / weightBuffer(xy)
```

Listing 4.3: Basic operations for each pixel

Efficient Parsing

The introduced calculations are done for each pixel in the influence area of a depth value. For each pixel the path length p is required, which is the sum of all guidance image values along the path. An example path is shown in Figure 4.12. The idea for an efficient algorithm is that every pixel in the influence area of a depth value is only visited once. When the pixels are visited in the correct order, the path length can be updated during each per-pixel operation. This can be done by starting with the pixels neighboring the depth value and then processing their neighbors. The idea is to visit the pixels along the path, avoiding to parse the path each time, Figure 4.13 illustrates the order of pixel parsing.
The GPU implementation assigns one depth value to each work-group. This indicates that all pixels in the sphere of influence of a depth value, are accessed by a single thread. The private memory buffer *pathLengthCache* contains all path lengths of the influenced pixels. When pixel is processed, the previously calculated path length is looked up in this buffer and the new one is written to the current position in the buffer. This is necessary, as the pixels have different predecessors, and are not necessarily on the same path. To ensure that the pixels are parsed in the correct order and to avoid unnecessary calculations, a look-up table is used. The look-up table gives a relative index of each pixel and its predecessor.

To generate the look-up table a Matlab script was created, enabling the generation of look-up tables for arbitrary filter kernel shapes and dimensions. The script parses all pixels of the filter kernel and samples all pixels along the path. If a pixel pair on the path has no look-up entry, it is added. This is a brute-force approach, but since 15x15 is the largest reasonable filter kernel dimension, it takes just a few seconds. After the Matlab script outputs a string containing the valid OpenCL C code of the look-up table, it can

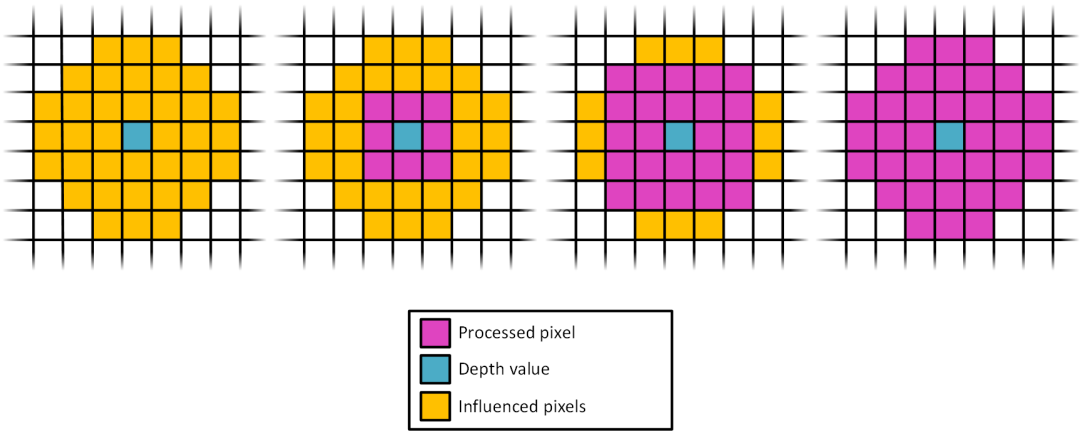

Figure 4.13: Diffusion parsing. All operations depend on each previous step.

be copied directly to the upscaling GPU program. By using pre-processor statements, multiple look-up tables with different filter kernel dimensions can be included in a single GPU program for evaluation. The look-up table is stored in constant memory. This is a read-only portion in global memory with fast access. The OpenCL program iterates the look-up table with a *for* loop. In each iteration, a different pixel is processed. The operations for each pixel are explained in the Section 4.5.3.

Local Memory Caching

The computation performance on GPUs is often restricted by memory transfers. The OpenCL memory model is introduced in Section 3.2.2. Besides the big global memory, there is the much faster local memory, which is shared among the members of a work-group. The local memory is ideally used, when work-items need to access the same data repeatedly. In case of the upscaling stage of the sensor fusion processing pipeline, the buffers *gradientImage*, *weightBuffer* and *upscalingResult* qualify to be used in local memory. Each work-group performs multiple accesses on a local area of these buffers. The obvious way would be to load parts of these global memory buffers into local memory, perform all operations, and then write it back. This is however a disadvantage of the proposed upscaling algorithm, because the location of these areas depends on the input data. Due to the mapping of depth values to the color image space, these depth values are mapped to different positions at every frame. If the complete area of the global memory buffers would be copied to local memory, it would differ at each frame. It would often be such a big area, that the local memory is too small. The solution is illustrated in Figure 4.14: Instead of copying the complete necessary area into local memory, only a part is copied. When a thread accesses a value, the GPU implementation checks if local memory contains a copy and uses it. Otherwise it redirects the access to global memory.

After the upscaling operations are finished within a work-group, the buffer segments in local memory are copied back to global memory. The complete process is called local memory caching. Local memory caching is in concurrency with the GPU memory caching. There exists already a hardware memory cache and the local memory caching might be unnecessary due to the copy overhead. However both caching strategies might be fruitfully

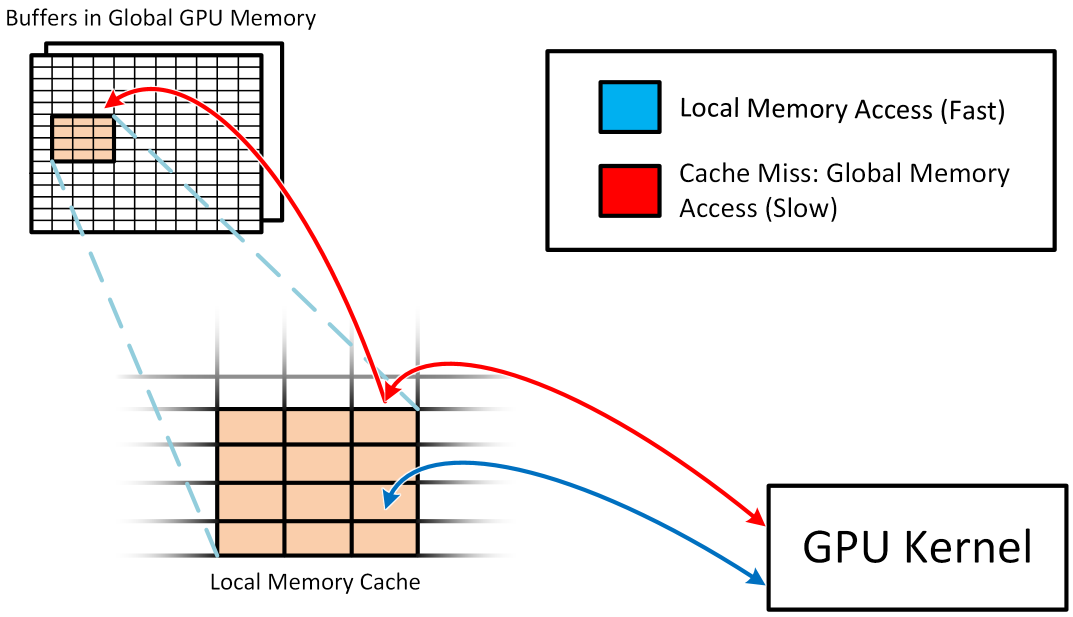

Figure 4.14: The concept of local memory caching

combined. This is evaluated in Chapter 5.3. During the implementation, a different aspect of local memory caching was discovered: If the vendor implemented the OpenCL programing model precisely, the introduced local memory caching would work. However it seems to be the case, that several work-groups are executed in parallel. This observation was experienced on a Nvidia Quadro NVS 4200M GPU on the development workstation and on the Adreno 430 GPU on the mobile platform. If more than one workgroup is executed, one work-group would write data to a global memory buffer, and the other workgroup would write their cached local memory over this data, resulting in corrupted global memory. This leads to the conclusion that just the read-only *gradientImage* buffer can be cached in local memory.

Chapter 5

Results

Figure 5.1: Left: Fused color and depth image; Right: 3D surface mesh

This chapter provides an extensive evaluation of the sensor fusion system. Firstly, the camera calibration is discussed in Section 5.1. A precisely calibrated camera system is fundamental to the color and depth sensor fusion. Section 5.2 illustrates the influence and discusses the best choice for parameters. An important consideration of this thesis is the feasibility study of ToF and color sensor fusion on mobile device, the performance is a crucial aspect and is evaluated in detail in Section 5.3. In certain situations, the depth data can be corrupted or of low quality. Section 5.4 evaluates how the sensor fusion system compensates for low quality depth data. The final sensor fusion results are presented in Section 5.5. The results are visualized and compared to other depth upscaling methods. All visual results were filtered with a salt and pepper noise filter, eliminating single-pixel depth artifacts.

For testing and evaluation, several test-sets were captured with the sensor fusion system. There is no ground-truth depth data available. This makes it hard to quantify the quality of the depth image in a comparable measure. However a visual inspection is sufficient to notice the introduced quality enhancements. The human eye can recognize, if objects in depth images are sharper (more resolution), or smoother (noise reduction). To aid the visual inspection the sensor fusion results are converted to textured 3D surface meshes, as shown in Figure 5.1.

5.1 Camera Calibration

The camera system was calibrated similar to a stereo camera setup, using the amplitude image of the ToF camera as 2D image. The Bouguet Matlab calibration toolbox [Bou15] was used for the calibration procedure. The implementation offers an uncertainty prediction for each parameter, which is approximately three times the standard deviation of all calculated values. The camera system was calibrated using 47 images of a checkerboard. Table 5.1 and 5.2 show the intrinsic parameters and the uncertainties.

Parameter	**Value**	**Uncertainty**
Focal Length X-Axis	541.208	1.514
Focal Length Y-Axis	542.515	1.398
Principal Point Offset X-Axis	316.197	2.361
Principal Point Offset Y-Axis	246.565	2.521

Table 5.1: The intrinsic parameters of the color camera

Parameter	**Value**	**Uncertainty**
Focal Length X-Axis	228.943	0.823
Focal Length Y-Axis	229.087	0.768
Principal Point Offset X-Axis	157.011	1.344
Principal Point Offset Y-Axis	142.722	1.369

Table 5.2: The intrinsic parameters of the ToF camera

The uncertainties are typical for the used cameras. The examples of the calibration toolbox show similar uncertainties. The uncertainties of the ToF camera are lower, because of the lower resolution. Figure 5.2 shows the re-projection error on the color image. The re-projection errors in this image are the X and Y deviations of checkerboard corners of the 47 ToF images, projected to the color image. During sensor fusion, the depth values are projected to the color image the same way, using the intrinsic parameters for transformation. A re-projection error under 0.5 pixels for X and Y means a pixel-perfect mapping.

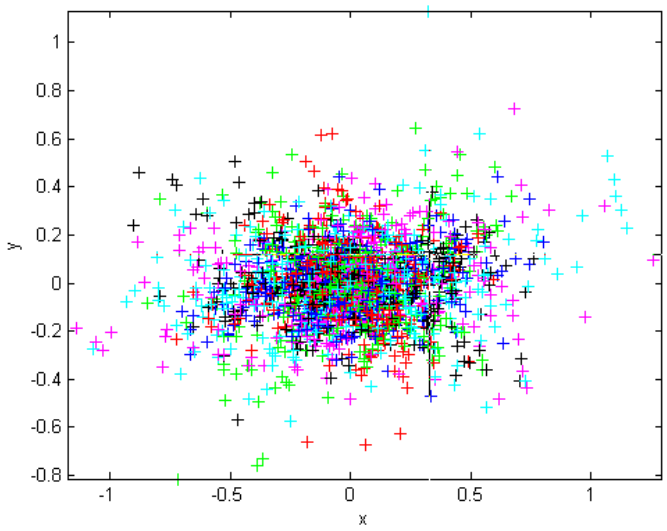

Figure 5.2: The re-projection error of the color image in pixels [Bou15]

5.2 Exploration of the Parameter Space

This section introduces the processing parameters and how they influence the results. Each subsection defines a set of rules to determine the optimal parameter. All further evaluations use the optimal parameters.

5.2.1 Interpolation Parameter Sigma

The interpolation parameter σ is introduced in Section 4.5.3. σ is a scalar value and is used during the depth data upscaling. The parameter regulates the decay of influence when there are color edges between a depth value and the current pixel. The following formula shows the influence of σ on the computation of the path weight w_n.

$$w_n = e^{\frac{-p}{\sigma}}$$

The smaller σ, the steeper is the exponential function and the larger the influence decay. The logical consequence: The smaller σ, the sharper is the upscaled depth images. A visual inspection confirms this. Figure 5.3 shows a depth image with different values for sigma. Some areas in the image appear black due to σ beeing too small. This is caused by depth weights, which are smaller than the 32 bit floating point variable can handle. The image in the center of the figure shows the optimum value and the right image shows blurred edges for large values of σ. The most suitable choice for σ is as small as possible.

5.2.2 Kernel Size

The kernel size is the radius of influence a depth value has on the result. The higher the size the more pixels are influenced by a depth value, which yields to longer computation.

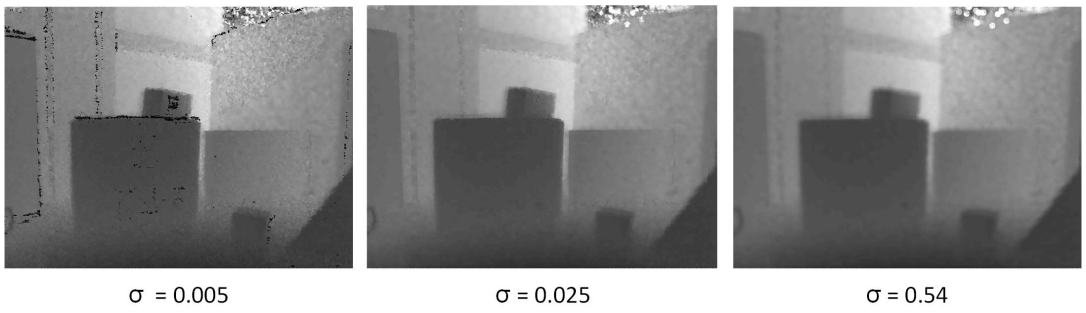

Figure 5.3: Evaluation of the parameter σ

A higher kernel size enhances noise-reduction and makes the sensor fusion algorithm more robust against invalid depth values. A smaller kernel size leads to immensely faster computation. Reducing the kernel size from 5 to 4 reduces the number of operations by 32%. This section reviews the visual influence of the kernel size. Section 5.3.2 discusses the influence on computation performance.

The maximum possible kernel size in the implementation is set to 7. Higher values do not lead to any significant improvement. Figure 5.4 shows 3 images with different kernel sizes. It can be clearly shown, that the depth noise is smoothed with an increasing kernel size, while the edges are preserved.

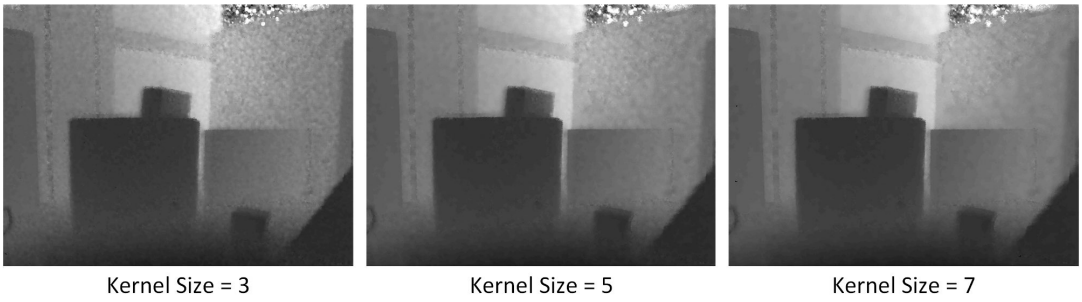

Figure 5.4: Evaluation of the kernel size

The difference between a kernel size of 3 and 5 is much more significant than 5 and 7. Since the kernel size of 5 is also noticeable superior to 4, it is chosen as a compromise between quality and performance.

5.2.3 Guidance Image

The guidance image is an edge image derived from the color image. It aids the depth upscaling and enables the creation of super-resolution depth images while reducing noise without loss. Section 4.5.2 describes this process.

An aspect of the guidance image computation is saturation restriction. As mentioned in Section 4.5.2, both the euclidean RGB distance and color saturation are used to create the guidance image. The saturation however can have drastic noise when the intensity (luminance) of a pixel is too low. A threshold is therefore introduced, limiting the use

of saturation to pixels with higher luminance. The threshold was chosen to be a third of the maximum possible luminance value of a pixel. The luminance is approximated by the sum of RGB values. Figure 5.5 shows the results of this procedure. On the lower images, it is possible to observe a much cleaner guidance image and a depth image with far less noise in this area. The conclusion is, that saturation restriction during the guidance image calculation is very is valuable and does not come with any disadvantages.

The algorithms to create the guidance image can be seen as a parameter of the sensor fusion computation. This section shows that the fast and simple algorithm introduced in Section 4.5.2, was the best choice. Figure 5.6 shows the edge detection algorithms and their influence on the result. The guidance images were created with several popular edge detectors and are shown on the left side of the figure. The middle demonstrates the upscaled depth images and on the right are magnifications.

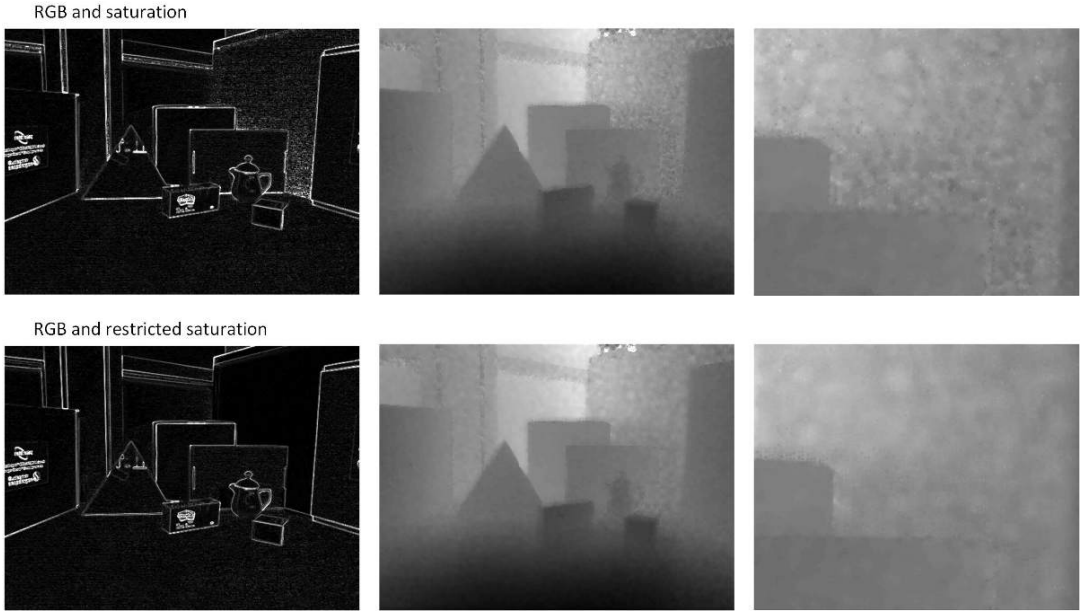

Figure 5.5: Saturation restriction; Left: The guidance images; Right: The resulting upscaled depth images

A visual inspection shows that implementing a rather simple edge detector is sufficient. It needs to be denoted that other edge detectors are binary, while the implemented detector creates a scalar value for each edge.

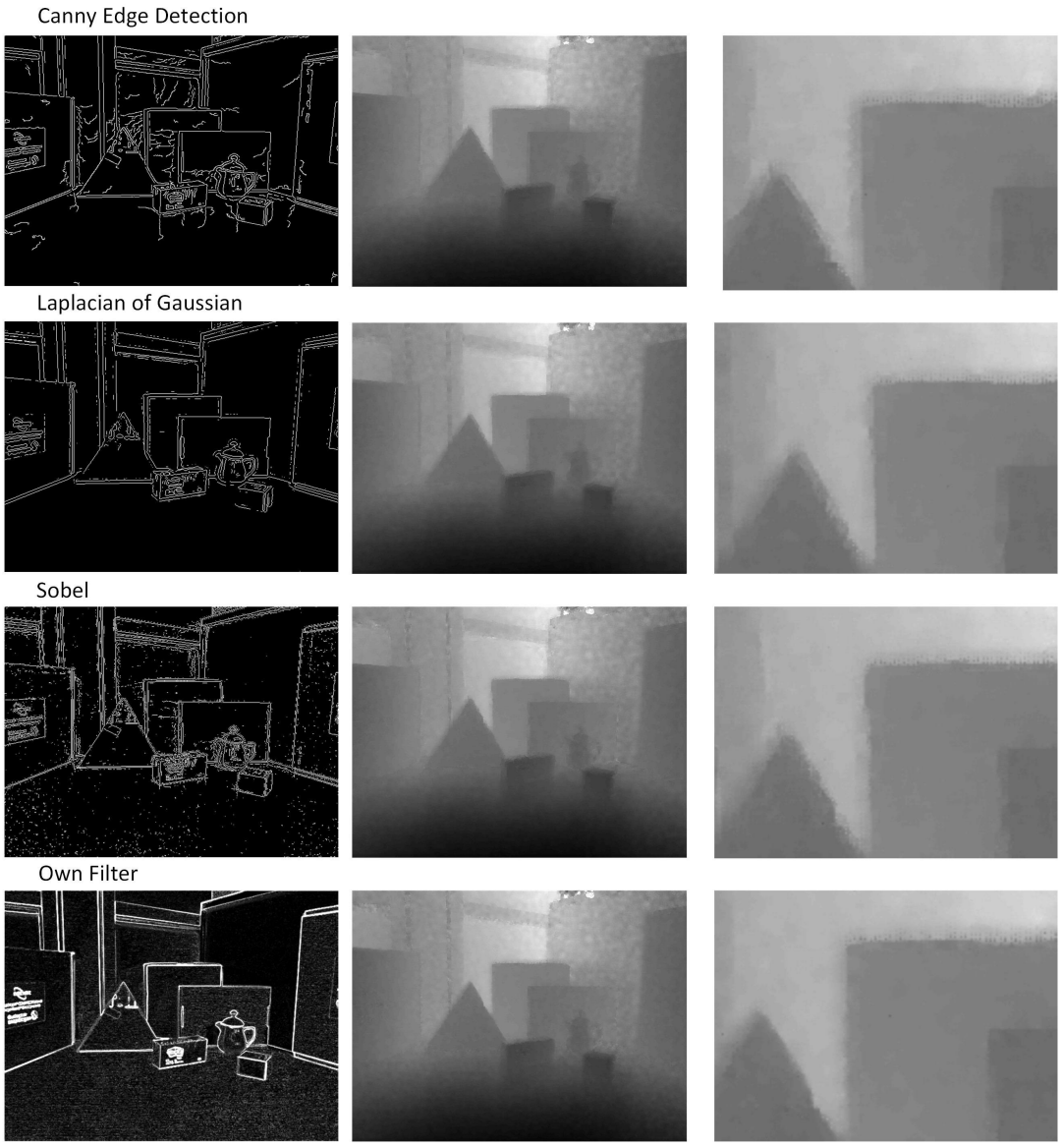

Figure 5.6: Evaluation of edge detectors for guidance image computation. Left: The guidance images; Right: The resulting upscaled depth images

5.3 Performance

The sensor fusion implementation aims to reach interactive (above 5 fps) frame-rate to demonstrate the feasibility of color and depth sensor fusion on mobile devices. The prototype accomplishes all sensor fusing processing operations below 100 ms, which can lead to more than 10 fps. This section evaluates all parameters from which performance is affected. It also gives details about the computational complexity and memory usage. Since the camera setup is experimental, the time dedicated for image acquisition is not accounted in the performance evaluation.

The execution time of each GPU processing pipeline step is visualized in a pie chart in Figure 5.7. Each GPU module was executed a number of times and the execution speed was averaged to retrieve the execution times. The best GPU parameter configuration was derived for each measurement. It is clearly shown that the depth upscaling is the most computationally intensive step.

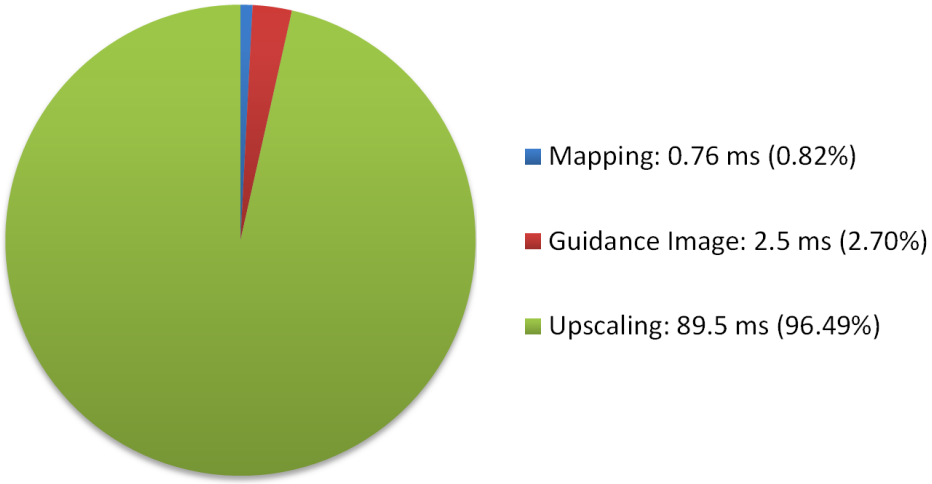

Figure 5.7: The execution times of the GPU processing pipeline stages

5.3.1 GPU Parameters

Mapping		Guidance Image		Upscaling	
Local Work-Group Size	**Execution Time [ms]**	**Local Work-Group Size**	**Execution Time [ms]**	**Local Work-Group Size**	**Execution Time [ms]**
12 x 64	0.76	20 x 48	2.5	24 x 2	89.5
6 x 128	0.78	16 x 48	2.6	48 x 1	90.25
18 x 32	0.79	16 x 60	2.9	12 x 4	92.25
3 x 256	0.81	40 x 10	2.9	6 x 8	100
24 x 32	0.81	40 x 20	2.9	32 x 2	100.5

Table 5.3: The execution times of the top 5 local work-group size configurations

The sensor fusion system optimizes its GPU computation parameters itself. This works by trying all possible combinations of local work-group sizes, until the fastest configuration is found. The local work-group size has a major influence on the computation, because it

defines memory access patterns, the shader processor occupation and cache usage characteristics. There are certain boundaries which reduce the number of valid local work-group sizes to 460 combinations. Only a small part of them yield to good performance, enabling the optimum size to be found within a few seconds.

Table 5.3 shows the top local work-group size configurations for all GPU processing pipeline steps. There is no general optimal local work-group size, since the top configuration varies a little bit on the data. The table also shows, that the execution speed of the top configurations only vary slightly and using the top configurations exclusively yields to good results.

5.3.2 Kernel Size

The relation between the kernel size and the quality of the upscaling results are discussed in Section 5.2.2. Figure 5.8 shows the execution times and the memory transfer volume for 3 different kernel sizes. The memory transfer volume is proportional to the computational complexity, as discussed in section 5.3.3. The figure shows that the execution time develops linear, while the memory transfer develops exponentially. The reason for this is the cache usage: A pixel is influenced by 10.5 depth values on average with a kernel size of 3. With a kernel size of 7 however, a pixel is influenced by 44 depth values. The more frequently the same memory address is accessed, the more operations are cached.

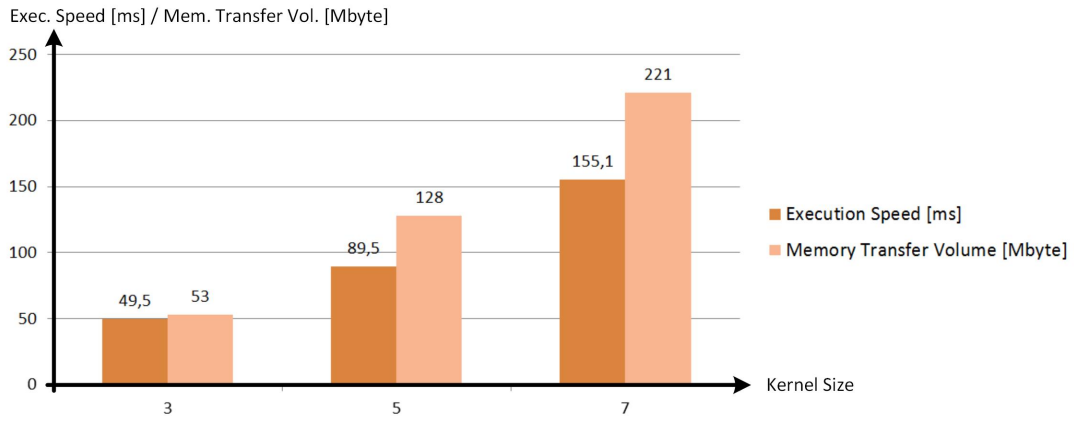

Figure 5.8: The memory transfer volume and execution time of the depth upscaling stage

5.3.3 Computational Complexity

The depth upscaling algorithm was developed for a mobile platform. The implementation fuses 640x480 color with 288x256 pixel depth images with 10 fps. This section analyzes the computational complexity of the guided diffusion upscaling. The complexity class of the upscaling algorithm is $O(k \cdot n_{color})$ where n_{color} is the number of color pixels and k how many depth values influence a pixel on average. k depends on the kernel size and depth resolution.

The most important aspect of GPU performance is the memory access. It is hard to quantify the access in bytes, because a lot of bytes can be read and written simultaneously.

Hence the number of memory access operations are also listed in this evaluation. The GPU execution speed also depends on memory locality due to the cache usage. A well utilized global memory cache may be faster than using local memory. Section 5.3.4 shows the results of the introduced local memory performance optimizations.

Kernel Size	**Memory Read [MOps.]**	**Memory Write [MOps.]**	**Transfer Volume [MByte/Frame]**	**Bandwidth for 30 fps [MByte/s]**
3	9.8	6.5	53	1595
4	15.1	10.0	82	2455
5	22.1	14.7	120	3603
6	30.1	20.1	163	4894
7	40.7	27.1	221	6615

Table 5.4: The memory operations and bandwidth of the depth upscaling algorithm for different kernel sizes

Table 5.4 shows the memory access operations for different kernel sizes per frame. This table considers the depth upscaling algorithm without optimizations. The values in the table assume a depth image with 288x256 pixel and a color image with 640x480 pixel.

The table also predicts the required memory bandwidth for a sensor fusion system, running with 30fps. While the maximum global memory bandwidth of a GPU can be measured, the estimated bandwidth does not include cached memory access. Cached memory access greatly reduces the required data transfer bandwidth, but is difficult to predict.

5.3.4 Local Memory Cache

The use of local memory during GPU computation can cause an increase in speedup. Section 4.5.3 explains the concept in detail. Using the local memory requires also a complete copy of the cached area from global memory. To explore the feasibility of local memory usage, the GPU program was executed with all possible work-group sizes. The cache size was chosen to cover a patch of 96x15 pixel, which lets the fastest (and most relevant) local work-group configuration profit the most. The diagram in Figure 5.9 shows the speedup of using local memory caching. A positive dT value, means that the local memory caching is y ms faster. On the x-axis are the top 20 fastest local-workgroup configurations are listed.
The diagram demonstrates that the difference is marginal. Because of the copy overhead, the local memory caching actually slows down the fastest configuration. The conclusion is that local memory caching can cause a performance gain, however the fastest configurations do not benefit. The reason for this is the GPU global memory cache, which seems large enough to make a local memory caching implementation redundant.

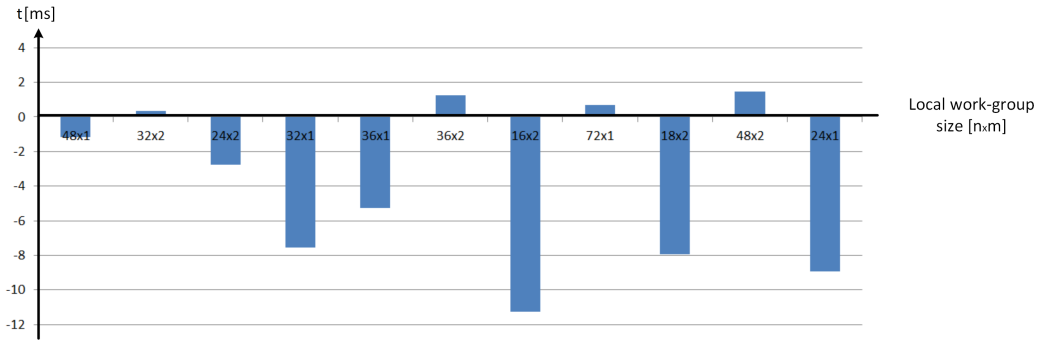

Figure 5.9: X-Axis: The top 20 fastest local work-group size configurations; Y-Axis: The execution time advantage compared to no local memory caching

5.4 Robustness

Another quality criteria of a color and depth sensor fusion system is robustness against poor sensor data. The fusion system prototype was not developed with extra measures against invalid depth data. The upscaling algorithm however can compensate invalid depth data to certain degree.

5.4.1 Holes

Holes in depth data were introduced in Section 2.2.3. The traditional cause of holes is due to a large disparity between cameras. This was avoided by mounting the cameras as close as possible (3 cm). Fast movements however, can cause invalid depth measurements similar to these holes. Since the camera system is not synchronized, the holes are simulated by creating a mask. The mask is applied to the original low-resolution depth image. A non-black value causes the GPU mapping implementation to ditch the current depth value.

Figure 5.10 shows how holes are interpolated. The simulated holes are lines to mimic holes caused by camera movement. The middle image clearly shows, that too large holes cannot be interpolated. The right image shows that increasing the kernel size aids hole interpolation.

5.4.2 Depth Resolution Reduction

Raw depth images will not always have a relatively fine resolution like in this sensor fusion system. This can be caused by the usage of an inferior depth camera, or downsampling to reduce noise. Another case can be a (temporal) depth resolution reduction to boost performance. Bisecting the depth image dimensions means a four times faster computation, see Section 5.3.3. Figure 5.11 shows the upscaled depth image with the original and reduced resolution. The depth image is not downsampled for this evaluation, only every second row and column is ditched by the mapping computation module.
The Figure shows that the upscaling noise reduction suffers a lot due to the reduced resolution. The noise reduction is worse, because each color pixel is influenced by fewer depth values. Due to the guided diffusion depth upscaling algorithm, the edges are still sharp.

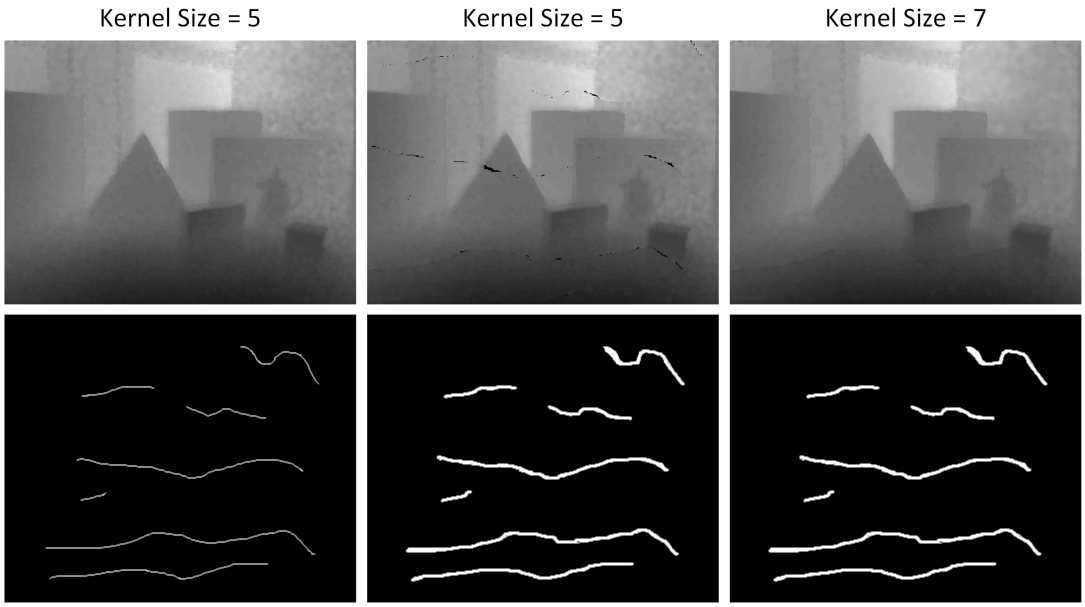

Figure 5.10: Interpolation of artificially created depth holes

5.4.3 Concurrency Evaluation

The GPU implementation does not use any kind of thread synchronization on purpose. Synchronization comes with performance loss and leads to much more complex GPU programs. By design, the upscaling algorithm does not have a lot of thread conflicts and the influence might not even be noticeable. Thread conflicts happen when a different GPU kernel changes a memory buffer, before a kernel has written its result. This section evaluates the influence of thread conflicts. Thread conflicts can be measured by accessing the memory buffers after the computation on each pixel and verify the consistency. Each time a conflict happens, the is pixel marked. It is important to note, that a thread conflict only affects one depth pixel influencing one image pixel. The concerned pixel does not turn invalid. The look-up table is designed to avoid this, but it can happen nevertheless. Table 5.5 lists the number of thread conflicts for 5 test images.

Test Image	**#Corrupted Pixels**	**Percentage of Corrupted Pixels**
1	131	0.043 %
2	52	0.169 %
3	0	0 %
4	1	0,003 %
5	0	0 %

Table 5.5: Corrupted Pixels due to GPU thread conflicts

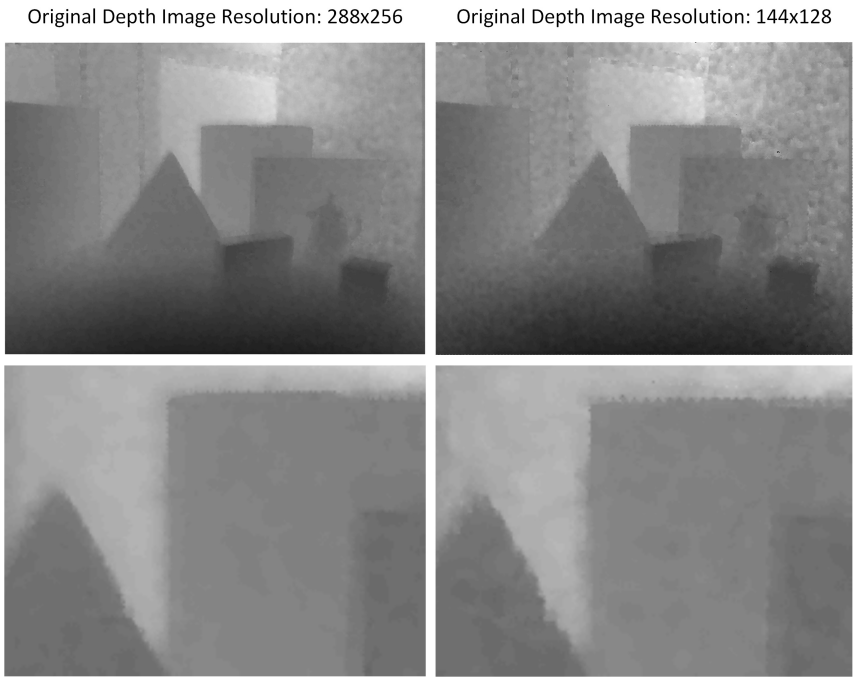

Figure 5.11: The sensor fusion result in comparison with a reduced original depth data resolution

Testset 1 and 2 contain a lot of invalid Time-of-Flight data. The reason is that the depth data is unprocessed and invalid values are not removed. If data is invalid, the depth values are heavily noised, and mapped very irregular to the color image. This is the main cause for thread conflicts. The low number of thread conflicts, verify the design choice of not using memory access synchronization.

5.5 Evaluation of the Sensor Fusion Result

This section presents the final sensor fusion results. The depth results are presented in gray-scale depth images and 3D surface meshes. The results are also compared with other up-scaling implementations.

Figure 5.12: The color images of the two test-sets

To demonstrate the fusion results in this section, two test-sets were captured: The first test-set is a shot of an open office shelf. This test-set contains many invalid pixels and demonstrates a realistic scene. Due to the larger distance, the depth data is a lot more noisy. Many areas are undefined, because they are out of the range of the ToF camera. Future implementations can avoid this problem by checking the signal strength of each depth value, and replacing invalid depth values. The second test-set shows a table with boxes and a piece triangular cardboard. All depth values are valid in this test-set. It simulates a future final form of the sensor fusion system, where all input depth pixels contain valid values.

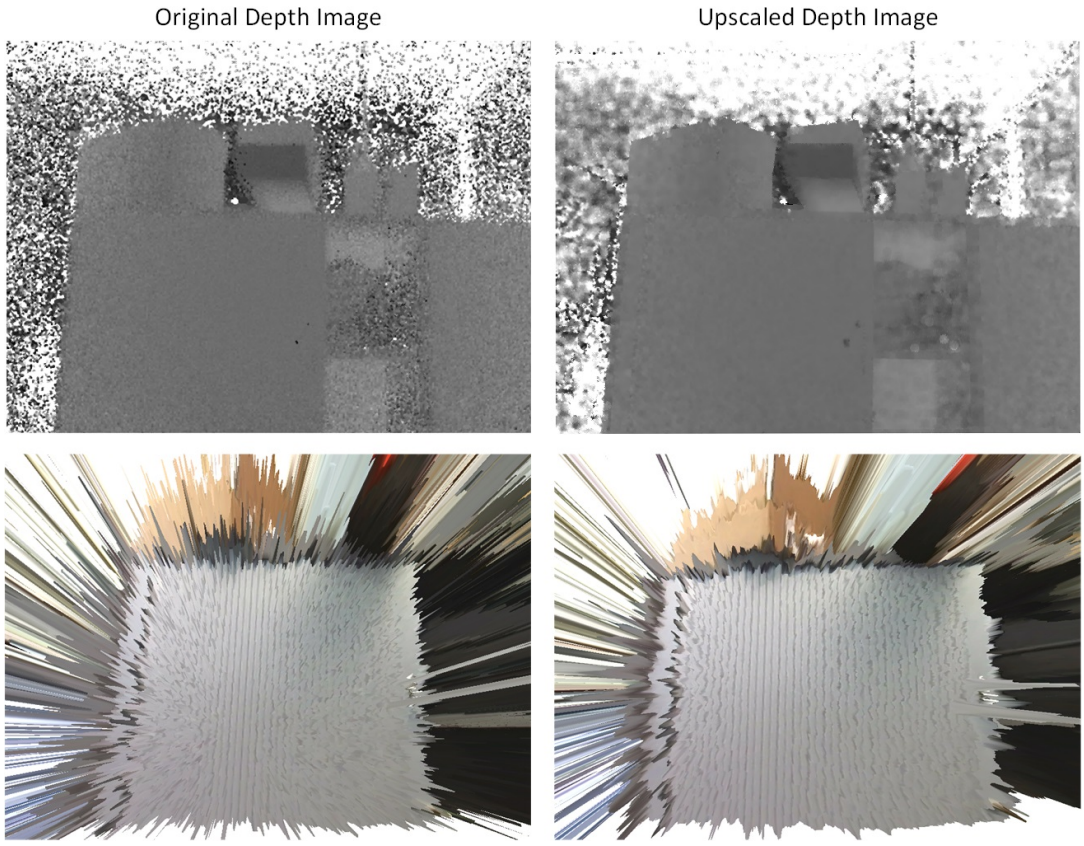

Figure 5.13: The sensor fusion result for the noisy test-set. Top: Depth images; Bottom: 3D surface meshes

Figure 5.13 shows the upscaling result on the first test-set (right). For comparison, the original depth image is upscaled using the nearest-neighbor method (left). The Figure shows a lot of background noise that is caused by invalid depth values. It also shows a degree of noise-reduction, but also demonstrates a flaw of the guided diffusion upscaling algorithm: Depth pixels on edges show only a slight increase in smoothness compared to the rest of the image. This might aid lossless denoising, but also leaves some noisy areas untouched.

Figure 5.14 shows the second test-set containing depth data with higher quality. In the middle image set, the fine differences of the 3D meshes are shown by re-lighting the surface with Meshlab. A massive gain in resolution can be observed, while the noise is reduced tremendously.

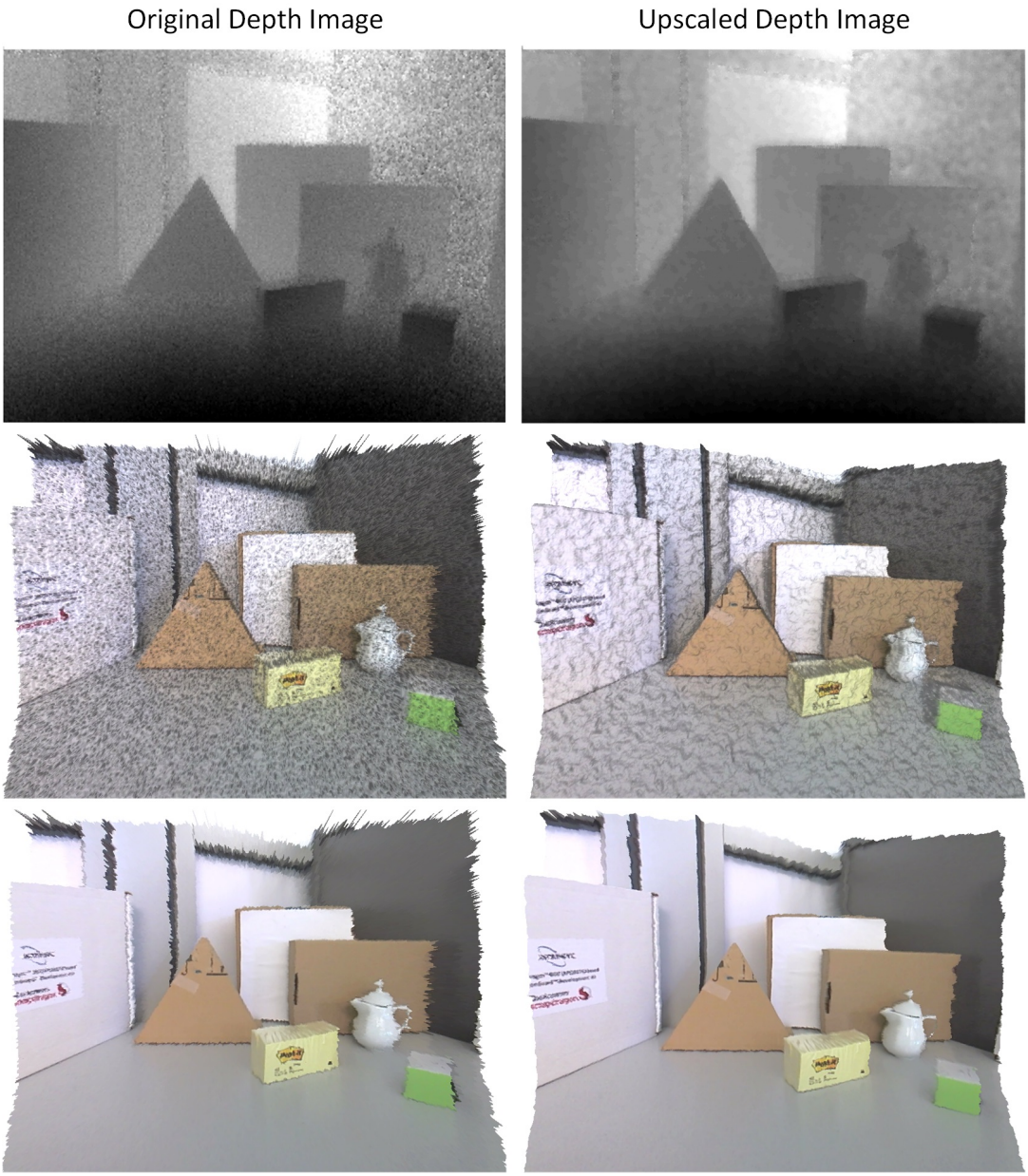

Figure 5.14: The sensor fusion result for the test-set 2. Top: Depth images; Middle: 3D surface meshes re-lighted; Bottom: 3D surface meshes without additional lightning

5.5.1 Comparison with the Joint Bilateral Filter

Figure 5.15: The joint bilateral filter [Kop+07] in comparison with the guided diffusion filter. From top to bottom: Depth image; Zoomed depth image; 3D surface mesh; Sensor fusion result with reduced original depth resolution

The joint bilateral filter by Kopf et al. [Kop+07] is a well-known image-guided upscaling method and is introduced in Section 2.2.2. This section is dedicated to showing the differences between this filter in quality and computational complexity.

Firstly, the computational complexity is analyzed. The guided depth diffusion method, introduced in this thesis, has a complexity of $O(n \cdot k)$. n is the number of pixels and k is how often a pixel is visited. The fundamental difference to the bilateral filter is that the guided depth diffusion algorithm is executed for every depth value, mapped in the color image. The bilateral filter is executed for every color value. There is a lot less depth than the color pixel. The bilateral filter has a complexity of $O(n \cdot k \cdot i)$, where n is the number of color pixels, k is the kernel size and i is the number of iterations.
The number of operations per pixel is identical with the kernel size k. Table 5.6 shows the comparison of the guided diffusion upscaling algorithm and joint bilateral filter. The operations per pixel is proportional with the computational complexity of both algorithms

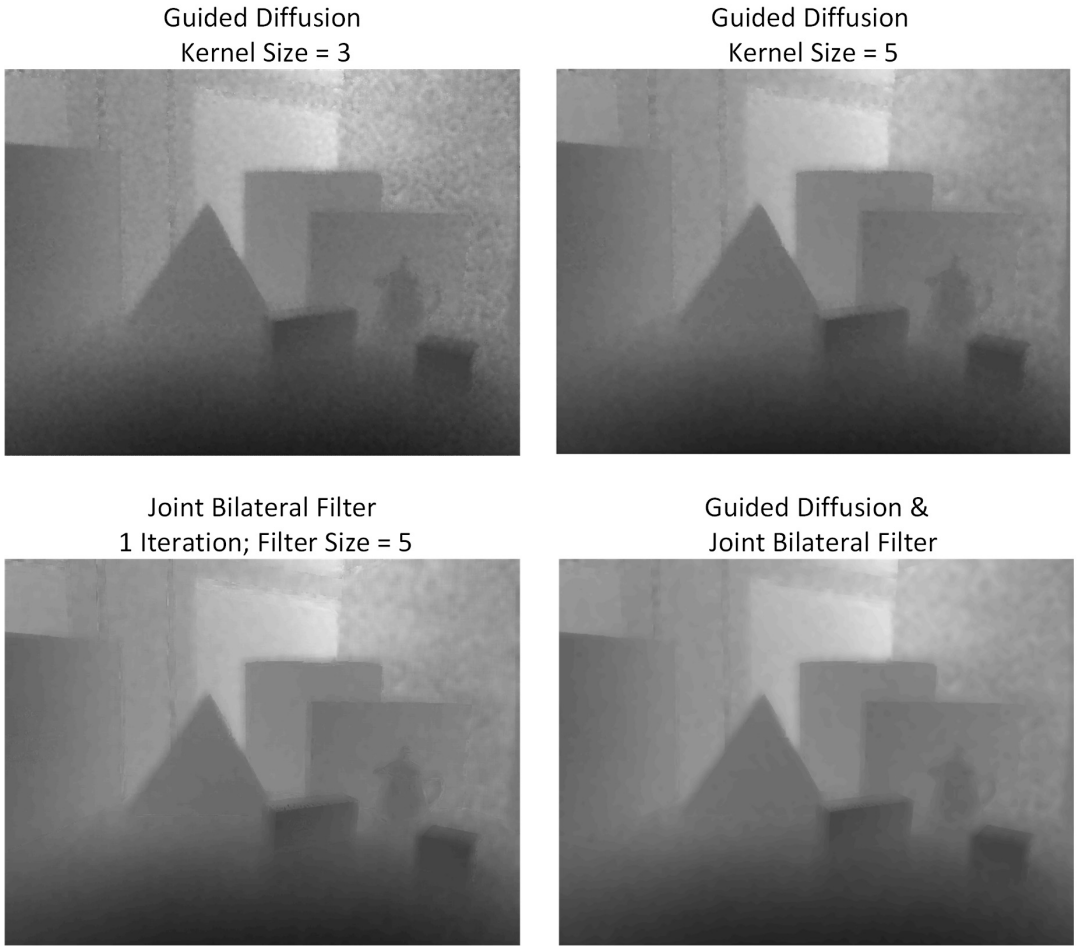

Figure 5.16: The bilateral filter can improve the image guided diffusion

and thus a good measure of performance. To evaluate the quality of the joint bilateral filter, it was re-implemented. After the best parameters were determined, the filter was applied to the data of test-set 2, as seen in Figure 5.15. The third row in the figure is a 3D surface mesh, textured with the color image. The last row is a depth image with reduced input resolution. The resolution was reduced to 144x128 pixels to simulate a large upscaling factor. The comparison shows good results for the bilateral upsampling, especially after 4 iterations, as seen on Figure 5.15 the noise reduction is especially excellent. The joint bilateral filter however is a lot more computationally complex and the 4 iteration version is not real-time capable. Color information can introduce errors. The detail image of Figure 5.15 shows an erroneous region on the side of the triangle. The lines in the surface mesh in Figure 5.15 reveal a loss of information on corners, such as the yellow box and the tea can. The results in the last row of the figure reveal good upscaling qualities, if the depth resolution is reduced. The original bilateral filter does not gain a lot performance, if the depth resolution is reduced.

It is shown in the comparison that the joint bilateral filter is a great local upscaling method. It can be better than guided diffusion in terms of noise reduction and can

Kernel Size	Number of Pixels per Kernel	Operations per Pixel Guided Diffusion	Operations per Pixel Joint Bilatheral Filter
3	44	10,56	44
4	68	16,32	68
5	100	24	100
6	136	32,64	136
7	184	44,16	184

Table 5.6: The number of operations per color pixel compared to the joint bilateral filter (1 iteration) [Kop+07]

produce sharper edges. It is however far more computationally intensive and can introduce erroneous regions and remove details.

An interesting possibility is a combination of guided diffusion upscaling with bilateral filtering. Figure 5.16 shows the depth image for various upscaling methods. The last image shows an image upscaled with guided diffusion with a kernel size of 3 and then filtered with the joint bilateral filter. The result is better than guided diffusion with a kernel size 5 and also, better than just one iteration of bilateral filtering. Upscaling with a kernel size of 3 instead of 5 speeds up the computation by the factor 2.4, but the combination with bilateral filtering will be much slower nevertheless.

5.5.2 Comparison with the Fast Minimax Path-Based Joint Depth Interpolation Method

This method from Dai et al. [Dai+15] proclaimed to be the best depth upsampling method with high performance (12/2014). While the principle makes sense, it has been shown to be not practically applicable in the sensor fusion system in this case. As shown in image 5.17, the algorithm has not features of noise reduction. The reliance on perfect depth values is the reason for its $O(n)$ complexity. The results support the choice in developing the proposed new algorithm.

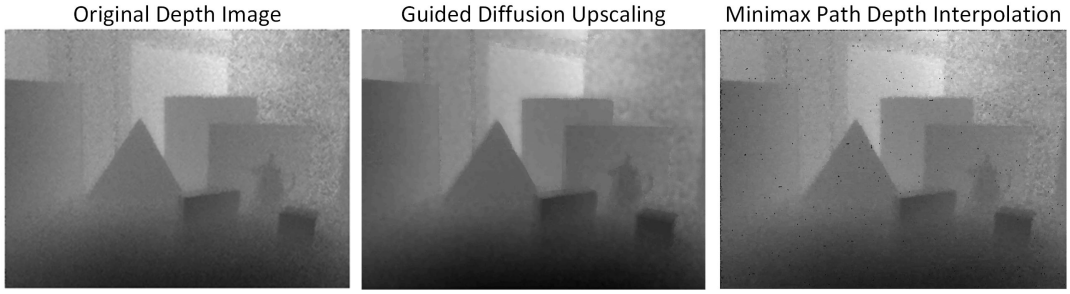

Figure 5.17: The results of the minimax path-based depth interpolation [Dai+15], compared to the original depth image and the proposed guided diffusion upscaling method

5.5.3 Comparison with the Anisotropic Total Generalized Variation Method

This section compares the guided diffusion with the anisotropic total generalized variation method by Ferstl et al. [Fer+13]. In this thesis, it was introduced in section 2.2.2. The method belongs to the category of global upscaling methods which iteratively solve an energy minimization problem. Global methods can produce the best depth images, but are too slow for interactive applications. Figure 5.18 illustrates the comparison which shows that a global method can produce great depth images using the sensor fusion prototype. This demonstrates how much quality improvement is theoretically possible, when performance disregarded.

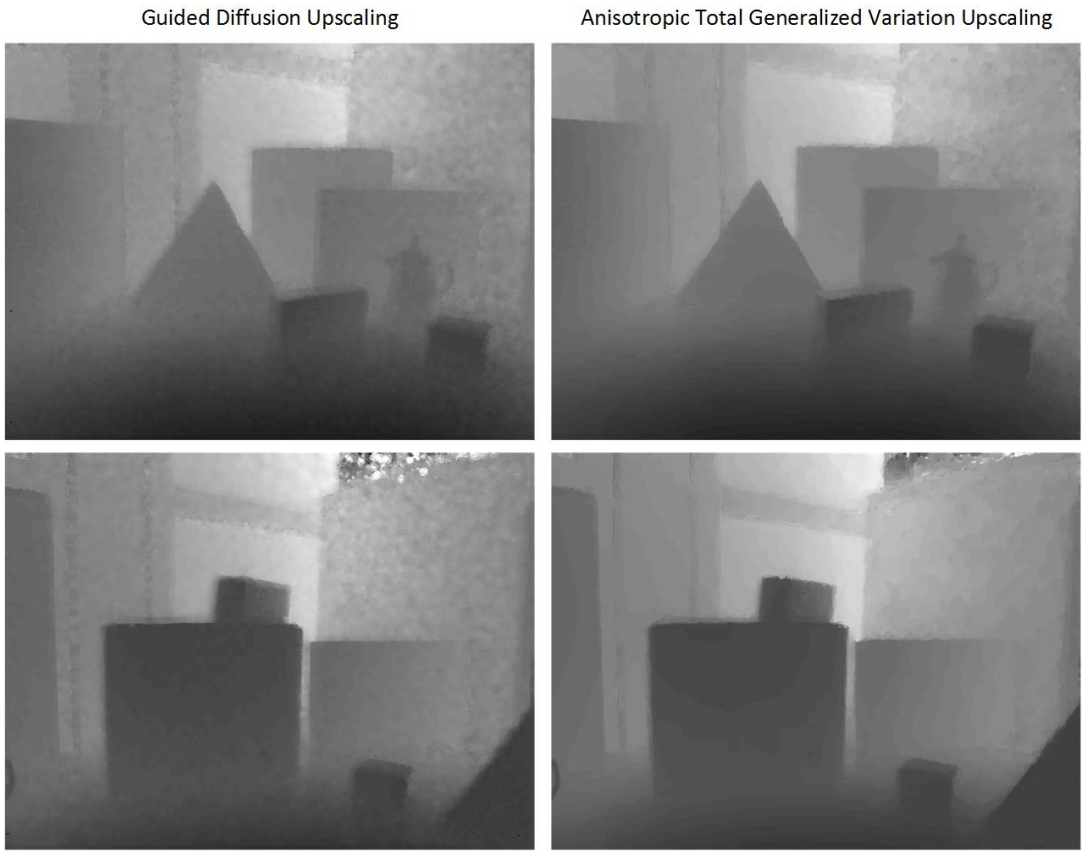

Figure 5.18: Comparison of guided diffusion and anisotropic total generalized variation upscaling methods

Chapter 6

Conclusion and Future Work

6.1 Conclusion

Depth sensing systems on mobile devices are a novelty and their capabilities and applications have only been partially explored. Fusion of data from a color and depth camera however, is well researched. There are many solutions which can increase depth resolution and reduce noise, but most of them are not designed to reach a high performance.

This thesis uses the state-of-the-art Snapdragon 810 platform to develop a prototype to show the feasibility of depth and color sensor fusion on mobile devices. It uses a Time-of-Flight image sensor, which is on the brink of being integrated into a new generation of smartphones and tablets. The output of this system is a color and depth image, where each color pixel is directly associated with each depth pixel. The output depth image is significantly sharper and less noisy than the original.

The cameras are calibrated like a 2D stereo camera system, using the 2D amplitude image of the ToF camera. The evaluation shows that the calibration is precise and the re-projection error within normal boundaries.

While the depth warping algorithms are found in standard literature, the interpolation algorithm is novel and was invented for this thesis. The algorithm is designed to generate a high-quality depth image, while not being computational complex. It is non-iterative and has an $O(n \cdot k)$ complexity. The algorithm is capable of being efficiently implemented on GPUs. An extensive evaluation demonstrates these these claims.

The complete sensor fusion processing pipeline was implemented on the GPU of the Snapdragon 810 platform with OpenCL. Due to the shared memory architecture, it was possible to copy the sensor data directly to GPU memory without redundant memory transfers. Utilizing the GPU on mobile devices for general purpose computation with the API OpenCL, was shown to be a feasible way to build a high performing prototype. As long as a device supports the OpenCL standard, the sensor fusion implementation is device-independent. The implementation automatically optimizes the computation parameters and maximizes the performance. Test and evaluation cycles for mobile GPU development are too much time-consuming to directly develop software for a mobile GPU. It is therefore necessary to develop GPU programs on a PC first then cross compile the framework for the mobile platform.

6.2 Future Work

The feasibility of color and depth sensor fusion on mobile devices has been verified by the development of a prototype in this thesis. Depending on the application, there are certain steps necessary to use sensor fusion method in practice. Future steps to improve quality, speed and applicability involve:

- **Dynamic Kernel Size**

  The upscaling GPU implementation is extended by the capability of using different kernel sizes. For values close to color image edges, the kernel size is smaller than for homogenous areas where noise needs to be reduced. This has the potential to enhance the performance and improve the visual quality of the depth image.

- **Partial Unguided Depth Interpolation**

  When there are no edges in the proximity of a depth value, it may be sufficient to use a simpler depth interpolation algorithm to enhance performance. The GPU offers a rich set of fast image interpolation methods.

- **Joint Bilateral Filter**

  As shown in Section 5.5.1, it is possible to enhance the sensor fusion result by applying the joint bilateral filter. While the guided diffusion upscaling can be sped up, the application of the bilateral filter remains computationally expensive.

- **Selective Sensor Fusion**

  There exist applications, where only small areas of the sensor fusion result are needed. These areas can be calculated rapidly by adapting GPU computation parameters.

- **Camera Synchronization**

  This step is absolutely necessary for a color and depth sensor fusion system. The color and depth camera must capture an image at the same time. When at least one camera can be controlled by software, synchronization is possible without hardware modifications.

6.2.1 Future Applications

Every application using depth data profits from the high quality depth images produced by color and depth sensor fusion. As mentioned in Section 2.4 numerous fields of computer vision benefit from color and depth information. It enables augmented reality to occlude virtual objects and computational photography to re-focus images without light-field sensors. Images can be segmented more robustly and 3D reconstruction can uncover finer details.

Bibliography

[Bai15] Mike Bailey. *OpenGL Compute Shaders*. 2015.

[BFL06] Yuri Boykov and Gareth Funka-Lea. "Graph Cuts and Efficient N-D Image Segmentation". English. In: *International Journal of Computer Vision* 70.2 (2006), pp. 109–131. ISSN: 0920-5691. DOI: 10.1007/s11263-006-7934-5. URL: http://dx.doi.org/10.1007/s11263-006-7934-5.

[Bou15] Jean-Yves Bouguet. Feb. 2015.

[Chr15] Henrik I Christensen. *SLAM Paper Repository*. 2015.

[CN10] Chunhui Cui and King Ngi Ngan. "Plane-based external camera calibration with accuracy measured by relative deflection angle". In: *Signal Processing: Image Communication* 25.3 (2010), pp. 224 –234. ISSN: 0923-5965. DOI: http://dx.doi.org/10.1016/j.image.2009.11.005. URL: http://www.sciencedirect.com/science/article/pii/S0923596509001350.

[Con15] Khronos Consortium. Mar. 2015.

[CP15] Neuralassembly Christopher Peplin. *Android Webcam Libary*. May 2015.

[CR04] Andrew Blake Carsten Rother Vladimir Kolmogorov. "Segmentation of depth image using graph cut". In: Aug. 2004.

[Dai+15] Longquan Dai et al. "Fast Minimax Path-Based Joint Depth Interpolation". In: *Signal Processing Letters, IEEE* 22.5 (May 2015), pp. 623–627. ISSN: 1070-9908. DOI: 10.1109/LSP.2014.2365527.

[DBC12] G. Danciu, S.M. Banu, and A. Caliman. "Shadow removal in depth images morphology-based for Kinect cameras". In: *System Theory, Control and Computing (ICSTCC), 2012 16th International Conference on*. Oct. 2012, pp. 1–6.

[Dru+15] Norbert Druml et al. "Time-of-Flight 3D Imaging for Mixed-Critical Systems". In: (July 2015).

[DT05] James Diebel and Sebastian Thrun. "An Application of Markov Random Fields to Range Sensing". In: *In NIPS*. MIT Press, 2005, pp. 291–298.

[FB81] Martin A. Fischler and Robert C. Bolles. "Random Sample Consensus: A Paradigm for Model Fitting with Applications to Image Analysis and Automated Cartography". In: *Commun. ACM* 24.6 (June 1981), pp. 381–395. ISSN: 0001-0782. DOI: 10.1145/358669.358692. URL: http://doi.acm.org/10.1145/358669.358692.

[FBK10] A. Frick, B. Bartczack, and R. Koch. "3D-TV LDV content generation with a hybrid ToF-multicamera RIG". In: *3DTV-Conference: The True Vision - Capture, Transmission and Display of 3D Video (3DTV-CON), 2010*. June 2010, pp. 1–4. DOI: 10.1109/3DTV.2010.5506604.

[FBS06] J. Fischer, D. Bartz, and W. Strasser. "Enhanced visual realism by incorporating camera image effects". In: *Mixed and Augmented Reality, 2006. ISMAR 2006. IEEE/ACM International Symposium on*. Oct. 2006, pp. 205–208. DOI: 10.1109/ISMAR.2006.297815.

[Fer+13] D. Ferstl et al. "Image Guided Depth Upsampling Using Anisotropic Total Generalized Variation". In: *Computer Vision (ICCV), 2013 IEEE International Conference on*. Dec. 2013, pp. 993–1000. DOI: 10.1109/ICCV.2013.127.

[FLM92] O.D. Faugeras, Q.-T. Luong, and S.J. Maybank. "Camera self-calibration: Theory and experiments". English. In: *Computer Vision — ECCV'92*. Ed. by G. Sandini. Vol. 588. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1992, pp. 321–334. ISBN: 978-3-540-55426-4. DOI: 10.1007/3-540-55426-2_37. URL: http://dx.doi.org/10.1007/3-540-55426-2_37.

[Fuc10] Stefan Fuchs. "Multipath Interference Compensation in Time-of-Flight Camera Images". In: *Pattern Recognition (ICPR), 2010 20th International Conference on*. Aug. 2010, pp. 3583–3586. DOI: 10.1109/ICPR.2010.874.

[GAL08] Sigurjon Arni Gudmundsson, Henrik Aanaes, and Rasmus Larsen. "Fusion of Stereo Vision and Time-Of-Flight Imaging for Improved 3D Estimation". In: *Int. J. Intell. Syst. Technol. Appl.* 5.3/4 (Nov. 2008), pp. 425–433. ISSN: 1740-8865. DOI: 10.1504/IJISTA.2008.021305. URL: http://dx.doi.org/10.1504/IJISTA.2008.021305.

[Gar15] Willow Garage. *OpenCV*. May 2015.

[Goo14] Google. *ATAP Project Tango – Google*. Feb. 2014. URL: http://www.google.com/atap/projecttango/.

[Han+13] Miles E. Hansard et al. *Time-of-Flight Cameras - Principles, Methods and Applications*. Springer Briefs in Computer Science. Springer, 2013, pp. I–X, 1–96. ISBN: 978-1-4471-4657-5.

[HCKH11] Daniel Herrera C., Juho Kannala, and Janne Heikkilä. "Accurate and Practical Calibration of a Depth and Color Camera Pair". English. In: *Computer Analysis of Images and Patterns*. Ed. by Pedro Real et al. Vol. 6855. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2011, pp. 437–445. ISBN: 978-3-642-23677-8. DOI: 10.1007/978-3-642-23678-5_52. URL: http://dx.doi.org/10.1007/978-3-642-23678-5_52.

[HFS07] B. Huhle, S. Fleck, and A. Schilling. "Integrating 3D Time-of-Flight Camera Data and High Resolution Images for 3DTV Applications". In: *3DTV Conference, 2007*. May 2007, pp. 1–4. DOI: 10.1109/3DTV.2007.4379472.

[HS07] H. Hirschmuller and D. Scharstein. "Evaluation of Cost Functions for Stereo Matching". In: *Computer Vision and Pattern Recognition, 2007. CVPR '07. IEEE Conference on*. June 2007, pp. 1–8. DOI: 10.1109/CVPR.2007.383248.

[Inc15] Intel Inc. Mar. 2015.

[Int15] Intel. *OpenCL™ Basic Tutorial for Android* OS*. 2015.

[Iza+11] Shahram Izadi et al. "KinectFusion: Real-time 3D Reconstruction and Interaction Using a Moving Depth Camera". In: *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology*. UIST '11. Santa Barbara, California, USA: ACM, 2011, pp. 559–568. ISBN: 978-1-4503-0716-1. DOI: 10.1145/2047196.2047270. URL: http://doi.acm.org/10.1145/2047196.2047270.

[JGM15] Alexander Porter James George and Jonathan Minard. Feb. 2015. URL: http://www.rgbdtoolkit.com.

[K.11] BOYDSTUN K. *Introduction OpenCL*. 2011.

[Kai+12] Bernhard Kainz et al. "OmniKinect: Real-time Dense Volumetric Data Acquisition and Applications". In: *Proceedings of the 18th ACM Symposium on Virtual Reality Software and Technology*. VRST '12. Toronto, Ontario, Canada: ACM, 2012, pp. 25–32. ISBN: 978-1-4503-1469-5. DOI: 10.1145/2407336.2407342. URL: http://doi.acm.org/10.1145/2407336.2407342.

[Kim+12] Wonjoo Kim et al. "A 1.5Mpixel RGBZ CMOS image sensor for simultaneous color and range image capture". In: *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2012 IEEE International*. Feb. 2012, pp. 392–394. DOI: 10.1109/ISSCC.2012.6177061.

[Kop+07] Johannes Kopf et al. "Joint Bilateral Upsampling". In: *ACM Trans. Graph.* 26.3 (July 2007). ISSN: 0730-0301. DOI: 10.1145/1276377.1276497. URL: http://doi.acm.org/10.1145/1276377.1276497.

[LHL11] B. Langmann, K. Hartmann, and O. Loffeld. "Comparison of Depth Super-Resolution Methods for 2D/3D Images". In: *International Journal of Computer Information Systems and Industrial Management Applications* 3 (2011), pp. 635 –645.

[Liu+13] Shaoguo Liu et al. "Kinect depth restoration via energy minimization with TV21 regularization". In: *Image Processing (ICIP), 2013 20th IEEE International Conference on*. Sept. 2013, pp. 724–724. DOI: 10.1109/ICIP.2013.6738149.

[LTT13] Ming-Yu Liu, O. Tuzel, and Y. Taguchi. "Joint Geodesic Upsampling of Depth Images". In: *Computer Vision and Pattern Recognition (CVPR), 2013 IEEE Conference on*. June 2013, pp. 169–176. DOI: 10.1109/CVPR.2013.29.

[Mat94] Tomomi Matsui. "The minimum spanning tree problem on a planar graph". In: *Discrete Applied Mathematics 58*. Mar. 1994.

[MGG12] Duane Merrill, Michael Garland, and Andrew Grimshaw. "Scalable gpu graph traversal". In: *In 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP'12*. 2012.

[ML] Mehdi Mekni and André Lemieux. "Augmented Reality: Applications, Challenges and Future Trends". In: ().

[Mö+05] Tobias Möller et al. "Robust 3D Measurement with PMD Sensors". In: *In: Proceedings of the 1st Range Imaging Research Day at ETH*. 2005, pp. 3–906467.

[Nai+13] Rahul Nair et al. "A Survey on Time-of-Flight Stereo Fusion". English. In: *Time-of-Flight and Depth Imaging. Sensors, Algorithms, and Applications*. Ed. by Marcin Grzegorzek et al. Vol. 8200. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, pp. 105–127. ISBN: 978-3-642-44963-5. DOI: 10.1007/978-3-642-44964-2_6. URL: http://dx.doi.org/10.1007/978-3-642-44964-2_6.

[Par+11] Jaesik Park et al. "High quality depth map upsampling for 3D-TOF cameras". In: *Computer Vision (ICCV), 2011 IEEE International Conference on*. Nov. 2011, pp. 1623–1630. DOI: 10.1109/ICCV.2011.6126423.

[Pet15] Peter. *Inverse Radial Distortion Formula*. 2015.

[Qua15] Inc Qualcomm. *Qualcomm Vuforia*. 2015.

[RH07] Thorsten Ringbeck and Bianca Hagebeuker. "A 3D time of flight camera of object detection". In: *Optical 3-D measurement techniques VIII : applications in GIS, mapping, manufacturing, quality control, robotics, navigation, mobile mapping, medical imaging, cultural heritage, VR generation and animation; papers presented to the conference organized at ETH Zurich, Switzerland, July 9 - 12, 2007. - Vol. 1*. Ed. by Armin Grün. 2007, pp. 1 –16.

[RK14] Jonathan Ragan-Kelley. "Decoupling Algorithms from the Organization of Computation for High Performance Image Processing". Ph.D. Thesis. Cambridge, MA: Massachusetts Institute of Technology, June 2014. URL: http://groups.csail.mit.edu/commit/papers/2014/jrkthesis.pdf.

[RT12] Takashi Nakamura Ryoji Tsuchiyama. *The OpenCL Programming Book*. Apr. 2012.

[SS02] Daniel Scharstein and Richard Szeliski. "A Taxonomy and Evaluation of Dense Two-Frame Stereo Correspondence Algorithms". In: *Int. J. Comput. Vision* 47.1-3 (Apr. 2002), pp. 7–42. ISSN: 0920-5691. DOI: 10.1023/A:1014573219977. URL: http://dx.doi.org/10.1023/A:1014573219977.

[Tec14] PMD Technologies. 2014.

[Tec15] Unity Technologies. *Unity*. 2015.

[TF03] A. Torralba and W.T. Freeman. "Properties and applications of shape recipes". In: *Computer Vision and Pattern Recognition, 2003. Proceedings. 2003 IEEE Computer Society Conference on*. Vol. 2. June 2003, II–383–90 vol.2. DOI: 10.1109/CVPR.2003.1211494.

[TM98] C. Tomasi and R. Manduchi. "Bilateral filtering for gray and color images". In: *Computer Vision, 1998. Sixth International Conference on*. Jan. 1998, pp. 839–846. DOI: 10.1109/ICCV.1998.710815.

[Yan+07] Qingxiong Yang et al. "Spatial-Depth Super Resolution for Range Images". In: *Computer Vision and Pattern Recognition, 2007. CVPR '07. IEEE Conference on*. June 2007, pp. 1–8. DOI: 10.1109/CVPR.2007.383211.

[YZ12] Jiangming Yu and Jieyu Zhao. "Segmentation of depth image using graph cut". In: *Fuzzy Systems and Knowledge Discovery (FSKD), 2012 9th International Conference on*. May 2012, pp. 1934–1938. DOI: 10.1109/FSKD.2012.6234121.

[Zhu+11] Jiejie Zhu et al. "Reliability Fusion of Time-of-Flight Depth and Stereo Geometry for High Quality Depth Maps". In: *Pattern Analysis and Machine Intelligence, IEEE Transactions on* 33.7 (July 2011), pp. 1400–1414. ISSN: 0162-8828. DOI: 10.1109/TPAMI.2010.172.