# Efficient Range and Image Data Processing – Algorithms and Software Paradigms

# Effiziente Verarbeitung von Tiefen- und Bilddaten – Algorithmen und Software-Paradigmen

DISSERTATION

zur Erlangung des Grades eines Doktors
der Ingenieurswissenschaften (Dr.-Ing.)

vorgelegt von
M.Sc. Dipl. Ing. (FH) Thomas Högg

eingereicht bei der Naturwissenschaftlich-Technischen Fakultät
der Universität Siegen
Siegen 2017

1. Gutachter: Prof. Dr. Andreas Kolb, Universität Siegen
2. Gutachter: Prof. Dr. Marc Stamminger, Friedrich-Alexander-Universität, Erlangen-Nürnberg
Vorsitzender: Prof. Dr. Udo Kelter, Universität Siegen

Tag der mündlichen Prüfung: 18.10.2017.

Printed on non-aging wood- and acid-free paper.
Gedruckt auf alterungsbeständigem holz- und säurefreiem Papier.

# Eidesstattliche Erklärung

Die vorliegende Dissertation wurde von mir selbständig angefertigt. Die verwendeten Hilfsmittel und Quellen sind im Literaturverzeichnis vollständig aufgeführt. Eingetragene Warenzeichen und Copyrights werden anerkannt, auch wenn sie nicht explizit gekennzeichnet sind.

Lautrach, den 01.12.2017

Thomas Högg

# Abstract

The growing demand towards industrial automation and autonomous systems requires more flexible technologies in different but interdependent domains of engineering. This thesis introduces and discusses two important areas: Time-of-Flight (ToF) camera data improvement (algorithm development) and related model driven engineering techniques (software development). In the last decades, these areas have been extensively studied and important progress was made.

The first part of this thesis discusses the special challenges of data quality improvement on a very deep layer of ToF cameras. It deals with different challenges related to the working principle of this kind of sensor. A new method for a fast motion artifact compensation for ToF cameras is presented. It is shown that the algorithm gives good results for simulated as well as real data while providing real-time performance. The second proposed algorithm deals with the automatic integration time estimation of ToF cameras. An online integration time adaption algorithm that works on a per-pixel basis and uses knowledge gained from an extensive analysis of the underlying inherent sensor behavior is introduced. Finally, an industrial real-time 3D car reconstruction example is presented. It shows how the data of three PMD (Photonic Mixer Device) cameras has to be preprocessed and combined, using an extensive depth data processing and filtering pipeline.

The second part of this thesis addresses the challenges of data related model driven software engineering. It introduces and contributes the new domain specific language GU-DSL and two data and image processing related extensions: GPGPU (General Purpose Computation on Graphics Processing Unit)-programming and CBSE (Component-Based Software Engineering) principles. The presented GU-DSL GPGPU extension contributes a convenient combination and mixture of textual and graphical model- and dataflow-driven design. Using a code generator, GU-DSL code can be transformed into C++, compiled and executed. All the GPU related features are encapsulated into a C++ Heterogeneous Computing framework. The GU-DSL CBSE system introduces a concept for component based software engineering in the domain of data- and image-processing. It proposes several new concepts for component- and component-instance-diagrams in combination with class- and activity-diagrams. Using a newly developed Rich Client Platform supporting a plugin based extension system, it shows how the GU-DSL CBSE concept can be realized and used in practice using C++. Exemplary, a simple processing pipeline is implemented to demonstrate the new concepts.

# Kurzfassung

Die wachsende Nachfrage nach industrieller Automatisierung und autonomen Systemen erfordert flexiblere Techniken in verschiedenen, aber stark voneinander abhängigen Bereichen des Ingenieurwesens. Die vorliegende Arbeit beschäftigt sich mit zwei wichtigen Bereichen: Verbesserung der Time-of-Flight (ToF) Kameradaten (Algorithmen-Entwicklung) und den damit verbundenen modellgetriebene Engineering Techniken (Software-Entwicklung). In den letzten Jahrzehnten wurde ausgiebig in diesen Bereichen geforscht und es konnten zahlreiche Fortschritte erzielt werden.

Der erste Teil dieser Arbeit beschäftigt sich mit den besonderen Herausforderungen der Datenqualitätsverbesserung von Time-of-Flight (ToF) Kameras. Es wird ein neues Verfahren zur schnellen Kompensation von Bewegungsartefakten von ToF-Kameras vorgestellt und gezeigt, dass der Algorithmus gute Ergebnisse für simulierte sowie reale Daten in Echtzeitausführung erzielt. Der zweite vorgestellte Algorithmus befasst sich mit der automatischen Anpassung der Integrationszeit von ToF Kameras. Es wird ein Algorithmus entwickelt, mit dessen Hilfe die Integrationszeit des Sensors auf einer „pro Pixel"-Basis im Live-Betrieb automatisch angepasst werden kann. Der erste Teil der Dissertation schließt mit einem Industrie-Beispiel ab. Dabei wird gezeigt, wie Fahrzeuge in Echtzeit unter der Verwendung von drei PMD (Photonic Mixer Device) Kameras nach umfangreicher Datenvorverarbeitung in 3D rekonstruiert werden können.

Der zweite Teil dieser Arbeit befasst sich mit den Herausforderungen der modellgetriebenen Softwareentwicklung im Bereich Bild- und Datenverarbeitung. Im Rahmen dieser Arbeit wird die domänenspezifische GU-DSL mit zwei Daten- und Bildverarbeitung relevanten Erweiterungen entwickelt: GPGPU-Programmierung und GPGPU (General Purpose Computation on Graphics Processing Unit)-Programmierung und Component Based Software Engineering (CBSE). Die GPGPU Erweiterung verwendet dazu eine Kombination aus textueller und grafischer Modellierung. Die Entwicklung findet dabei datenfluss- und modellgetrieben statt. Mittels Code-Generator, kann der GU-DSL Code in C++ transformiert, anschließend kompiliert und ausgeführt werden. Alle GPU relevanten Funktionen sind dazu in ein C++ Heterogeneous Computing Framework gekapselt. Das GU-DSL CBSE System ist ein Konzept zum komponentenbasierten Software Engineering. Es werden neue Ansätze für Komponenten- und Komponenten-Instanz-Diagramme in Kombination mit Klassen- und Aktivitätsdiagrammen vorgeschlagen und umgesetzt. Mit Hilfe einer plugin-basierten Rich Client Platform, wird exemplarisch demonstriert wie diese neuen Konzepte in C++ umgesetzt werden können. Abschließend wird eine vereinfachte Verarbeitungspipeline implementiert, um die neuen Konzepte zu evaluieren.

# Contents

# 1 Introduction

The requirements on software systems have risen dramatically in the last years. Especially the usage of complex software architectures for technical applications, such as image processing or similar (sensor-) data processing tasks, has heavily increased due to cheaper and more powerful hardware.

In the domain of distance data measurement, significant advances have been made. An example are Time-of-Flight (ToF) sensors such as the Photonic Mixer Device (PMD) camera. Offering a cheap and elegant way to measure depth/distance data, they become more and more important for the computer vision and graphics domain and also for industrial applications. Having advantages, such as high performance and no mechanical overhead compared to e.g. laser scanners, they also have many problems, especially in accuracy and noise behavior. Most of these errors can be corrected well by applying good calibration models and pre-filtering (e.g. low-pass filtering). However, artifacts arising from dynamic scenes are still not resolved satisfactorily. Moving objects in scenes cause a blur effect (motion artifacts) in acquired depth images. A fast movement leads to strong artifacts, related to the sensor's working principle which is based on the sequential acquisition of four so-called phase images in order to generate a depth map. Artifacts occur in areas where corresponding phase image values do not refer to the same object position. Noise and motion result in an incorrect distance calculation, which is why ToF data denoising has always been an important discipline in depth data processing.

Several techniques have been established during the last years. Methods for outlier removal and outlier correction have been developed to improve acquisition quality of noisy data. Denoising and optimization can be applied at different stages: at image acquisition level and/or during data processing. Previous works have shown that combining both optimization stages gives the best result. All these techniques require fast processing. This can be achieved by using Graphic Processing Unit (GPU)s and related frameworks like Nvidia Compute Unified Device Architecture (CUDA) [NVI16] or the more generic and standardized Open Computing Language (OpenCL) [Khr16]. However, increasing data processing performance via parallelization of tasks leads to an increase of programming complexity and reduction of maintainability. This kind of programming has special requirements due to the parallelization of tasks. Hence, there must be special control structures on the one hand and simple ways to allow parallel execution of operations on the other hand. Therefore, such systems are usually split into two parts: a host (e.g. a PC) and multiple devices (e.g. a graphics card) which is the case for OpenCL and CUDA. Executable device programs (kernel) are based on a version of the International Organization for Standardization (ISO) C99 standard, extended by types and functions. Calculations are performed by work-items arranged in work-grids. Data always has to be transferred between host memory and device kernel memory, which can lead to big bottlenecks if this process is performed too often. This means the developer also has to decide when and how often the memory synchronization is performed.

Besides hardware improvements, there were also important advances in the domain of software engineering. The topic of rising complexity is a general problem in software engineering. Several methods have been introduced to overcome this during the last years. As example the Unified Modeling Language (UML) was introduced by the Object Management Group (OMG) and accepted by the ISO (International Organization for Standardization) as a standard in 2000. It allows to model high-level abstractions of real-world problems by using graphical descriptions. Different kinds of diagrams such as class-, activity- or state-machine diagrams are the basis of modeling the system's structure and functionality. The usage of UML can improve the general software quality and can reduce the Time-to-Market. Another way to improve software development are Domain Specific Languages (DSLs). They have become more and more important in the past years due to tools simplifying the language development, for example xText [EEK$^+$16] (textual modeling) and the Graphical Modeling Framework (GMF) [EC16] (graphical modeling). DSLs can be divided into two types: external (completely new and independent language) and internal (using a host language such as C)[EEK$^+$12]. Both kinds of DSLs improve the readability and the formulation of domain-specific, often real world problems.

While UML can be extended by profiles (for e.g. adding new types and/or model elements), DSLs provide the possibility to even start the formulation of problems from scratch, which means writing a completely new language.

## Problem Statement

Summarizing the problems stated in the previous sections, several interesting challenges arise in the two different but closely connected domains of ToF data denoising and data processing that should be mentioned here. Common issues, that virtually all active optical systems encounter, include

**Low reflective objects** result in low optical signals which reduce the Signal-to-Noise Ratio (SNR), thus degrading the measurement reliability.

**Highly specular surfaces** may lead to oversaturation effects, which severely alter the signal values resulting in wrong distance measurements.

Further ToF specific issues, that need to be taken into account, include

**Systematic intensity-related distance errors,** i.e. the mean (and not only the variance) of the measured distance is influenced by the total amount of incident light (Sec. 2.1.2.2).

**Systematic distance (wiggling) errors** occur because the theoretically required sinusoidal signal is not achievable in practice (Sec. 2.1.2.2).

**Flying pixels** occur in spatial regions with inhomogeneous depth; here superimposed phase shifts result in mixed signals leading to wrong distance values (see Sec. 6.5.1.3).

**Motion artifacts:** In case of motion, individual pixels no longer relate to the same object point during the successive acquisition of the four phase images $P_i$; this leads to motion artifacts at object boundaries and in regions of inhomogeneous reflection (see Sec. 6.5.1.1).

**Lateral intensity attenuation** in the image plane results from the non-uniform illumination of the scene.

The stated points bring up several interesting and complex challenges. A very general problem is the usage of ToF cameras in combination with Computer Vision based image registration algorithms. Firstly, ToF cameras have a comparably low resolution. Secondly, the data characteristics are very different from standard 2D grayscale or RGB-cameras. For example, the PMD-intensity data is not a true gray data, it is a measure of the amount of reflected light and depends on the infrared reflectivity of the reflecting material.

Besides the stated ToF problems, there are also important and significant challenges in the domain of Model Driven Engineering, e.g. when modeling the ToF-denoising algorithms. Graphical modeling and also external DSLs are unfortunately hardly used in the domain of image processing and computer vision. The community mainly wants to concentrate on the development of algorithms. But in most cases they have to begin their work from scratch and start with the development of GUIs and the abstraction of data and image processing interfaces. It is widely known that GUI development can rapidly grow to become a long and error-prone task, which often leads to loss of focus. Having the possibility of using a specially adapted development environment and toolchain like the one proposed in this thesis, the focus can be shifted back to the initial scope of algorithm development.

## Contributions

The introduction has shown the most important and significant challenges in the domain of Depth Data Processing and Software Engineering. Thus, the corresponding specific contributions of this thesis are twofold and divided as follows.

### Depth Data Processing Contributions

1. Compensation of Motion Artifacts
   - A new algorithm to perform a fast real-time motion compensation with high frame rates (above 50 FPS). The focus lies on high flexibility to allow the algorithm to be either computed in parallel on a GPU using CUDA or to simply be ported to small devices like a Field Programmable Gate Array (FPGA) preprocessing platform. [HLK13a, LHK13]
2. Automatic Integration Time Estimation
   - A new approach to automatically determine the best integration time for arbitrary scenes using the knowledge of underlying inherent sensor behavior and properties. The approach benefits from a detailed sensor data analysis and integrates this knowledge into a novel algorithm that is more flexible and stable than a proportional feedback control system, especially in unknown, arbitrary scenes. [HBK15]
3. A Complex Depth Data Processing Pipeline
   - A novel approach for online acquisition and reconstruction of a vehicle's outer hull. The key features of the system are the integration of three active range sensing ToF cameras based on the PMD principle, an appropriate preprocessing of the sensor data, registration, data fusion and geometry extraction. All data processing is done on GPUs to minimize reconstruction time. [HLK13b]

### Data Processing Related Software Engineering Contributions

1. GU-DSL
   - A novel Domain Specific Language, specially designed for data and image processing tasks. [HFKK15]
2. Model Driven General Purpose Computation on Graphics Processing Unit (GPGPU) Programming

- Special GPGPU programming related GU-DSL textual and graphical language features to simplify and improve the error prone task of GPU algorithm development. [HFKK16]

3. Component-Based Data And Image Processing Architectures

    - A special extension of GU-DSL allowing for component-based textual and graphical modeling of data and image processing algorithms and systems [HKK16]
    - A novel Architecture Definition Language (ADL) to simplify architecture definitions [HKK16]
    - A C++ Component-Based Software Engineering (CBSE) system to provide a tested default runtime environment for data and image processing systems and to show an exemplary implementation of a system that can be developed with GU-DSL. [HKK15]

## Outline

Initially, this thesis gives an overview of depth measurement and related software engineering problems in *Chapter 1*. *Chapter 2* provides the necessary fundamentals to understand the parts of depth data processing and the related software engineering techniques.

The structure of Part I consists of the following chapters:

*Chapter 3* gives an introduction to ToF related problems.

*Chapter 4* shows how ToF motion artifacts can be reduced using a novel algorithm that performs fast real-time motion compensation with high frame rates, especially on GPUs or FPGAs.

*Chapter 5* introduces a new approach to automatically determine the best ToF integration time for arbitrary scenes.

*Chapter 6* shows a novel approach for online acquisition and reconstruction of a vehicle's outer hull using three ToF cameras in an industrial example application.

*Chapter 7* summarizes the results of the previous depth data processing chapters and concludes Part I.

The structure of Part II consists of the following chapters:

*Chapter 8* gives an introduction to data processing related software engineering problems.

*Chapter 9* introduces GU-DSL, a generic DSL for textual and graphical modeling of data and image processing problems.

*Chapter 10* shows how three different image processing algorithms can be implemented for GPGPU processing using GU-DSL.

*Chapter 11* introduces a new component-based data and image processing approach using GU-DSL.

*Chapter 12* summarizes the results of the previous model driven engineering chapters and concludes Part II.

The last chapter (*Chapter 13*) of this thesis concludes this work with a summary and conclusion.

# 2 Fundamentals

This chapter introduces the most important and relevant fundamentals of the ToF technology, such as working principle, calibration, error analysis and data processing. Additionally, it also shortly explains Domain Specific Languages.

## 2.1 Time-of-Flight Cameras

The following sections give a brief introduction to the functionality of PMD ToF cameras. It gives a basic overview of the working principle, effects and also parameters.

### 2.1.1 Time-of-Flight Principle

ToF cameras, such as the *PMD*, estimate distances from each pixel to related points on a target using the phase shift of intensity modulated, incoherent infrared (IR) light, induced by the time of flight. The scene is irradiated by the camera's illumination unit with a modulated optical signal, generated by a modulator with modulation frequency $f$, which is then reflected off the target back to the sensor. Each pixel correlates the incoming optical signal $s(t)$ with the reference signal $r(t)$.

Several phase delays are acquired in order to demodulate the correlation function and thus retrieve the distance-related phase shift. Theoretically, three phase delay images are necessary to demodulate the correlation function, however phase shift ToF devices usually sample the phase images four times, in order to keep a good Signal-to-Noise Ratio. Moreover, PMD-based devices acquire two phase images $P_i^A, P_i^B$ at one point in time, where in case of perfect pixel behavior $P_i^B = P_{(i+2) \mod 4}^A$. Internally, the camera device proceeds with $P_i = P_i^A - P_i^B$ in order to suppress hardware inhomogeneities [LNL+13].

Based on the four phase images $P_{i=0-3}$, the phase shift $\phi$, the amplitude $A$ and the intensity $I$ can be retrieved:

$$\begin{aligned}\phi &= \operatorname{atan2}(P_3 - P_1, P_0 - P_2)\\ I &= \frac{P_0 + P_1 + P_2 + P_3}{4}\\ A &= \frac{1}{2} \cdot \sqrt{(P_3 - P_1)^2 + (P_0 - P_2)^2}.\end{aligned}$$

The resulting distance $D$ is then calculated using the angular modulation frequency $\omega$ and the speed of light $c \approx 3 \cdot 10^8 ms^{-1}$:

$$D = \frac{c}{2\omega}\phi.$$

ToF cameras have a limiting range measurement capability which is called unambiguous distance range $\delta$. This is directly related to the modulation frequency:

$$\delta = \frac{c}{2f}.$$

An additional concept is the Suppression of Background Illumination (SBI) of the PMD-cameras. It is a mechanism to enlarge the ratio between the distance carrying signal and the uncorrelated extraneous light. The same amount of unnecessary charge carriers within pixel channels A and B (e.g. the symmetric steady component in both channels) are removed what of course affects the signal behavior. As a result, the cameras can be used e.g. in direct sun light. The PMD CamCube and its corresponding Software Development Kit (SDK) indicate that the SBI became active by a flag. Note: the presented measurements and algorithms in this thesis don't consider the SBI.

### 2.1.2 Time-of-Flight Calibration

ToF calibration is one of the most important preprocessing steps and is usually done once at setup configuration. Two different calibration processes will be presented in this section. One to deal with usual calibration problems such as intrinsic parameters and extrinsic camera position, the other covers the depth measurement provided by a ToF camera.

#### 2.1.2.1 Intrinsic/Extrinsic Parameter Estimation

Precise intrinsic parameters and extrinsic camera positions are crucial for applications to achieve correct spatial data fusion. Using an additional high-resolution RGB camera can drastically improve the estimation of intrinsic (focal length, principle point, lens distortion) and extrinsic (position, orientation) parameters of low resolution ToF devices, as shown by Schiller [SBK08]. Providing a free Multi-Camera Calibration software [MIPG16] which uses a full calibration model of a ToF camera, it is possible to determine intrinsic and also extrinsic camera parameters. Furthermore, it also provides ToF depth correction function parameters which can be used to overcome bias errors and also the so called wiggling error. Experiments for the estimation of the intrinsic and extrinsic parameters can be found in Sec. 2.2.2 and Sec. 2.2.3.

#### 2.1.2.2 Time-of-Flight Distance Correction

Systematic distance (wiggling) errors occur because the theoretically required sinusoidal signal is not achievable in practice. The wiggling error is corrected using the PMD CamCube SDK (detailed information about wiggling correction can be found in [LSKK10]), since the system presented in Chapter 6 uses three PMD cameras with different modulation frequencies and different integration times. It provides a precise correction of the wiggling error for each camera configuration. Another error source related to ToF devices is the intensity-related distance error [LSKK10], which is of great interest, since the application is dealing with arbitrary cars (e.g. different color reflectivities). The basic idea is to use a complex acquisition protocol utilizing the different checkerboard reflectivities in the test rig. A high-resolution RGB camera is mounted onto the ToF device allowing the estimation of "ground-truth" distances thanks to the RGB checkerboard size. This is possible by the usage of the intrinsic parameters and the Field Of View (FOV) of the RGB camera and the exact known size of the checkerboard. See Fig. 2.1 for a comprehensive description of the influence of measured intensity on the distance error. For more details, see [LSKK10] and Sec. 2.2.1.

The method described in [LSKK10] has been extended in [HLK13b] in order to correct the intensity related depth error of the ToF device using a different wiggling correction method. Since the measured intensity of a ToF device is linearly dependent on its integration time (see Sec. 2.2.1, Fig. 2.2), the final distance correction is extracted in relation to the integration time:

$$D_c^p = D_m^p - \delta_{calib}\left(I_m^{p,intT} \cdot \frac{intT_{calib}}{intT}\right),$$

where

- $D_c^p$ is the corrected distance of pixel p
- $D_m^p$ the initial measured distance of pixel p with wiggling correction
- $intT$ the current integration time
- $intT_{calib}$ the integration time used in the calibration process
- $I_m^{p,intT}$ the measured intensity of pixel p using the integration time $intT$
- $\delta_{calib}$ intensity-related distance function modeled as a quadratic polynomial during the calibration process. Note that this function is computed using the measured distance of each pixel to its corresponding ground-truth.

Detailed experiments and results can be found in Sec. 2.2.1.

## 2.2 Time-Of-Flight Camera Error-Analysis and Parameter Estimation

This section describes the main experiments performed to setup the physical and data processing system. It provides a detailed description as to how the necessary parameters, such as the camera position and also the system thresholds, are determined.

### 2.2.1 Systematic Error Correction

As mentioned in Sec. 2.1.2, systematic errors like the wiggling error or intensity related distance errors have to be corrected. Since the PMD camera SDK already provides a good wiggling correction, the focus is the optimization of bias and intensity related errors. The necessary experiments are described in Sec. 2.1.2.

In order to adjust the PMD distance based on the intensity, a precise acquisition should be operated. Here different checkerboards with different reflectivities (100 % dark, 80 %, 60 %, 40 %, 20 %, white) are used to describe the complete intensity range. Each pattern was fixed on a plane wall and several acquisitions were done between 1 and 3 meters. The camera system, composed of an RGB (resolution 1280x960px, focal length 12mm) and a PMD camera (resolution 200x200px, focal length 8mm), was moved approximately 5cm between each measurement which leads to roughly 40 different Cartesian distances acquired. Note that the camera's movements does not need to be precise since the reference distance has been computed precisely using the 2D cameras and the checkerboard geometry. For each Cartesian distance acquisition, two integration times were used (250, 500 μs).

Fig. 2.1 shows the distance error in relation to different light-absorption patterns. The main difficulty of a reliable PMD intensity calibration is, to retrieve precise reference distance measurement in a first step. A high-resolution RGB camera was mounted and strongly fixed to the top of the PMD camera. Using the MIP Multi-Camera Calibration tool [MIPG16], the relative transformation between the PMD camera and the color camera has been computed as well as both intrinsic parameter sets. The extrinsic parameters of the color camera have been computed using its intrinsic parameters and the checkerboard's 2D corner points.

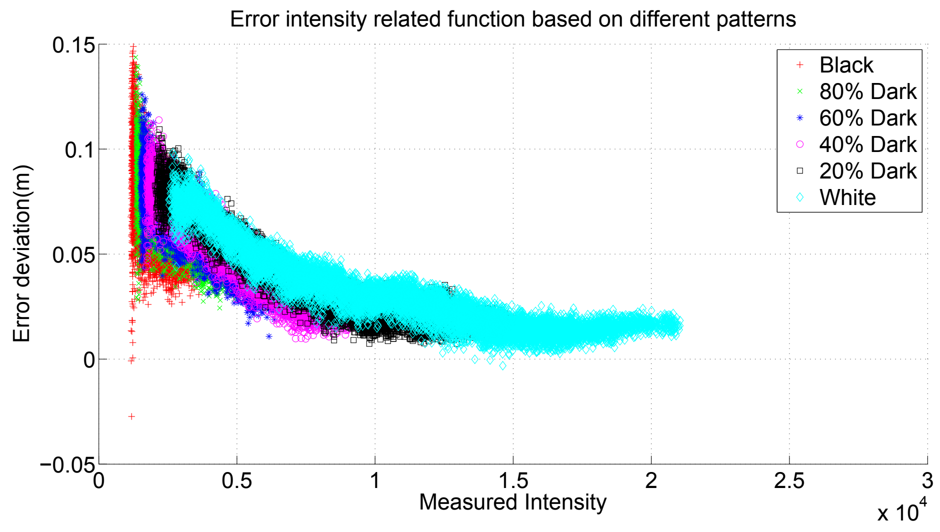


Figure 2.1: Error distance measurement of a PMD camera in relation to the measured intensity (using six light-absorption patterns). [HLK13b]

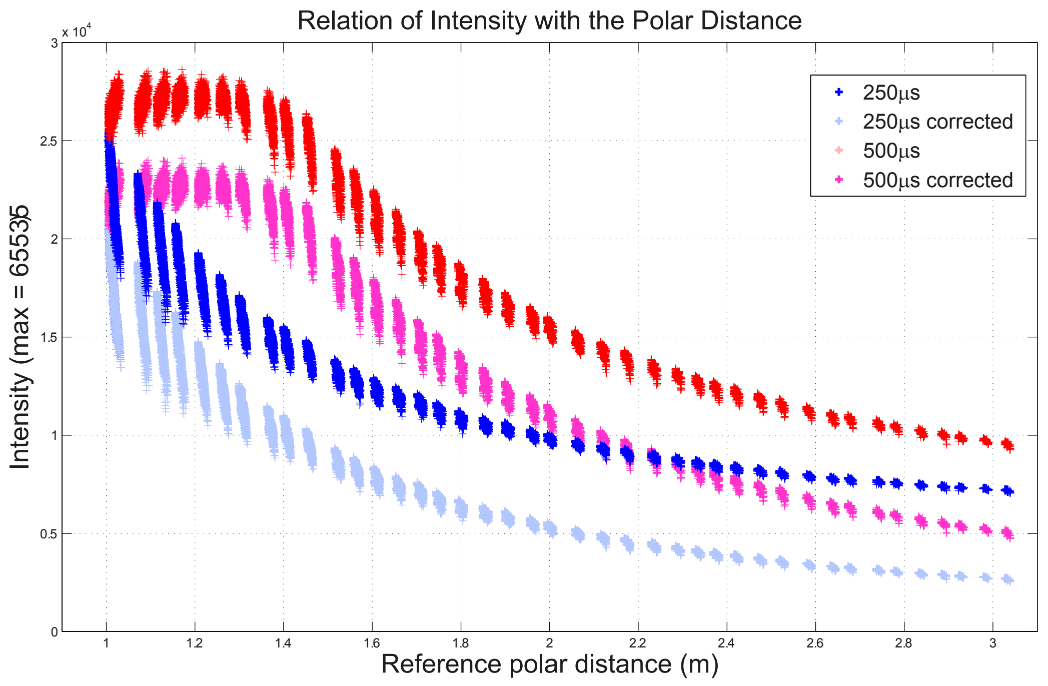


Figure 2.2: Red and blue plot: distance-dependent intensity function (white pattern); Magenta and light blue plot: distance-dependent intensity function offset corrected using the black intensity PMD image (white pattern). [HLK13b]

Fig. 2.2 shows the PMD intensity response (red and blue plot) of the bright white pattern in relation to the polar distance reference. Each set of points is described with its mean error and the corresponding standard deviation. It shows two different plots, one using 250 µs integration time (in blue) and the other 500 µs (in red). Note the saturation effects (nearly constant intensity) for the distance acquisition with less than 1.4 m and an integration time of 500 µs. This graph directly shows that, even by using a twice the integration time, the intensity is not doubled. This is due to an intensity offset in the PMD intensity measurement which can be easily corrected using a "black image" PMD acquisition (see Fig. 2.2, magenta and light blue plot). It illustrates the corrected intensity by subtracting the offset intensity from the black image. It clearly shows the **linear** relation of the intensity to the integration time. The same relation can be seen using a completely dark pattern (100 % black, see Fig. 2.3).

Fig. 2.4 shows the intensity-related distance correction for a low reflectivity pattern. Each red error bar (mean and standard deviation of all measured distance pixels) represents a specific reference distance acquisition. Note the high uncertainty for distances greater than 2.5 m using a low reflectivity object. It can be seen that applying the distance correction reduces the overall error.

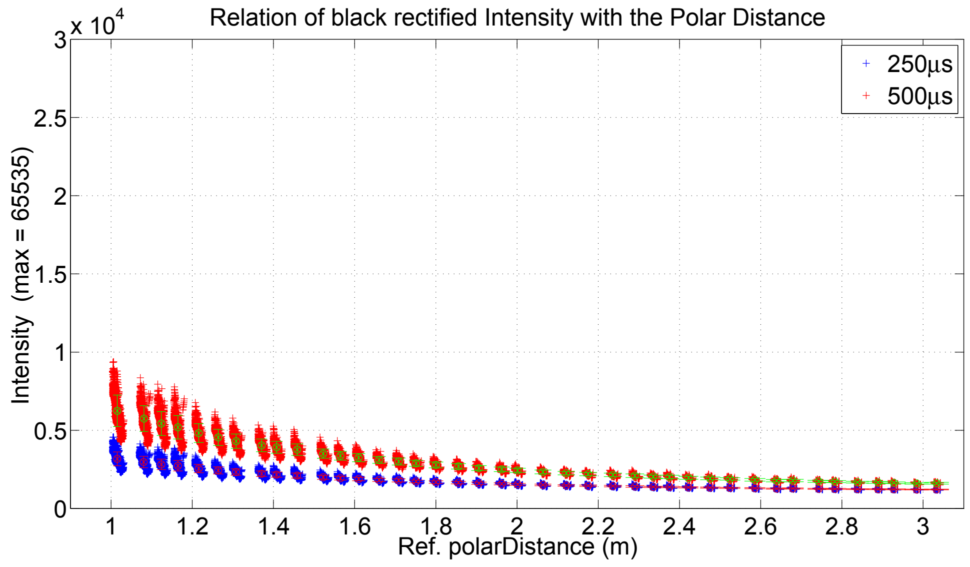


Figure 2.3: Distance-dependent intensity function corrected using the black intensity PMD image (100 % dark pattern). [HLK13b]

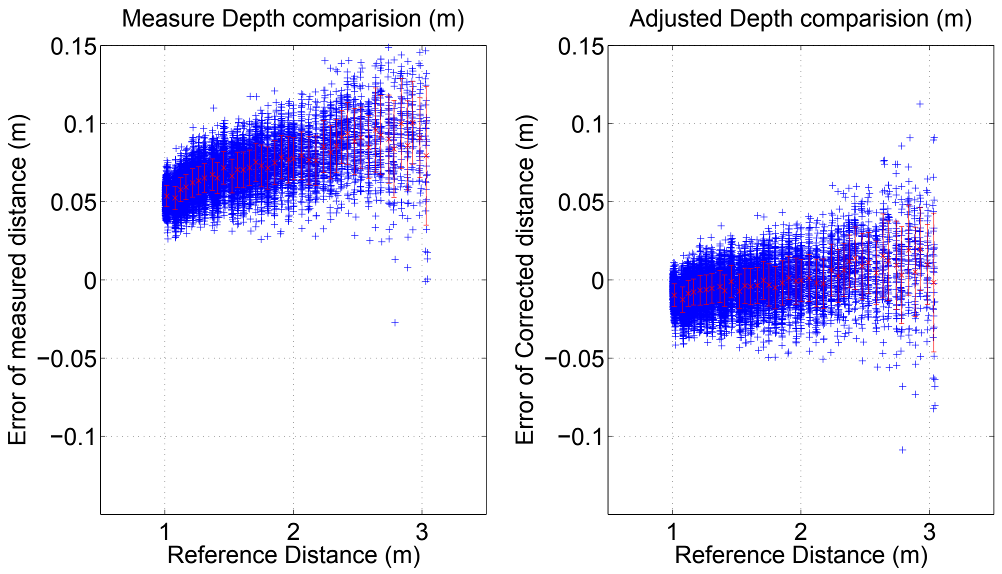


Figure 2.4: Left: intensity-related distance error using a black pattern (for measurement between 1 and 3 meters). Right: corresponding distance error after correction using $\delta_{calib}$. [HLK13b]

### 2.2.2 Intrinsic Calibration

The intrinsic calibration is the process of estimating camera parameters describing the lens distortion, focal length and the principle point. Knowing these parameters allows for lens undistortion and the conversion from ToF polar into Cartesian distances. The calibration is done using the tool from Schiller [MIPG16]. Detailed information about the used algorithm can be found in [SBK08].

The input for the calibration method is a sequence of checkerboard images, all taken from different poses. The goal is to cover the whole sensor area as good as possible. This allows for a good correction of the distortion, especially near image borders where the degree of distortion is the highest.

| Parameter | Value |
|---|---|
| Image width | 200 px |
| Image height | 200 px |
| Focal length X | 176.14 px |
| Focal length Y | 176.70 px |
| Center X | 103.83 px |
| Center Y | 99.06 px |
| Radial distortion X | -0.295140 |
| Radial distortion Y | 1.191152 |
| Tangential distortion X | 0.004619 |
| Tangential distortion Y | -0.001630 |

Table 2.1: Exemplary intrinsic calibration results of a PMD camera. [HLK13b]

Table 2.1 shows the intrinsic calibration result for the PMD camera used with 70° FOV lens.

### 2.2.3 Extrinsic Calibration

The extrinsic parameters, i.e. the transformation between the two cameras, are estimated similarly to the intrinsic data. An image sequence with different poses of the checkerboard is acquired. The only difference is that the checkerboard always has to be visible to the top mounted RGB camera (used as reference camera [SBK08]) and the camera, for which the relative transformation to the RGB reference should be determined.

The RGB camera provides the reference coordinate system for the whole setup, which means that all point clouds are transformed into this coordinate system.

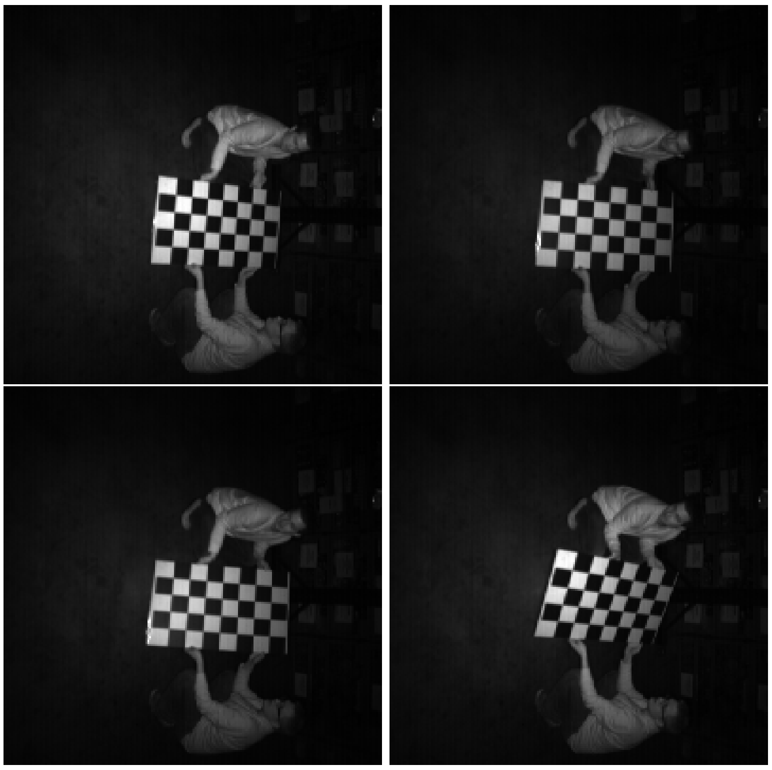


Figure 2.5: An exemplary partial sequence of checkerboard images to cover the whole sensor area for the extrinsic calibration.[HLK13b]

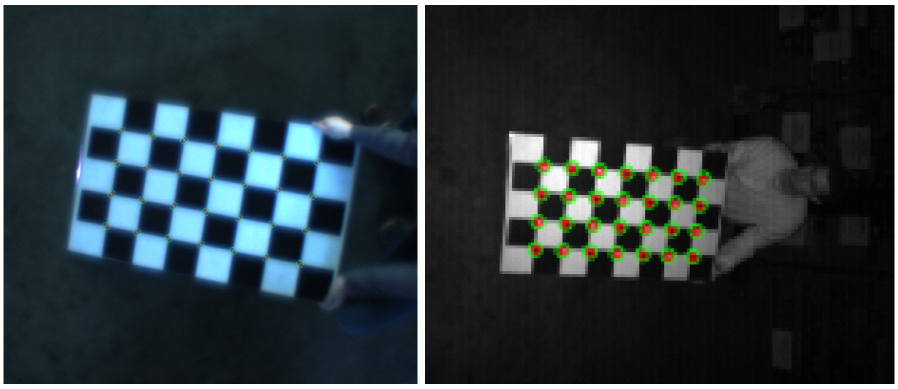


Figure 2.6: Reprojection results after applying the estimated transformation between the RGB (left image) and the PMD (right image) camera. [HLK13b]

The green circles in Fig. 2.6 represent the detected corners in the RGB image, the red circles show the corners of the PMD image. It can be seen that the transformation estimation between the RGB and PMD camera works well. The remaining reprojection error over the whole image sequence (see Fig. 2.5) can be seen in Table 2.2.

| Camera Type | Value |
|---|---|
| RGB camera | 0.717589 px |
| PMD camera | 0.356076 px |

Table 2.2: Reprojection error of a PMD camera. [HLK13b]

## 2.3 Iterative Closest Point Algorithm

The Iterative Closest Point (ICP) algorithm is widely used for alignment of 3D data. It is able to minimize distances between corresponding points of two different datasets by iteratively refining a 3D rigid body transformation, which transforms the source data towards the destination data. It was introduced by Besl and McKay [BM92], who established correspondences by pairing closest points from the two datasets. Then, "the sum of the squared distance between points in each correspondence pair is minimized" [Low04a]. This is also known as *point-to-point error metric*.

Independently, Chen and Medioni [CM92] proposed a similar approach, where the sum of squared distances from each source point to the tangent plane at the destination point is minimized. This is also known as *point-to-plane error metric*. The usage of the algorithm can be seen in the industrial 3D reconstruction application shown in Chapter 6.

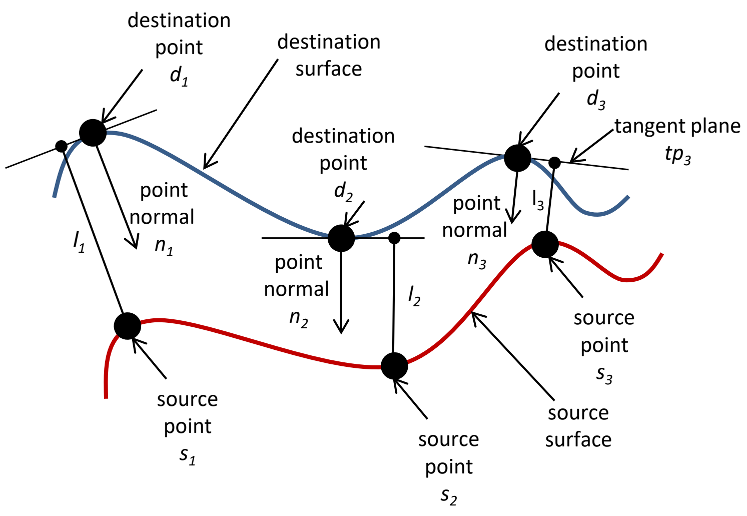


Figure 2.7: Point-to-plane metric between two surfaces. Red: source surface, blue: destination surface. $s_i$ and $d_i$, $i \in \mathbb{N}$, indicate sample points of the source and destination surface, $n_i$ the normals at the destination points, $tp_i$ the tangent planes and $l_i$ the distances of the source points to their corresponding planes as proposed by Low [Low04a, HLK13b]

### 2.3.1 Point-to-Point Error Metric

The point-to-point metric tries to minimize the sum of squared distances between corresponding points ($s_i$,$d_i$). The minimization problem can be formulated with the following equation:

$$E = \sum_i (Rs_i + t - d_i)^2$$

where $s_i$ is a source point, $d_i$ the corresponding destination point, $R$ the required rotation matrix and $t$ the required translation vector.

This can be solved using standard linear least-squares methods, such as the singular value decomposition, as "for an error metric of this form, there exist closed form solutions" [RL01].

### 2.3.2 Point-to-Plane Error Metric

The point-to-plane metric tries to minimize the sum of squared distances from each source point to the tangent plane at the destination point. The minimization problem can be formulated with the following equation:

$$E = \sum_i ((Ms_i - d_i) \bullet n_i)^2$$

where $s_i$ is a source point, $d_i$ the corresponding destination point, $n_i$ the unit normal vector at $d_i$ and $M$ the required transformation matrix including rotation and translation.

This equation has to be solved using non-linear methods, because for this error metric, "no closed-form solutions are available" [RL01] (see Fig. 2.7).

## 2.4 Domain Specific Languages

Domain Specific Languages offer an elegant way to model data processing tasks in a more abstract way using a textual or graphical representation or a combination of both. As an example, UML was introduced by the OMG and accepted by the ISO (International Organization for Standardization) as a standard in 2000. It allows to model high-level abstractions of real-world problems by using graphical descriptions. Different kinds of diagrams, such as class-, activity- or state-machine diagrams, are the basis of modeling structure and functionality.

DSLs are divided into two types: internal and external [Fow10, EEK$^+$12].

**Internal DSLs** use the existing infrastructure of host programming languages. Examples for such DSLs are OpenCL or Open Graphics Library (OpenGL). Both languages are a C dialect, extending C to their requirements.

**External DSLs** however are designed from scratch after a full analysis of the problem description. Using special keywords, abstractions and control structures, they are able to illustrate complex problems mostly in simpler and more compressed forms. A big drawback is that the whole infrastructure such as parsers, lexers and compilers has to be built. Well-known examples are the unix shell scripts or the Structured Query Language (SQL).

Both kinds of DSLs improve the readability and the formulation of domain-specific, often real world problems. Using tools such as xText [EEK$^+$16] (textual modeling) and/or GMF [EC16] (graphical modeling) can help significantly in creating specific, problem related languages.

### 2.4.1 DSL - Advantages

The usage of a Domain Specific Language has several advantages over the usage of generic (programming-) languages as e.g. C++:

- Full domain relation
- Reduction/Hiding of technical code
- Improved readability
- Possibility to validate for the specific domain
- More simple to learn for domain experts, even non-programmers

### 2.4.2 DSL - Disadvantages

Besides the mentioned advantages in the previous section a Domain Specific Language has also several disadvantages:

- Missing language features can stop the full development process
- The quality of the DSL depends on the quality of the language developer
- A DSL is not standardized in most cases, which makes it difficult to find new experts
- Special trainings for the new DSL are necessary

- The DSL development is maybe bound to design tools which have no long term support
- Additional effort to maintain the newly developed tools is often necessary

### 2.4.3 Development

The previous sections have given an overview of several advantages and disadvantages of Domain Specific Languages. But still DSLs make sense in many scenarios. Their development can be split into three phases:

1. Language Definition
2. Sentence Development
3. Sentence Evaluation

**Language Definition** During this phase, an alphabet of domain specific keywords and rules how to combine these keywords has to be designed.

**Sentence Development** In this phase, domain experts have to design sentences (using the alphabet from the language definition) to formulate and solve their domain problems.

**Sentence Evaluation** The last phase evaluates the domain specific sentences from the development phase either by transforming the language into another DSL or by directly evaluating it. This can be performed by using code generators, interpreters or compilers.

The design of new DSLs is possible in two different ways:

- By designing custom tools: lexer, parser and compiler/interpreter and additional development tools
- By using tools as e.g. Xtext [EEK+16] which automatically create a new set of tools

**Design of Custom Tools** Designing custom tools has the advantages that it will exactly fit to the problem description. The tools can be faster and better maintainable as generated tools. But at the same time, these tools have a big overhead especially for the development time because special experts for lexer, parser and/or compiler development are necessary and the project can rapidly grow in complexity.

**Using DSL Creation Tools** These kind of tools have the advantage that DSLs can also be quickly designed from developers who are maybe no experts for formal language design. The developer has just to create a language description. Using this description, the tools are able to create parts or the full set of required engineering tools. But these tools have also several disadvantages as e.g. the dependency on their creation tools. Furthermore the maintainability of automatically generated tools is sometimes difficult.

# Part I

# Time-of-Flight Algorithms

# 3 Introduction

Time-of-Flight (ToF) sensors like the PMD camera [PMD16] offer an elegant way to measure depth data. However, artifacts arise during capturing. Image denoising has always been an important discipline in image processing. Several techniques have been established during the last years. Methods for outlier removal and outlier correction have been developed to improve acquisition quality of noisy data. Denoising and optimization can be applied at different stages: at image acquisition level and/or during data processing. Previous works have shown that combining both optimization stages gives the best result [LKS$^+$13][LNL$^+$13]. Compared to other industrial suited range sensing systems like laser-scanners, ToF cameras provide fully lateral 3D information at high frame rates, additional grayscale information and full eye safety. Furthermore, cheap industrial cameras, that can handle difficult environment conditions, are currently under development and are already available with a resolution of 176×132 pixels. [IFM16]

## Problem Statement

There are several challenges in using ToF range sensing cameras, that should be mentioned. Common issues, that virtually all active optical systems encounter and which are specially related to the topics of this thesis will be shown in this section.

A huge problem for ToF sensing are low or high reflective objects. Low reflective objects result in low optical signals which reduce the SNR, thus degrading the measurement reliability. Highly specular surfaces however, as e.g. reflectors at cars (see also Chapter 6), often lead to oversaturation effects and cause a wrong distance measurement. Further ToF specific issues, that need to be taken into account, are different ToF error types. A typical error is the systematic intensity-related distance error. The mean and not only the variance of the measured distance is influenced by the total amount of incident light (Sec. 2.1.2.2). An other type is the systematic distance (wiggling) error. It occurs because the theoretically required sinusoidal signal is not achievable in practice (Sec. 2.1.2.2). Flying pixels are the next type of error. It occurs in spatial regions with inhomogeneous depth (see Sec. 6.5.1.3). The next error type is visible only in case of motion: motion artifacts. Individual pixels no longer relate to the same object point during the acquisition of the four phase images. This leads to motion artifacts at object boundaries and in regions of inhomogeneous reflection (see Sec. 6.5.1.1). An other error is a result from the non-uniform illumination of the scene. It causes a lateral intensity attenuation in the image plane.

The stated points bring up several interesting and complex challenges. A very general problem is the usage of ToF cameras in combination with computer vision based image registration algorithms. Firstly, ToF cameras have a comparably low resolution, secondly, the data characteristics are very different from standard 2D grayscale or RGB-cameras. For example, the PMD-intensity data is not a true gray data, it is a measure of the amount of reflected light and depends on the infrared reflectivity of the reflecting material.

Corresponding to the significant challenges in the domain of depth data processing, several contributions are made with this thesis. These contributions are the Compensation of Motion Artifacts, an Automatic Integration Time Estimation Algorithm and a Complex Depth Data Processing Pipeline for the 3D reconstruction of a moving car.

## Outline

The structure of this part of the thesis consists of the following four chapters:

*Chapter 4* shows how ToF motion artifacts can be reduced using a novel algorithm developed to perform a fast real-time motion compensation with high frame rates especially on GPUs or FPGAs.

*Chapter 5* introduces a new approach to automatically determine the best ToF integration time for arbitrary scenes.

*Chapter 6* shows a novel approach for online acquisition and reconstruction of a vehicle's outer hull using three ToF cameras in an industrial example application.

*Chapter 7* summarizes the results of the previous depth data processing chapters and concludes the first part of this thesis.

# 4 Compensation of Motion Artifacts

Time-of-Flight sensors as the PMD camera [PMD16] offer an elegant way to measure depth data. However, artifacts arising from dynamic scenes are still not resolved satisfactorily. Moving objects in scenes result in a blur effect (motion artifacts) in acquired depth images. A fast movement leads to strong artifacts, related to the sensor's working principle which is based on the sequential acquisition of four so-called phase images in order to generate a depth map (see also Sec. 2.1.1). Artifacts occur in areas where corresponding phase image values do not refer to the same object position, resulting in an incorrect distance calculation.

This dissertation proposes a new algorithm to perform a fast real-time motion compensation with high frame rates (above 50 FPS). The focus lies on high flexibility to allow the algorithm to be either computed in parallel on a GPU using CUDA [NVI16] or to simply port it to small devices like an FPGA preprocessing platform. To fulfill these requirements, a linear movement with constant motion between the four consecutive phase images is assumed. Hence, the algorithm still allows to correct motion in all directions assuming this linear behavior for each individual pixel (see Sec. 4.3). Invalid pixels are replaced by corresponding values of the spatial neighborhood. This leads to simpler and faster processing compared to standard methods shown in Sec. 4.1. For the evaluation, a PMD CamCube 3.0 with a resolution of $200 \times 200$ pixel is used. The big advantages of the proposed method are the possibility of an automatic motion detection, a search direction restriction, the system performance and also the repeatability of results in different applications (other algorithm often don't support an arbitrary degree of freedom or have at least performance problems with it, see Sec. 4.1).

*Publications: Real-Time Motion Artifact Compensation for PMD-ToF Images [HLK13a]; Real-Time Motion Artifacts Compensation of ToF Sensors Data on GPU [LHK13]*

## 4.1 Related Work

In the last years, several methods have been proposed to detect and compensate motion artifacts.

Hussmann et al. [HHE11] introduce a motion compensation for linear object motion on a conveyor belt. Areas of motion artifacts are identified using phase image differences. These areas are binarized for each individual difference image using a threshold. The length of motion is determined by processing each line of the binary images and counting the lines with white pixels. Once knowing the length, every phase image is moved accordingly before the distances are calculated. The algorithm is implemented exemplary on an FPGA platform, but it is restricted to a linear motion in a range between $90 - 100cm$ due to the small object size and the camera's field of view.

Schmidt [Sch11] proposes a method handling motion artifacts as disturbances in the raw data. Motion artifacts are calculated for each phase image using a temporal derivative. High temporal derivatives of the raw data are then replaced by previously valid values. An advantage compared to Hussmann et al. is the arbitrary degree of freedom. Lee et al. [LKKK12] propose a similar approach where they detect motion artifacts by temporal-spatial coherence of neighboring pixels directly on the hardware level.

Another method was proposed by Lindner et al. [LK09]. This method computes a dense optical flow to compensate spatial shifts between subsequent phase images (three flow calculations). Lefloch et al. [LHK13] proposed a method improving this approach. Necessary computation steps can be reduced to two flow calculations. The missing step is replaced by a polynomial approximation. One big disadvantage is the system performance. The optical flow computation is a very time consuming task and thus is a heavy burden for real-time processing, if further processing tasks need to be performed.

The proposed method uses particular parts from Lindner and Lefloch [LK09, LHK13] (flow field) and Hussmann [HHE11] (binarization of the motion area). The algorithm restricts the motion to blurred areas only and optimizes the flow field detection.

## 4.2 A Method for Fast Linear Motion Compensation

This section starts with an analysis of the origin of motion artifacts and continues with a detailed description of the proposed method.

### 4.2.1 Problem Analysis

ToF-cameras like the PMD camera have the advantage to be able to acquire full distance-/depth-images of the whole scene at once. This is done using a sequence of four phase images, as described in Sec. 2.1.1 and shown in Fig. 4.1.

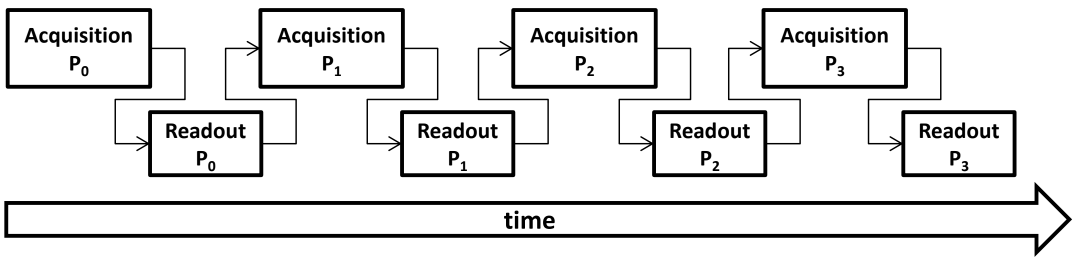


Figure 4.1: Schematic view of the acquisition process of a PMD frame using four phase images. [HLK13a]

One full phase acquisition is split in two parts: acquisition and readout. The acquisition time is equal to the integration time set, the readout time of actual PMD cameras is stated as about $3.5ms$. Ideally all four phase images would simultaneously be recorded. In reality the acquisition is done sequentially (see Fig. 4.1). Motion artifacts typically arise in areas of unmatching raw phase values due to motion (see. Fig. 4.2). It mainly occurs at object boundaries and in regions of inhomogeneous reflection. This effect becomes more extensive the faster an object moves, the closer the object is to the camera and the longer the scene is exposed (high integration times) [LK09].

Fig. 4.2 shows the default demodulation of a car (top images), moving from the right to the left and of a moving hand (bottom images). In both scenes, the blurred areas are marked red. It can be seen that especially these areas contain many motion artifacts.

Blurred areas in depth maps lead to incorrect distance computations. The goal of motion compensation approaches is the elimination of these areas in order to minimize errors. The motion during a single acquisition is not considered here and is nearly negligible for small integration times ($< 1ms$).

### 4.2.2 The Motion Compensation Approach

The proposed method works on a per pixel basis allowing arbitrary motion directions. It is divided into several steps, starting with a phase normalization. The normalization is done to compensate the sensor's pixel gains and to equalize the inhomogeneous illumination of the scene. This is necessary due to the block-matching like working principle of the approach and to obtain comparable raw values. In a second step the area of motion is estimated to improve the processing time. The motion direction is then determined by a correspondence search in the spatial neighborhood. Once

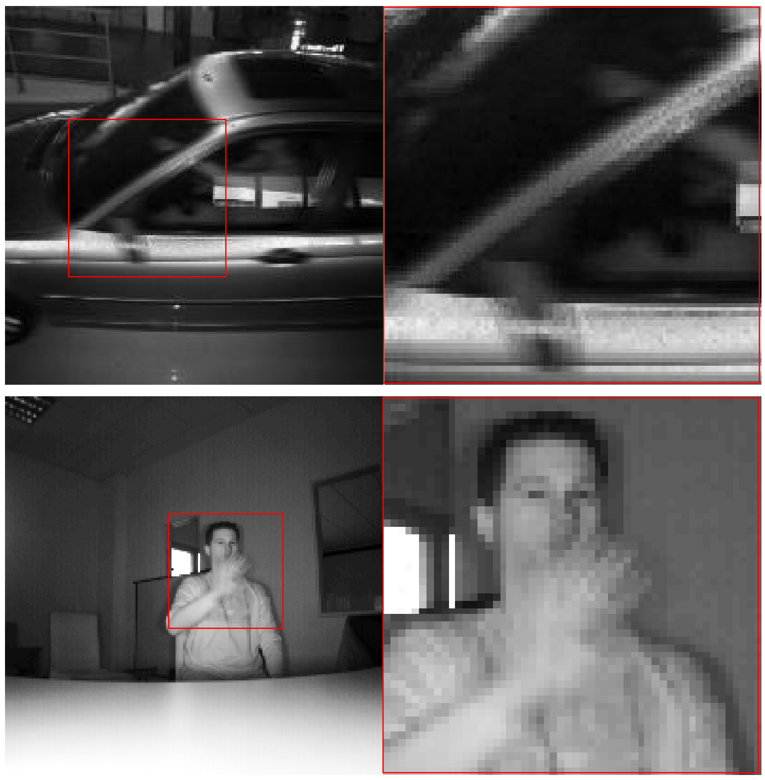


Figure 4.2: Top: demodulation of the car's phase image sequence and a corresponding closeup. Bottom: a moving hand scene and a corresponding closeup. Left: motion areas are marked red. Right: The closeups of the marked motion areas.[HLK13a]

knowing this kind of flow field, the raw values can be corrected. The processing pipeline can be seen in Fig. 4.3 and will be explained in the following sections.

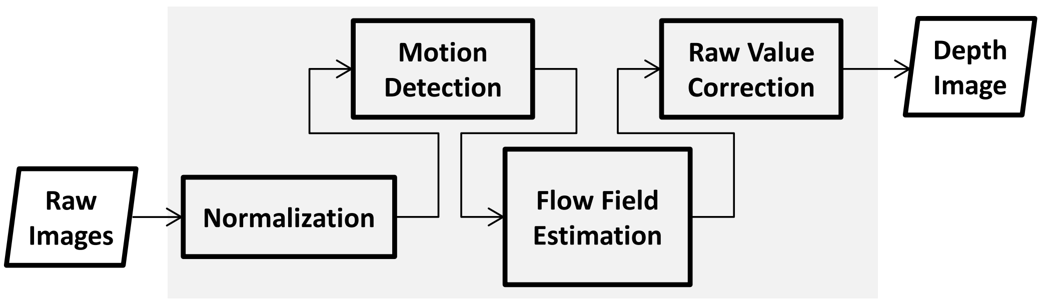


Figure 4.3: The motion detection and processing pipeline used for the algorithm. [HLK13a]

#### 4.2.2.1 Phase Normalization

According to the behavior and design of PMD cameras, there are several aspects for the pixel correspondence search, which have to be taken care of. One point is the radial light attenuation. Images become darker from the center to the border. Another aspect is the difference in pixel gains, which has to be individually corrected for each sensor and every pixel.

To compensate these two problems, Lindner et al. [LK09] propose to create a set of reference images with different reflectivities at different distances and to pixel-wise apply the fit intensity correction functions:

$$f_{P_A}(P_{A_i}) = \tilde{P}_{A_i},\ f_{P_B}(P_{B_i}) = \tilde{P}_{B_i} \text{ with } i = 0 \ldots 3 \tag{4.1}$$

This allows to minimize the following equation:

$$\sum_{i=0}^{3} (\tilde{P}_{A_i} + \tilde{P}_{B_i}) = h_{\text{ref}}. \tag{4.2}$$

The brightest pixel in a homogeneous surface is taken and used as reference intensity, the fitting functions are assumed to be logarithmic as $f_X(X_i) = a\sqrt{X_i + b} + cX_i + d$.

Applying these corrections improve the search as shown by Lindner et al. [LK09].

#### 4.2.2.2 Motion Detection

Since the motion estimation is a computationally intensive task, an important preprocessing step is to detect areas of apparent motions first. Motion can be detected using the changes in the total per-pixel intensity for the subsequent phase images, i.e.

$$P_i^+ = P_{A_i} + P_{B_i} \tag{4.3}$$

$$M = \sum_{i=1}^{3} \left| P_i^+ - P_0^+ \right| \tag{4.4}$$

In a next step, the estimated motion image $M$ is binarized

$$B = M > \theta \tag{4.5}$$

where $B$ is the binary image and $\theta$ a threshold value that is determined experimentally. Tests have shown that for $\theta = 650$ (about 1% of the maximum of $P_{A_i/B_i} = 65535$) the results are the most reliable. Fig. 4.4 shows the motion image and its corresponding binary image. White areas (ones) on the right side indicate unmatching raw values.

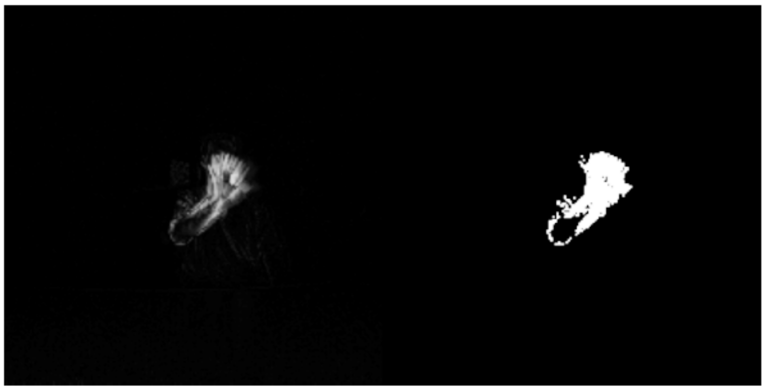


Figure 4.4: The moving hand from Fig. 4.2 with extracted motion artifacts. Left: the motion image $M$ calculated using Equation 4.4. Right: the binarized image $B$ thresholded using Equation 4.5. [HLK13a]

#### 4.2.2.3 Motion Direction Estimation using Block Matching

Inspired by the idea of the optical flow motion estimation, a block matching algorithm is used to determine a 2D vector displacement map ($D(u,v)$) without subpixel precision. Each pixel value represents a unique displacement vector $(u,v)^T$ in $D$.

The approach assumes a linear motion between all raw phase images with a constant velocity (see Sec. 4.3). A pixel-wise motion displacement is estimated for all detected invalid pixels in $B$. Therefore a motion window around every invalid pixel is defined, which limits the detectable motion around these pixels. The window is assumed to be square with an odd size between 3 and 11 pixels (**M**otion **W**indow **S**ize, MWS). Vectors from the center (the invalid pixel) to all neighbors are calculated and scaled according to the phase image index. Let $(dx,dy)$ be a single delta for a possible pixel correspondence shift between adjacent phase images, then the respective shifted phase values for the $i$-th phase image are given as:

$$\begin{aligned} P_{shifted,i,dx,dy}(x,y) &= P_i(x+i\cdot dx, y+i\cdot dy) \\ i &\in \{0,1,2,3\} \end{aligned} \tag{4.6}$$

$P_i$ and $P_{shifted,i}$ represent the particular phase image with index $i$. $dx$ and $dy$ are the applied deltas from the center (see also Fig. 4.6) with a maximum value of:

$$dx_{max} = dy_{max} = (MWS-1)/2 \tag{4.7}$$

and an odd MWS. The maximum euclidean pixel distance $l$ between two corresponding points of phase image $P_0$ and $P_3$ is given as:

$$l = 3\cdot \left\| \overrightarrow{(dx_{max}, dy_{max})} \right\| \tag{4.8}$$

In compliance with Equation 4.8 and some knowledge about the expected motion in a scene, the Motion Window Size can be preset to optimize the system performance. Setting $MWS = 5$ lead to reliable results in most of the situations (see Sec. 4.3). Fig. 4.5 shows how search vectors are defined and introduces the coordinate system exemplary for a $5 \times 5$ motion window.

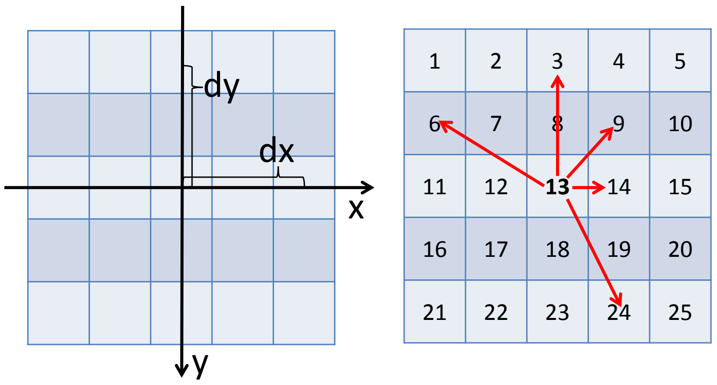


Figure 4.5: Left: the coordinate system and an example offset $dx = dy = 2$. Right: the grid cells are numbered in a row-wise order. Index '13' indicates the start position. Five sample vectors are presented here. [HLK13a]

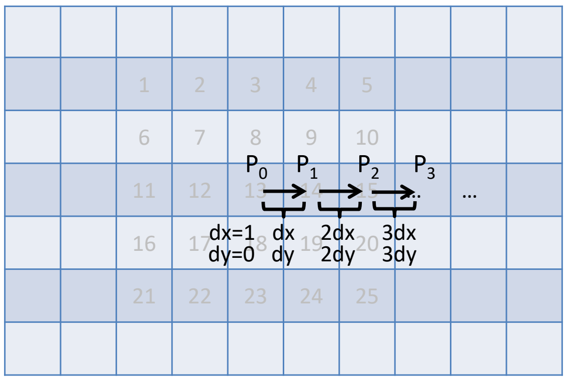


Figure 4.6: An exemplary flow for a shift $dx = 1$ and $dy = 0$. This leads to a maximum shift between the four phase images of 3 pixels. [HLK13a]

For the estimation of the best corresponding flow, experiments have shown, that the Sum of Squared Differences (ssd) is a good method to determine and calculate all possible flow vectors within the defined motion window for all $dx$ and $dy$ combinations:

$$ssd_{index}(x,y,dx,dy) = \sum_{i=1}^{3}(P_0(x,y) - P_{shifted,i}(x,y,dx,dy))^2 \qquad index \in \left\{0,1,...,MWS^2\right\} \tag{4.9}$$

Accordingly, the best correspondence value has the minimal deviation from the first phase image. So the final flow vector for the currently processed pixel can be expressed as:

$$V(u,v) = (dx(u,v), dy(u,v))^T = \operatorname{argmin}(ssd(x,y,dx,dy)) \tag{4.10}$$

The number of possible vectors $MWS^2$ is leading to a time complexity $T(MWS) = \mathcal{O}(MWS^2)$. A quadratic complexity allows only small Motion Window Sizes (about $11 \times 11$) to perform the algorithm in real-time. To overcome this problem, the search direction can be restricted to an initial or mean direction from a previous frame.

### 4.2.3 Search Space Reduction

An additional performance optimization can be achieved using a search space reduction as can be seen in Fig. 4.7. Therefore the mean direction angle of a previous frame is used as initial guess for the current motion. The direction angle $\varphi_{(u,v)^T}$ for one pixel is calculated between the positive x-axis $\overrightarrow{x}$ and the corresponding flow vector $\overrightarrow{f(u,v)}$. The mean motion direction angle $\varphi$ is defined as average of all estimated flow vector direction angles. Assuming a small motion between two consecutive frames, the amount of change of the mean direction angle is small. Now using this assumption, all pixels (x′, y′) in the search window whose position vector has an angle in the range of $\varphi \pm \rho/2$ ($0° \leq \rho \leq 359°$) are taken into consideration for the motion estimation. The raw phase value correction is then applied to the reduced flow field as described in Sec. 4.2.5.

All possible flow direction angles in a motion window can be precalculated which offers simple portability to small platforms, e.g. an FPGA, by using lookup-tables.

An alternative to using the mean direction angle of the previous frame can be the usage of a local direction vector average around the motion window. This can be combined with the vector length to make a comparison more meaningful.

### 4.2.4 Flow Field Optimization

In order to improve the robustness of the approach, a possible optimization is optionally applied taking the spatial neighborhood into account. A median filter is used

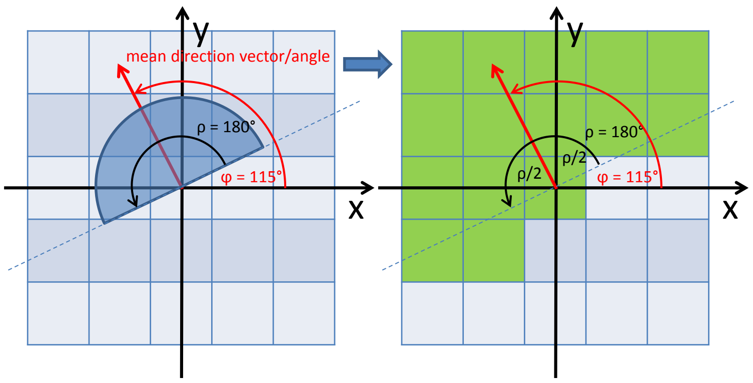


Figure 4.7: Left: the mean motion vector ($\varphi = 115°$ in this example) calculated in a previous frame (red arrow) and an exemplary search space reduction to $\rho = 180°$. Right: valid motion vectors resulting from the reduction are marked green. [HLK13a]

to filter outliers. For this, all motion vectors in a neighborhood with size $MWS$ are considered. Median filtering is then performed using the direction angle as defined in Sec. 4.2.3. The vector length is kept.

### 4.2.5 Raw Phase Value Correction

Once the flow field ($V(u,v)$) is determined, it can be applied to the raw phase values $P_{A_{0-3}}$ and $P_{B_{0-3}}$ according to Equation 4.6. Each vector of the flow field is applied to its corresponding raw value. After this data correction, the depth values can be reconstructed according to the principle described in Sec. 2.1.1. The results and evaluation can be seen in Sec. 4.3.

## 4.3 Results

The following subsections give detailed information about the motion compensation results obtained with the proposed method. To perform a comprehensive analysis, the evaluation is split in two parts. In the first part, a quantitative evaluation is done to allow for the comparison of results against ground-truth data. Using artificial scenes gives reproducible and reliable results. In the second part, a qualitative evaluation in real scenes is shown. Having the disadvantage that generally no ground-truth models are available, a visual evaluation makes it possible to see if the correction could successfully be applied.

### 4.3.1 Quantitative Results

The method has been tested in a variety of scenes of different complexities (simulated and real environments). A robustness and peformance evaluation can be done using simulated data (see Table 4.2 and Table 4.3). Similar to Lefloch et al. [LHK13] different data sets generated with a simulator [KK09] have been used. The data sets are generated without white noise, but with flying pixels and motion (translation and/or rotation). Statistic evaluations are done with the tool CloudCompare[1]. The first data set is a buddha figure, the second is a dragon. Both figures are used as input for the simulator. An artificial, planar wall is placed in a distance of 4 meters. In front of this wall, in a distance of about 3 meters, the figures are placed. This setup provides reliable ground-truth data. To obtain motion data, several different images at different camera positions are acquired. The camera is transformed between each individual phase acquisition. For the buddha, a simple lateral camera translation of 1cm (about $2m/s$ motion speed) is used. In the dragon figure setup, the camera is rotated 1 degree (about $200°/s$ angular velocity) around the z-axis (line of sight).

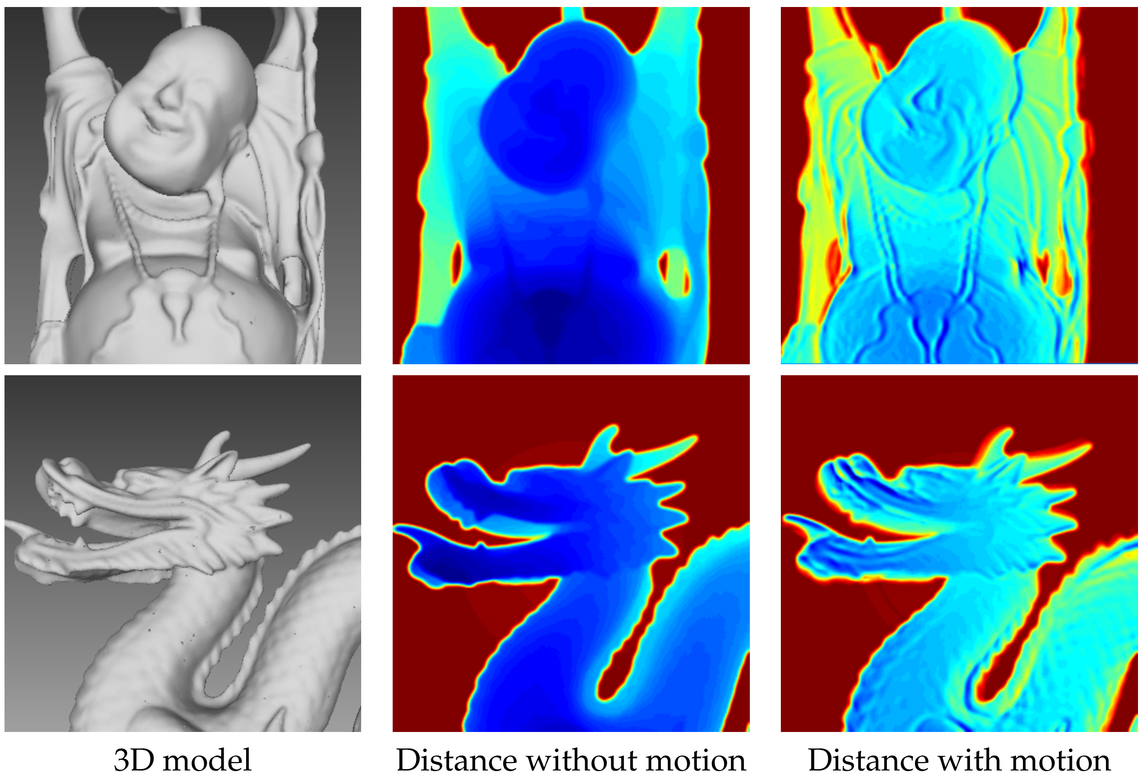


Figure 4.8: Two different data sets (buddha (top) and dragon (bottom)) that have been used for the robustness evaluation of the approach. Left: the ground-truth 3D model. Center: The Cartesian distance image without any motion. Right: the Cartesian distance image with motion. [HLK13a]

Using the ground-truth of Fig. 4.8, comparable results between the different motion compensation approaches (see also Fig. 4.9) can easily be generated.

[1]http://www.danielgm.net/cc/

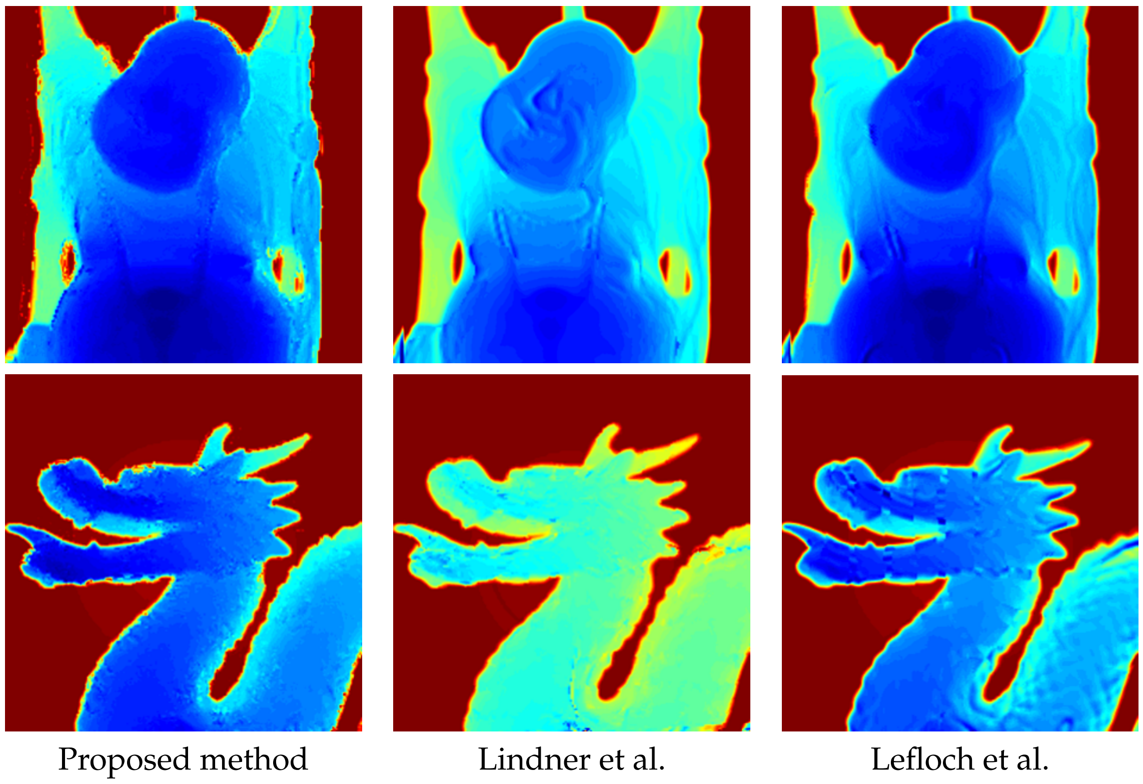


Figure 4.9: The results of the three evaluated methods. As can be seen, all the methods yield good results visually when comparing them to the ground-truth distance with no motion shown in Fig. 4.8. [HLK13a]

| **Scenes** | **Distance errors from ground-truth (cm)** | | | |
|---|---|---|---|---|
| | Static (no motion) | | Dynamic (with motion) | |
| | Mean | Sigma | Mean | Sigma |
| *Buddha* | 0.64 | 3.12 | 5.96 | 9.32 |
| *Dragon* | 1.25 | 4.62 | 7.75 | 14.37 |

Table 4.1: The deviation of the ground-truth depth data (flying pixels included) of the buddha and dragon scene from the underlying meshes (before correction). It shows the mean distance error and the deviation for the static and the dynamic scene.[HLK13a]

Table 4.1 shows a statistical evaluation of the buddha and dragon scene without motion compensation where static background pixels are discarded. The dynamic scene is created as previously described.

For further evaluation, several different setups are created to verify the quality of the proposed method. The first test shows the behavior of the algorithm with different settings for the Motion Window Size (MWS, see Sec. 4.2.2.3), neighborhood filtering (NH, see Sec. 4.2.3), motion area estimation ($\theta$ with a maximum of 65.535) and also search space restriction ($\rho$) (see Sec. 4.2.2.2). Table 4.2 and Table 4.3 contain the test results of the different setups and show the detailed behavior of the proposed algorithm using different parameter sets. Especially the remaining depth error compared to the ground-truth data and the system performance is highlighted. It can be seen that as expected, the best results are given without any limitation and restriction of the search space ($\theta = 0\%$, $\rho = 0$). The mean error of the buddha motion scene is reduced from 5.96$cm$ (±9.32$cm$) to 1.14$cm$ (±3.02$cm$), the dragon scene is corrected from a mean error of 7.75$cm$ (±14.37$cm$) to 2.07$cm$ (±5.65$cm$). Furthermore it can be seen that with an increasing $\theta$ the correction performance gets better, but the quality decreases. Another fact that becomes apparent is that using the neighborhood flow smoothing technique also improves the mean error compared to the initial error, but with the disadvantage of losing performance: with the same settings and neighborhood filtering the buddha scene can be corrected in 16.13$ms$, without neighborhood filtering it takes only 11.57$ms$. A similar behavior can be seen for the dragon scene in Table 4.3. In addition, restricting the algorithm to a maximum direction deviation also improves the correction quality (mean error) and the system performance. This can be achieved by the rejection of a large number of search vectors in the motion window (Maximum number of vectors $MWS = 5$: 1.000.000; $MWS = 7$: 1.960.000). Up to 36% of the possible directions are rejected ($MWS = 7$, $\rho = 90°$, 1.960.000 direction search vectors, rejected directions between 110770 and 716877) in the buddha scene and up to 30% ($MWS = 7$, $\rho = 90°$, 1.960.000 direction search vectors, rejected directions between 118916 and 583470) in the dragon scene. Please note that the execution time is an average value of 100 measurements. Furthemore it can be seen that the mean motion direction $\varphi$ most closely approximates the expected linear translation of the buddha scene of 180°.

Additionally the algorithm is compared against the methods proposed by [LK09] and [LHK13]. The approach reduces the mean error of the buddha scene to 1.14cm (±3.02$cm$), compared to Lindner 1.13$cm$ (±4.39$cm$) and Lefloch 1.46$cm$ (±4.40$cm$). For the dragon scene, the remaining mean error with the method is 2.07$cm$ (±5.65$cm$), for Lindner 2.26$cm$ (±7.57$cm$) and for Lefloch 3.14$cm$ (±8.00$cm$). The results between the three compared methods are nearly equal, but the newly proposed method can score with the execution time, which is about half the time of the method from Lefloch et al. and an eighth of Lindner et al.

The result of the tested setups is a good correction compared to the input mean error that can be seen in Table 4.1. Furthermore in comparison with Lindner and Lefloch, the method gives slightly better (dragon scene) or nearly equal (buddha scene) results and is also suitable for real-time applications with a framerate of 50–100 FPS allowing for additional data processing. Note: compared to the evaluation of Lefloch et al. [LHK13], a smaller clamping distance (3.85$m$) is used to remove the wall, explaining the slightly different mean and sigma values. A good default parameter set is a threshold $\theta = 1\%$ and $MWS = 5$. Another helpful setting is a direction restriction to the mean motion direction. The default settings and the search area restriction significantly optimize the system performance and the motion compensation quality. The tests were executed on an Intel Core i7-3770K CPU @ 3.50 GHz and an NVIDIA GeForce GTX 680, 2GB graphics card using CUDA.

| **Buddha scene** | | | | **Distance errors from ground-truth corrected** | | | | |
|---|---|---|---|---|---|---|---|---|
| MWS | NH | $\theta$(%) | $\rho$ (°) | **Mean** (cm) | **Sigma** (cm) | $\varphi$ (°) | Rejected directions (%) | ⌀ Time (ms) |
| **5** | - | 0.00 | - | **1.15** | 3.10 | - | 0 | 12.58 |
| 5 | - | 1.00 | - | 1.70 | 3.54 | - | 0 | 11.57 |
| 5 | - | 5.00 | - | 3.49 | 4.70 | - | 0 | 10.59 |
| 5 | - | 8.00 | - | 4.22 | 5.18 | - | 0 | 10.13 |
| 5 | x | 1.00 | - | 1.38 | 3.05 | - | 0 | 16.13 |
| 5 | - | 1.00 | 90 | 1.34 | 3.33 | 206.36 | 11.89 | **9.79** |
| 5 | x | 1.00 | 90 | 1.46 | 3.24 | 223.98 | 11.08 | 13.18 |
| 5 | - | 1.00 | 180 | 1.34 | 3.38 | 179.21 | 21.24 | 11.08 |
| 5 | x | 1.00 | 180 | 1.27 | 3.04 | 190.33 | 21.24 | 13.49 |
| **7** | - | 0.00 | - | **1.14** | 3.02 | - | 0 | 25.04 |
| 7 | - | 1.00 | - | 1.16 | 3.08 | - | 0 | 23.54 |
| 7 | x | 1.00 | - | 1.34 | 3.03 | - | 0 | 31.86 |
| 7 | - | 1.00 | 90 | 1.35 | 3.35 | 174.34 | 36.58 | **12.15** |
| 7 | x | 1.00 | 90 | 1.35 | 3.13 | 178.90 | 36.58 | 21.16 |
| 7 | - | 1.00 | 180 | 1.35 | 3.37 | 179.74 | 24.38 | 15.18 |
| 7 | x | 1.00 | 180 | 1.37 | 3.04 | 187.10 | 24.38 | 23.53 |
| Method Lindner et al. | | | | | | | | |
| - | - | - | - | 1.13 | 4.39 | - | - | 71.87 |
| Method Lefloch et al. | | | | | | | | |
| - | - | - | - | 1.46 | 4.40 | - | - | 25.60 |

Table 4.2: The statistic evaluation of the buddha scene and the behavior of the mean error in relation to different parameters. Statistics are shown for different Motion Window Sizes, neighborhood filtering (NH) on(x) and off(-), binarization thresholds $\theta$ (in percent relative to 65.535) and a search space restriction $\rho$.[HLK13a]. The tests were executed on an Intel Core i7-3770K CPU @ 3.50 GHz and an NVIDIA GeForce GTX 680, 2GB graphics card using CUDA

### 4.3.2 Qualitative Results

This part of the evaluation shows the behavior of real environments and applications. Two different setups are built. Unfortunately, there are no ground-truth values for these real world data sets, therefore they are limited to a visual comparison. The first scene shows a moving hand as can be seen in Fig. 4.10. The hand is moved very fast from one side to the other. The figure shows how the blurred images are corrected using the proposed method. Furthermore the images also show the motion area and direction restriction ($MWS = 5, \theta = 1\%, angle = 90°$). It can be seen that the blur is fully corrected.

| **Dragon Scene** | | | | **Distance errors from Ground-truth corrected** | | | | |
|---|---|---|---|---|---|---|---|---|
| MWS | NH | $\theta$(%) | $\rho$ (°) | **Mean** (cm) | **Sigma** (cm) | $\varphi$ (°) | Rejected directions (%) | ⌀ Time (ms) |
| **5** | - | 0.00 | - | **2.08** | 5.70 | - | 0 | 12.09 |
| 5 | - | 1.00 | - | 2.09 | 5.71 | - | 0 | 11.97 |
| 5 | - | 5.00 | - | 2.43 | 6.04 | - | 0 | 11.57 |
| 5 | - | 8.00 | - | 2.62 | 6.01 | - | 0 | 10.89 |
| 5 | x | 1.00 | - | 2.22 | 5.51 | - | 0 | 14.31 |
| 5 | - | 1.00 | 90 | 2.12 | 5.49 | 186.68 | 25.93 | **9.40** |
| 5 | x | 1.00 | 90 | 2.37 | 5.75 | 206.36 | 11.89 | 11.89 |
| 5 | - | 1.00 | 180 | 2.16 | 5.68 | 182.36 | 17.29 | 9.83 |
| 5 | x | 1.00 | 180 | 2.33 | 5.70 | 198.20 | 13.13 | 13.13 |
| **7** | - | 0.00 | - | **2.07** | 5.65 | - | 0 | 21.44 |
| 7 | - | 1.00 | - | 2.07 | 5.65 | - | 0 | 20.55 |
| 7 | x | 1.00 | - | 2.35 | 5.48 | - | 0 | 26.45 |
| 7 | - | 1.00 | 90 | 2.44 | 5.63 | 210.65 | 8.98 | **11.30** |
| 7 | x | 1.00 | 90 | 2.17 | 5.66 | 184.70 | 29.77 | 17.61 |
| 7 | - | 1.00 | 180 | 2.18 | 5.77 | 184.02 | 19.85 | 14.28 |
| 7 | x | 1.00 | 180 | 2.42 | 5.64 | 201.35 | 10.55 | 20.67 |
| Method Lindner et al. | | | | | | | | |
| - | - | - | - | 2.26 | 7.57 | - | - | 80.76 |
| Method Lefloch et al. | | | | | | | | |
| - | - | - | - | 3.14 | 8.00 | - | - | 24.07 |

Table 4.3: The statistical evaluation of the dragon scene and the behavior of the mean error in relation to different parameters. Statistics are shown for different Motion Window Sizes, neighborhood filtering (NH) on(x) and off(-), binarization thresholds $\theta$ (in percent relative to 65.535) and a search space restriction $\rho$.[HLK13a]. The tests were executed on an Intel Core i7-3770K CPU @ 3.50 GHz and an NVIDIA GeForce GTX 680, 2GB graphics card using CUDA

The second evaluated scene contains a car moving laterally in front of the camera. Motion occurs mainly on edges, the mirror and the wheels. The visually determined movement direction is about 180°. The algorithm is parameterized with $MWS = 5, \theta =$

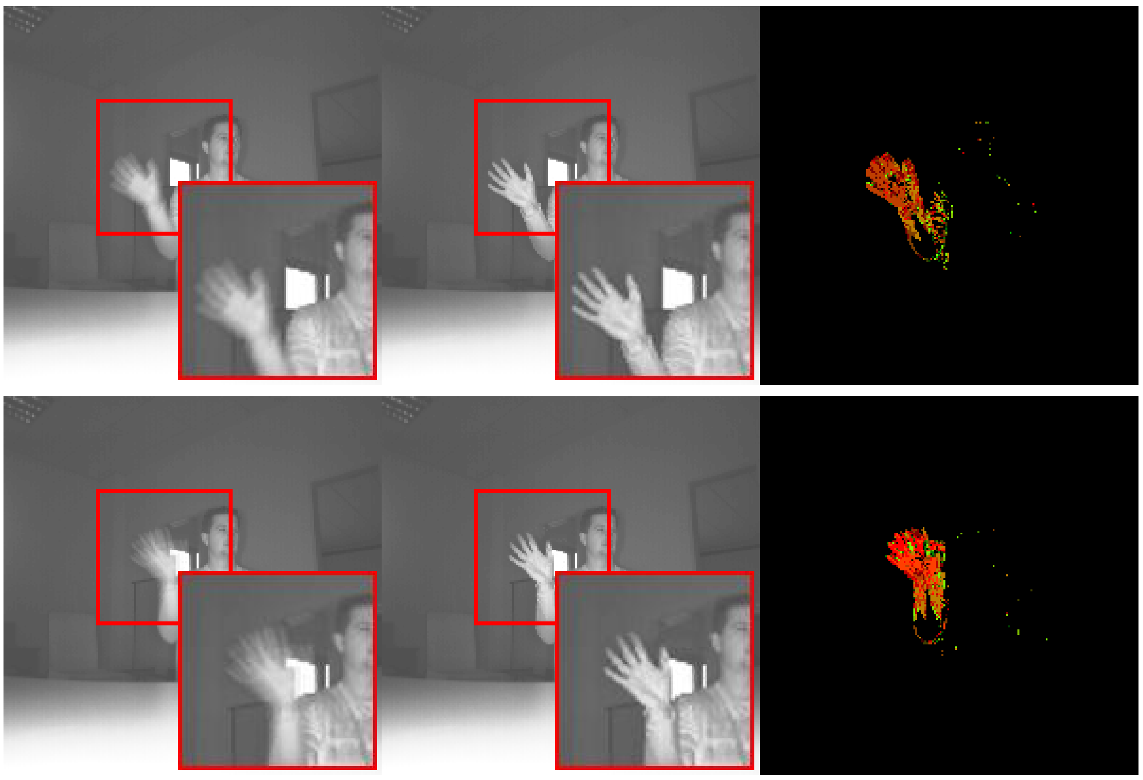


Figure 4.10: Hand scene: the left column contains images with motion artifacts, the middle column contains the corresponding motion compensated images using the proposed method and the right column contains the related flow images (red: horizontal motion, green: vertical motion). The mean estimated motion direction for the top row is $\varnothing = 125.08°$, for the bottom row $\varnothing = 91.91°$. [HLK13a]

$1\%, angle = 90°$. Area, direction and also the correction is successfully applied which leads to the expected results as can be seen in Fig. 4.11.

## 4.4 Summary

In the previous sections a new method for a fast motion artifact compensation for ToF cameras was presented. The approach is based on several assumptions such as linear motion between the four consecutive phase images and also absolute phase intensities $(P_{A_i} + P_{B_i})$ of the PMD camera. The algorithm uses a thresholding and binarization method to restrict the artifact correction area to spaces where in fact motion occurs. Furthermore an approach to find pixel correspondences in a local neighborhood (motion field estimation), a local search area minimization by tracking the mean motion direction of a previous frame and an optional motion field smoothing is proposed. It is shown that the algorithm yields good results for simulated data

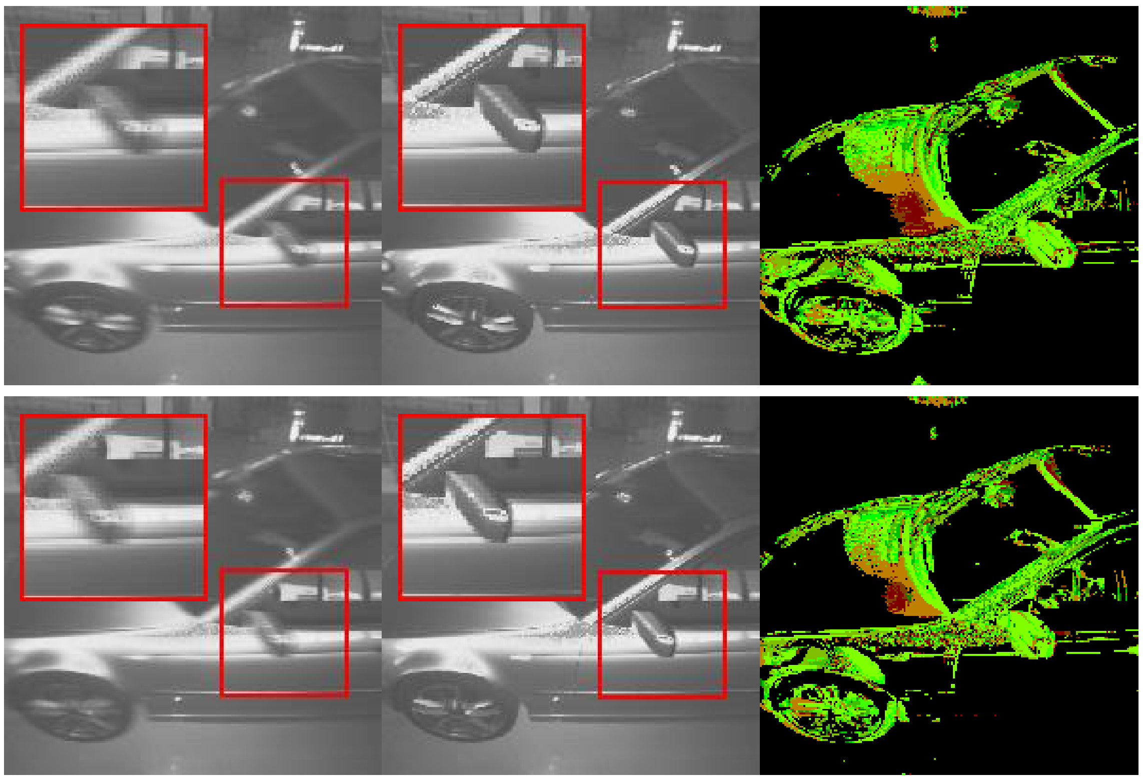


Figure 4.11: Car scene: the left column contains images with motion artifacts, the middle column contains the corresponding motion compensated images using the proposed method and the right column contains the related flow images (red: horizontal motion, green: vertical motion). The mean estimated motion direction for the top row is $\varnothing = 138.20°$, for the bottom row $\varnothing = 142.71°$. [HLK13a]

(linear and non linear motion) and also for real data. Furthermore it is shown that the results are comparable to the mean value correction of Lindner et al. [LK09] and Lefloch et al. [LHK13] and that the algorithm can work in real-time.

The proposed method could be extended for support of phase image motion correction. Furthermore the threshold $\theta$ can be automatically adapted via statistics of the observed scene. The algorithm is also designed in a way that allows for easy porting to embedded hardware like an FPGA (no subpixel flow, possibility of lookup-tables for the search area reduction and parallelization).

# 5 Automatic Integration Time Estimation

Time-of-Flight (ToF) data denoising has always been an important discipline since the initial development of this technique. Several methods have been established during the last years. Algorithms for outlier removal and outlier correction have been developed to improve acquisition quality of noisy data. Denoising and optimization can be applied at different stages: at image acquisition level and/or during data processing. Previous works have shown that combining both optimization stages gives the best result [LKS+13][LNL+13]. This chapter presents a new approach to automatically determine the best integration time (where the noise is lowest) for arbitrary scenes using the knowledge of underlying inherent sensor behavior and properties. The approach benefits from a detailed sensor data analysis and integrates this knowledge into a novel algorithm that is more flexible and stable than a proportional feedback control system ([MWSP06]) especially in unknown, arbitrary scenes. While prior work [MWSP06][GPT10] concentrates on a global optimization of intensities or amplitudes, this approach focuses on a per-pixel based improvement. The results compared to previous approaches in regard of adaption performance and thus in reduction of the mean error over time are significantly improved. For evaluation, the PMD CamCube 3.0 has been used. However, the findings are applicable for other sensors as well since different ToF sensors depict a similar behavior [LHL12]. The PMD working principle can be found in Sec. 2.1. This approach comprises the following contributions:

- A per-pixel online auto integration time estimation algorithm
- An extensive sensor behavior analysis
- A ToF evaluation scheme based on real physical data for data evaluation

*Publication: Online Improvement of ToF Camera Accuracy by Automatic Integration Time Adaption [HBK15]*

## 5.1 Related Work

In this section important existing work related to ToF error sources, noise reduction and automatic integration time estimation will be discussed.
The measurement quality of ToF cameras is influenced by several factors. Foix et al. [FAT11] have shown different systematic errors such as depth distortion (wiggling error), pixel-, amplitude-, temperature- and also integration time related errors. In [LNL$^+$13], several methods are discussed on how ToF noise can be reduced. They give an overview on how errors and noise occurs and state that a longer integration time causes a higher amplitude due to more incident light and in this case to e.g. oversaturation. This enhances the Signal-to-Noise Ratio and the depth variance. Several other works concentrate on the performance and measurement uncertainty of ToF sensors [LHL12], [EHK14].
Noise is an unavoidable source of measurement uncertainty. Its reduction has been studied very well during the last years. Several techniques have been established mainly concentrating on denoising in a post processing step. Methods for detecting and repairing defective areas have been developed and presented. Those approaches work either on raw data or on the final amplitude, intensity or depth image ([LKS$^+$13],[JPP07], [EOHM10], [JSHWY14]).
While most of the works for data denoising concentrate on post processing, data improvement can also be achieved by optimizing the integration time as proper saturation yields a high Signal-to-Noise Ratio and thus reduces noise. May et al. [MWSP06] present an approach for dynamic integration time estimation by approximating an overall mean intensity using a proportional controller. Gil et al. [GPT10] propose an automatic integration time adaption approach for visual servoing of mobile robots by approximating a mean amplitude. This is related to [MWSP06], but optimized for robots.

## 5.2 PMD Sensor Analysis

The main goal of this approach is the reduction of the overall error due to inappropriate integration times in arbitrary scenes. As previous work has shown (see Sec. 5.1), this can be achieved by preventing or at least by minimizing under- and oversaturation to increase the number of usable data points. To get a better understanding on how amplitudes, intensities and the distance error correlate with each other, an initial PMD sensor analysis is necessary. For this purpose eight metal plates with different colors and reflectivities are used: black, metallic, blue, green, red, silver, yellow and white. These plates are fixed on a planar wall at a distance of approximately 1 meter and are recorded with a distance and intensity calibrated PMD camera with an integration time range between 50 and 8000 $\mu$s with a step size of 1 $\mu$s. These measurements are used for the sensor data evaluation in the next sections. The polar ground-truth for the error analysis has been calculated per pixel, using the median

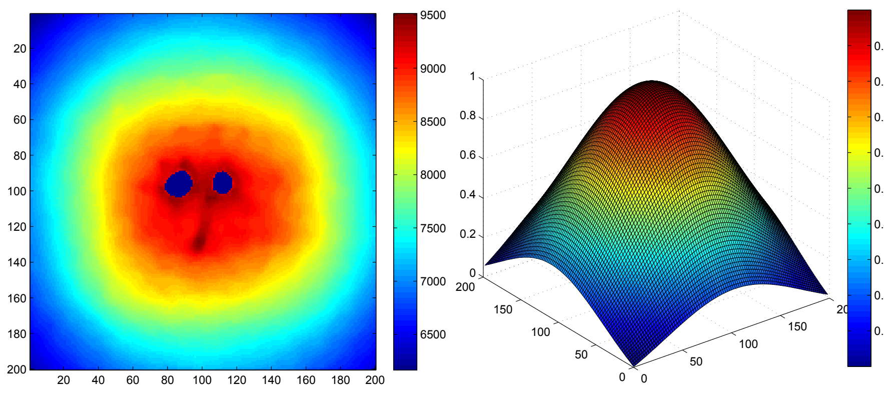


Figure 5.1: The left image shows an intensity image from the test scenario (see Sec. 5.2). Two dark blue circular spots in the center have been invalidated due to very high oversaturation, caused by a total reflection. The right image shows a weight map with values of a Gaussian distribution that is used to weight pixels in the algorithm (see Sec. 5.3.1). [HBK15]

of various hundred measurements with different integration times while omitting under- and oversaturated values, and a final 2D median filtering step. Using a reference plane fitted to temporally averaged Cartesian distance data yielded very similar results.

### 5.2.1 Spatial Intensity Distribution

Pixels near the image border yield a much lower intensity and amplitude, even when the measured distance is approximately the same (see Fig. 5.1). Such areas are much harder to properly saturate. Often very high integration times are needed. But choosing such high integration times causes an oversaturation in the image center while properly saturating the image borders. If all pixels are weighted equally when estimating the optimal integration time, the integration time will be optimized to what most pixels need to be properly saturated. Since most pixels reside outside the image center, the calculation will yield an integration time that optimizes the image borders and oversaturates the image center. If oversaturation causes a growth in error to the same degree as non-optimal saturation, then optimizing areas where most pixels reside, namely image borders, will actually reduce the overall error. However, if a system detects and excludes oversaturated values, such an approach will reduce the number of valid points. Also, the image center often captures more important objects than the image borders and should thus be regarded as more important.
As a result, a weight map has been incooperated into the approach (see Sec. 5.3.1). This allows to weight each pixel according to its importance. As Fig. 5.1 shows, a Gaussian value distribution has been used for the weight map, normalized between 0 and 1.

## 5.2.2 Intensity and Amplitude Behavior

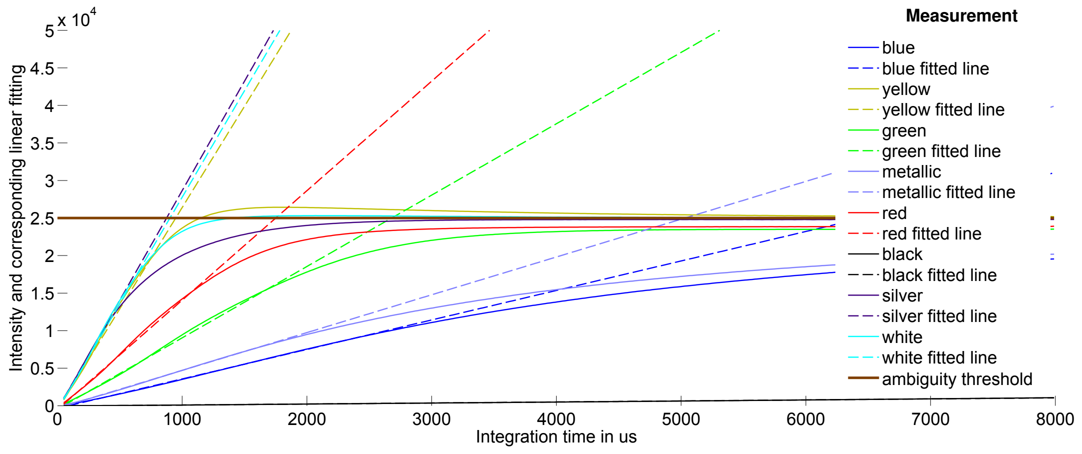


Figure 5.2: Intensity to integration time and corresponding lines fitted to the intensity's linear behavior. Values above the ambiguity threshold (brown line) are considered ambiguous. [HBK15]

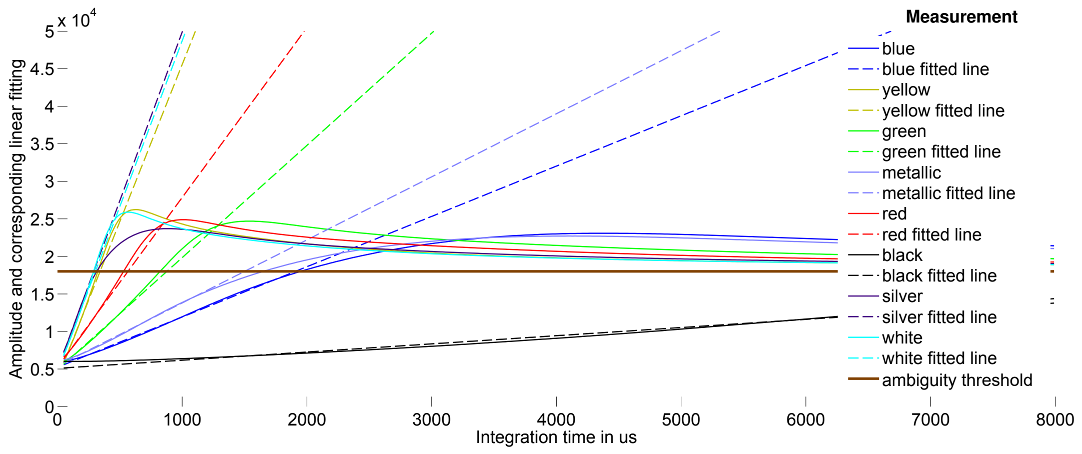


Figure 5.3: Amplitude to integration time and corresponding lines fitted to the amplitude's approximately linear behavior. Values above the ambiguity threshold (brown line) are considered ambiguous. [HBK15]

Fig. 5.2 shows the intensity as function of the integration time and linear fits to the intensity-curves based on their initial linear behavior. When oversaturation occurs (starting with values around 10000), the intensity starts deviating from its linear behavior (dashed lines) and reduces its gradient, which in some cases (e.g. white or silver) becomes even negative. Areas where the intensity function is not strictly increasing are considered ambiguous, as they cannot be mapped back to a distinct integration time, and reside above the ambiguity threshold (brown line). Also note

the black measurement's extremely low reflectivity.
Fig. 5.3 shows the amplitude as function of the integration time and, as with the intensity functions, linear fits to the curves based on their initial linear behavior. It can be seen that with an increasing integration time, the amplitude values reach a peak and then decrease in value. Compared to the intensity, the amplitude has a much larger area of ambiguity. Also, the amplitude behaves less linearly than the intensity as its gradient becomes a bit larger before the values reach the peak.
In both figures, the effects of oversaturation can be seen as deviation from linear behavior. This has also been observed by May et al. for the Swiss Ranger SR-2 ToF camera and denoted as "oversaturation gap" [MWSP06, p.3].
As already explained, values above the ambiguity threshold cannot be mapped back to a distinct integration time. This essentially means that, from their value alone, they cannot be distinguished between being slightly or strongly oversaturated.
Overall the intensity shows to be much less susceptible to ambiguous behavior compared to the amplitude. Also, its linear behavior is much more consistent.

### 5.2.3 Amplitude-Error Correlation

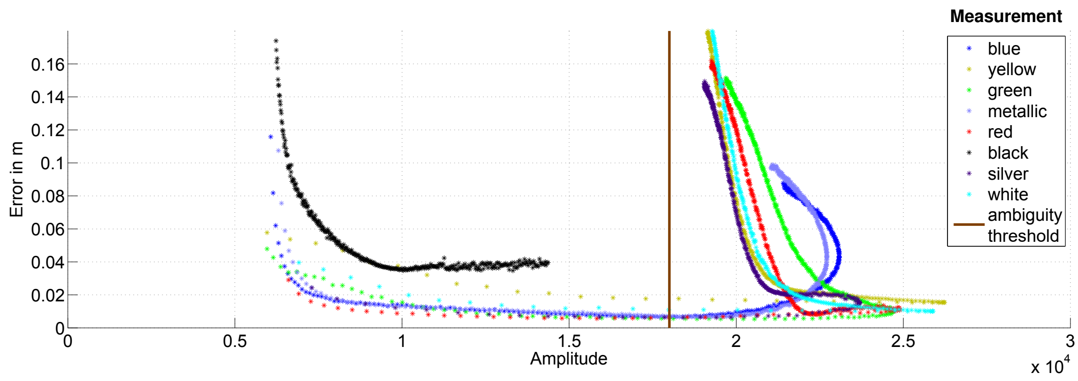


Figure 5.4: Error to amplitude. Amplitude values on the right of the ambiguity threshold (brown line) can have both small and very large errors, but cannot be differentiated due to the ambiguous behavior explained in Sec. 5.2.2 and Fig. 5.3. [HBK15]

In the effort to improve quality by automatic integration time estimation, the error behavior in regard to the integration time and resultant intensity and amplitude has to be analyzed.

Fig. 5.4 shows the correlation between the average error and the amplitude. Amplitude values above 18000 can have both small and very large errors, but cannot be differentiated due to the ambiguous behavior explained in Sec. 5.2.2 and Fig. 5.3. However, it can be seen that for unambiguous amplitude values, the error is smallest between 10000 and 18000. This is true for all measurements, even black.
May et al. show that the "most precise mean accuracy could be acquired with an

integration time located near the amplitudes maxima" [MWSP06, p.3]. This is true for both their Swiss Ranger SR-2 and the used PMD CamCube 3.0, however, as already shown in Fig. 5.3 and Fig. 5.4, values near the maxima are ambiguous and thus cannot be used. May et al. also omit these values but attribute this to the fact "that the image has a non-neglective saturation at the amplitudes maximum" [MWSP06, p.4].
Choosing an amplitude a bit too low only causes the error to rise slightly while choosing an amplitude too large causes values to exceed 18000. Such values cannot be differentiated between being only slightly or strongly oversaturated (see Fig. 5.4) and may carry large errors.
It can be argued that every value in-between is a viable candidate for the optimal amplitude. E.g., if omitting a substantial portion of the image due to oversaturation in order to optimize the remaining values is conceivable, choosing a value close to 18000 is adequate. However, if preventing oversaturation is the main goal, a value close to 10000 is better suited for this task.

## 5.2.4 Amplitude-Intensity Mapping

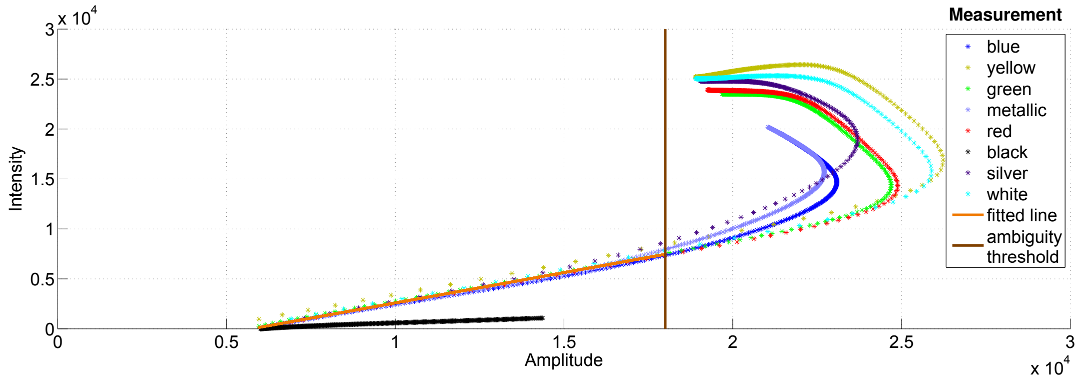


Figure 5.5: Intensity to amplitude. In non-oversaturated areas, the intensity and amplitude are directly proportional to one another. The orange line represents the mapping function between amplitude and intensity values. Values to the right of the ambiguity threshold (brown line) are considered ambiguous (see Fig. 5.3). [HBK15]

Fig. 5.5 shows the correlation between the intensity and the amplitude. It can be seen that there is a linear relationship between the amplitude and the intensity, as long as values stay within ranges outside of oversaturation (amplitude ≤ 18000). By fitting a line (orange) for amplitude values below 18000 (values above cannot be mapped by a function as they are ambiguous; see Fig. 5.3 ), a mapping between the amplitude and intensity is established. The black measurement exhibits a unique behavior that distinguishes it from the other colors. This stems from its unique intensity behavior that was already observed in Fig. 5.2. Since this behavior is an exception and cannot be easily compensated, the black measurement is not applicable for amplitude-intensity matching and thus omitted from the fitting process.

### 5.2.5 Intensity Correction

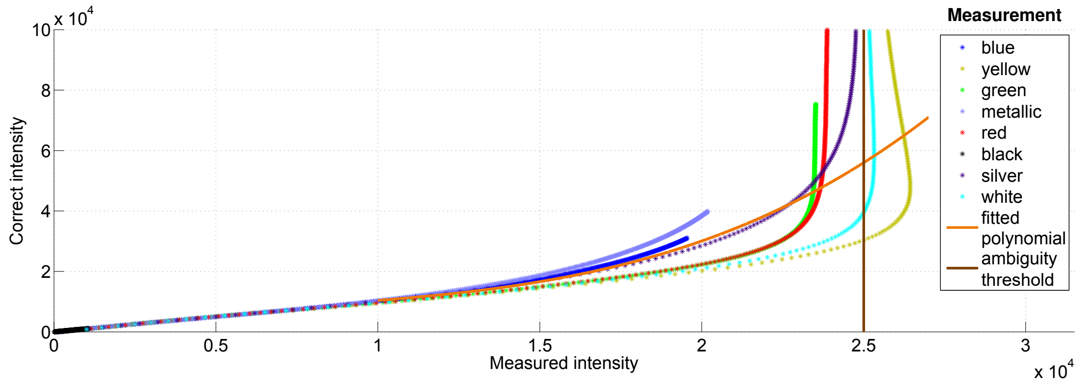


Figure 5.6: Measured intensity to correct (linearized) intensity. The orange line represents the correction polynomial for intensity values. [HBK15]

The correct intensity is defined as the value an intensity would have reached if the oversaturation did not have any effects regarding the linear behavior. Fig. 5.6 shows the correlation between the measured and the correct intensity. The correct intensity is derived from the intensity and its corresponding line that has been fitted to non-oversaturated values (see Fig. 5.2). The difference in values shows the enormous discrepancy between oversaturated measurements and what their values would have been if they had not been oversaturated.
To compensate for this discrepancy, a 3rd-degree polynomial is fit to the data left of the ambiguity threshold (see Fig. 5.6) but only for values above 10000 (see Equation 5.3). This polynomial serves as correction function that allows to approximate an intensity's actual value in case of oversaturation.

## 5.3 The Proposed Method

Optimizing a ToF camera's integration time for the current scene is an important aspect of reducing sensor errors from both under- and oversaturation. Fig. 5.7 shows how the appropriate choice of integration time significantly reduces noise-related errors in a scene.
In this section an approach is presented that chooses the optimal integration time for a ToF camera. It uses the knowledge of sensor specific characteristics and works on arbitrary scenes, which can contain randomly placed objects with different reflectivities.

### 5.3.1 Algorithm

The basic idea of the proposed algorithm is to calculate the optimal integration time for the next frame on a per-pixel basis (expecting that the best possible integration

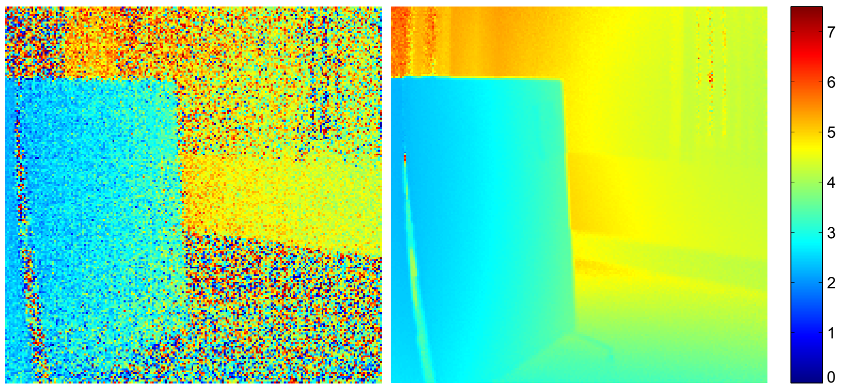


Figure 5.7: The figure shows two different measurements of the same office scene in jet color encoding. The left image uses a low integration time of 50 $\mu$s while the right image uses a better fitting integration time of 2000 $\mu$s). [HBK15]

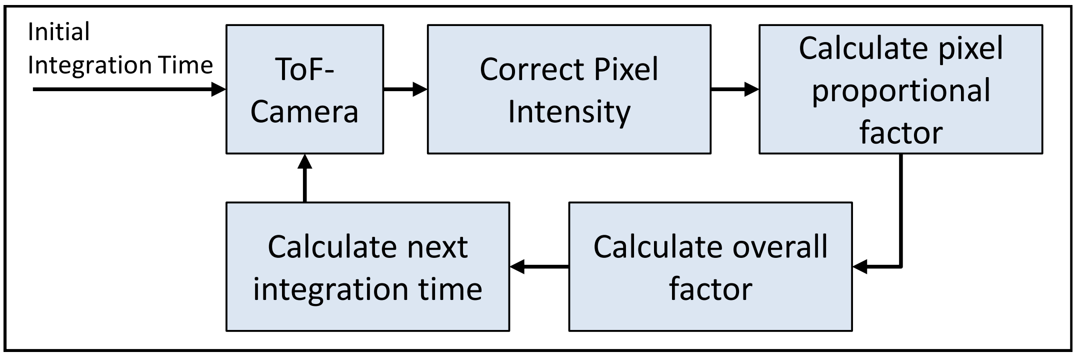


Figure 5.8: The principle schema of the integration time estimation algorithm. [HBK15]

time for the pixel should be achieved) instead of averaging the whole image as prior work does. To accomplish this, the knowledge about specific sensor behavior regarding intensity, amplitude and distance error has been used. The algorithm schema can be found in Fig. 5.8.

Having a look at Fig. 5.2 and Fig. 5.3 it can be seen that neither the intensity nor the amplitude behave linear. The intensity shows to be much less susceptible to ambiguous behavior compared to the amplitude. Also, its linear behavior is much more consistent (see Sec. 5.2.2). Considering this and the deviation from the linear behavior (which can also be described as degree of oversaturation) brings up the main idea of the algorithm. The complex intensity behavior (nonlinear) is mapped to a linear function, which makes it ideal for approximating a desired intensity by estimating a proportional factor (see Sec. 5.2.5).

The amplitude has a direct correlation with the distance error (see Sec. 5.2.3), but only intensities can reliably be approximated not amplitudes. However, there

is a linear correlation between the amplitude and intensity (see Sec. 5.2.4). So, by defining an ideal amplitude, correlating with the smallest error, an ideal intensity can be derived that is to be approximated.
With this knowledge, the optimal integration time (the integration time at which the error is lowest) $t_{\text{opt}}$ per-pixel can be calculated in the following manner:

- Perform **evaluation and preprocessing** steps (optimal amplitude estimation, estimate amplitude/intensity mapping and the intensity linearization function)
- Perform the **online integration time estimation**

### Evaluation and Preprocessing Steps

1. Determine the optimal amplitude $A_{\text{opt}}$, i.e. the amplitude where the error is the lowest. This is explained in detail in Sec. 5.2.3.

2. Determine the optimal intensity $I_{\text{opt}}$ that corresponds to the optimal amplitude $A_{\text{opt}}$ using the linear amplitude-intensity mapping

$$m(x) = \sum_{i=0}^{1} a_i x^i \tag{5.1}$$

$$I_{opt} = m(A_{opt}) \tag{5.2}$$

   a linear function fit to amplitude and intensity data (see Sec. 5.2.4).

3. Determine the intensity correction function

$$h(x) = \sum_{i=0}^{3} a_i x^i \tag{5.3}$$

   a polynomial of 3rd degree, fit to the intensity's measured and correct values. The extraction of correct intensity values and its connection to measured ones is explained in detail in Sec. 5.2.5.

### Online Integration Time Estimation

Having determined $I_{\text{opt}}$, $m$ and $h$ during the preprocessing steps, the optimal integration time $t_{\text{opt}}$ per-pixel can be calculated:

1. Calculate the corrected intensity from the current intensity, using the intensity-correction function:

$$I_{\text{corr},x,y} = h(I_{\text{curr},x,y}) \tag{5.4}$$

2. Calculate the proportional factor:

$$f_{x,y} = \frac{I_{\text{opt}}}{I_{\text{corr},x,y}} \tag{5.5}$$

3. Calculate the optimal integration time:

$$t_{\text{opt},x,y} = f_{x,y} \cdot t_{\text{curr}} \tag{5.6}$$

This yields an individual optimal integration time for each pixel. Now an overall optimal integration time for the whole image can be calculated. For this task a weight map is used, incorporating a compensation for sensor-specific behavior and regarding pixels near the image center as more important than near image borders (see Sec. 5.2.1). The weight map based calculation of the optimal integration time for the whole image is done in the following manner:

1. Calculate the pixel weight from the weight map and a pixel based gain factor:

$$w'_{x,y} = w_{x,y} \cdot g_{x,y} \tag{5.7}$$

2. Calculate the optimal integration time for the whole image as a weighted average of the pixel specific optimal integration times:

$$t_{\text{opt}} = \frac{\sum w'_{x,y} \cdot t_{\text{opt},x,y}}{\sum w'_{x,y}} \tag{5.8}$$

Compared to prior work on integration time estimation, the proposed algorithm has several advantages. It estimates the integration time on a per-pixel basis. This enables one to use only portions of the image or even apply per-pixel weighting, counteracting sensor properties and allowing to adjust importance of certain image regions (see Sec. 5.2.1). Additionally, it uses knowledge gained from an extensive analysis of the underlying inherent sensor behavior regarding intensity, amplitude and distance error. This knowledge is used to minimize the overall error and to prevent oversaturation or at least escape from it quickly. This works well in presence of highly various reflectivities and quick changes in the scene.

## 5.4 Algorithm Evaluation

The previous sections have presented the new algorithm (see Sec. 5.3.1) and depicted an extensive analysis of the underlying inherent sensor behavior (see Sec. 5.2). This section compares the approach to the work proposed by May et al. [MWSP06] and details differences in quality, quantity and adaption speed. For further evaluation purposes, the parameters estimated in Sec. 5.2 are used. These parameters can be found in Table 5.1.

| Parameter | Value(s) |
|---|---|
| Optimal amplitude ($A_{\text{opt}}$) | 18000 |
| Optimal intensity ($I_{\text{opt}}$) | 7353 |
| Amplitude-intensity-mapping function ($m(x)$) | 0.6067281 -3475.016<br>$6000 \leq x \leq 18000$ |
| Intensity-correction function ($h(x)$) | 2.2026e-10 4.417871e-05<br>0.4276858 1100.069<br>$10000 \leq x \leq 25000$ |
| Gain factor for all pixels | 1.11 |

Table 5.1: The table shows the estimated sensor and correction parameters. [HBK15]

### 5.4.1 Evaluation Method

Two kinds of evaluation are performed. In the first measurement, a static scene with metal plates of different colors/reflectivities (black, metallic, blue, green, red, silver, yellow and white) in a distance of approximately 1 $m$ (see Sec. 5.4.2) is used. However, since the results have been very similar for all plates, the plots for the red plate are presented exemplarily. In the second measurement, an arbitrary office scene with the camera orientation changing in-between every frame by about 15° around the x-, y- and/or z-axis, representing a highly dynamic scene scenario (see Sec. 5.4.3) has been recorded.
For both scenes, each frame has been recorded with the full spectrum of integration times, ranging from 50 to 8000 $\mu$s. This allows one to use a scene for reproducible tests with different and differently parametrized auto integration time estimation approaches. The evaluation scheme chooses the integration time for the next frame according to the optimal integration time calculated in the previous frame. To achieve a fair and meaningful comparison, the simulation has been performed with various initial integration times.

### 5.4.2 Static Metal Plate Scene Evaluation

Fig. 5.9 compares the approach of May et al. (left column) to the new algorithm (right column). The static scene has been recorded for 20 frames. Five different initial integration times have been used, covering strong under- and oversaturation as well as average saturations. The development of the integration times, the mean error to the ground-truth (see Sec. 5.2) and also the number of well saturated values (amplitude values between 250 and 18000) in the image is compared over the course of the algorithm. It can be seen that May's algorithm slowly converges to the optimum after about 8 frames while this approach is already close to the optimum after 3 frames. Additionally it can be seen, that the new algorithm has a smaller mean error (0.013 $m$) compared to May et al. (0.027 $m$). Also the number of well saturated pixels is higher for the proposed algorithm (19000) compared to May et al. (17000).

The approach's fast adaption to the proper integration time, especially in areas of oversaturation (the first 3 frames) and the resultant lower error can be attributed to the fact that the intensity values are corrected before calculating the proportional factors.

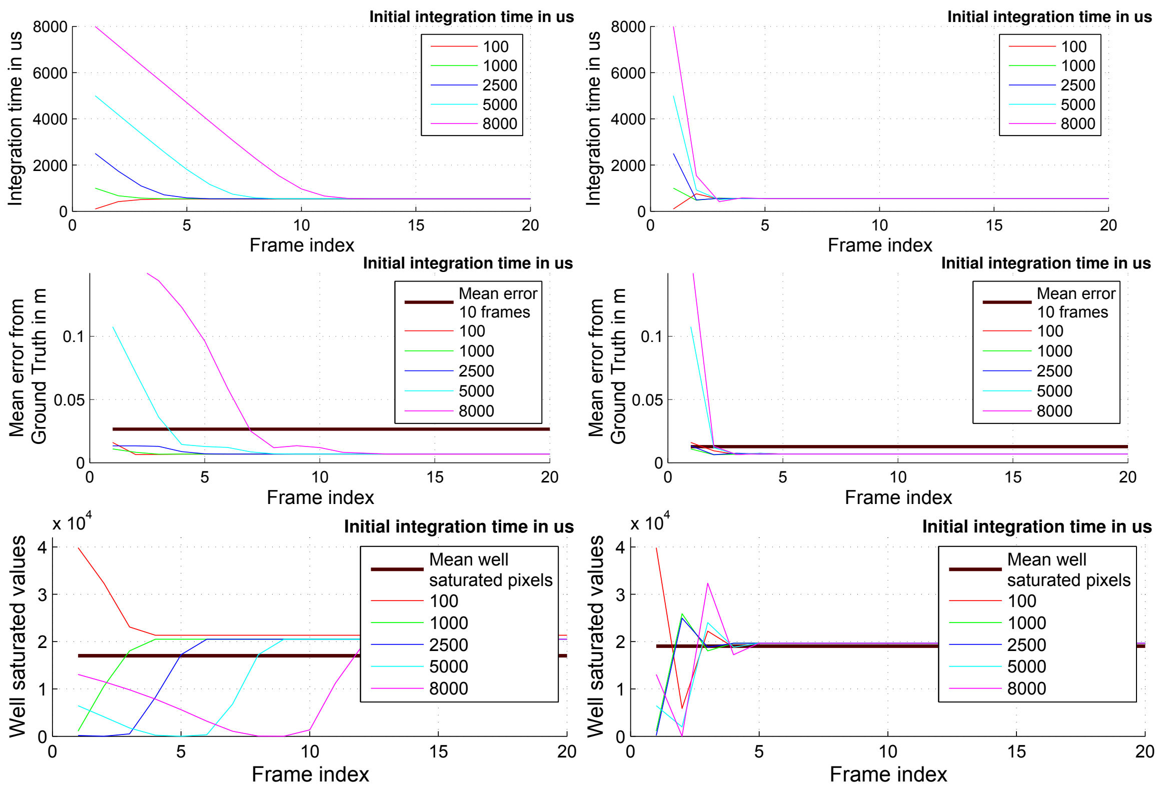


Figure 5.9: Comparison between the approach of May et al. (left column) and the new approach (right column). A static scene capturing a red metal plate from a distance of 1 *m* is recorded for 20 frames. The integration time (top), the mean error (center) and the number of well saturated pixels (bottom) are compared. [HBK15]

### 5.4.3 Dynamic Scene Evaluation

Fig. 5.10 compares the approach of May et al. (left column) to the new algorithm (right column) in the highly dynamic office scene over the course of 35 frames and with 5 different initial integration times. The novel approach converges within 8 frames to the global optimal integration time, while the algorithm of May et al. needs up to 20 frames. However, the mean error is approximately the same for both approaches (0.047 *m*). The number of well saturated pixels is higher for the new algorithm (35700) compared to May et al. (33500).

This shows several things. As explained in Sec. 5.2.1, if oversaturation causes a growth in error to the same degree as non-optimal saturation, then optimizing areas

where most pixels reside, namely image borders, will actually reduce the overall error. Also, just because amplitudes reach beyond the ambiguity threshold, they are not necessarily oversaturated and thus carry only small errors.
Overall this new approach shows a faster adaptability, especially in oversaturated scenes, which can, like with the static scene, be attributed to the intensity correction.

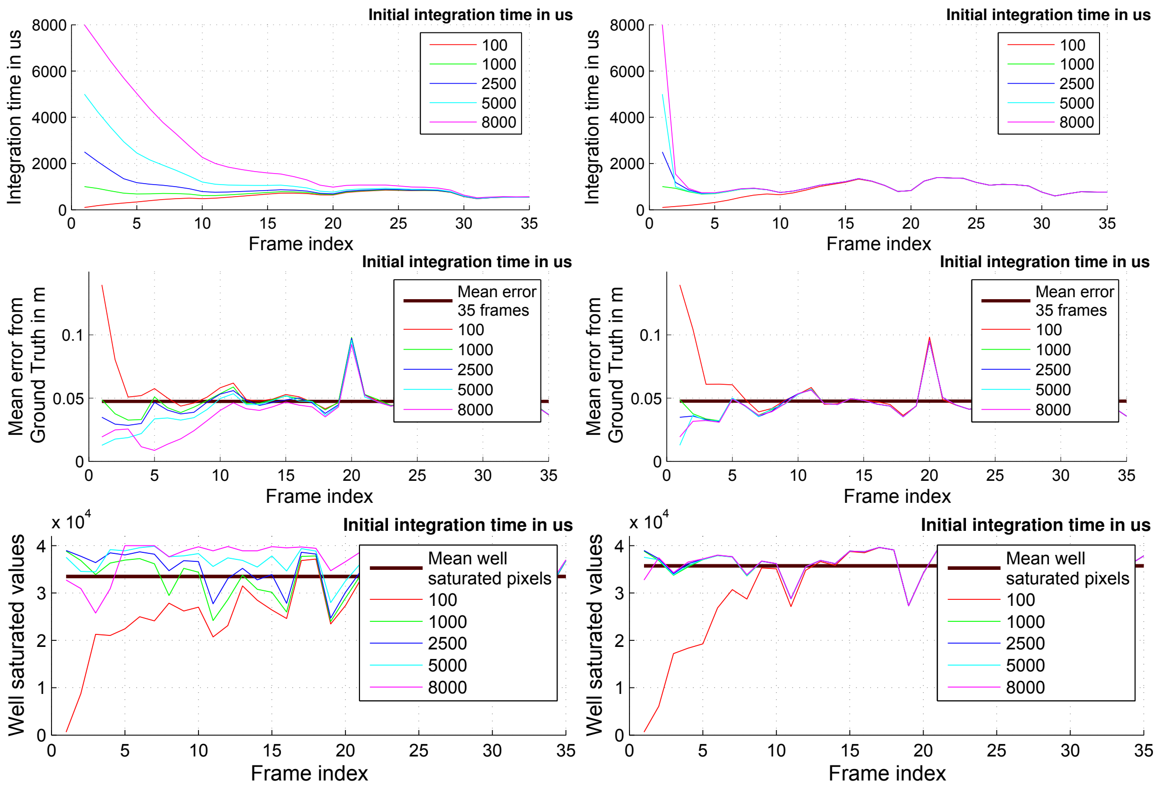


Figure 5.10: Comparison between the approach of May et al. (left column) and the new approach (right column). A highly dynamic office scene is recorded for 35 frames. The integration time (top), the mean error (center) and the number of well saturated pixels (bottom) are compared. [HBK15]

## 5.5 Summary

This chapter presented a novel online integration time adaption algorithm that works on a per-pixel basis and uses knowledge gained from an extensive analysis of the underlying inherent sensor behavior regarding intensity, amplitude and distance error to reduce the overall error, to prevent oversaturation and to minimize the adaption time. It also works well in presence of various reflectivities and quick changes in the scene. The per-pixel character enables one to use only portions of the image or even apply pixel-specific weighting, counteracting sensor properties (e.g. spatial intensity

distribution) and allowing to adjust importance of certain image regions. Overall, this represents a significant improvement over previous methods. Furthermore an evaluation scheme has been introduced that allows to perform reproducible and comparable tests with different and differently parametrized auto integration time estimation approaches.
Future work will concentrate on applying and optimizing this method to other ToF sensors.

# 6 3D Car Reconstruction – *An Industrial Application*

The global development regarding new vehicle registrations clearly shows an increasing demand towards the automotive service infrastructure. In parallel, economic issues put more and more pressure onto producers of electromechanical engineering systems to offer sustainable and efficient systems. This trend can for example be seen in the final report of the Automechanika, Frankfurt 2012 [Fra16]. Thus, to be able to persist in the market of car wash systems, appropriate technical innovations are required in order to improve on *effectivity*, i.e. the quality of the washing result, *efficiency*, i.e. the amount of employed energy, water, detergent and time, as well as *safety*, i.e. prevention of damages to the car and/or the washing system (e.g. due to car superstructures).

Current car wash systems are generally controlled by light barriers and power measurement sensors. The horizontal fixtures and fittings are controlled by the usage of sensors. These sensors are all directly placed on the movable parts of the system, e.g. the height control of the roof brush and the dryer are realized with light barriers. In contrast to the horizontal brushes, the position of the vertical brushes is regulated with the contact pressure by monitoring the current consumption of the brush unit. A higher contact pressure implies a higher current consumption of the brush unit. Actual values lower than the nominal value move the brush towards the vehicle. Crossing the nominal value moves the brush away from it, thus implementing an inertial regulating system. This approach is not optimal in many ways. For example, geometric variations in the car profile can not properly be reflected by a single brush pressure force, resulting in either imperfect cleaning results or increased risks of damages [AG16].

To achieve improvements in terms of effectivity, efficiency and safety, this section proposes a concept for an automatic, contactless online 3D measurement of vehicles. Knowing the car's shape in advance allows for a "global" optimization of the wash process. The approach utilizes *ToF cameras* for online optical distance measurement. Compared to other industrial suited range sensing systems like laser-scanners, ToF cameras provide fully lateral 3D information at high frame rates, additional grayscale information and full eye safety. Furthermore, cheap industrial cameras, that can handle the difficult environment conditions (water and detergent), are still under development. [IFM16]

The technological and scientific challenges are in the scope of the camera setup and data processing to fulfill the required accuracy of at least 5 to 10 cm and also to process the data in real time.

This work focuses on the calibration process and the extensive data preprocessing required due to reflections (diffuse, specular) and environmental influences (light, wetness, temperature). Furthermore, a prototype 3D reconstruction system, that can be used in a car wash system, is proposed and prepared.

---

*Publication: ToF Camera Based 3D Point Cloud Reconstruction of a Car [HLK13b]*

## 6.1 System Overview

The acquisition system is composed of three electronically synchronized ToF cameras (PMD camcube 3.0), all mounted on an arch (see Fig. 6.1). Each camera observes different parts of the passing car (left, right and top) and provides images at a rate of 20 frames per second with a resolution of $200 \times 200$ pixels (exposure time between 2 ms and 10 ms).

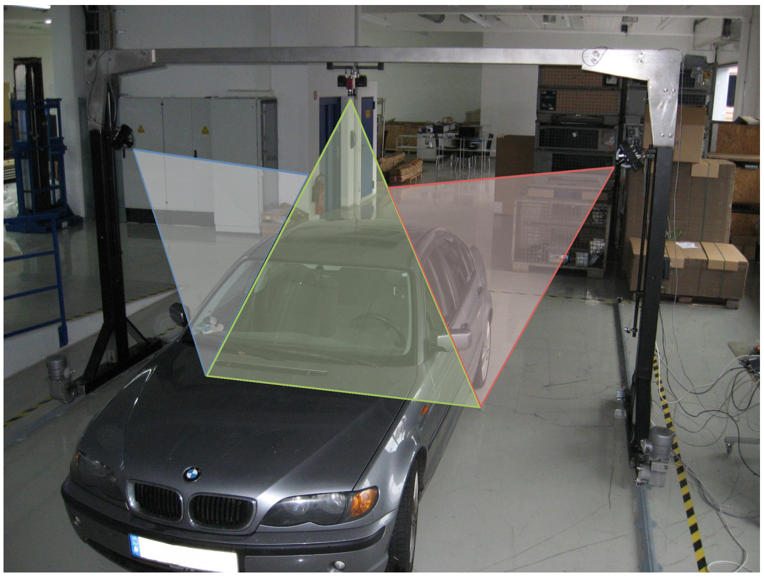


Figure 6.1: Hardware set-up of the current prototype. Left, top and right: mounted PMD cameras. Top: additional RGB camera mounted for the extrinsic calibration. [HLK13b]

Note that an additional RGB camera was fixed to the top mounted $\text{PMD}_{\text{top}}$ camera for the extrinsic calibration process. Further information can be found in section Sec. 2.1.2.1.

The ToF data processing concept comprises several steps (see also Fig. 6.2):

1. **Acquisition** from three synchronized ToF cameras.
2. **Data Preprocessing**, i.e. optimization and segmentation of the ToF data.
3. **Fusion** of the three point clouds.
4. **Registration** of the acquired image sequences.
5. **Accumulation** of the registered point data to the accumulated data from prior frames.
6. **Reconstruction** of the final point cloud.

The data processing modules are mainly implemented using CUDA to fulfil the above stated performance requirements.

## 6.2 Contribution

The contribution of this work is twofold. On the *system level*, a novel approach for online acquisition and reconstruction of the outer vehicle hull is presented. The key features of the system are the integration of three active range sensing ToF cameras based on the PMD principle [PMD16], an appropriate preprocessing of the sensor data, registration, data fusion and geometry extraction (Fig. 6.2). All data processing is done on GPUs in order to achieve a fast reconstruction.

On the *technical level*, this work presents the following contributions, which are mainly due to the specific data characteristics present in the application scenario:

- a data preprocessing concept, including handling of interferences caused by the multiple concurrent ToF measurements (see Sec. 6.5.1),
- robust outlier detection and segmentation in the raw range images in the presence of a comparably high noise level and strongly varying reflectivities (see Sec. 6.5.1.2 and Sec. 6.5.2),
- registration of low resolution data using a multiple ToF camera setup (see Sec. 6.5.1),
- an analysis of PMD cameras in a multi camera setup with an industrial background (Sec. 6.6) and
- a full 3D data scan of moving objects (for example a car as presented here, see Sec. 6.7).

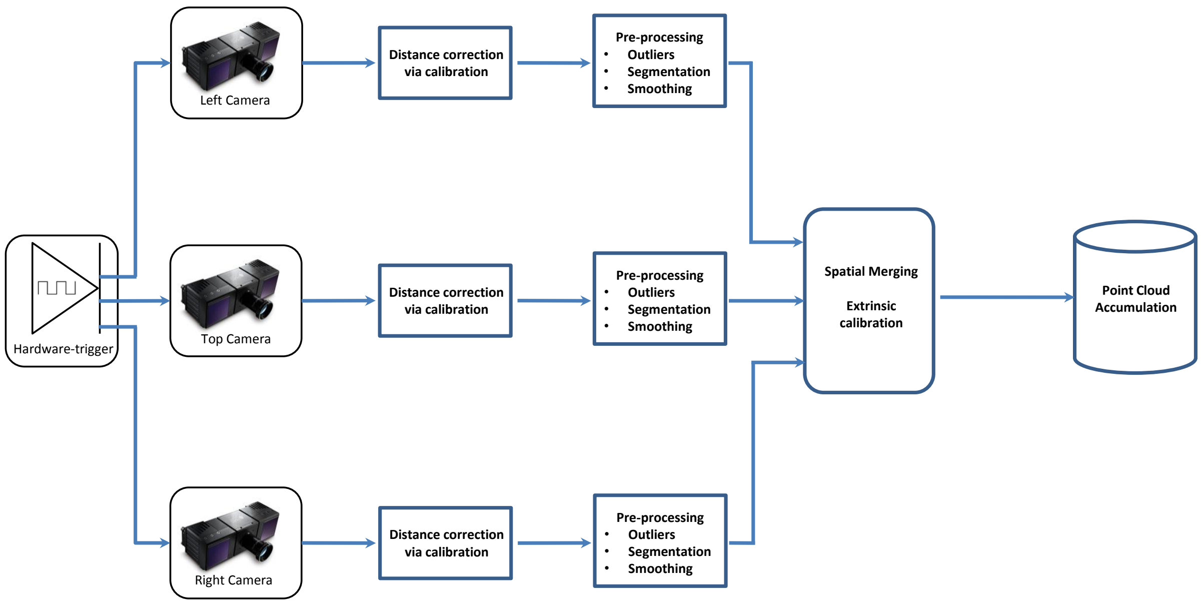


Figure 6.2: Data processing pipeline. [HLK13b]

## 6.3 Related Work

In this section, the prior work related to 3D reconstruction in general and specifically to correspondence finding and registration is presented.

3D reconstruction of real world objects has been a well studied field of research for more than two decades. Thus, many approaches are available to perform image registration by handling multiple point sets and images. State of the art approaches are divided in two kinds of image and point cloud-registration, using either a *global* or *local matching metric* [Bes88, BM92, BSGL96].

*Global matching metrics* try to minimize the matching error over the whole data set. This leads to a global optimal registration transformation for the new incoming data, but in general it is slow. In contrast, the *local matching metric* tries to find an optimal transformation just by comparing spatial data. If the starting point for the registration is well known, the local matching can result in a good solution and is normally much faster than a global metric. However, without a good starting position, it leads to false matching and misalignment.

In this work, a local metric is used to perform the point cloud registration, with the assumption of a small motion between two consecutive PMD frames.

*Image registration* methods have been studied for years [Bro92, ZF03]. The majority of the methods are implemented using the four steps shown in [ZF03]: feature detection, feature matching, transform model estimation and image resampling.

In literature, for *image based feature detection*, many different methods for detection, description and matching of features are available. Well known and often used detectors and descriptors are the Harris Corner Detector [HS88], Good features to track [ST94], SIFT (Scale Invariant Feature Transform) [Low04b], SURF (Speeded-Up Robust Features) [BETG08], BRIEF (Binary Robust Independent Elementary Features) [CLSF10] and Optical Flow [HS81, LK81].

This work shows that these detectors are not working well for this situation due to the missing corners and edges in the design of cars (see Sec. 6.7.1).

*Object based registration methods*, are usually based on point set merging. The commonly used algorithm is the ICP introduced by Besl [BM92]. It formulates the matching process as an iterative least square minimization problem by finding the minimal error between all corresponding points of two point clouds (point-to-point metric, see Sec. 2.3.1). Depending on the initialization, the algorithm converges to good results in the local neighborhood. One of the biggest disadvantages is the brute force finding of point correspondences. In Zhang [Zha94] the search was sped up by the usage of a k-d tree.

A more efficient version of the ICP was introduced by Chen [CM92] using a point-to-plane metric (see Sec. 2.3.2 and [Low04a]).

Under the assumption that the displacement and angle between the source and destination surface is small, this ICP version converges much faster than the point-to-point ICP [Low04a]. A more detailed description can be found in Sec. 6.5.3.3.

Based on these two kinds of ICP, many approaches to construct full point clouds and 3D meshes are described in literature [RHHL02, CSC+10, Sim96]. [IKH+11] and [NIH+11] proposed a very interesting version of the ICP (KinectFusion) using a Microsoft Kinect camera [Mic16c].

This work focuses the point cloud registration on the algorithm published in [IKH+11, NIH+11] and extends it by a preprocessing pipeline which can be found in Sec. 6.5.1.

## 6.4 System Challenges

In the following sections an overview on hardware and application specific challenges is given to set up a concept of an online 3D reconstruction system for the acquisition of cars. A real, industrial system has different challenges compared to lab-systems, which are designed to work under controlled environmental conditions. Many specific tasks have to be considered and restrictions are often not acceptable. E.g. the assumption of a static background, equal lighting conditions or linear motion do not hold in real situations. The next sections give an overview of special application related challenges.

### 6.4.1 Car Materials

An important issue when measuring and reconstructing cars is the kind of materials and surfaces that occur. Statistically, the white, silver and black paintings are the preferred colors (especially metalloid and glossy effects) over the past years. This leads to the problem of selecting proper exposure times for the ToF cameras, i.e. dark/black cars need long exposure times in order to get sufficient signals, bright cars need short exposure times in order to prevent oversaturation effects.

Additionally, other materials frequently used for cars are reflectors, commonly used in car lights and glass. Reflectors reflect the ToF camera's incoming active infrared light, which may lead to oversaturation of this area. Glass, on the other hand, is transparent to the infrared light, thus the camera sees the interior of the car which leads to additional unwanted and noisy data.

As cars are made of the previously mentioned materials and coatings, it is necessary to find an optimal integration time to get as much reliable data as possible. (For this work an experimental determined integration time value of 2500 $\mu$s has been used; see also Chapter 5)[1].

Another big problem is the design of cars. In general smooth surfaces are used to model the geometry. However, state-of-the-art 2D intensity and color based image registration algorithms like SIFT, SURF or Optical Flow rely on sufficient edges and corners in order to detect features robustly. See also Sec. 6.5.1.1, Sec. 6.5.3 and Sec. 6.7.1.

### 6.4.2 Car Wash Environment

As the car moves under the camera arch, motion artifacts occur. Another big challenge are the random environment conditions that occur in car wash systems. Beside spray and detergent in the air, reflecting materials such as metal parts in the background, ground grids on the floor or moving objects in the background are influencing the sensor and cause additional noise or may confuse the system. These challenges are solved by good segmentation and preprocessing as e.g. the motion compensation (see Chapter 4, Sec. 6.5.1 and Sec. 6.5.2).

### 6.4.3 Performance

To be able to bring the system to market, beside the price, the performance of the system is one of the most important points. It is not acceptable if the washing time is extended. At least it should be equal or faster. So the main requirement is a real-time system that provides the prepared data as fast as possible to the Programmable Logic Controller (PLC).

[1]Note: The Automatic Integration Time Estimation Algorithm has been development as a requirement from the car reconstruction project afterwards.

## 6.5 Data Processing Concept

The goal of the data processing stage is to build a geometric representation of the car by analyzing the data from the three ToF cameras. For this purpose, the basic calibration methods are applied to each camera input frame to reduce systematic errors (see Sec. 2.1.2). Afterwards, advanced preprocessing methods that further improve the raw data are used (see Sec. 6.5.1).

The corrected and improved data is then segmented into relevant parts (the car) and irrelevant parts (background); see Sec. 6.5.2. The relevant parts of the three input data frames are then passed to the transformation step, where they are put into a single coordinate system using the extrinsic data from the initial system calibration (see Sec. 6.5.3). This representation of the three merged camera frames is then passed to an accumulation step that transforms and merges all input data into the single model of the car (see Sec. 6.5.4.2).

### 6.5.1 Preprocessing Stages

Preprocessing of ToF depth data is the basis for a successful registration and merging. Related to the way how ToF cameras acquire their depth data (extraction of four phase images), an initial motion compensation is required (see Sec. 6.5.1.1). Furthermore, the depth data contains many outliers, mainly due to noise and so called flying pixels (see Sec. 6.5.1.3). Flying pixels occur in inhomogeneous areas as for example in edge jumps between foreground and background objects. The sensor area covers more than one distance which results in a distance in between of them (see upper part of Fig. 6.8). Lindner [LLK08] and Sabov [SK10] introduce approaches to reduce these error pixels. Beside noise and inhomogeneities another issue is the interference between the independent light sources of the individual cameras (See Fig. 6.3 and Fig. 6.4). The here presented algorithms are sequentially applied in this order: Motion Compensation, Median Filter, Gaussian Filter, Outlier Removal.

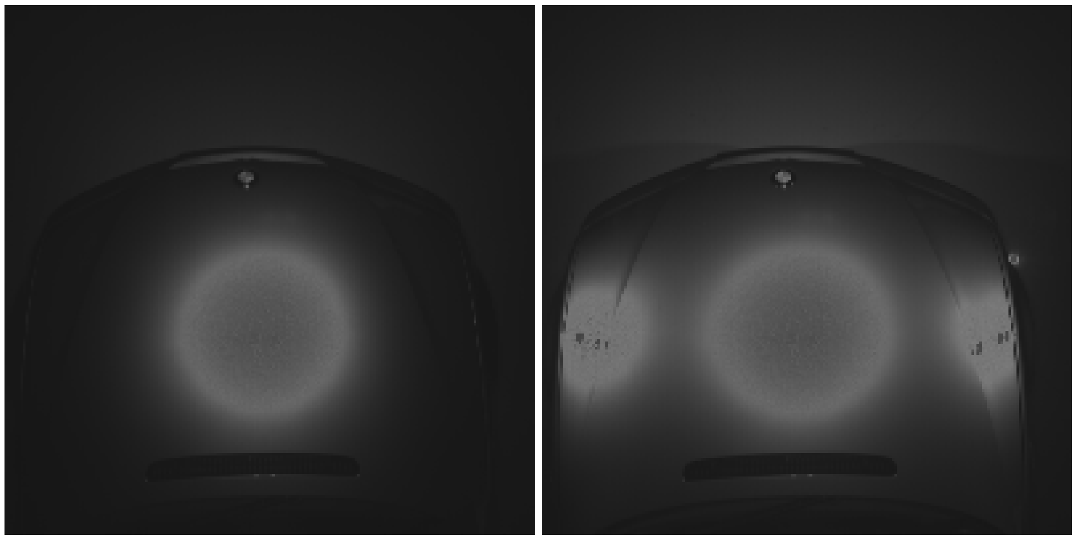


Figure 6.3: Left: single camera data acquisition (intensity image). Here saturation is present in areas which directly reflect the light (white/light gray areas). Right: acquisition of multiple cameras (intensity image). Overexposure is also visible in areas on the left and right side. [HLK13b]

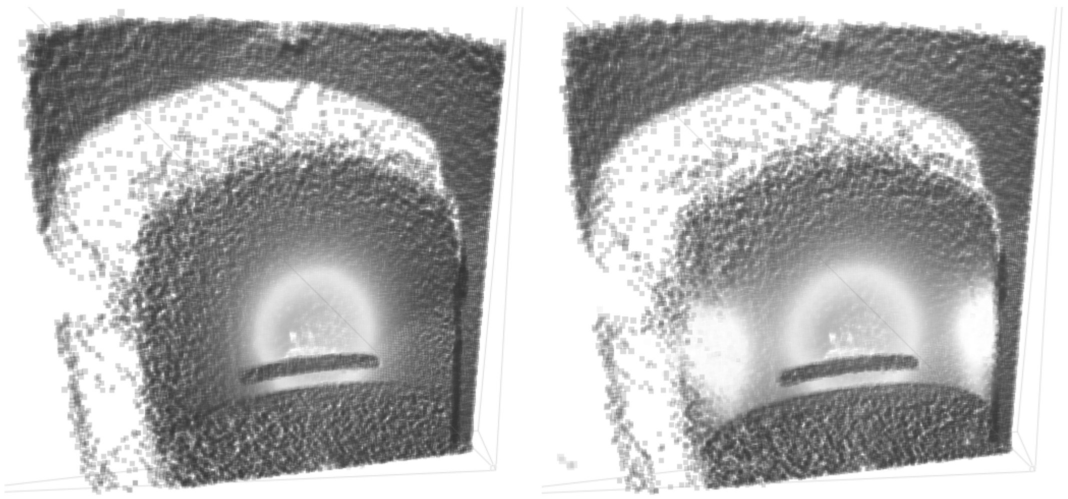


Figure 6.4: The two images show the depth data acquisition result of a single camera; Left: distance data acquisition with only one active camera. Oversaturation is only visible at the center. Right: distance data acquisition with multiple active cameras. The oversaturation is also visible on the left and right side. The impact of the interference in the distance measurements can be seen in the white holes in the right image. [HLK13b]

#### 6.5.1.1 Motion Artifacts and Compensation

As seen previously, the PMD technology works in the way that four subsequent images are acquired. It leads to motion artifacts in dynamic scenes (induced by object or camera movement). In this application the car is thus sampled four times, each time at a different location due to its lateral motion.

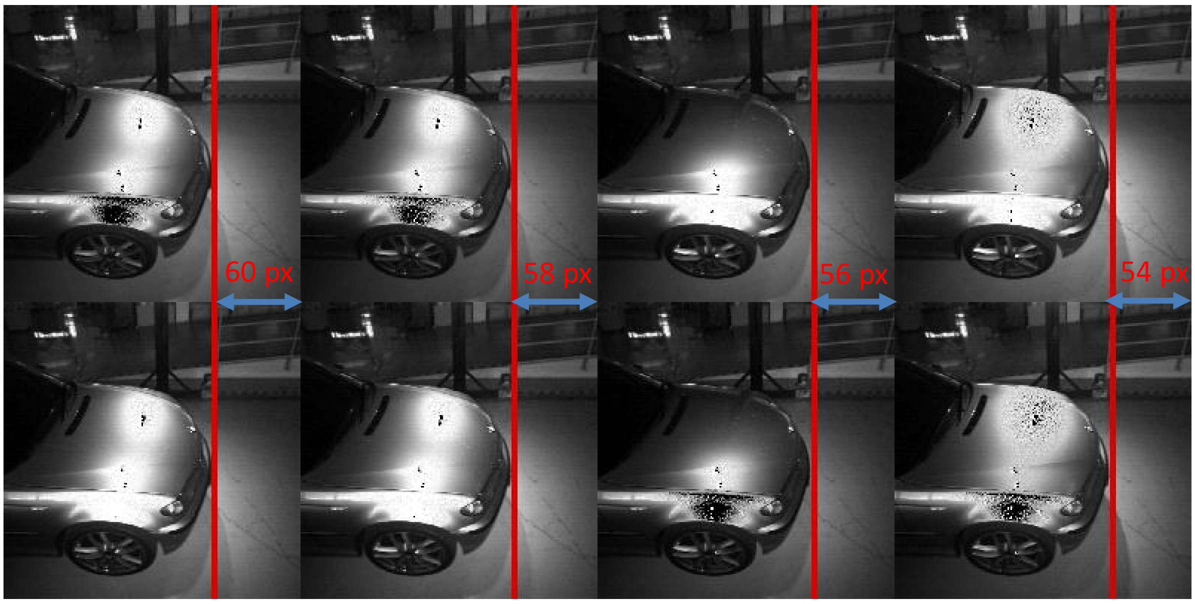


Figure 6.5: The four phase images with marked movements of the car (about 6 pixels between phase 1 and 4). Top: phase channels $P_{A1} - P_{A4}$. Bottom: phase channels $P_{B1} - P_{B4}$. [HLK13b]

Motion artifact correction was introduced by Lindner [LK09] using an Optical Flow approach. It works according to the following principle: the algorithm tries to track surface points (on a per pixel basis) between all the phase images of the PMD image (see Fig. 6.5). This allows to determine the correct phase values used for the distance calculation. A detailed explanation can be found in *Chapter 4*.

The PMD sensor measures two different raw images at the same time: a shifted reference signal $P_{A_i}$ and the inverted signal image $P_{B_i}$ (Sec. 2.1.1). Internally these two signals are subtracted and give the resulting phase image $P_i$ (see Fig. 6.5).

The Anisotropic Huber-L1 Optical Flow (see [WTP$^+$09]) is applied to the raw intensity values ($P_{A_i} + P_{B_i}$) by estimating the flow between the reference phase image $P_0^+$ and the images $P_1^+$ - $P_3^+$. The raw values are then resampled according to the calculated flow vectors and then processed by the standard PMD demodulation scheme. The result can be seen in Fig. 6.6.

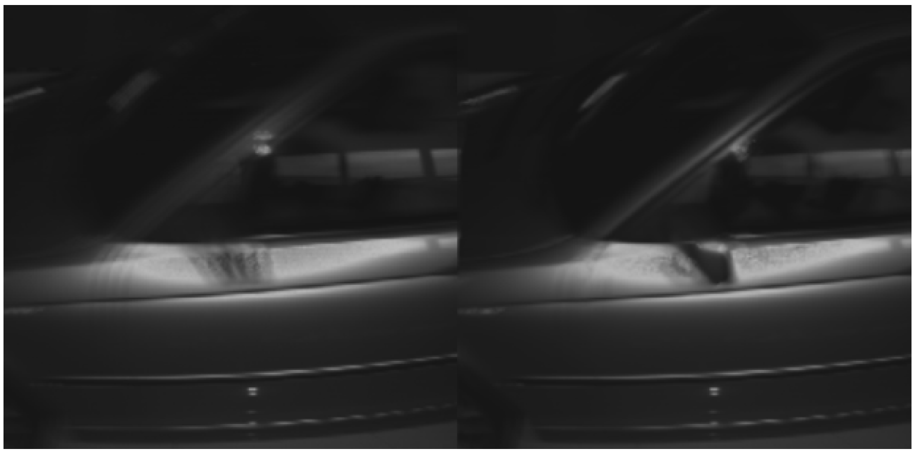


Figure 6.6: Left: motion artifacts of the moving car, e.g. at the mirror (intensity image). Right: successful motion compensation. No more artifacts are visible at e.g. the mirror (intensity image). [HLK13b]

#### 6.5.1.2 Noise Reduction

An important and non-negligible problem is the noise of a PMD sensor. Beside the Fixed Pattern Noise (FPN) [FAT11, HHE11], unreliable depth measurements occur due to low (small SNR, undersaturation) or high amplitudes (oversaturation). The rest of the noise is usually interpreted as white Gaussian noise [FB07].

**Fixed Pattern Noise** Fixed-pattern-noise [HHE11] is an offset (additive noise, bias) and gain (multiplicative noise) error in the PMD pixel. The gain factor can be handled by the wiggling correction [HHE11]. The additive noise can be removed using the so-called black image unique for each chip. The black image is then subtracted from each acquired PMD image.

**Under- and Oversaturation** Low and high amplitude values have an important impact on the quality of the sensor measurements. Low amplitudes (i.e. undersaturation) are caused for various reasons. One reason is due to a low reflective Near Infrared (NIR)-light area that is currently covered by a sensor pixel. As described in Sec. 6.4.1 a low reflective material has a high absorption rate which leads to a reduced amount of reflected light. Using an averaged integration time (to get the best exposure for the whole scene), a low amplitude will be measured for those pixels. The second reason is the light attenuation due to large distances between objects and camera. Areas far away from the camera, or distances greater than the unambiguous range of the PMD camera (see Sec. 2.1.1) result in low amplitudes. Due to the high noise level of PMD pixels and the low amplitude values, the Signal-to-Noise Ratio is very low. The counterpart is oversaturation. It leads to high amplitude values in these areas. Here the raw values of the sensor are considered to identify overflow. Both can basically identified using thresholding [Rap07]. Thus the measured distances from low and high amplitude pixels are considered to be unreliable and can be removed with this method:

$$O_{x,y} = A_{x,y} < \theta_{min\_amplitude} \; OR \; A_{x,y} > \theta_{max\_amplitude} \tag{6.1}$$

where $(x, y)$ is a pixel coordinate, $O_{x,y}$ the outlier mask, $A_{x,y}$ the measured amplitude, $\theta_{min\_amplitude}$ the adaptive minimal and $\theta_{max\_amplitude}$ the maximal allowed amplitude for the given environment.

As described in the introduction of this section, even if different modulation frequencies are used for all cameras (e.g. 19 MHz, 20 MHz, 21 MHz), interference influences will be still present for a multiple ToF camera system. The result is an oversaturation in all illuminated areas (see Fig. 6.3 and Fig. 6.4). These areas can also be determined and segmented using the simple thresholding as described above. An other method (Automatic Integration Time Estimation) to reduce under- and oversaturation effects has been shown in Chapter 5.

#### 6.5.1.3 Outlier Detection and Removal

**Outliers** Beside under- and oversaturated pixels, outliers are also related to noise, moving objects in the scene, texture/material changes or flying pixels.

In many cases they can be detected and removed using neighboring pixel information.

**Median** Another well working algorithm to remove outlier is the median filter. Like the previously discussed algorithms it also takes the neighborhood into account. A sliding window with a fixed size of e.g. 5 x 5 pixels is used to perform the filtering.

The median is a good way to eliminate outliers. It is robust towards single outliers within the window area, due to the fact that the high and low values are moved (sorted) to the start and end of the value sequence.

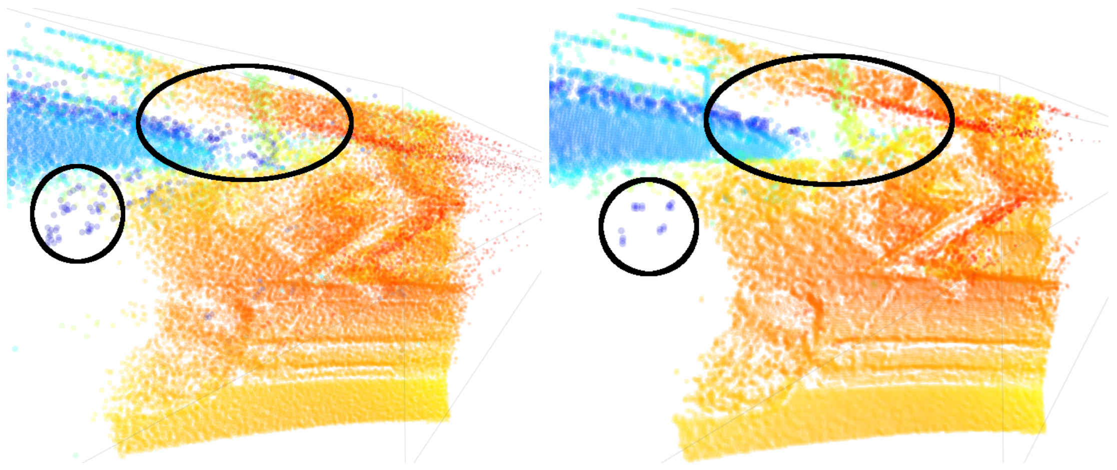


Figure 6.7: Left: no median filter applied. Right: same image as left with 5 x 5 median filter applied. [HLK13b]

**Outlier Removal Filter** Another simple outlier removal method is a threshold based method on a 3×3 neighborhood check. The binary outlier classification function $f_{outlier}$ is defined as following:

$$f_{outlier}(i) = \begin{cases} 0 & \text{if} \left| D_i - D_{x,y} \right| < \theta_{range} \\ 1 & \text{else} \end{cases}$$

$D$ are distance values and $\theta_{range}$ is the outlier threshold. A good, experimentally determined threshold for this application is about 0.05 m.

$$O_{x,y} = \begin{cases} \text{true} & \text{if } count(\sum_{i=1}^{8} f_{outlier}(i)) > \theta_{neighbors} \\ \text{false} & \text{else} \end{cases}$$

$O_{x,y}$ defines a possible outlier at position $x, y$ while $\theta_{neighbors}$ defines the minimum number of pixels similar to the currently evaluated pixel $D_{x,y}$. If the minimum number of neighbors $\theta_{neighbors}$ (in this application $\theta >= 3$) is reached, the current evaluated pixel is marked as a valid pixel. Note that this approach also eliminates most of the flying pixels (see Fig. 6.7 and Fig. 6.8).

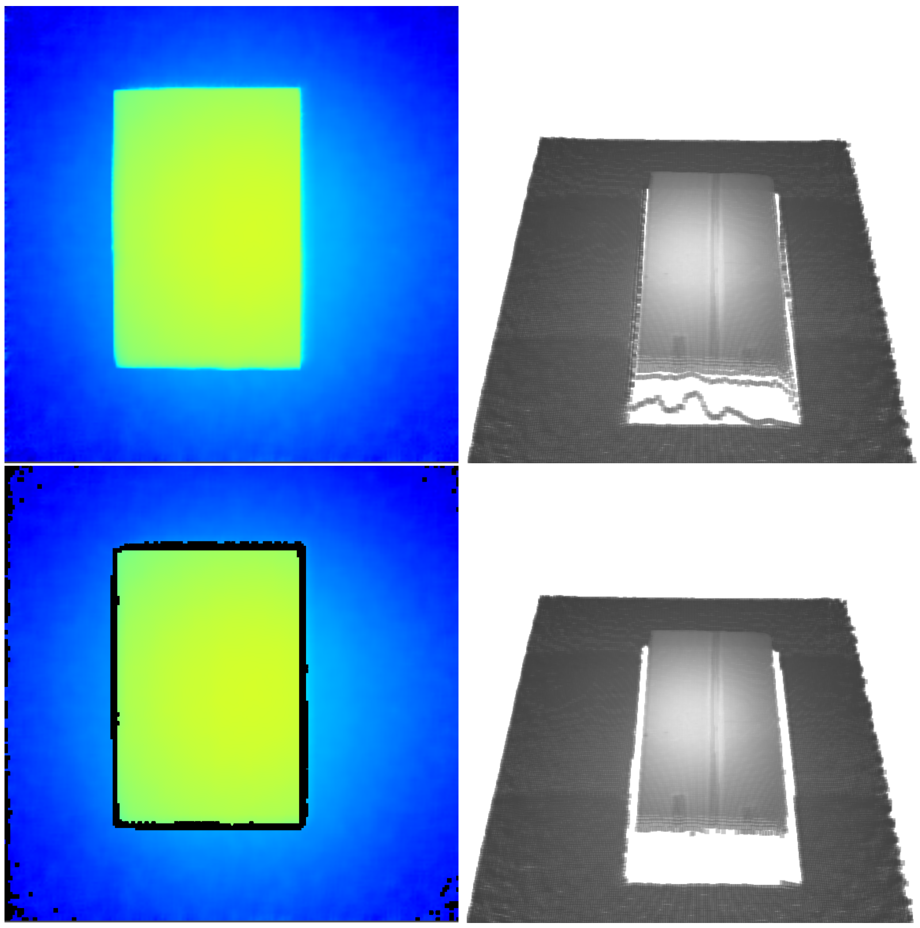


Figure 6.8: Top: calibrated depth data of a simple box acquired from PMD$_{top}$-camera (2D color jet map and associated 3D point cloud representation). Bottom: outlier removal applied (outliers are marked black in the 2D image). [HLK13b]

The outlier removal is performed in a second step. After median and Gaussian filtering, outliers such as flying pixels are still present. The distance thresholded outlier filtering presented here removes these remaining points as can be seen in Fig. 6.8.

**Smoothing** Due to the noise influence in the depth values, values around the center pixel remain often unstable. Assuming that a single pixel is part of a surface, distance changes in the neighborhood indicate edges. If the pixel cannot uniquely be assigned to one of these edges, it should be corrected in a way that it is assignable to e.g. the closest surface. This assumption makes spatial denoising necessary and has been studied for decades: e.g. diffusion techniques [PM90], wavelet methods [RKN00] or smoothing using Gaussian filters. Due to its simplicity compared to other filter techniques, the Gaussian filter has become a standard in digital image processing for smoothing and spatial denoising:

$$G_{\sigma,v}(p,q) = exp\left(-\frac{\left\|p-q\right\|^2}{2\sigma^2}\right)$$

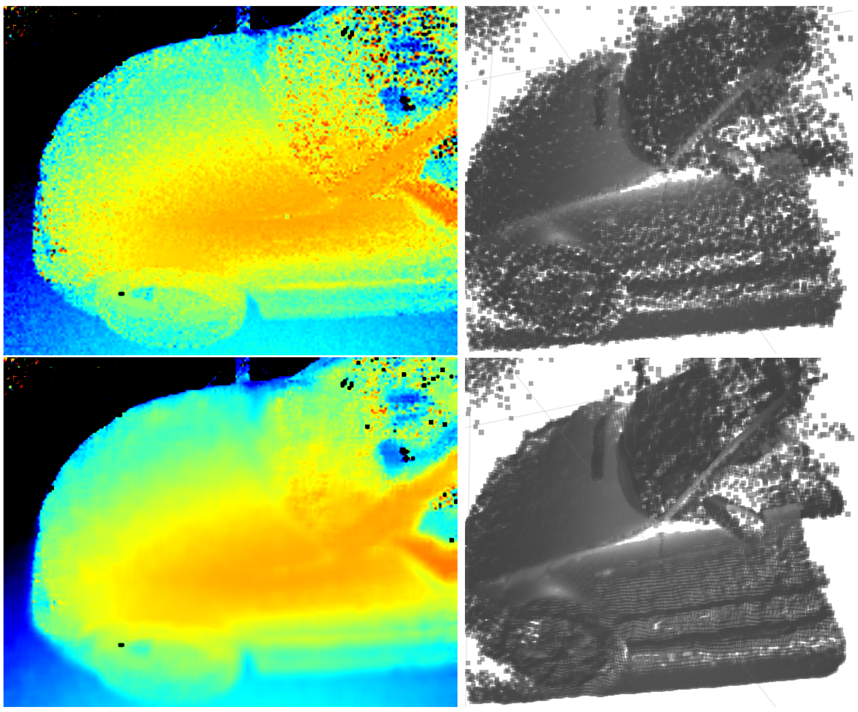


Figure 6.9: Top: calibrated depth data from the $\text{PMD}_{\text{left}}$ camera (2D color jet map and associated 3D point cloud representation). Bottom: Gaussian Filter applied. [HLK13b]

The experiments with PMD distance data have shown, that $\sigma = 4$ (spatial relation) yield good filtering results in the case of noisy data, but has the negative effect of blurring images strongly. This is especially a problem in the area of contours or distance discontinuities. Tomasi et al. [TM98] introduced an edge-preserving filter known as Bilateral Filter to counter this effect. But it could not improve the results in this application.

### 6.5.2 Segmentation

In addition to preprocessing, segmentation takes an important role before performing the full registration of the moving car. Because the background can vary over the day and the camera alignment can differ between setups (e.g. open scenes where the unambiguity range of the PMD camera is violated, see Sec. 2.1), a more dynamic segmentation (than just a distance clamping) is strongly required. Since registration works in a way that point correspondences of two subsequent images are taken and aligned, it works quite well for a static scene where only cameras move. As long as a moving object in the scene has to be tracked and registered, the standard approach fails and causes misalignment to the object of interest. The reason for the misalignment is the usage of correspondences in the static part of the images. To avoid misalignement, a pre-segmenation of the image data into back- and foreground via simple binary classification is required. This process is called background subtraction and has been studied and used for several decades. Fig. 6.10 schematically shows the main background subtraction principle.

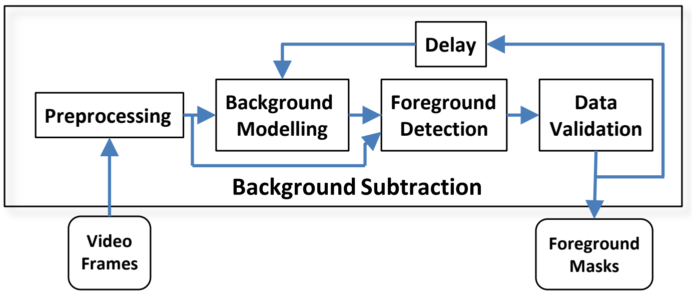


Figure 6.10: Schematic view of the standard background subtraction process. [HLK13b]

These algorithms are generally divided into four steps. The *Preprocessing* (see Sec. 6.5.1) allows to reduce noise in the data by modeling the fixed noise and applies some smoothing filters. The second step is known as the learning phase and consists of the extraction of a *Background Model* $B(x,y)$. It must be robust against environmental changes such as lighting and at the same time sensitive enough to discriminate from the moving objects [CK05, McI00], thus a median background image (without the object to extract) is calculated:

$$B(x,y) = \text{med}\left(I_t(x,y)\right), \; with \; t = 0...n, \; n \in \mathbb{N}$$

The next step is the *Foreground Detection*. Using the background model and the current image, a boolean mask $S_t(x,y)$ can be defined using a simple image subtraction operation:

$$S_t(x,y) = \left|I_t(x,y) - B(x,y)\right| > \theta$$

In most of the approaches, the threshold $\theta$ is experimentally determined, according to the requirements specified. The last step is the *Data Validation*. Most background models neglect the relationship and influences between neighboring pixels. In this step the detected foreground is validated and corrected by using a morphological opening operation to eliminate single pixels.

This kind of thresholding can be easily applied to the intensity and distance data of the PMD image. This additional knowledge is combined with a boolean operation to receive a more stable boolean segmentation mask $S_t(x,y)$:

$$S_t(x,y) = S_I(x,y)\ AND\ S_D(x,y)$$

with $S_I(x,y)$ as the intensity and $S_D(x,y)$ as the distance segmentation mask. $S_t$ contains valid foreground pixels only.

In a last step, a morphological closing is applied in combination with a contour detection algorithm to close holes. The results can be seen in Fig. 6.11, especially in the left image.

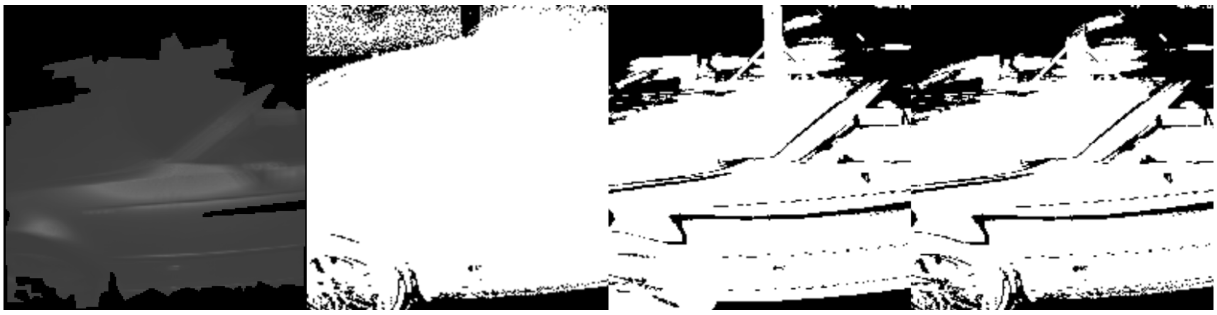


Figure 6.11: Left to right: segmented background image after applying the final morphological closing; distance segmentation; intensity segmentation; combination of distance and intensity segmentation. [HLK13b]

The presented algorithm works exactly in the way as shown in Fig. 6.10.

### 6.5.3 Registration

The registration process is the final part in the system after the correction and segmentation of the data. As base algorithm, the KinectFusion ICP approach has been chosen due to its working principle of e.g. a dense correspondence search (see also Sec. 6.7.1 why a sparse method does not work in this case). Furthermore it assumes small transformations between to consecutive frames and has real time capabilities. For the registration process, it is assumed to have the same transformation for all three cameras in the system. For this reason, the following three registration steps are executed for one camera only (e.g. the left camera):

1. Normal estimation
2. Correspondence search
3. Registration using the ICP algorithm

The following sections will describe the important steps in detail.

#### 6.5.3.1 Normal Estimation

It is the initial step of the correspondence search. A point normal represents the surface orientation at the corresponding point. To calculate stable point normals the neighborhood is taken into account. This can be done by simply using *central differences*.

This kind of normal calculation is very fast and gives satisfying results concerning the stability (see Fig. 6.12). However, at the cost of performance, the quality and stability can be improved by using more advanced surface normal estimation techniques such as a principal component analysis.

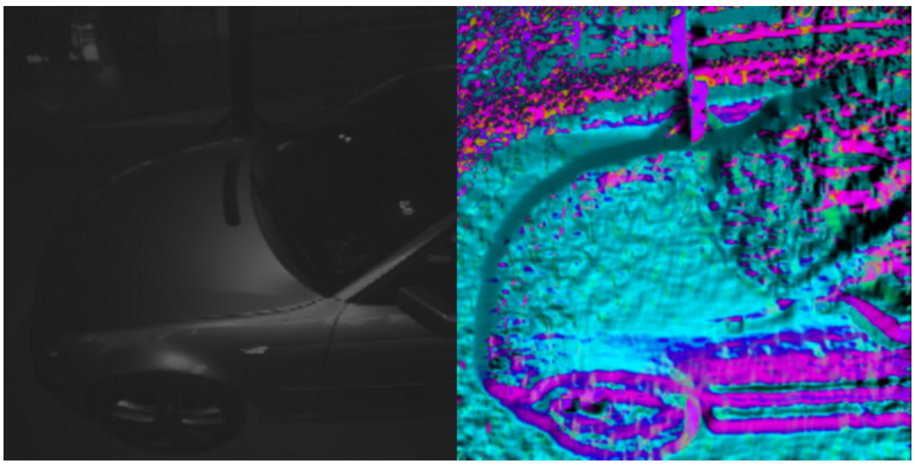


Figure 6.12: A PMD intensity image on the left and its corresponding normal image using central differences on the right. [HLK13b]

#### 6.5.3.2 Correspondence search

It is the second step in the registration. In general, good correspondences give good alignment results with the ICP algorithm. The algorithm presented here uses the Point-To-Plane metric that can be seen in Fig. 2.7.

This part of the registration is normally the most time consuming step. The point-to-plane metric can be implemented in real-time, assuming small motions between two consecutive images, as shown by Izadi and Newcombe [IKH+11, NIH+11]. Here the correspondence search is done using a 3D representation of the current accumulated surface $P_M$ (containing all valid points at this time) and an image to register $P_R$. An additional step needs to be performed before the correspondence search can be applied. The image points $P_R$ have to be transformed to an accumulated camera position (based on previous registered valid transformation). Then the points of both data sets (model and image) are projected onto a 2D image using the known camera intrinsic parameters. Under the assumption of small movements only, possible point correspondences are points at the same 2D-image location $P_M(x,y)$ and $P_R(x,y)$. To reject invalid correspondences the following two conditions are defined:

1. Similar point normal
2. Similar distance

Therefore thresholds for the angle ($\theta_{angle}$) and distance deviation ($\theta_{distance}$) are defined. Experiments with the PMD cameras for this application have shown that an angle of 20° and a distance of 0.2m as deviation give good results.

#### 6.5.3.3 The ICP

The Iterative Closest Point is an often used and well studied algorithm. Over the last two decades it became the preferred algorithm to merge point clouds. Using the method proposed by [IKH+11, NIH+11], it is possible to implement the point-to-plane metric using CUDA. The big advantage is that this method does not require any special data structure, thanks to the simple projection to a 2D image. This kind of data can easily be represented as linear or 2D-memory on the GPU and makes real-time processing possible.

The algorithm is formulated as a common least square minimization problem:

$$M_{opt} = \underset{M}{\operatorname{argmin}} \left( \sum_i \| (M \cdot s_i - d_i) \bullet n_i \|^2 \right) \tag{6.2}$$

where $M$ is the current transformation matrix composed of the translation vector $T\left(t_x, t_y, t_z\right)$ and a rotation matrix $R(\alpha, \beta, \gamma)$; $M_{opt}$ the optimal transformation matrix, $s_i$ a source point, $d_i$ a destination point and $n_i$ the unit normal vector.

The equation system is solved by an iterative linear approximation. Except for the final Cholesky decomposition of the reduced equation system (6×6 linear system), all the calculations are done in parallel on the GPU as described in [IKH+11]. The algorithm initializes the transformation $M$ with the identity matrix, which means that the first processed frame is the starting point. All transformations are then relative to this. The absolute transformation for the current frame $F_{t=n}$ is an accumulation of the single, small transformations between the frames $F_{t=0}$ to $F_{t=n-1}$.

What has be shown here is, that the point-to-plane metric ICP is able to robustly register 3D image data from low resolution (200×200 pixel) PMD images. With a suitable preprocessing step, the depth data from the $\text{PMD}_{\text{left}}$ and $\text{PMD}_{\text{right}}$ cameras contain enough reliable information of the car's surfaces to allow robust alignment of consecutive images.

### 6.5.4 Model Integration and Accumulation

After successful registration, the data has to be integrated into a model. The model $P_M$ contains all the available point data from frame $F_{t=0}$ till $F_{t=n}$ (n = current frame). Therefore the current frame data $P_D$ has to be integrated into the model. Two different approaches for the final point cloud are available:

1. Using the volumetric representation of KinectFusion provided by [IKH$^+$11, NIH$^+$11].
2. Integrating the point clouds into an independent point cloud representation of all three cameras.

#### 6.5.4.1 KinectFusion: The volumetric representation and integration

In this approach the method presented by [IKH$^+$11, NIH$^+$11] has been used. They use a volumetric representation, that appears as a 3D voxel grid data structure. The points are registered in the grid using a Truncated Signed Distance Function (TSDF) [IKH$^+$11, NIH$^+$11, CL96]. This makes it possible to encode the uncertainty of ToF data directly in the $512^3$ voxel volume. It is a continuous and iterative updating process which averages measured distances to the assumed smooth surface. Referring to the ICP, it allows to have a realistic, smooth model $P_M$ for the point and normal extraction of the next iteration step.

The implementation shows that this approach does not only work for high-resolution depth map cameras ($640 \times 480$ resolution for the Kinect camera) as presented by Izadi and Newcombe [IKH$^+$11, NIH$^+$11], but also for the low-resolution data of PMD cameras (see Fig. 6.13).

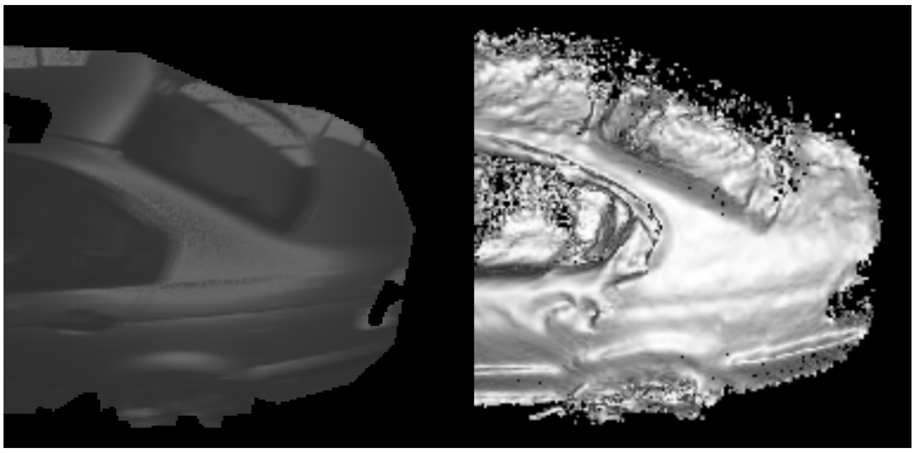


Figure 6.13: Left: segmented intensity image of the car provided by the PMD$_{\text{right}}$ camera and filtered of outliers. Right: the corresponding volumetric representation. [HLK13b]

#### 6.5.4.2 Integration of All Cameras and Accumulation

In general, the presented image based ICP, in combination with a voxel grid, is a performant but high memory (512MB) consuming algorithm. Applying the ICP for the three separate PMD cameras would be a big overhead in performance and memory consumption. For that reason, only one camera information is used to retrieve the

registered transformation instead of using three separate ICPs. Furhtermore using the information of the three PMD cameras would make the image based correspondence search impossible.
Experiments show that only the left or right camera is usable for the registration. This is due to the top camera providing too little information of the car's surface which leads to false correspondences (planar regions). The whole data is then merged using the transformation of the ICP presented here and the known extrinsic transformation from the calibration process. These transformations are applied to the three preprocessed point clouds (left-, top- and right-PMD-camera). Integrating frame after frame leads to a fully accumulated point cloud. Since the same transformation is used for all the three point clouds, a camera synchronization mechanism is required. An electronic synchronized trigger circuit is used. This ensures that all images are taken at the same time. An example of a fully accumulated car point cloud can be seen in Fig. 6.14.

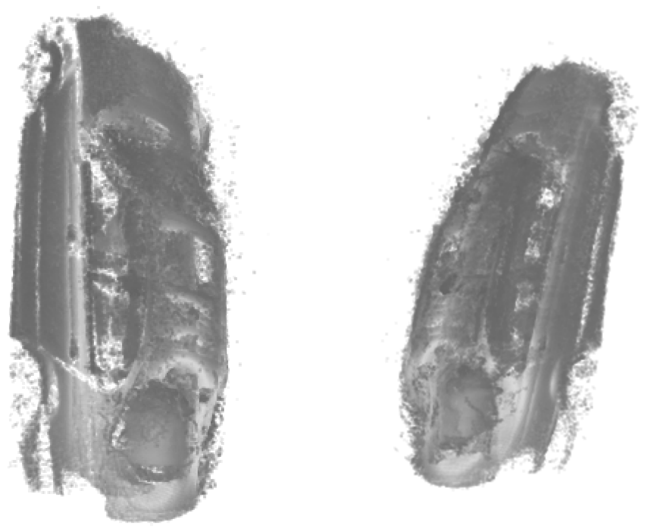


Figure 6.14: Left: left view of the car point cloud. Right: right view of the car point cloud. [HLK13b]

## 6.6 Hardware Setup and Calibration

This section describes the main experiments performed to setup the physical and data processing system. It goes into a detailed description how the necessary parameters like the camera position and also the system thresholds are determined. One of the critical points to allow for successful depth registration is the hardware setup, especially the camera orientation. Fig. 6.15 shows two possible angles for the left camera. It exemplary shows the FOV and which areas of a car are covered.

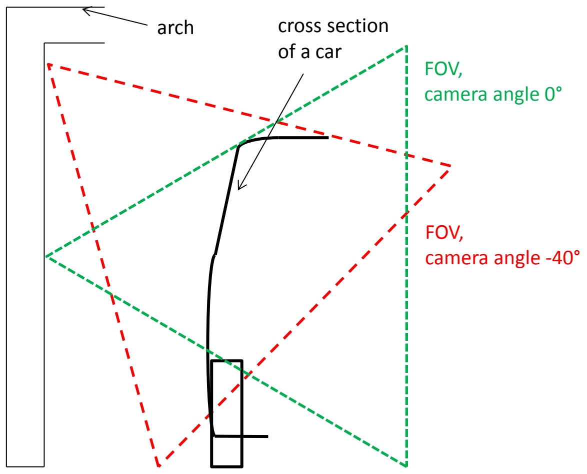


Figure 6.15: Sketch of the left camera setup with two different camera angles. [HLK13b]

A well known restriction of the ICP [IKH+11, NIH+11] is that flat surfaces do not provide enough correspondences for the algorithm to converge successfully. The experiments with different camera angles, in combination with the registration algorithm presented in Sec. 6.5.3, have exactly shown this behavior. So it has been decided to use a side camera angle of -40° to cover the roof and the side of the measured car.

To receive a complete system calibration, the cameras are extrinsically calibrated using the method shown in Sec. 2.2. The result is a transformation between RGB (which represents the reference coordinate system, Fig. 6.16) and PMD camera, as can be seen in Table 6.1, where the translation is in the unit *meter*.

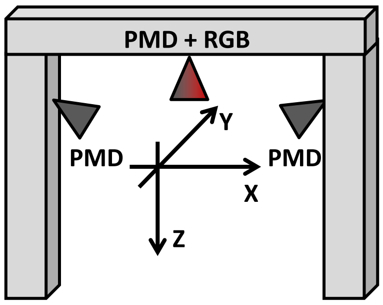


Figure 6.16: This figure shows the reference coordinate system related to the top mounted RGB camera. The transformation matrix in Table 6.1 is relative to it. [HLK13b]

| Rotation | | | Translation | Axis |
|---|---|---|---|---|
| 0.621669 | 0.034382 | 0.782525 | -1.97145 | X |
| 0.0206714 | 0.997968 | -0.0602701 | -0.0490192 | Y |
| -0.783008 | 0.053644 | 0.619695 | 0.319617 | Z |
| 0 | 0 | 0 | 1 | |

Table 6.1: Transformation matrix between the left PMD camera and the top-RGB camera (translation part in meter). [HLK13b]

This calibration process is performed for the left, top and right camera respectively.

## 6.7 Results

The previous sections have introduced a hardware setup, mechanisms, algorithms and a processing pipeline to reconstruct a point cloud of car. Fig. 6.14 shows the fully reconstructed point cloud. On the roof side a lot of noise is still visible. Reasons for this are the inherent noise of the cameras as well as measurements of points (e.g. background objects) that lie beyond the unique distance for ToF (in general 7.5m with a frequency of 20 MHz).

One goal is to achieve real-time data processing. Tests were executed on an Intel Core i7-3770K CPU @ 3.50 GHz and an NVIDIA GeForce GTX 680, 2GB graphics card. It gives the following timing results for the individual processing modules:

| **Module/Filter type** | **Average time of 110 frames in ms** | **Average time of 41 frames in ms** |
|---|---|---|
| Motion Compensation | 25.315521 | 26.26309 |
| Median | 0.986592 | 0.953216 |
| Segmentation | 0.15550 | 0.17341 |
| Outlier removal | 0.10756 | 0.11094 |
| Bilateral Filter | 0.69742 | 0.46539 |
| PMD to point cloud | 0.95627 | 0.91442 |
| Point cloud extrinsic merge | 0.78604 | 0.61030 |
| ICP Bilateral Filter | 0.86885 | 0.47001 |
| ICP registration filter | 15.62808 | 10,04618 |
| Point cloud transformation | 0.14510 | 0.12066 |
| Point cloud accumulation to model | 1.00250 | 0.83022 |
| **Total time** | **46.649433** | **40.957836** |

Table 6.2: Execution timing for 110 frames (all frames) and for 41 frames (valid frames only)[HLK13b]

Table 6.2 shows the average execution time for one of the cameras (the other two are similar) and the time needed for registration. The 110 processed frames also contain images that cannot be processed or which will cause the fail of the ICP (invalid frames; either the car is not visible or there are too few valid pixels). The 41 frames contain valid frames only. It can be seen that the average execution time for the ICP is 5 ms lower compared to the 110 frames. This is related to the rejection of 40 % of the frames. In consequence thereof, the ICP fail rate is lower which leads to the better average ICP registration performance.

The result is an average execution time for all frames and all cameras of about $41ms$. This gives a theoretically possible frame rate of about 25 frames per second. The realistic frame rate, containing additional processing overhead (memory allocation and transfer from CPU to GPU) and considering the restrictions of the PMD cameras (see Sec. 6.1) is about 20 frames. The here provided timing results are reproducible and also applicable for other cars than the one used for the experiments. This shows that the presented solution can handle the data in real-time.

The average number of acquired data points is about 4.4 million. Due to outlier removal, filtering and smoothing the average final number of processed points is around 0.9 million. This is a rejection rate of 80 %. The results show that this is still enough for a successful registration.

The error comparison against real cars is still a problem because there is no access to real CAD models from the car manufacturers. First comparisons to the dimensions (e.g. from the technical manual) of a car show deviations of 3 to 10 % to these numbers.

Despite the lack of detailed reference data, some evaluation about the accuracy can be made. Fig. 6.17 shows the reconstruction of a Peugeot 206. The real wheel distance (physically measured) is about 2.45 m. Data from the reconstruction process shows it to be 2.52 m (measured with the tool MeshLab), which is a deviation of about 3 %.

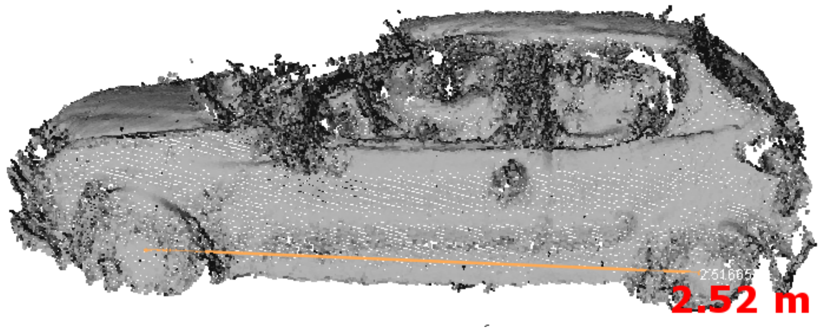


Figure 6.17: Wheel distance of a measured Peugeot 206. [HLK13b]

Furthermore, a BMW E46 has been reconstructed with the proposed processing pipeline (see Fig. 6.18). The real wheel distance is about 2.72 m. The reconstruction result is 2.65 m (measured with the tool MeshLab), which is also a deviation of about 3 %.

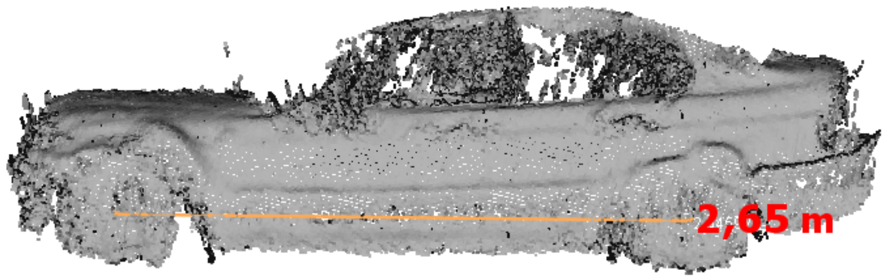


Figure 6.18: Wheel distance of a measured BMW E46. [HLK13b]

As a last car, a Skoda Octavia II has been reconstructed with the pipeline (see Fig. 6.19. The real wheel distance is about 2.55 m. The reconstruction result is 2.49 m (measured with the tool MeshLab), which is a deviation of about 2 %. It can be seen, that it is difficult to measure the wheel distance in the point cloud due to the lack of information in the wheel area. This is a common problem for all measurements.

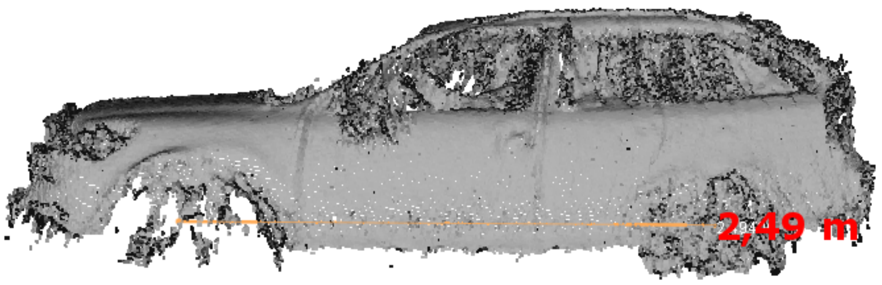


Figure 6.19: Wheel distance of a measured Skoda Octavia II. [HLK13b]

### 6.7.1 Feature Based Image Registration

The section is focused on different feature detection algorithms using a masked image that has been created with the proposed segmentation method (see Fig. 6.20). The red circles indicate features, the size represents their reliability and the green lines connect matches.

It is shown that SURF [BETG08] is not working in the feature rare environment of car measuring. The same results as in Fig. 6.20 were visible for SIFT and Good Features To Track. This shows that it is not possible to use approaches well known from Stereo Vision for image registration in this field of work [LF96].

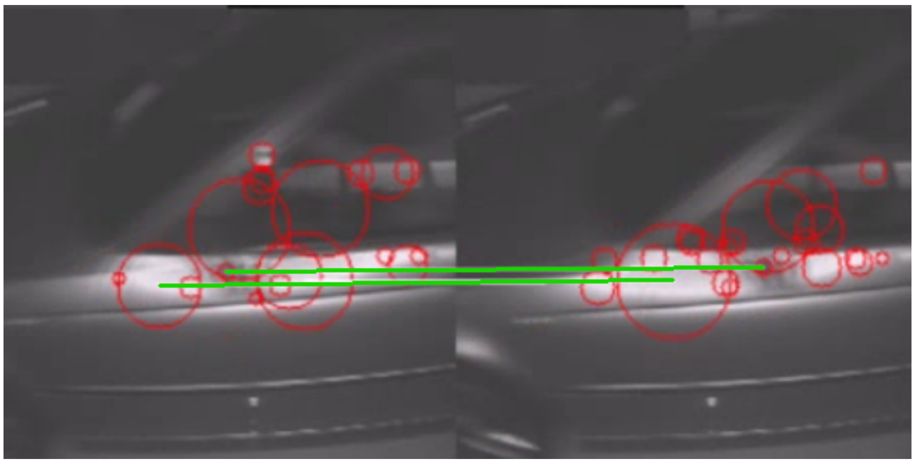


Figure 6.20: SURF extraction of moving car. Features are mainly found in oversaturated ares or reflections. [HLK13b]

### 6.7.2 Result Summary

Summarizing the results from this section it can be seen, that the presented algorithm already gives acceptable results in performance and quality. It can be seen that the applied outlier removal already gives good results but still has to be optimized.

Furthermore, a proper working background subtraction is one of the most important parts and has to be improved. Even though there is a lot of potential for optimization and improvement of quality, the point cloud can be used for further processing where the extraction of following information can be performed:

- Bounding box to determine length, width and height
- The car contour of the sides and the height (see Fig. 6.21)

Fig. 6.17, Fig. 6.18 and Fig. 6.19 show that, especially in regions of windows and wheels, problems still remain. This has to be optimized in future work. Due to the inertial behavior and the flexibility of the brushes in a car wash system, the achieved accuracy is sufficient, but has still to be improved.

Furthermore the results show, that the usage of a dense ICP is a good way to perform this kind of 3D car reconstruction (see also Sec. 6.7.1).

The speed of the car (between 1 $m/s$ and 3 $m/s$) has been a typical speed (about preferred walking speed) when entering a wash system. Higher speeds cause the ICP to fail due to the large lateral distances between consecutive images. In this case, the standard dense ICP as used for the KinectFusion algorithm fails.

## 6.8 Summary

A GPU based data processing pipeline has been presented allowing the real-time data fusion of three synchronized PMD cameras. The pipeline supports multiple pre-processing steps to optimize incoming depth data. Also the presented challenges such as over- and undersaturation, flying pixels, motion artifacts and the rough environment, that are specific to this project, are successfully handled by the extensive preprocessing. The presented pipeline can handle different kinds of materials, such as reflectors, glass and metallic car paintings by performing outlier removal.

Another important contribution is the extension to the standard segmentation algorithm. The logical conjunction of the depth and intensity data in combination with the morphological operation and the contour estimation stabilize the algorithm.

Furthermore it was shown that the existing KinectFusion algorithm works with low resolution ToF depth images. Large moving objects like cars can be sampled and reconstructed to a full point cloud by driving the car through a measurement arch. While some aspects require a lot of optimization, it was shown that a contactless 3D measurement and reconstruction of a car is possible by using PMD-ToF cameras.

In a next step, each preprocessing step has to be optimized to avoid misalignment of the ICP algorithm. Therefore especially the background subtraction has to be optimized to avoid wrong point correspondences and to fully remove the restriction of a static background model. Furthermore the 3D point cloud has to be rendered to a contour representation to make it usable for a PLC device (Fig. 6.21).

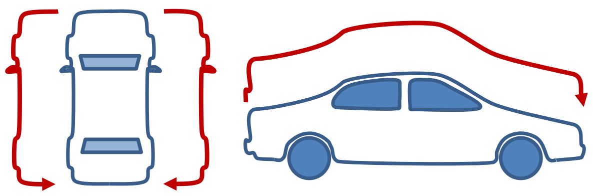


Figure 6.21: The car contour: the red lines are the extracted contours that can be used and processed by a connected PLC device. [HLK13b]

In detail, the height and side contour has to be extracted by rendering the point cloud from top (side contour) and side (height contour). Additionally these extracted contours can be interpreted to predict invalid abnormal superstructures like roof racks as mentioned in the beginning. Having the full point cloud of the car opens up new research areas such as point cloud interpretation (localization of windows and wheels) and 3D mesh reconstruction to use the models, e.g. for usage in a CAD tool. To improve the final point cloud and to optimize the contour extraction, a resampling of the points can be helpful.

Finally it can be said, that the first measurement results fulfill the required accuracy (size deviation of about 5 to 10 cm) and also the performance requirement (real-time processing). Further improvements, as e.g. the optimization of the normal estimation, outlier removal, contour extraction and the finalization of the contour extraction, will be done in the future.

# 7 Depth Data Processing Summary

The previous chapters have shown two ways how ToF data can be improved. Furthermore an industrial application using this specific sensor knowledge has been presented. This chapter summarizes the results and contributions of Part I.

First a new method (see Chapter 4) for a fast motion artifact compensation for Time-of-Flight cameras was presented. The approach is based on several assumptions such as linear motion between the four consecutive phase images of the PMD camera. The algorithm uses a thresholding and binarization method to restrict the artifact correction area to spaces where actual motion occurs. It is shown that the algorithm gives good results for simulated data (linear and non linear motion) and also for real data. Furthermore a big advantage of this algorithm is that it can work in real-time with an execution time of about 10 ms and a frame rate of up to 100 FPS). However, there is still a lot of optimization potential, e.g. support for phase image motion correction. Additionally the threshold can be automatically adapted via statistics of the observed scene.

The second proposed algorithm (see Chapter 5) deals with the automatic integration time estimation of ToF cameras. A novel online integration time adaption algorithm that works on a per-pixel basis is presented. It uses knowledge gained from an extensive analysis of the underlying inherent sensor behavior regarding intensity, amplitude and distance error to reduce the overall error, to prevent oversaturation and to minimize the adaption time. Working well in presence of various reflectivities and quick changes in the scene, the per-pixel character also allows to use only portions of the image or even apply pixel-specific weighting, counteracting sensor properties (e.g. spatial intensity distribution) and allowing to adjust importance of certain image regions. Overall, this represents a significant improvement over previous methods.

As third and last contribution of this part of the thesis, an industrial 3D car reconstruction example has been presented (see Chapter 6). A GPU based data processing pipeline has been introduced allowing the real-time data fusion of three synchronized PMD cameras. The pipeline supports multiple preprocessing steps to optimize incoming depth data. Also the presented challenges like over- and undersaturation, flying pixels, motion artifacts and the rough environment, that are specific to this project are successfully handled by the extensive preprocessing. The presented pipeline can handle different kinds of materials, such as reflectors, glass and metallic car paintings by performing outlier removal. By using KinectFusion based on the ICP algorithm with the preprocessed low resolution data, it was possible to reconstruct a full 3D model of moving cars.

Summarizing the results of the previous chapters and the contributions of this thesis, it is shown that the ToF technology is an interesting, helpful and promising technology that can be used in various scientific and industrial areas. But still a lot of work has to be invested, especially to improve the data quality of this kind of sensor.

# Part II

# Model Driven Software Engineering Paradigms

# 8 Introduction of Model Driven Engineering

The complexity of image processing tasks has risen during the last years for several reasons. On one side, more and more industrial machines profit from image processing support, e.g. by using a camera system for visual inspection. An example can be found in Chapter 6. On the other side, camera systems and the processing components (as e.g. desktop PCs) have become better and cheaper. The complexity and demand for a short Time-to-Market requires improving development processes.

During the past years, several techniques have been established in software engineering but are rarely used for image processing. The two most important approaches are graphical modeling and DSLs. Graphical modeling gives modelers the possibility to create a relatively simple, high-level, iterative graphical architecture design, where all relevant parties (programmers and non-programmers such as project leaders) can have an abstract but graphical view on the system in development. This can help with identifying and solving problems at an early stage. For this purpose UML was introduced by the OMG and accepted by the ISO as a standard in 2000 [OMG16e]. It allows to model high-level abstractions of real-world problems by using graphical descriptions. Different kinds of diagrams such as class-, activity- or state-machine-diagrams are the basis of modeling the system's structure and functionality. This improves general software quality and reduces the Time-to-Market.
DSLs are the second important approach. They have become more and more important in the past years for improving software development due to tools simplifying the language development, e.g. xText [EEK+16]. DSLs are divided into two types: internal and external [Fow10, EEK+12]. Internal DSLs use the existing infrastructure of host programming languages. Examples for such DSLs are OpenCL or OpenGL. Both languages are a C dialect, extending C to their requirements. External DSLs however are designed from scratch after a full analysis of the problem description. Using special keywords, abstractions and control structures, they are able to illustrate complex problems mostly in simpler and more compressed forms. A big drawback is that the whole infrastructure such as parsers, lexers and compilers has to be built. Well-known examples are the unix shell scripts or SQL. Both kinds of DSLs improve

the readability and the formulation of domain-specific, real world problems. While UML can be extended by profiles (for e.g. adding new types and/or model elements), external DSLs provide the possibility to even start the formulation of problems from scratch, which means writing a completely new problem related language. The approach presented here combines the best of both worlds using xText to design a DSL and GMF [EC16], allowing graphical modeling of data processing and vision problems as those presented in Part I.

The following chapters present a novel DSL based on Eclipse xText (textual modeling) in combination with Eclipse GMF (graphical modeling). It is developed from scratch, while using some concepts from Java and C#. Furthermore, it adopts the idea of encapsulating classes and flow-models using diagrams as done by UML, but in a textual and graphical form. This has the advantage of giving developers the freedom of modeling as they see fit (round-trip engineering). But at the same time, the DSL forces the developer to keep special structures and requirements using flow modeling. This helps reducing recurring mistakes. The DSL is specially designed to make the model-to-text transformation as easy as possible and to support C++ as intermediate textual representation in a first step.

## Problem Statement

Both approaches, graphical modeling and also external DSLs, are unfortunately hardly used in the domain of image processing and computer vision. The community mainly wants to concentrate on the development of algorithms. But in most cases they have to begin their work from scratch and start with the development of GUIs and the abstraction of data and image processing interfaces. It is widely known that GUI development can rapidly grow to become a long and error-prone task, which often leads to loss of focus. Having the possibility of using a specially adapted development environment and toolchain as the one proposed in this thesis, the concentration and focus can be shifted back to the true scope of algorithm development.

This thesis makes several contributions which are the result from the significant challenges in the domain of Model Driven Engineering for data and image processing. These contributions are the generic Domain Specific Language GU-DSL supporting textual and graphical design and development of data and image processing related algorithms [HFKK15]. Furthermore on the basis of GU-DSL, a Model Driven GPGPU Programming approach is introduced to improve and simplify the error prone task of GPU algorithm development [HFKK16]. Finally a special extension of GU-DSL is contributed supporting Component-Based Data And Image Processing Architectures in a textual and graphical representation [HKK16].

# Outline

The structure of Part II of this thesis consists of the following chapters:

*Chapter 9* provides the necessary fundamentals of GU-DSL.

*Chapter 10* shows how three different image processing algorithms can be implemented for GPGPU processing using GU-DSL .

*Chapter 11* introduces a new component-based data and image processing approach using GU-DSL.

*Chapter 12* summarizes the results of the previous model driven engineering chapters and concludes Part II.

# 9 GU-DSL – A Generic Domain Specific Language

The development of Domain Specific Languages becomes more and more popular, since the release of tools like xText and GMF. These tools lower the barrier for programmers to develop application related languages which can simplify their problem descriptions and solutions.

GU-DSL is such a Domain Specific Language, based on Eclipse xText and GMF. It is especially designed for textual and graphical, object-oriented modeling of data and image processing problems as described in the chapters of Part I. Different from other languages in this domain, it is designed in a way that model and flow driven problem solutions (important for image processing) are the main focus while other languages often concentrate on the compression and simplification of code only.

The domain of image processing has requirements as e.g. the following:

- A higher level representation (as e.g. a graphical model) but with low level data access (e.g. memory) at the same time
- Access to a textual language representation to lower the entrance level for low level programmers
- High performance execution of tasks/functions
- Support of reference types
  - Simplification of data manipulations
  - Performance issues

GU-DSL is developed from scratch, but using some well established concepts from Java and C# to fulfill these requirements. Using this syntax, it is easy to learn and it offers an abstract way for textual modeling of tasks, especially in the area of data and image processing. Furthermore, it is adopting the idea of encapsulating classes and flow-models using diagrams as done by UML, but in a textual and graphical form. This has the advantage of giving developers the freedom of modeling as they see fit.

But at the same time, the DSL forces the developer to keep special structures and requirements using flow modeling. This helps reducing recurring mistakes. The DSL is specially designed to make the model-to-text transformation as easy as possible and to support C++ as intermediate textual representation.

The language introduces several new or/and adapted concepts providing the necessary functionality to fulfill the previously stated requirements:

- A diagram based, object oriented, textual modeling language
- Class-Diagram support
- Activity-Diagram support
- An expression language allowing to implement sequential code sections within classes, class-methods and activity-diagram nodes
- A graphical modeling toolchain, based on Eclipse GMF

The following sections will give an introduction into the GU-DSL's concepts and functions as a basis for Chapter 10 and Chapter 11.

*Publication: GU-DSL – A Generic Domain Specific Language for Data- and Image Processing [HFKK15]*

## 9.1 Language Features and Concepts

The following section will introduce the most important features of GU-DSL, allowing the implementation of image- and data-processing algorithms.

### 9.1.1 Structural Modeling using Class-Diagrams

Modern object oriented programming languages in general use classes and namespaces to form an abstraction of real-world objects. This allows structuring software by encapsulating functionality belonging together. As long as there are only a few number of classes or objects, it is easy to keep track. Class-diagrams are a good tool to model complex structures. While allowing the design of individual class structures (variables and methods), they also provide the possibility to visually show object interconnections as associations or inheritance between related classes.

This concept has been adopted and allows the definition of classes, interfaces and attributes (used to visualize meta-information). Listing 1 shows the definition of a simple diagram with an abstract image class, an example for class inheritance and interface implementations. This is in general well-known from languages like C# and Java and forms the base of the new system.

```
ClassDiagram TypeDefinitions
{
  // Interface definition
  public interface IImage
  {
    public bool load(string filename);
    public bool save(string filename);
  }

  // Interface implementation
  public abstract class Image implements IImage
  {
    public int width;
    public int height;

    public bool load(string filename);
    public bool save(string filename);
  }

  // Class Extension
  public class ExtImage extends Image
  {
    public void filterNoise();
    public void smooth();
  }
}
```

**Listing 1** Extended class diagram showing interfaces and class inheritance. [HFKK15]

Besides classes, enumerations are another important kind of structure allowing value grouping (Listing 2).

```
enum ImageFormat
{
  Format_RGB32 = 2,
  Format_ARGB32 = 3,
}
```

**Listing 2** Definition of an enumeration. [HFKK15]

Enumerations can be used as independent types as the built-in types **byte**, **int**, **float**, **real**, **string**, **bool** and **void**.

Method parameters and fields can have qualifiers applied. This gives the possibility to restrict or to grant access rights and define the visibility (Listing 3).

```
// A constant, public field
public const int i = 0;
// A protected reference field
protected ref real j;
// A private static field
private global int g = 100;
// A public constant reference to a static field
public global const ref int k = ref g;
// A public static method
public global void memcpy(ref void dst, ref void src, int numBytes);
```

**Listing 3** Field, method and parameter access qualifiers and visibility. [HFKK15]

To allow modeling of meta-information, e.g. to support a more flexible code generation, attributes (comparable to C# attributes and Java annotations, see Listing 4) have been introduced.

```
ClassDiagram Attributes
{
  public attribute CppType
  {
    public string name;
  }

  public attribute GeneratorVisibility
  {
    public bool visible = true;
  }
}
```

**Listing 4** Definition of generator attributes. [HFKK15]

Attributes can be used on classes, methods, fields and enumerations (see Listing 5).

```
[Attributes.GeneratorVisibility(visible = false)]
public abstract class Image{}
```

**Listing 5** Usage of generator attributes. [HFKK15]

Besides simple classes, interfaces and enumerations, class- and interface-templates as well as reference types are supported. Using these definitions, a complete structural representation of image- and data-processing algorithms can be modeled.

### 9.1.2 Definition of Behavior Modeling

The second important part is the definition of object and system behavior. UML provides several kinds of diagrams (activity-, state-machine or sequence-diagrams). This approach uses two incorporating methods. The first one uses an expression language (similar to Xbase [EEK+12]), allowing sequential coding and lowering the barrier using the presented approach. A second method uses an extended activity-diagram allowing flow-modeling. The diagram fully supports and incorporates the expression language to solve the main drawback of rapidly rising visual complexity of pure activity-diagram modeling, as stated in the introduction.

### 9.1.3 Behavior Modeling using Expressions

As mentioned previously, the expression language uses the base concepts of Xbase and other well-known programming languages. It is based on the types introduced in Sec. 9.1.1 and is specially designed in a way to make the text-to-text (T2T) transformation (code generation) as easy as possible. It has special domain-specific extensions, e.g. references to improve algorithm performance. The next sections will introduce the most important expression features.

**Variable Expressions** The expression grammar allows to declare variables. For a detailed description have a look at Appendix A.

**Other Expressions** The language also supports basic expressions like assignments (=), equality checks (==, ! =), logical operations (&&, ||), bitwise operations (&, |), comparisons (>=, <=, >, <) and expression grouping in blocks ({}) that are used in the same way as in C/C++.

All the expressions previously described can be attached to class methods to allow sequential behavior modeling. Furthermore, it is possible to use the grammar in activity-diagram nodes introduced in Sec. 9.1.4.

### 9.1.4 Behavior Modeling using Activity-Diagrams

The previous section introduced the expression language allowing simple textual behavior modeling. This section shows how to use textual activity-diagrams to give the developer the possibility to abstract algorithms or behavior in a more object oriented manner. Activity-diagrams provide an easy way to model data flow or single system parts. They support concurrency, conditional decisions and also loops. The UML activity-diagrams have been extended by the following features to make them more reusable:

- Class-assignments
- Diagram input parameter (like in UML 2.x)
- Activity-diagram variable definitions
- Action-node definition
- Expression support within nodes

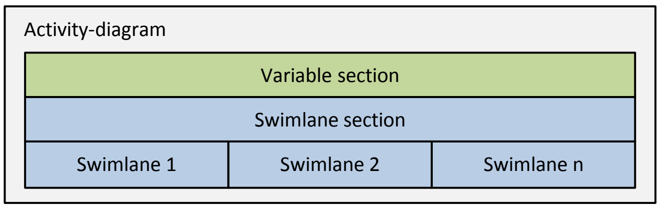


Figure 9.1: The two sections of an activity-diagram. [HFKK15]

As can be seen in Fig. 9.1, an activity-diagram consists of a variable section defining local variables and an arbitrary number of swimlanes. Swimlanes are elements to split an activity-diagram into several concurrent sub-processes. This approach assumes that an activity diagram describes the flow within a class method and is called as shown in Listing 6.

```
call behavior AdMemberAccess("C:\\example.png",
                             "C:\\example_filtered.png");
```

**Listing 6** Call of an activity-diagram. [HFKK15]

Listing 8 shows an example diagram containing important diagram features, which are described in the next sections.

**Class-assignments** This implementation allows assigning owner classes to either a diagram or to the individual swimlanes. This offers the possibility to directly interact with classes. It provides access to all class member variables and methods as can be seen in Listing 8 (load- and save-node).

**Diagram input parameter** The UML defines activity diagram input and output parameters as well as objects, which is an essential extension to allow the re-usage of diagrams. Listing 8 shows the re-usage of these parameters as object flow between nodes.

**Activity-diagram variable definitions** Another newly introduced feature is the usage of local variable definitions. Class-assignments allow access to class member-variables. But in most cases, this isn't enough. The definition of local variables that can be used in the same way as member variables for transition guards and also within expression statements is possible.

**Action-node definition** Action-nodes are the main modeling nodes. They provide the basic functionality and allow visualizing all kind of problems. The connection between nodes is realized as either guarded ([] =>) or non-guarded (=>) transitions (see Listing 7). Transitions are always evaluated after the full execution of the node's content.

```
// An example action with introducing transitions
action example_action()
{
  // The => operator defines a transition to another node without any restriction
  => nonguardedAction;

  // [i < 10]: transition guard; The content within the
  // parentheses is evaluated and decides if the transition
  // has to be executed
  [i < 10] => guardedAction;
}
```

**Listing 7** An example action node with transitions. [HFKK15]

This provides the possibility to model conditional and loop flows, but most of the time it is complicated and confusing. UML has already introduced special kinds of nodes such as forks, joins and decisions. But from this point of view, it is not enough to provide a simple programming interface. The additional and extended node types are introduced in the following sections. Furthermore, three different kinds of action-node declarations are available. The first one represents a method call of an assigned class (e.g. load or save in Listing 8), while the second method (filterImage) shows the possibility of defining collections of expression statements. The third possibility is defining action-nodes that are assumed to be a method call, but they don't have to be members of a class. Having the same signature as method-call nodes, they are treated in a special way by the code generator. This allows the reduction of generated code by reusing them in a second occurrence as a simple method call.

**Other node-types** Besides the shown new nodes, **start**-, **final**-, **decision**-, **fork**- and **join**-nodes are also supported.

```
ActivityDiagram AdMemberAccess(string filenameLoad,
                               string filenameSave)
{
  // Diagram specific variable declarations
  public int i = 0;

  swimlane Swimlane1, owner TypeDefinitions.Image
  {
    start S1 { => load(filenameLoad); }

    // The action represent the load method
    // of the Image class
    action load(string name)
    {
      i = 0;
      => filterImage;
    }

    // A simple action-node doing some filtering
    action filterImage
    {
      i = i + 1;
      // Do some filtering

      // Recursive node call ==> lowwop
      [i < 10]  => filterImage;
      // Call the save method
      [i >= 10] => save(filenameSave);
    }

    // The action represent the save method
    // of the Image class
    action save(string name)
    {
      => Finish;
    }

    final Finish

  }
}
```

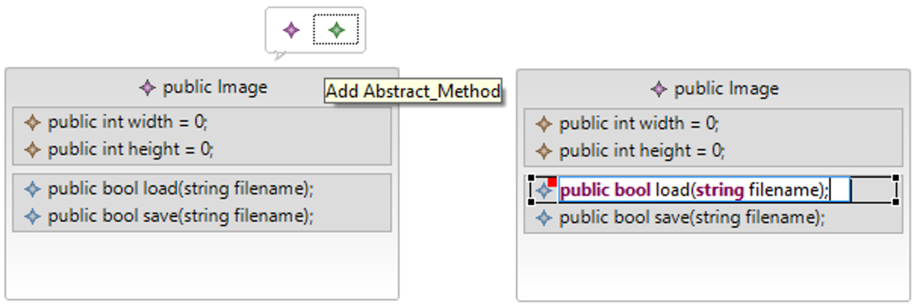
**Listing 8** An example activity-diagram. [HFKK15]

## 9.2 The Combination of Graphical And Textual Modeling

The preceding sections have shown the most important novel features of GU-DSL. In this section, it is described how graphical editors incorporate the underlying DSL and especially the expression language. The editors allow the placement of as many class- and activity-diagrams as possible next to each other. This provides a better overview and allows keeping structural- and behavior-models close together, but still separated. The graphical editors are combined with the textual editors in a very convenient way, to make editing as easy as possible (see also Fig. 9.2).

## 9.2.1 Class-Diagrams

Class diagrams can be modeled in a typical UML-like style which is shown in Sec. 9.1.1. Classes, interfaces, attributes and enumerations are placeable, movable and resizable objects from a predefined toolbox. Methods, fields and enumeration values are added by the default GMF functionality directly from the graphical editor (see Fig. 9.2, left). This editing mechanism is extended, allowing in-place editing, with auto-completion and syntax highlighting (comparable to [MN16]), which is the ideal add-on and improvement for easy programming in model-driven development scenarios and tools introduced here. E.g., a method is added and can directly be declared, parameters can be defined and also the implementation can be carried out (see Fig. 9.2, right). This behavior is implemented for all modifiable parts in the diagrams. Inheritance, associations etc. can also be created using the toolbox.



```
public Image
public int width = 0;
public int height = 0;
public bool load(string filename);
public bool save(string filename);
```

```
public Image
public int width = 0;
public int height = 0;
public bool load(string filename);
public bool save(string filename);
```

Figure 9.2: Left: adding a new new method to a class. Right: modifying a method using the in-place xText editor.

As well known from other editors and state-of-the art tools, the class diagrams support all kinds of data- and structure-types (classes, interfaces, enumerations, etc.) and corresponding connections as extensions (inheritance) and interface-implementations.

## 9.2.2 Activity-Diagrams

The other kind of diagram that can be designed in a graphical editor is the activity-diagram. As could be seen in Listing 8, the activity-diagrams are split into two main sections: a variable and a swimlane section.

#### 9.2.2.1 Variable Section

The variable section is a novel GU-DSL specific extension, introduced because local variables are unavoidable. The sections have been implemented as xText-editor analogously to the already shown class-method editors in Fig. 9.2. The variables are strung together in a multi-line editor and can be written as described in the DSL specification.

#### 9.2.2.2 Swimlane section

The swimlane section consists of an area allowing the placement of multiple, concurrent swimlanes. Properties, as e.g. the owning class, can be changed in the Eclipse property-editor. Due to the direct interoperability between class- and activity-diagrams, only accessible objects can be chosen. This helps to reduce bugs and is one of the biggest benefits in using the underlying DSL as basis for the graphical editors. In the case of the swimlane owner class, the choice is limited to classes. A swimlane itself is the main modeling area of an activity-diagram. It is the parent of all available diagram nodes and allows adding and modifying nodes by drag-and-drop.

#### 9.2.2.3 Node Types

In Sec. 9.1.4, the most important node types of the DSL have been introduced. All of the textual nodes are also available in the graphical modeling environment. This makes it easy for developers to also share code within a team. That means team members can decide on their own whether they prefer textual or graphical programming (round-trip engineering). This is a big advantage compared to other model-driven development systems. All graphical elements, except initial and final nodes, are constructed in the same way, in order to keep the usage as easy as possible.

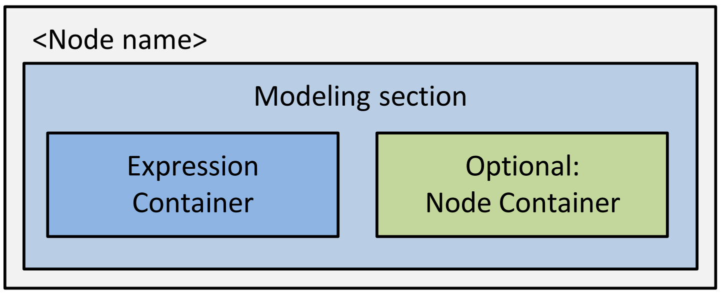
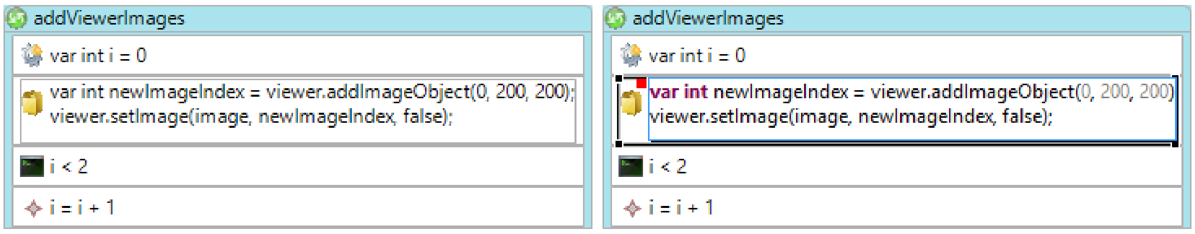


Figure 9.3: The principle schema of graphical nodes.
nodes

All node types start with a name section. The modeling section contains two parts. The first part is the expression container (see also Fig. 9.4). It allows entering expression statements using the full textual editor functionally in the same way as for pure textual modeling. This simplifies behavior modeling and bridges the gap between the complexities of solely textual or graphical modeling. So it is up to the developer to decide how fine or how coarse the graphical diagram is designed, either splitting all statements into single nodes or combining larger statements within one big node (see Fig. 9.4).



```
addViewerImages
var int i = 0
var int newImageIndex = viewer.addImageObject(0, 200, 200);
viewer.setImage(image, newImageIndex, false);
i < 2
i = i + 1
```

```
addViewerImages
var int i = 0
var int newImageIndex = viewer.addImageObject(0, 200, 200)
viewer.setImage(image, newImageIndex, false);
i < 2
i = i + 1
```

Figure 9.4: Left: An action-node. Right: Activated xText-Editor.

The second, but optional part is a node container section. Node types such as loop- and conditional nodes allow for modeling hierarchical flows using nested nodes. This provides opportunity to also model complex scenarios in a graphical way instead of switching over to textual modeling. But developers should keep in mind that pure graphical modeling rapidly ends up in an unclear representation.

### 9.2.2.4 Node Connections

Nodes can directly be connected with other nodes using directed node transitions (see Appendix B). As explained in Sec. 9.1.4, transitions can be guarded. This is realized in the graphical modeling view using labels. So every transition has an additional label, that is directly editable using a syntax highlighting text editor. This makes modeling different flow paths easy.

# 10 Model Driven GPGPU Programming

Computing on GPUs became popular about one decade ago (an example can be found in Chapter 6). However, increasing data processing performance via parallelization of tasks leads to an increase of programming complexity and reduction of maintainability. GPGPU programming poses special requirements due to the parallelization of tasks. Hence, there must be special control structures on one hand and simple ways to allow parallel execution of operations on the other hand. Therefore, such systems are usually split into two parts: a host (e.g. a PC) and one or multiple devices (e.g. a graphics card). For example, this is the case for OpenCL and CUDA. Executable device-programs (kernel) are based on a version of the ISO C99 standard [AMD16], extended by special types and functions. Calculations are performed by work-items arranged in work-grids (see Fig. 10.2). Data always has to be transferred between host memory and device OpenCL-Kernel memory, which can be a big bottleneck if this process is performed too often. The experience shows that algorithm development on the GPU is often an empiric, incremental process when trying to achieve the best performance. Abstracting algorithms into a generic, dataflow-driven form using the proposed language GU-DSL allows to automatically analyze the flow graph and optimize it with respect to best performance. While even small algorithms for classic sequential problem solving rapidly grow in complexity, CUDA and OpenCL also impose the whole initialization, memory-managed and code execution to the developer, which can be an error-prone and sophisticated task. Hence, the new type-safe, dataflow-driven textual and graphical programming language GU-DSL in combination with a C++ based Heterogeneous Computing (HC) framework has been developed. An additional Object Contstraint Language (OCL) [OMG16d] based code validation helps to reduce errors already during the modeling stage. A code generator generates fast C/C++ code that is compiled to a platform dependent executable. The underlying HC library allows to delegate GPU initialization, memory management and also code execution to the HC framework. Additionally, automated tests and performance optimizations can be performed by the code generator analyzing the flow graph.

This work contributes a convenient combination and mixture of textual and graphical model- and dataflow-driven design by extending GU-DSL class- and activity-diagrams to fulfill the explained requirements (e.g. reducing complexity, high performance, parallelization, special control structures, host/device management, kernel and memory management, Region Of Interest (ROI) handling) of GPGPU programming , which is unique in this domain. The main focus lies on the improvement of GPU code structure and not just its simplification. Therefore novel concepts like adding of meta-information by using attributes in class-diagrams, are introduced. Furthermore, new node types such as conditional-, loop- and calculation-nodes, in combination with an expression language in activity-diagrams, are added. The contribution includes the following ideas, tools and frameworks (see also Fig. 10.1):

- A data and image processing language with special GPU related language constructs, allowing dataflow-driven textual, object oriented programming
- Novel modeling concepts like conditional-, loop-, calculation- and loop-window-nodes
- GPU related control structures and types
- A code generation framework
- A GPU accessing and programming framework using OpenCL

Especially activity-diagrams and expressions give the developer a good possibility to abstract algorithms or behaviors in an object oriented manner. Activity-diagrams provide an easy way to model data flow or single system parts (see Listing 8). They support concurrency, conditional decisions and also loops. Besides action-nodes, transitions and transition guards (see Listing 7), special new features are proposed, to make GU-DSL's activity-diagrams more reusable and adaptable for GPGPU programming:

- Diagram input and output parameters
- Transition arguments
- Calculation-, loop- and loop-window-nodes
- GPU specific data types and keywords also supporting shared memory
- Special control structures, e.g. to support the simple processing of a Region Of Interest

Using the proposed new concepts with the textual and graphical editors, combined with a C++ code generator, it can reduce the complexity of data processing tasks as will be shown in the remaining sections. The language and the surrounding framework can be used to create an almost platform independent vision software and especially improve GPGPU development.

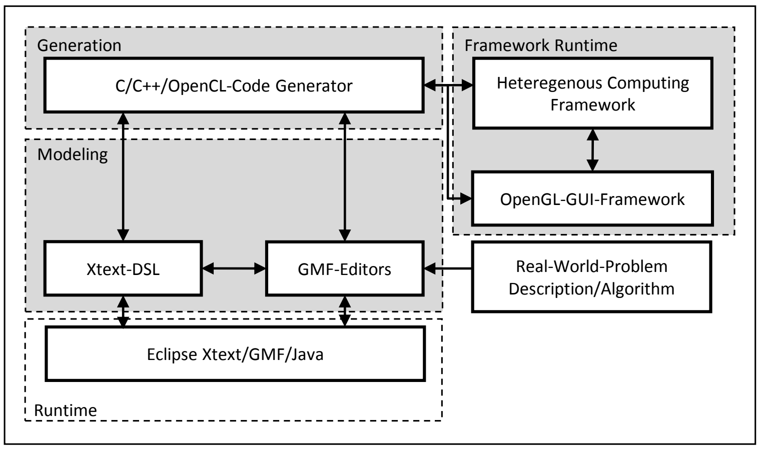


Figure 10.1: Overview of the proposed system, highlighted in light gray. [HFKK16]

*Publication: Flow Driven GPGPU Programming combining Textual and Graphical Programming [HFKK16]*

## 10.1 Related Work

This chapter presents GU-DSL with a special GPGPU related extension. The next sections will have a look at the most important work directly related to this approach.

An interesting approach is OptiML [SLB+11]. It is a DSL with a Matlab-like syntax and has been especially designed for CUDA GPU interoperability and allows for a simplified and more abstract way of GPU modeling.

In 2011, Han and Abdelrahman presented hiCUDA [HA11], a directive-based language. Using specially designed compiler directives, it reduces the necessary overhead for memory allocation and performance optimizations.

Forma [RHG15] is another DSL specially designed for image processing applications. Written code can be translated to GPUs and multi-core CPUs also supporting their individual special architectural features.

Another important language is HIPAcc [MRH+16] allowing to design image processing kernels and algorithms using a high-level DSL. High-level code can be translated into different GPU-types using a new compiler.

Different from the approach presented here, the shown related work just tries to simplify the GPU programming while GU-DSL integrates all development steps required for CPU and GPU interaction and programming, which is unique in this domain. Furthermore, compared to other works, GPGPU programming is brought to a high-level, dataflow-driven textual and also graphical abstraction which can additionally improve and speed-up development. The main focus lies on the improvement of GPU code structure and not just its simplification as most related work does.

In 2012, Kehrer et al. [KBKR12] have presented a system for modeling GPU applications, showing how it can improve GPGPU programming. Therefore they use the Eclipse Modeling Framework and a code generator. The difference to this approach is that they base their work directly on the Eclipse Modeling Framework domain model, restricting them to graphical modeling. In contrast to it, GU-DSL also supports generic and GPU-specific textual and graphical modeling specially adapted to the needs of this domain.

Besides the GPU related work, also other interesting related work about data and flow modeling exists.

Using flow diagrams is a common practice in behavior modeling of data and computer vision processing systems. Schumacher et al. [SHGR11] propose an approach to allow the modeling and code generation of signal processing systems to extend activity-diagrams in order to improve support for repetitive and recursive process modeling.

In [SP09], Sulistyo and Prinz show how to combine top-down (recursive model refinements) and bottom-up (extending programming environments) modeling approaches to allow full code generation, resulting in an executable system. Therefore they recursively optimize an activity-diagram from a very coarse to a fine model representation, which can then be transformed into an executable using predefined rules.

An important work directly related to the presented approach is the work of Efftinge et. al [EEK+12]. They present Xbase as a reusable part of xText, allowing the usage of expressions in a control flow. Furthermore, they provide full interoperability between Java and Xbase based DSLs. This gives engineers the possibility to start the development of new DSLs from a generic base. Parts of the Xbase Extended Backus Naur Form (EBNF) notation are used for the expression language.

Engelen and Van Den Brand [EvdB10] propose an approach to textually represent activity-diagrams and transform them into the diagram's graphical UML representation. Scheidgen [Sch08] shows how textual modeling can be integrated into Eclipse Graphical Modeling Editors. It maps the textual and also the graphical elements to meta-model objects and allows the transformation between both representations. This approach uses GMF and xText to combine and transform between the textual and graphical representations.

Another important tool that has to be mentioned here is LabVIEW [Ins16], developed and maintained by National Instruments. It is a visual programming system, providing reusable function blocks, that can be connected and parameterized in an Integrated Development Environment (IDE). It also supports visual CUDA Programming, allowing algorithm development, but it is mainly limited to graphical modeling.

Besides CUDA and OpenCL, Microsoft also introduced a native programming model called C++ AMP [Mic16a]. This library is based on DirectX and directly integrates itself into the C++ runtime environment, using parallel programming capabilities of both CPU and GPU. But still, the complexity of parallel algorithm development is not sufficiently reduced.

## 10.2 A Generic Data- and Image-Processing-Language for GPGPU-programming

Modern object oriented programming languages in general use classes and namespaces to form an abstraction of real-world objects. This allows structuring software by encapsulating functionality belonging together. The following sections will show the special novel GPGPU *GU-DSL* extension fulfilling the special GPU requirements as parallelization, special control structures, host/device management, kernel and memory management and ROI handling (see also Chapter 10).

### 10.2.1 GPGPU Behavior Modeling using Expressions and Activity-Diagrams

This section shows how to use textual activity-diagrams and expressions to give the developer the possibility to abstract algorithms or behaviors in a more object oriented manner. Activity-diagrams provide an easy way to model data flow or single system parts as stated in the introduction.

#### 10.2.1.1 Class-Assignments

GU-DSL allows the assignment of classes to activity-diagrams or to individual swimlanes as shown in Sec. 9.1. This allows the developer to directly interact with the owner class. It provides access to all class member variables and methods as can be seen in Listing 8 (e.g. the *load*-node). There are several ways on how GPU support can be added to the DSL. The first possibility is extending the DSL itself, but with the drawback that adding new features always requires rebuilding the DSL. A second possibility is the definition of a GPU class interface containing all GPU related methods. This allows a simple extension of the range of functions while keeping the DSL simple. To make the functionality available for algorithm design, a two-step approach is used. In the first step, a new class containing all GPU related filters has to be modeled extending the GPU interface class. Next, the class is assigned as owner to all required and implemented activity-diagrams.

#### 10.2.1.2 Diagram Input Parameter, Transition-Arguments and Guards

One of the most important system features is the implementation of diagram input and output parameters, as well as objects, which are an essential extension to allow the re-usage of diagrams. The second important newly introduced feature are transition arguments. They are passed along a transition (data- and object-flow) to the

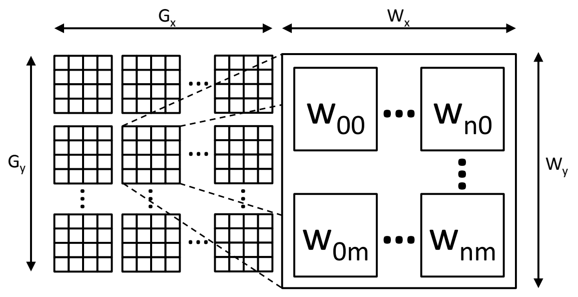


Figure 10.2: 2D work-grid G (left) with an exemplary work-group *w* (right) (based on [AMD16]).

destination action, allowing to call the action multiple times having different entry parameters. This is the key feature for the presented GPGPU programming system as will be seen in Sec. 10.2.1.3. Listing 8 shows the implementation of the *load*-method introduced in Listing 1 using input parameters and transition arguments between nodes. Besides arguments, transitions can be guarded by guard expressions. This allows to design multiple flow paths, which e.g. is required for loops.

#### 10.2.1.3 Calculation nodes

Calculation nodes are a GPGPU specific extension. As stated in the previous sections, algorithms or data processing tasks have to be split in multiple, small sub-tasks that can be executed in parallel. The developer has to decide if the problem is treated as 1D, 2D, or 3D, since the proper number and dimension of work-groups depend on this. For example, in image processing tasks, $n_{pixels} = width_{img} \times height_{img}$ have to be processed.

In this case, the problem can be assumed to be 2D. Applying the filtering step means that a work item has to be created for every pixel on the GPU (see Fig. 10.2). The number of work-groups $n_{work-groups}$ can be chosen in the following manner, assuming $16 \times 16 = 256$ work-items per work-group:

$$\begin{aligned} n_{work-groups_x} &= ceil(width_{img}/16) \\ n_{work-groups_y} &= ceil(height_{img}/16) \end{aligned}$$

Among other things, the ideal number of work-items depends on the GPU hardware capabilities. The calculation-nodes allow passing the dimension (dimension-x, dimension-y, dimension-z; in this case the image width and/or image height, z can be ignored at this time) along the incoming source transition (see Listing 9). For a more detailed description of work-groups and work-items have e.g. a look at [AMD16].

```
// Pass the image size along the transition to the calc-node.
// It is used to determine the ideal number of work-groups and work-items
=> filterData (image.width(), image.height(), 0);
```

**Listing 9** Transition arguments.

The ideal number of work-groups and work-items is chosen by the underlying heterogeneous-computing-system and its declaration can be done like in Listing 10.

```
// Definition of GPU calc-node. Note: The work-size
// parameters don't have to be declared
calc filterData()
{ // call the GPU filter that is modeled in a separate class
  call Filter::filterBilateral(imageContainer,
        outImageContainer, 13.0f, 10.0f, 50.0f);

  => showFilteredImage;
}
```

**Listing 10** An exemplary calculation node.

Knowing that the ideal number of items cannot always be chosen automatically, a second type of calculation node has been added allowing to manually pass the global- and local-processing kernel ranges (see Listing 11).

```
// Definition of GPU calcRang-node with two parameters. Note: The global-
// and local work-size parameters don't have to be declared
calcRange filterDataRange(int arg1, int arg2) { .... }
```

**Listing 11** Defintion of a range based calculation node.

As already shown for the standard calculation node, the dimensions are passed along the incoming transition. Instead of the three arguments, six arguments can be passed to the node, representing the global-range (x, y, z) and also the local-range (x, y, z). This kind of node is used in the GPU-reduction example shown in Sec. 10.4.

#### 10.2.1.4 Loop- and Loop-Window-Nodes

Image processing tasks often require looping across image areas to set neighboring pixels into relation. Typically, this kind of problem is solved using two or more nested for-loops. For this reason, two new additional node types:

- Loops: a loop-node is a simple node comparable to either a for- or a while loop well known from Java or C++
- Loop-Windows: a loop-window is the representation of an arbitrary number of nested loops

Loop-nodes allow the direct definition of loops within flow diagrams. Instead of the more complex and confusing connection of several action nodes to a ring, loop-nodes have up to four sections: a setup-, a test-, an increment- and an expression area. Depending on which of the first three sections are used, the node is either interpreted as for-, while- or do-while-loop. Loop-Windows are the second new node type that has been newly designed in the DSL. They allow to directly model nested loops within a single node. Depending on the number <*n*> of declared setups, the node is interpreted as <*n*> nested loops (see Listing 12).

```
// Declaration of a loop window using two different control variable types.
// The loopWindow node directly defines two nested loops compared to e.g. C++
loopWindow filterKernel(var int u = -n_radius, var float v = (float)-n_radius;
                        u < n_radius, v < n_radius;
                        u = u + 1, v = v + 1.0f)
{
  => writeOutput;
}
```

**Listing 12** Definition of a loop-window node.

Using these two kinds of nodes gives a much better system overview and improves the diagram maintainability. Both types are used in the examples in Sec. 10.4.

#### 10.2.1.5 GPU specific data types and keywords

Programming on GPUs has special functional requirements for programming languages. Fundamental data types, as *int*, *float* etc., have been well known for a long time. But this is in most cases not enough for 2D and 3D image processing tasks. Special data types such as *int4* or *float4* can help making complex tasks better understandable. Imagine a 4-channel image (red, green, blue, alpha; RGBA). Reading this data would result in four single values, but *float4* can simply group the values into one single structure. For this purpose, OpenCL and also CUDA add some special vector-data-types. Natively the following types *t* are supported: (u)short<*n*>, (u)int<*n*>, (u)long<*n*>, float<*n*>, double<*n*> with <*n*> $\in \{2,3,4,8,16\}$. Also, special image-types, *image2D_<t>*, *image3D_<t>*, respectively buffer-related types, *buffer2D_<t><n>* and *buffer3D_<t><n>*, with <t>$\in$ { c(char), uc(unsigned char), s(short), us(unsigned short), i(int), ui(unsigned int), l(long), ul(unsigned long), f(float), d(double)} and <*n*> $\in \{2,3,4,8,16\}$, have been added to the DSL. Adding type information to data types solves the drawback of the original OpenCL types (*image2D_t*, *image3D_t* and default vector-data-types), where the underlying type or at least the dimension is not available in a kernel call. Knowing the underlying type allows the validation and verification of parameter types before and during the code generation process, which can of course reduce the number of typical OpenCL kernel memory access violations. Furthermore, image access qualifier flags (*write* and *read*) have been added to the DSL allowing to pass image buffers with read, write or read-write access to an activity-diagram, respectively to a kernel call.

In Fig. 10.2 the concept of work-grids and -groups has been introduced. Each work-item within a work-group is processed separately. But very often, the exchange of information is required during execution. To share data between work-items, shared memory is used. The keyword *shared* has been added for this purpose. Declaring a method parameter with this keyword forces the code generator and OpenCL to reserve a special shared memory block for each work-group. The size in bytes of the memory block is passed along the node-transition as an argument instead of a variable.

#### 10.2.1.6 New control structures

Beside the previously introduced GPU data-types, the need for additional novel control structures has been identified. Similar to the introduced loop- and loop-window node-types, two new control structures have been added to the GU-DSL expression language part: *loopwindow2D* and *loopwindow3D*. Both of them are interpreted as nested for-loops in 2D and 3D respectively. This simply shortens the expression parts within nodes and helps avoiding mistakes.

## 10.3 Heterogeneous Computing and Code Generation

The DSL is designed in a way to provide best support and compatibility to C and C++, while supporting extensions of managed and type safe languages like Java or C# and it is also compatible with them. C and C++ are used as an intermediate state to transform the model to an executable. The generator is designed to meet the requirements of GPGPU programing as well as possible which will be shown in the next sections.

### 10.3.1 Heterogeneous Computing

Due to the fact that GPGPU framework initialization is a frequently recurring task, this code and additional processing code, such as kernel calls (see Fig. 10.5) and GPU memory management (see Fig. 10.6), has been encapsulated into a framework called Heterogeneous Computing. Furthermore, an OpenGL GUI framework is used, developed from scratch using Qt [QT16] as base user interface, to give the possibility to show and manipulate images processed by the GPU. The flow of code processing and executable-generation can be seen in Fig. 10.3. Starting from the graphical and/or from the textual model, the model is transformed to C++ and OpenCL code using an Xtend-based code generator. It fully incorporates the GUI- and the HC-framework. Finally, the generated code is compiled to an executable file. The Heterogeneous Computing framework works with a multi-layer principle (see Fig. 10.4). Structurally, there is the native OpenCL implementation (or/and other GPU frameworks in the future, *Layer 1*), encapsulating the available platforms and computing devices (e.g. the graphic cards) (*Layer 2*). *Layer 3* abstracts the wrappers to a common framework, hiding platform specific implementations from the programer. On top of this abstraction layer, the simplification *Layer 4* has been added, giving programmers the opportunity to use simple kernel and memory containers, instead of the more complex framework layer.

Depending on the requirements, it is possible to mix *Layer 3* and *Layer 4* and to switch between them. This means that the designer/developer has always full access to the underlying HC-framework of *Layer 3* to e.g. manually do some kind of kernel- or memory management.

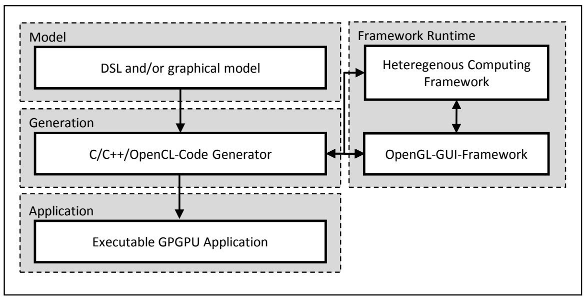

Model
DSL and/or graphical model
Generation
C/C++/OpenCL-Code Generator
Application
Executable GPGPU Application
Framework Runtime
Heteregenous Computing Framework
OpenGL-GUI-Framework

Figure 10.3: Code generation flow starting from the model as entry point. [HFKK16]

The two most important framework classes will be introduced here. The first one is the KernelCall class. It abstracts the setting of arguments, the kernel execution and has additional timing information (see Fig. 10.5). This class wraps the framework kernel *HCKernel* and is used by the code generator for the generation of calculation-nodes.

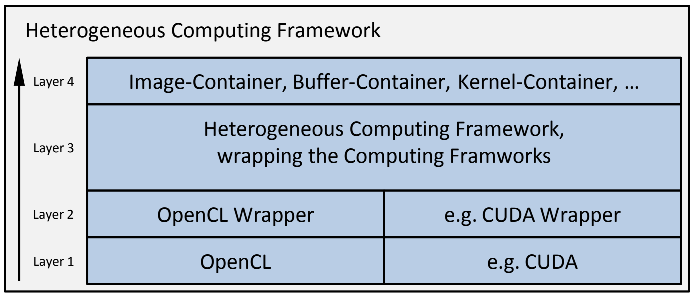

Heterogeneous Computing Framework
Layer 4
Image-Container, Buffer-Container, Kernel-Container, ...
Layer 3
Heterogeneous Computing Framework, wrapping the Computing Framworks
Layer 2
OpenCL Wrapper
e.g. CUDA Wrapper
Layer 1
OpenCL
e.g. CUDA

Figure 10.4: The multi-layer schema of the proposed Heterogeneous Computing Framework.

Other important classes are the image- and buffer containers wrapping GPU memory and allowing CPU/GPU memory copy and synchronization (see Fig. 10.6) and are used often in these examples. The framework internally allocates the memory, stores the containers and handles the full lifecycle, which makes it unnecessary to manually release it (but it is still possible) and allows the reusage without a reallocation.

Another important concept introduced with the computing framework are validators. Every memory container owns a unique validator that can be extended using rules. During the generation it is possible to verify the initialization by checking all registered validation rules, as e.g. checking the dimensions or type. Furthermore this concept is added to the KernelCall class (see Fig. 10.5).

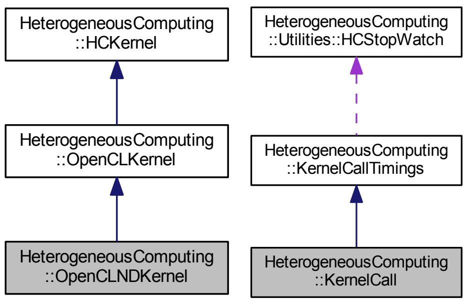

HeterogeneousComputing ::HCKernel
HeterogeneousComputing ::OpenCLKernel
HeterogeneousComputing ::OpenCLNDKernel
HeterogeneousComputing ::Utilities::HCStopWatch
HeterogeneousComputing ::KernelCallTimings
HeterogeneousComputing ::KernelCall

Figure 10.5: Left: the kernel container inheritance diagram. Right: the kernel call inheritance diagram of the GPGPU framework.

The code generator is able to add parameter specific rules to the KernelCall class instance, allowing for runtime validation against the wrapped OpenCL kernel parameters. This can reduce errors due to wrong parameter types.

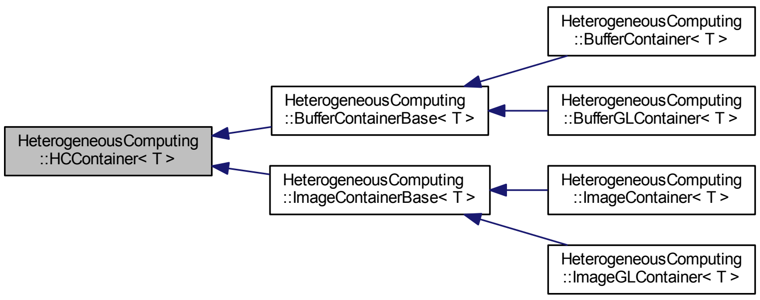

HeterogeneousComputing ::HCContainer< T >
HeterogeneousComputing ::BufferContainerBase< T >
HeterogeneousComputing ::ImageContainerBase< T >
HeterogeneousComputing ::BufferContainer< T >
HeterogeneousComputing ::BufferGLContainer< T >
HeterogeneousComputing ::ImageContainer< T >
HeterogeneousComputing ::ImageGLContainer< T >

Figure 10.6: The memory container inheritance diagram of the GPGPU framework.

### 10.3.2 The Code Generator

This section will introduce the most important features of the code generator and how special GPU control structures, presented in the previous sections, are transformed from GU-DSL to OpenCL and C++.

#### 10.3.2.1 Main Diagram

The code generator itself has special extensions supporting the generation of executables. Applications need a special *main*-function that is called during start-up. It is the entry point for all kinds of executables and has the base logic for program execution. The generator always tries to find an activity-diagram with the name *Main* to find the entry point of the model. An alternative solution can be to define and bind a special entry attribute to the diagram, similar to the class attributes described in the next section. This main diagram is compiled to the C++ main function and builds the start-up of the model.

#### 10.3.2.2 Class Attributes

A second special feature of the code generator is the possibility of hiding classes from generation or treating them in a special way, which is necessary for this approach's OpenCL code generation (see Sec. 10.3.2.3 - OpenCL support and Listing 14). Before a class or interface is compiled, the generator checks if the developer has put one or both of the following attributes on the top of the class:

- GeneratorVisibility
- CppType

The *GeneratorVisibility* attribute (see Listing 13) decides if the class is generated to C++-code. *CppType* allows mapping a modeled class to an already existing C++ type, including additional import files. In combination, the two attributes allow the re-usage and the full C++ class and framework interoperability, especially with the *Heterogeneous Computing* and GUI framework.

```
// Hide the class from being generated as additional C++ code, but map all the occurrences
// to the already existing C++ type: HeterogeneousComputing::Image2DContainer
[Attributes.GeneratorVisibility(visible = false),
Filtering.Attributes.CppType(name="HeterogeneousComputing::Image2DContainer")]
public class Image2DContainer<T> { /*some code*/  }
```

**Listing 13** Usage example of generator attributes. [HFKK16]

#### 10.3.2.3 OpenCL support

OpenCL provides predefined keywords, types and functions. The interoperability between the modeling toolchain and OpenCL can be achieved using the following mechanism: an interface with the name *OpenCL* has to be defined and marked invisible for generation: *GeneratorVisibility(visible = false)*. The interface defines all available OpenCL functions and classes implementing this special interface will now be interpreted as OpenCL file by the generator (see Listing 14). Activity-diagrams receive OpenCL support by assigning the OpenCL extension classes as owner (see Sec. 10.2.1.1). OpenCL itself is a C dialect and can easily be created in a similar way to the other expressions from the GU-DSL expression-language. Once the GPU side is generated, the host side (CPU) has to be generated. Therefore, all the calculation nodes are generated as methods filling a *KernelCall*-class instance with all the required arguments and the execution call. The problem size estimation is done in an underlying worksize-calculator. So just the 1D-, 2D- and 3D- problem size is passed along a transition to the calculation node. The ideal local ranges can then automatically be chosen.

```
// All declarations have to be invisible for the generator and are just used for modeling
ClassDiagram GPU
{
  // Hide the enumeration from generation
  [Generator.Attributes.GeneratorVisibility(visible = false),
     Generator.Attributes.CppType(name="sampler_t")]
  enum sampler_t
  {
    CLK_NORMALIZED_COORDS_FALSE,
    CLK_ADDRESS_REPEAT,
    CLK_FILTER_NEAREST
  }

  // ... All other OpenCL type declarations

  import "<QImage>";
  import "<vector>";

  import Generator.Attributes.*;

  // Hide the interface from generation
  [GeneratorVisibility(visible = false)]
  public interface OpenCL
  {
    public int get_global_id(int dimension);
    public int get_global_size(int dimension);

    public float4 read_imagef(image2D_f image, sampler_t sampler, int2 coordinate);
    public void write_imagef(image2D_f image, int2 coordinate, float4 value);

    // ... All other OpenCL declarations
  }
}
```

**Listing 14** OpenCL interface declaration.

#### 10.3.2.4 Activity-Diagram Generation

In general, activity-diagrams are comparable to directed graphs. They can be transformed in a recursive way. Starting from the initial-node, the diagram can recursively be traversed from node to node until the final-node is reached. Special method-nodes, accepting in- and out-parameters are generated as methods, while simple nodes are generated in place. But this concept would be too simple, if performance and code reduction were one of the goals. Furthermore, it has several drawbacks such as code redundancy and the missing loop detection. Both stated problems are handled by the code generator. Code redundancy can be reduced by a look-up in a graph-node-table. Therefore, the activity-diagram is treated as a directed graph and stored in a graph-node look-up table to analyze all reachable nodes from a single node. Once a transition of a node is generated, a backward look-up in the graph-table is performed to check, if the transition target is also reachable unguardedly by a preceding node. If this is the case, the code is not generated again at this place. This reduces code redundancy enormously. The second problem are loops. For loop-detection a way similar to the one for redundancy reduction is used. For this a forward look-up in the graph-node-table is performed. It is checked, if the current node is reachable by a following node. If this is the case, a while-loop is generated. This is implemented for an arbitrary number of nested nodes.

#### 10.3.2.5 Generation of Control Structures and OpenCL keywords

In Sec. 10.2 several new control structures, OpenCL types and keywords have been introduced. The GPU code generator part is able to handle these structures, especially the loop-windows and the shared memory. Loop-nodes are generated either as while-loop (if only the condition section is filled), as do-while-loop, if a specific do-while-loop-flag is set, or a for-loop. To simplify nested loop usage, additional loop-window-nodes that are directly generated as <n> nested for loops have been defined, depending on the number of loop control variable declarations in the loop setup section. Another important feature that is handled by the code generator is the OpenCL shared memory. The desired size is directly passed as a kernel argument to tell the OpenCL-compiler to allocate shared memory accessible to all work-items within a work-group. The OpenCL specific DSL-data-types are first translated into the correct OpenCL names and types and can then directly be created and used.

## 10.4 Evaluation

The previous sections introduced the proposed modeling framework in a generic way. An overview of the general possibilities using the textual language and the corresponding graphical editors has been given. In this section it will be shown how three different problems can be modeled and implemented with this Computer Aided Software Engineering (CASE) toolchain. Two different noise-reducing image processing filters (as e.g. described in Sec. 6.5.1) have been implemented: a *Mean* Filter (averaging the neighborhood pixel-values, see Sec. 10.4.1) and the more complex, edge-preserving *Bilateral* Filter (see Sec. 10.4.2). Furthermore, a typical reduction problem will be shown and analyzed (see Sec. 10.4.3).

### 10.4.1 Mean Filter

A *Mean* Filter is a rather simple filter. It allows for noise reduction by averaging a neighborhood window of fixed size. Depending on this window size, images are smoothed to a smaller or larger degree. In general, the filter size should be an odd number. This ensures, that the window center can represent the pixel that is to be filtered. Noise reduction is a recurring task and this filter can be parallelized easily. It was taken as a first example and reference implementation using GU-DSL. Every pixel can be handled on its own. The neighborhood pixel look-ups are realized using the previously introduced loop-window-node. This keeps the filter clear and short. The only special point that should be kept in mind is that border pixels (pixels where the filter window does not fully fit into the image) have to be handled in a special way (e.g. by using the original value or setting them to black). The corresponding textual activity diagram can be found in Listing 15 and its graphical counterpart is available in Fig. 10.7.

### 10.4.2 Bilateral Filter

The *Bilateral* Filter [TM98] is another kind of noise reduction filter and has an edge preserving characteristic. It can be implemented in a very compact way similar to the *Mean* Filter using the 2D loop-window nodes. It has the same restrictions as all the other neighborhood related convolution filters for border pixels (see Sec. 10.4.1). Due to the Gaussian weighting functions, it is much slower compared to the *Mean* Filter, as can be seen in Sec. 10.4.5, despite using heavy parallelization. The important parts of the corresponding textual activity diagram can be found in Listing 16.

### 10.4.3 Reduction

Reduction problems are typical and required data processing tasks on GPUs. The general principle of reduction is to take a data array and to reduce it to a single remaining element. Depending on the aims of the reduction (building a sum, finding the minimum or maximum, ...), the elements are reduced in different ways (by e.g. arithmetic or logical operations). The reduction is performed on a per work-group basis, meaning all work-groups are reduced to a single element each. Using OpenCL-thread synchronization, the number of work-items is reduced step by step by first reading from global memory to shared memory and performing e.g. a comparison. The winning work-item then continues this comparison until one work-item per work-group remains. This remaining value is finally written back to global memory. In a second step, the remaining values can be reduced to a single element (see e.g. [AMD16]). A special kind of reduction has been chosen, which essentially searches for the array-index of the first value $v$ that satisfies the condition $v \geq x$. The full implementation can be found in the appendix (see Listing 34).

### 10.4.4 Implementation Details

The implementation itself can be done using the elements introduced during the previous sections. Depending on the preferences of the developer, the textual or the graphical editors can be used. For all three examples, a Main-activity-diagram (see Sec. 10.3.2.1 and Listing 30) has been used for the control flow. The diagram is divided into several sections, such as GPU buffer initialization and synchronization, the viewer setup, processing and displaying results. Beside the heavy usage of the expression language, the most important newly introduced language features used are loop- (viewer image initialization), calc- (*Bilateral* and *Mean* Filter execution) and calcRange-nodes (*findIndex*-execution). The usage can be seen in the full Main-activity-diagram in the appendix (see Listing 31 and Listing 32). While the *Bilateral* and *Mean* filter calculation nodes benefit from the framework's work-size calculator, the *findIndex*-reduction is formulated as 1D problem. So the optimal work-range calculation is implemented in the Main-diagram. Necessary framework C++ classes (GPU buffer, viewer, Qt, ...) are mapped using the attribute-mechanism shown in

Sec. 9.1.1. On the GPU side of implementation, an activity-diagram called by the execution Main-diagram is used, using the OpenCL implementing filter classes (and class assignments) (see Sec. 10.3.2). Special features used are the loop-window nodes, shared memory, as well as the newly introduced data-types (see Sec. 10.2.1).

```
ActivityDiagram AdFilterMean(image2D_uc inBuffer, image2D_uc outBuffer, int kernelSize)
{
  import OpenCL.GPU.*;

  public int gidx;
  public int gidy;
  public int n_radius;
  // Other variables
  // ...

  swimlane P1, owner Filtering.Filter2.MeanFilter
  {
    start S1
    {
      => init;
    }
    // Initialize filtering
    action init
    { // Read the global work-item coordinates
      gidx = get_global_id (0);
      gidy = get_global_id (1);
      n_radius = kernelSize/2;
      coordinate = new int2 (gidx , gidy);

      => checkValid;
    }
    // Check border values
    action checkValid
    {
      var int dst_width = get_global_size (0);
      var int dst_height = get_global_size (1);
      outValue = new int4( 0, 0, 0, 255 );
      // Check using guarded transitions, if a border
      // pixel is processed and possibly write an invalid output value
      [ ( gidx >= (dst_width - n_radius) ) || ( gidy >= (dst_height - n_radius) ) || ( gidx < n_radius ) || (
     gidy < n_radius ) ] => writeOutputInvalid;
      [] => filterKernel;

    }
    // Use the new filter window node to process a 2D window of pixels
    loopWindow filterKernel(var int u = -n_radius, var int v = -n_radius;
                            u < n_radius, v < n_radius;
                            u = u + 1, v = v + 1)
    {
      var int i = gidx + u;
      var int j = gidy + v;
      var int2 localCoordinate = new int2 ( i , j );
      var int4 currentPixi = read_imagei ( inBuffer , _sampler , localCoordinate );
      var float3 currentPixf = new float3(currentPixi._x/255.0f, currentPixi._y/255.0f, currentPixi._z/255.0f);
      accumulated = accumulated + currentPixf;
      count = count + 1;

      => writeOutput;
    }
    // Write a default output value, and finish
    action writeOutputInvalid
    {
      write_imagei(outBuffer, coordinate, outValue);

      => f;
    }
    // Write the mean filtered value and finish
    action writeOutput
    {
      var float3 outValuef = accumulated/count;
      outValue = new int4(outValuef._x*255, outValuef._y*255, outValuef._z*255, 255);
      write_imagei(outBuffer, coordinate, outValue);

      => f;
    }
    final f
  }
}
```

**Listing 15** Important parts of the Mean Filter activity-diagram using several of the newly proposed features. [HFKK16]

```
ActivityDiagram AdFilterBilateral(image2D_uc inBuffer, image2D_uc outBuffer, int kernelSize)
{
  // Declarations ...

  swimlane P1, owner Filtering.Filter2.BilateralFilter
  {
    // Other node definitions ...

    // Use the new filter window node to process a 2D window of pixels
    loopWindow filterKernel(var int u = -n_radius, var int v = -n_radius;
                            u < n_radius, v < n_radius;
                            u = u + 1, v = v + 1)
    {
      var int i = gidx + u;
      var int j = gidy + v;
      var int2 localCoordinate = new int2 ( i , j );
      var int4 currentPixi = read_imagei ( inBuffer , _sampler , localCoordinate );
      var float3 currentPixf = new float3(currentPixi._x/255.0f, currentPixi._y/255.0f, currentPixi._z/255.0f);
      accumulated = accumulated + currentPixf;
      count = count + 1;

      => writeOutput;
    }

    // Other node definitions ...
  }
}
```

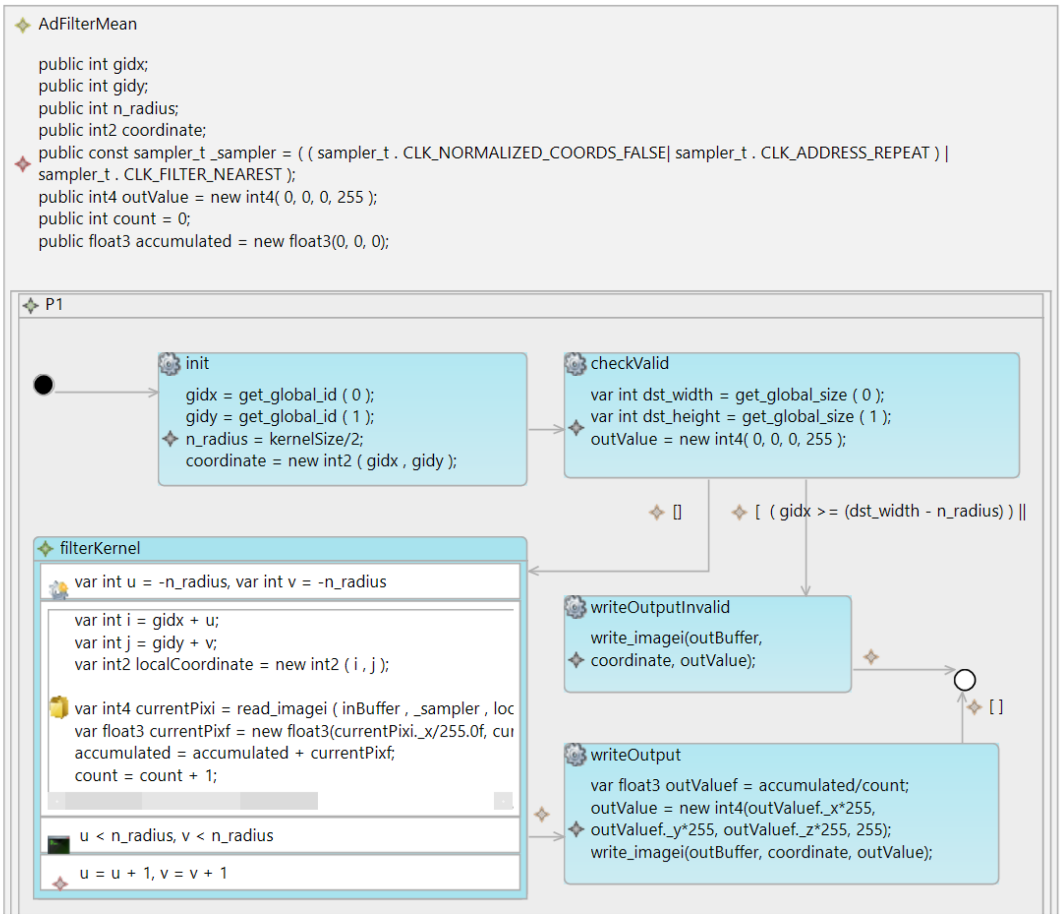
**Listing 16** Important parts of the Bilateral Filter activity-diagram. Most parts are comparable to the mean filter implementation of Listing 15.

### 10.4.5 Results

This section will show the results of the previously described examples. It mainly focuses on a performance analysis of the generated OpenCL code in comparison to manually written OpenCL code.

#### 10.4.5.1 Performance Analysis

The analysis starts with a performance comparison of generated OpenCL kernel calls against manually optimized OpenCL kernel calls. There are two goals of this analysis. The first one is to evaluate the influence of the proposed framework on the execution time, while the second one is to analyze the speed of the generated OpenCL code. Therefore a sequential native C++ application has been implemented doing all the OpenCL initialization by hand (optimized executable). This reduces the overhead of the object oriented GPU framework. Additionally, all three examples have been implemented by hand (OpenCL code manually coded). In parallel the same functionality has been implemented using the proposed CASE toolchain (executable and OpenCL code generated). To make the comparison more meaningful, the graphical implementation was designed using a similar code structure, but as an activity-diagram. Fig. 10.8 shows the execution time and the lines of code (LOC) results achieved on an Intel(R) Core(TM) i7-4770K CPU @ 3.50GHz, with a NVIDIA GeForce GTX 770 graphics card by averaging 1000 iterations. The image used for *bilateral* and *mean* filtering is "Lena" (512x512 pixels), generally used for image processing demonstrations (it can be found in the image database of the University of Southern California [USC16]).



```
public int gidx;
public int gidy;
public int n_radius;
public int2 coordinate;
public const sampler_t _sampler = ( ( sampler_t . CLK_NORMALIZED_COORDS_FALSE| sampler_t . CLK_ADDRESS_REPEAT ) |
sampler_t . CLK_FILTER_NEAREST );
public int4 outValue = new int4( 0, 0, 0, 255 );
public int count = 0;
public float3 accumulated = new float3(0, 0, 0);
```

```
gidx = get_global_id ( 0 );
gidy = get_global_id ( 1 );
n_radius = kernelSize/2;
coordinate = new int2 ( gidx , gidy );
```

```
var int dst_width = get_global_size ( 0 );
var int dst_height = get_global_size ( 1 );
outValue = new int4( 0, 0, 0, 255 );
```

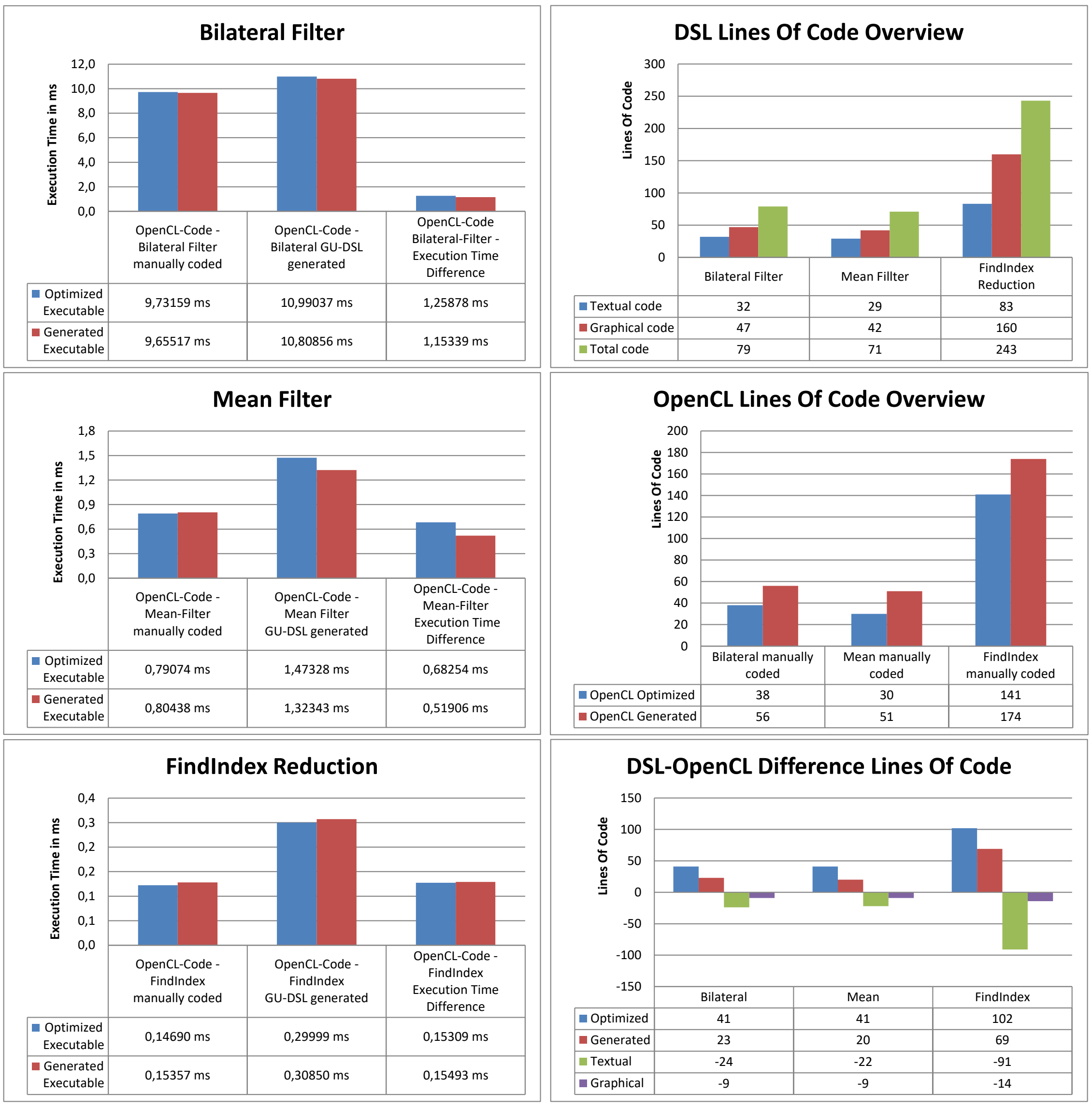
```
var int u = -n_radius, var int v = -n_radius

var int i = gidx + u;
var int j = gidy + v;
var int2 localCoordinate = new int2 ( i , j );

var int4 currentPixi = read_imagei ( inBuffer , _sampler , loc
var float3 currentPixf = new float3(currentPixi._x/255.0f, cu
accumulated = accumulated + currentPixf;
count = count + 1;

u < n_radius, v < n_radius

u = u + 1, v = v + 1
```

```
write_imagei(outBuffer,
coordinate, outValue);
```

```
var float3 outValuef = accumulated/count;
outValue = new int4(outValuef._x*255,
outValuef._y*255, outValuef._z*255, 255);
write_imagei(outBuffer, coordinate, outValue);
```

Figure 10.7: Important parts of the graphical Mean Filter activity-diagram using several of the newly proposed features. [HFKK16]

On the left top of Fig. 10.8, it can be seen that the average execution time of the optimized *Bilateral* Filter is 9.73159 *ms* and the generated version takes 9.65517 *ms*. In comparison to it, the execution times are shown that have been reached with the fully generated executable. The optimized OpenCL code runs within 10.99037 *ms* while the generated OpenCL kernel takes 10.80856 *ms*. This means, that the optimized code requires 1.25878 *ms* and the generated 1.15339 *ms* more time for execution in the fully generated code (this is about 12% slower compared to the optimized code). The left center figure shows the different execution times of the *Mean* Filter, where the average execution time of the optimized *Mean* Filter in the optimized environment is 0.79074 *ms* and for the generated OpenCL code 0.80438 *ms*. In the generated executable, the execution time is 1.47328 *ms* for the optimized OpenCL and 1.32343 *ms* for the generated code. The generated OpenCL code is 0.68254 *ms* slower in the optimized executable, while it is 0.51906 *ms* slower in the generated executable. This is about 87 %, respectively 65 % slower compared to optimized code. The third example (the *findIndex*-reduction) can be seen in the left bottom figure. The optimized



**Bilateral Filter**

| | OpenCL-Code - Bilateral Filter manually coded | OpenCL-Code - Bilateral GU-DSL generated | OpenCL-Code Bilateral-Filter - Execution Time Difference |
|---|---|---|---|
| 🟦 Optimized Executable | 9,73159 ms | 10,99037 ms | 1,25878 ms |
| 🟥 Generated Executable | 9,65517 ms | 10,80856 ms | 1,15339 ms |

**DSL Lines Of Code Overview**

| | Bilateral Filter | Mean Fillter | FindIndex Reduction |
|---|---|---|---|
| 🟦 Textual code | 32 | 29 | 83 |
| 🟥 Graphical code | 47 | 42 | 160 |
| 🟩 Total code | 79 | 71 | 243 |

**Mean Filter**

| | OpenCL-Code - Mean-Filter manually coded | OpenCL-Code - Mean Filter GU-DSL generated | OpenCL-Code - Mean-Filter Execution Time Difference |
|---|---|---|---|
| 🟦 Optimized Executable | 0,79074 ms | 1,47328 ms | 0,68254 ms |
| 🟥 Generated Executable | 0,80438 ms | 1,32343 ms | 0,51906 ms |

**OpenCL Lines Of Code Overview**

| | Bilateral manually coded | Mean manually coded | FindIndex manually coded |
|---|---|---|---|
| 🟦 OpenCL Optimized | 38 | 30 | 141 |
| 🟥 OpenCL Generated | 56 | 51 | 174 |

**FindIndex Reduction**

| | OpenCL-Code - FindIndex manually coded | OpenCL-Code - FindIndex GU-DSL generated | OpenCL-Code - FindIndex Execution Time Difference |
|---|---|---|---|
| 🟦 Optimized Executable | 0,14690 ms | 0,29999 ms | 0,15309 ms |
| 🟥 Generated Executable | 0,15357 ms | 0,30850 ms | 0,15493 ms |

**DSL-OpenCL Difference Lines Of Code**

| | Bilateral | Mean | FindIndex |
|---|---|---|---|
| 🟦 Optimized | 41 | 41 | 102 |
| 🟥 Generated | 23 | 20 | 69 |
| 🟩 Textual | -24 | -22 | -91 |
| 🟪 Graphical | -9 | -9 | -14 |

Figure 10.8: Left side: the figures show the average execution time of generated OpenCL code in comparison to manually optimized OpenCL code in a raw OpenCL-executable (manually coded) and an executable using the proposed framework (GU-DSL generated). Right side: the figures show the lines of code overview of all examples. Top: lines of GU-DSL code split in textual (pure implementation), graphical (e.g. node declarations and transitions) and total code. Center: lines of OpenCL code optimized and generated. Bottom: the difference between the lines of code of GU-DSL and OpenCL. [HFKK16]

OpenCL code requires 0.14690 *ms* and the generated OpenCL code requires 0,15357 *ms* in the optimized executable. Using the fully generated executable, the optimized OpenCL code can be executed within 0.29999 *ms* and the generated code within

0.30850 *ms*. This means that compared to the optimized executable, the generated code is slower by 0.15309 *ms* and 0.15493 *ms* respectively (which is about half as fast). In summary, it can be inferred from these results that the purely generated OpenCL code is comparable to the manually optimized code in performance, at least for algorithm implementations with larger execution times (e.g. the *Bilateral* Filter). For algorithms with short execution times, the acceptability depends on the system requirements. The explanation of the high percentage differences is simple and can be deduced from the *Bilateral* Filter. For short or fast kernels, the proportionate cost for passing OpenCL kernel arguments and kernel calls is rather high. The slower the kernel runs, the less this overhead carries weight. This can be a typical result for object oriented programming in comparison to fast, but often confusing sequential development. This means in total that the generated OpenCL code itself can be fully accepted in the performance domain, while the overhead of the underlying HC framework can unfortunately not be completely ignored. But it is acceptable in many cases.

#### 10.4.5.2 Lines Of Code Analysis

The second interesting evaluation are the lines of code used for representing the algorithms (see Fig. 10.8). It is clear, that high-level structuring causes additional lines of code, which is generally applicable for high-level programming languages. In this section GU-DSL code (right top and right center in Fig. 10.8) and also generated versus optimized OpenCL code (right bottom figure) will be compared. In the right top figure the number of required lines for the examples can be seen, being split into three different textual metrics: the total number of necessary lines, pure textual code and also the graphical, respectively the activity-diagram code. In the right center figure, optimized OpenCL code is compared against the GU-DSL generated code. The right bottom figure concludes the lines of code analysis showing the differences of lines of code between GU-DSL and OpenCL code. It can be seen that 79 LOC are necessary for the *Bilateral* Filter, 71 LOC for the *Mean* Filter and 243 LOC for the *findIndex*-reduction using GU-DSL. Having a look on the optimized OpenCL code 38 LOC are required for the *Bilateral* Filter, 30 LOC for the *Mean* Filter and 141 LOC for the *findIndex*, while the generated code uses 56 LOC (≈ 47 % more) for the *Bilateral* Filter, 51 LOC (≈ 70 % more) for the *Mean* Filter and 174 LOC (≈ 23 % more) for the *findIndex*-reduction. The reasons for these differences are related to the way how the code-generator works. Every node is encapsulated into a block of curly braces. This leads to a huge number of potentially unnecessary braces but has no influence on performance. Finally, when comparing the GU-DSL code against the OpenCL code, it can be seen that 41 LOC (optimized, ≈ 108 %) and 23 LOC (generated, ≈ 41 %) more are required for the *Bilateral* Filter, 41 LOC (optimized, ≈ 136 %) and 20 LOC (generated, ≈ 39 %) more lines for the *Mean* Filter and 102 LOC (optimized, ≈ 72 %) and 69 LOC (generated, ≈ 39 %) for *findIndex*-reduction. At a first glance, these numbers seem to be very high, but when splitting the GU-DSL code into a pure textual (expressions) and separated graphical part (diagram-, node- and transition-definitions), it can be seen that the difference isn't that large. That means in detail that depending on the algorithm length, the additional graphical amount of LOC raises. But this is compensated using the graphical modeling approach presented in this thesis where the nodes just have to be placed and filled with necessary expressions. It can be said that a LOC overhead is present using the textual variant of GU-DSL, but in combination with the graphical editors, the overall advantages prevail.

### 10.4.6 Evaluation Summary

In total, GU-DSL and the corresponding generator have of course a non negligible overhead in both performance and lines of code (LOC). However, a performance gap using strong object orientation, as done with the proposed HC framework, will always occur independently to this approach (see also Sec. 10.4.5.1). Furthermore, it has been shown that the performance of the generated OpenCL code compared to the optimized code is approximately identical. Besides the performance comparison also a LOC comparison has been executed. It could be seen, that depending on the algorithm length, the additional graphical amount of LOC rises. However, this is compensated using the presented graphical modeling approach depicted in this thesis where the nodes just have to be placed and filled with necessary expressions. It can be said that an LOC overhead using the textual variant of GU-DSL is present, but in combination with the graphical editors, the overall advantages prevail. The LOC can mostly be ignored using the graphical model-driven development approach in combination with the built-in editors. Under observance of some simple points (splitting flows not to finely granular, using graphical editors and using the provided framework as often as possible), the presented approach for GPGPU programming has more advantages than disadvantages and it has potential improving the Time-to-Market by reducing many common mistakes.

## 10.5 Summary

In the previous sections, the novel GU-DSL GPGPU language extension has been introduced in combination with graphical editors, allowing developers and modelers to design fully runnable GPGPU applications in a model driven way. The modeling toolchain is based on Eclipse-xText to design the novel DSL and its text editors. Eclipse-GMF is used to develop the corresponding graphical designers. In combination with the code generator and the high-level abstraction *Heterogeneous Computing* OpenCL framework, it is possible to easily develop fast image processing filters, e.g. a Bilateral Filter or even more complex processing methods (e.g. the *findIndex*-reduction). An import result is that too fine- or coarse-granular modeling of algorithms minimizes the advantages (e.g. clarity and maintainability) of using the proposed toolchain. A good mixture between expression grouping in a node and nodes that represent single operations has to be found. If a developer keeps this in mind, then this toolchain can really improve the development of complex image and data processing GPGPU systems. It structures the code that has to be written and can assist the developer in avoiding common mistakes.

The future work will focus on CUDA support, further case studies and the support of automatic test- and documentation generation, which will additionally increase the significance of the presented novel DSL and editors in the sense of model-driven GPGPU development.

# 11 Component-Based Data And Image Processing Architectures

Setting up new data and image processing systems (e.g. such as Chapter 6) is an always recurring task. Having predefined DSLs and tested generic default runtime architectures, supporting functionality as Graphical-User-Interface (GUI) and algorithm interaction can catalyze development. As an add-on on top of a data processing runtime, Component-Based Software Engineering can help separating data processing problems and algorithms into packed, reusable software components. After designing such a component, engineers and developers are able to instantiate it and connect it to other components using predefined interfaces. However, one of the biggest drawbacks in the domain of data and image processing is that no real standard architecture has been established yet.

This work proposes an external DSL in combination with graphical modeling based on GU-DSL [HFKK15]. Furthermore, the complete infrastructure of a data and image processing runtime environment in combination with a C++ CBSE system is introduced, allowing algorithm development to remain in focus. The newly proposed language extensions are designed in a way to best fit the object and model oriented concept of GU-DSL.

GU-DSL and the C++ CBSE system realize a component approach supporting and introducing the following novel main features and the necessary infrastructure:

- A language concept and implementation of components using ports and interfaces based on xText
- Component- and novel component-instance-diagrams
- An exemplary C++ based CBSE system mapped to the GU-DSL features
- A prototype-based factory pattern for dynamic object registration and generation

- A Rich Client Platform (RCP) (console and GUI based) supporting a plugin based extension system (GUI and components)

*Publications: Abstracting Data and Image Processing Systems using a Component-Based Domain Specific Language [HKK16]; Component based data and image processing systems: A conceptual and practical approach [HKK15]*

## 11.1 Related Work

This thesis presents GU-DSL with a component extension and a C++ CBSE framework allowing for Component-Based Software Engineering. The next sections will have a look at some important related work.

The basis of this proposed work is GU-DSL [HFKK15]. It is a generic image processing DSL allowing engineers to develop image processing algorithms and tools in an abstract and simplified manner. It has a Java and C# like syntax supporting class- (structural modeling) and activity-diagrams (behavior modeling). Furthermore, it includes an expression language allowing sequential algorithm development.

Several other related languages and software engineering frameworks have been developed during the last years. Architecture Definition Languages play an important role for this development, allowing description of hardware as well as software architectures. GU-DSL and its infrastructure is exemplary compared to the most important of them.

A very early approach of formal architecture definitions is *Wright* [AG97]. It was introduced by R. Allen et. al in 1997, proposing ways how components can be interconnected.

In [MDK92], Magee et al. have introduced *Darwin*, a configuration language that allows for grouping process instances (an early kind of modern components) communicating by message passing.

Another standardized example of an ADL is *AADL* [FLVC05]. It was developed especially for automotive embedded hard- and software real-time systems, supporting several different types as devices, buses, processors (hardware-side) and threading or data processing (software-side). Furthermore, it can encapsulate visible functionality into connectable components.

UML [OMG16e] is another standardized, ADL like language supporting structural, sequential and behavioral graphical modeling of real world problems. It is proposed to be a generic applicable system and architecture language, giving architects the possibility of graphically designing and solving problem descriptions.

Besides ADLs, also other related component-based approaches and frameworks have been installed in several domains of software engineering in the past two decades.

An early example of a component-based robotic controlling framework is Smart-Soft [SW99b, SW99a] on basis of CORBA (Common Object Request Broker Architecture, supported by the OMG [OMG16a]), originally developed in the late 1990s. They propose concepts and patterns how electronic and software components of a robot can interact and how they can be designed in a reusable way. Furthermore, they introduce concepts of event driven communication and also model driven approaches [SHLS09].

In 2002, the CORBA Component Model (CCM) [OMG16b], based on CORBA 3.0, has been released supporting component-based distributed architectures and services.

Another widely spread approach of component-based architectures is Microsoft's Component Object Model (COM) and its distributed version (DCOM) [Mic16b]. It is mostly language independent using instantiable interfaces for an abstract view of distributed components.

Other interesting image-processing frameworks are OpenCV [Ope16] and the Point Cloud Library (PCL) [PCL16]. Both libraries have the main focus on providing simple interfaces to generic 2D and 3D image processing algorithms and viewers. But in general, to use the frameworks in real applications, the GUI and algorithm usage has to be manually coded. Using the GU-DSL attribute concept proposed in [HFKK15] it is easy to integrate this kind of library into the system.

In contrast to the mentioned languages and component systems, GU-DSL and the implemented framework provides the full functionality necessary for developing fast, interactive image processing pipelines. Furthermore, the framework supports the concept of round-trip engineering necessary for arbitrary switching between graphical and textual modeling. It specially focuses on object oriented textual and graphical modeling in the sense of model driven engineering. Starting with object abstractions (from object to class to component) up to the final system generation using the proposed C++-CBSE framework, the full development process is assisted.

Besides the presented ADLs, frameworks and concepts, many other systems exist, e.g. LabView [Ins16] with its dataflow visual programming language. But no system comparable to the proposed one in the domain of data and image processing has been available until now, supporting its own DSL (graphical and textual), an ADL, a code-generator and a corresponding C++-CBSE runtime system.

## 11.2 System Concept and Overview

GU-DSL is a diagram based, object oriented modeling language supporting structural- and behavior-modeling in a textual and graphical, model driven way. Apart from using class- and activity-diagrams to model objects and system behaviors, it also gives architects the possibility for sequential behavior coding using its embedded expression language (see [HFKK15]).

This chapter shows the component-based extension to GU-DSL. For component-

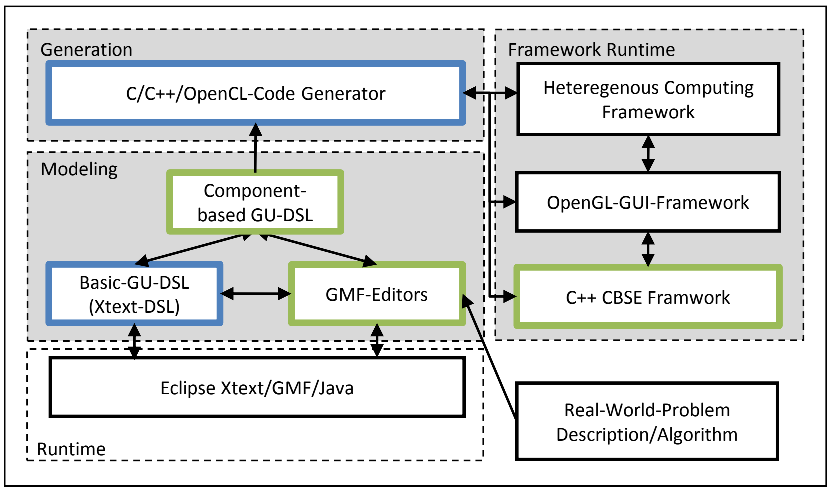


Figure 11.1: The principle schema of the component-based GU-DSL framework. [HKK16]

oriented modeling, the basic concepts specified in literature as interfaces, classes, components, ports and interface connections are picked up and integrated in a novel way into GU-DSL. Hence, special novel keywords, structures, component implementations and communication concepts have been added as will be shown in the following sections.

Fig. 11.1 shows the principle setup of the GU-DSL framework. Novel added features, like the component support, are highlighted in green, while the extended schema parts are marked blue. The system comprises three collaborating parts: modeling, code-generation and the runtime framework. The framework itself is also split into three parts: a Heterogeneous Computing framework (see also Chapter 10), based on OpenCL, an OpenGL GUI framework (allowing the interactive visualization of images or data) and the newly introduced C++ CBSE framework.

Data and image processing tasks can rapidly grow in complexity. The abstraction towards reusable components helps handling this. Having a look at such a processing system (see the examples in Chapter 6 and Sec. 11.5) shows, that it is in general a pipeline system starting with a source (often a camera, sensor or a database), continuing with filters (e.g. noise reduction or data evaluation) and finishing with sinks (e.g. showing or storing data). In software-engineering, this problem description can be mapped by the *pipe and filter architectural pattern*, allowing pipeline based data processing. It serves as a basis for the novel concept of a CBSE system as will be shown in the next sections. A detailed description of the particular GU-DSL framework parts can be found in Sec. 11.3.

Components form the basis of the system. Interfaces, known e.g. from UML, allow communication between components. The direction depends on the interface definition and can be either uni- or bi-directional. GU-DSL is defined with the following restrictions:

- **Sources:** outgoing communications (uni-directional)
- **Filters:** incoming and outgoing communications (bi-directional)
- **Sinks:** incoming communications (uni-directional)

Ports are used as flexible input and output points of components by providing required interfaces. Every port is uniquely assigned to a component and can be connected to other valid ports, allowing a predefined, direct communication as shown in the later sections.

## 11.3 GU-DSL – Component-Based Engineering

This section shows all the new features required for understanding the concept and novel ideas of component-based engineering using GU-DSL. In [HFKK15] and Chapter 9, the basic principles of GU-DSL have been introduced supporting classes, interfaces and activity-diagrams. But for the component concept, several extensions have to be implemented. Components are a well established concept in software engineering, however, the novel contribution of this approach is the way how all the different diagrams and language features interact to reduce errors (e.g. bugs, side-effects) and to speed up development using this tested and reusable framework. Especially the way how components interact with classes, types and methods using behavior-diagrams or sequential coding are unique in this domain and will be shown in the following section. Additionally, the proposed component extension can be used as a standalone approach for pure architectural design and modeling in the sense of Architecture Definition Languages, even without the GU-DSL basic functionality.

To better support reusable components, the system is split into two main parts:

1. Component-diagrams
2. Component-instance-diagrams

As typical for ADLs, this gives architects the opportunity of splitting a system into an architectural modeling layer (component-diagrams, where the basic system setup can be defined) and an instantiation layer (component-instance-diagrams, where components can be instantiated). This can be partly compared to the OMG Meta-Object-Facility (MOF, a four layer modeling architecture [OMG16c]).

The following sections will introduce all necessary new language features.

### 11.3.1 Interface Definitions

A central role in the CBSE system is played by interfaces. GU-DSL already supports them, but for the novel component extension, they are not sufficient. To make the system type- and communication-safe, two new keywords have been added: *processor* and *provider*. As shown in Listing 17, the new keywords can directly be applied to either defining an output-interface (provider) or an input-interface (processor). The keywords classify the new interface types to be usable with component ports, as will be shown in Sec. 11.3.3.

```
Interface:
  visibility=('public'|'protected'|'private')? 'interface'
    ('provider'|'processor')? name=ID
    '{' (methods+=Method)* '}'
```

**Listing 17** Component interface definitions. [HKK16]

An automatic communication specialization has been added in a way, that a provider-interface can talk to a corresponding processor-interface only. For this purpose, there is a restriction that a *provide*-method-signature needs a corresponding *process*-method-signature as can be seen in the example in Listing 26. It simplifies the usage, makes automatic code generation easier and allows for complete model validation. Which processor and which provider interface correspond to each other can be defined in the component-diagram by connecting ports (see Sec. 11.3.3).

## 11.3.2 Component Definitions

Components and their simple (re-)usage are the central new contribution of this thesis. In this approach, they are directly interconnected with the class-system known from class diagrams and are the base of the system architecture definition.

Therefore, an important extension in the GU-DSL's class diagram is added. The new keywords ***source***, ***filter*** and ***sink*** can be used in a similar way as the standard *class*-keyword (see Listing 18). Stemming from the domain of data and image processing, these special keywords characterize the components as specified in Sec. 11.2.

```
BaseClass:
  visibility=('public'|'protected'|'private')?
    ('class'|'source'|'filter'|'sink') name=ID
    '{'
        (attributes+=Attribute)*
        (methods+=Method)*
    '}'
```

**Listing 18** Component class definitions. [HKK16]

Once having a component-class defined, it is used as basis for the final component. Designing components this way gives architects the possibility to split the definition (component-diagram) from the concrete implementation (class-diagram, activity-diagram). Components are designed in the newly introduced component-diagrams (Listing 19). As for class-diagrams, there are also the three main types (*source*, *filter*, *sink*) available. To support generic components that cannot be mapped to these three types, the *class*-keyword is also available. The component definition syntax (see Listing 19) itself is quite simple: it is the previously defined fully qualified name (Fqn) used in the class-diagram (see Listing 1 and Listing 18).

```
Diagram:
  'ComponentDiagram' name=ID
  '{'
      (components+=Component)*
  '}';

Component:
  ('class'|'source'|'filter'|'sink'| name=[BaseClass|Fqn]
  '{'
    (ports+=Port)*
    (connections+=Connection)*
  '}';
```

**Listing 19** Component-diagram and component definitions. [HKK16]

As other GU-DSL diagram types (e.g. class- and activity-diagrams), component-diagrams support nesting of additional component-diagrams in arbitrary depth.

```
Component:
  ('class'|'source'|'filter'|'sink'| name=[BaseClass|Fqn]
  '{'
      (components+=Component)*
  '}';
```

**Listing 20** Nested component definitions.

For embedded components, the same design principles are applicable as for all other components. That means the underlying classes have to be defined first and ports have to be used in the same way (see Sec. 11.3.3).

#### 11.3.2.1 Component Interfaces

Components need interfaces to be able to communicate. Hence, two types of component interface implementations are provided:

1. Interfaces by inheritance
2. Anonymous interfaces

**Interfaces by Inheritance** They are the default interfaces that should be used during the component definition. In this case, the component classes inherit from the necessary interfaces and provide all the methods that can later be used by ports and during the component functionality implementation. An important feature is that, in this case, the interfaces' methods have to be implemented only if they require a special implementation. Otherwise, they are automatically available assuming a default implementation. This has the advantage of interface methods being directly accessible and usable without any additional code. By default, the code generator interprets missing interface method implementations and fills the missing implementation part with signal based interface-method calls (see Sec. 11.4.2). However, if a special kind of implementation is required, the interface methods can also be implemented using GU-DSL.

**Anonymous Interfaces** They are a special version of interface implementations. In this case, the component class does not have to implement the interface, but the interface method can still be called directly, e.g. in the sequential expression language introduced with GU-DSL [HFKK15]. Therefore a new kind of interface call has been added:

```
InterfaceCall:
  ('call' ('interface')? )?  interfaceName=Fqn '.' methodName=ID
    '(' parameterValues+=ParameterValue
     (',' parameterValues+=ParameterValue)* ')'
```

**Listing 21** Anonymous interface calls. [HKK16]

Anonymous interfaces are completely loose and the full interface definitions do not have to be available during design time. This gives designers the possibility to model components without having the full interface code. It can also be used to call interface methods from arbitrary places during component execution.

### 11.3.3 Port Definitions and Connections

Ports are the door to the outer and inner world of components. Allowing data exchange between components and nested components, ports provide the ability to define interfaces that handle how data can be transferred to and from a component. Therefore the previously defined interfaces (see Listing 17) can be chosen. The port definition within a component has a simple syntax (see Listing 22).

```
Port:
  ('async' ('buffered')? )? 'port' name=Fqn
    '(' connectors+=Connector
     (',' connectors+=Connector)* ')'

Connector:
  (Processor | Provider);

Processor:
  'processor' name=Fqn ':' interfaceref=[Interface|Fqn];

Provider:
  'provider' name=Fqn ':' interfaceref=[Interface|Fqn];

Connection:
  '[' name=ID 'source' sourceport=[Connector|Fqn] '==>' 'target' targetport=[Connector|Fqn] ']';
```

**Listing 22** Port and connection definitions. [HKK16]

The special preceded keyword *async* characterizes the port to be asynchronous, which means communication is forced to be non-blocking. Compared to standard blocking ports, a called interface method directly returns to the component and its sequential execution. A special variant of asynchronous ports are buffered ports. Incoming data can be buffered and is forwarded to the component if needed.

The detailed usage of components in combination with interfaces and ports can be found in the example in Sec. 11.5.

Ports play an important role when designing the system. By connecting two ports (source ==> target), it is defined which interfaces and which interface methods can be connected in the component-instance-diagram (see Sec. 11.3.4).

Once all components, interfaces, ports and connections are defined, the architecture definition is completed and the instantiation phase can begin as will be shown in the next sections.

### 11.3.4 Component Instance Definitions

Once components have been designed and interface/port wiring is completed, the architectural definition can be used in the component-instance-diagram. The component-instance-diagram is an important new feature added to be able to reuse components and to guarantee a type- and interface-safe component wiring. The diagram can be compared to the object-diagram known from UML, but it is specially designed for component-based engineering. Splitting the instantiation from the definition of types and objects is a well-known practice in software engineering, but is not fully established for component-based ADL approaches (or also UML) in this form.

The separation of definition and instantiation ensures two things:

1. Defined components can simply be reused and parametrized
2. Interface connections can be checked for validity during design

Using the component and port definitions of the component-diagram, the component instantiation can be realized as shown in Listing 23.

```
ComponentInstanceDiagram:
  'ComponentInstanceDiagram' name=ID
  '{'
      compinstances+=ComponentInstance*
  '}';

ComponentInstance:
  name=ID 'instantiates' componentType=[BaseClass|Fqn]
      '(' (fieldName+=ID '=' fieldValue=FieldValue
          (',' fieldName+=ID '=' fieldValue=FieldValue)*)? ')'
  '{'
      (ports+=CID_Port)*
      ((bindings+=Binding)*)
      (compinst+=ComponentInstance*)?
  '}';
```

**Listing 23** Component instance definitions. [HKK16]

While the component definition just defines fields and methods in the class declaration, the instantiation step assigns a unique instance name and initialization values to a component and its fields. Three points are necessary to completely instantiate a component. The first one is a unique name. As second point, the component type has to be assigned using the fully qualified name of a component defined in the component-diagram. The third important point is the assignment of new initialization values. They can directly be assigned within two parentheses after the type specification by a name-value combination.

Besides the component instantiation, the other important step is having connections allowing the communication and data-exchange between components. For this purpose, instances of ports are created first. Newly instantiated ports can then be wired. This syntax can be seen in Listing 24.

```
CID_Port:
  name=ID ':' type=ID
  '(' connectors+=CID_Connector (',' connectors+=CID_Connector)* ')' ';';

CID_Connector:
  (CID_Processor | CID_Provider);

CID_Processor:
  'processor' name=ID '=' processorInterfaceType=Fqn
  '(' (fieldName+=ID '=' fieldValue=FieldValue (',' fieldName+=ID '=' fieldValue=FieldValue)*)? ')'
      ';

CID_Provider:
  'provider' name=ID '=' providerInterfaceType=Fqn
  '(' (fieldName+=ID '=' fieldValue=FieldValue (',' fieldName+=ID '=' fieldValue=FieldValue)*)? ')'
      ';

Binding:
  ('[' 'source' sourceport=[CID_Connector|Fqn] '==>' 'target' targetport=[CID_Connector|Fqn] ']');
```

**Listing 24** Component connection definitions. [HKK16]

Connections are possible between processor and provider ports only. This is guaranteed by the architecture definition specified in the component-diagram. Validity checks can be applied by an Object Contstraint Language (see [OMG16d]) model validator (used in this approach) or any other kind of validator. Depending on the final kind of code generation and how ports are implemented on the target platform, it is also possible to initialize and assign values to port instances. This can e.g. be necessary for buffered ports shown in Listing 22, giving the opportunity to define the buffer size and so forth.

The connections are created between instantiated ports. Only port instances are used as source or target. The direction is unidirectional and it doesn't matter if the source is a provider and the target a processor or vice versa.

### 11.3.5 Component Initialization and Execution

While the previous sections have shown how components can be defined in general, this section shows how they can be implemented. Hence, three methods are automatically available. The first one is responsible for initialization (***init()***) and it is assumed to be called during the instance initialization, while the second method is called during the component execution ***process()***. The third important method is the ***cleanup()*** and is responsible to release e.g. memory or other resources. It is important to know that the ***process()***-method is assumed to be executed in a thread but this depends on the code-generator and the underlying CBSE framework. The implementation of all three methods is performed in the corresponding component class, either by using activity-diagrams or sequential code as shown in the example in Sec. 11.5.

### 11.3.6 Graphical Design Assistance

Besides the textual new features, special graphical editors have also been added allowing full graphical modeling in the sense of model driven development. Using graphical editors, system architects profit from the higher level of abstraction. Multiple lines of code can be added by drag and drop operations from a toolbox, placing e.g. complete components on a component-diagram. Also, connections can intuitively and interactively be added by simply selecting source and target ports. Sequential coding of method implementations is added using in-place editors supporting the full GU-DSL language specification, especially the expression language. This gives architects the possibility to decide on their own if they prefer either textual, graphical or mixed design. Since textual and graphical representations are fully compatible (round-trip engineering), the designer starts e.g. with textual modeling and can go over to graphical modeling and vice versa. This can facilitate the entry to this new kind of programming language and the component-based engineering. An example of the graphical design system can be seen in Sec. 11.5.

### 11.3.7 Summary

The previous sections have introduced the novel textual and graphical language contributions as provider and processor interfaces, components, ports, component-instances and also component- and component-instance-diagrams. It has been shown how components can be designed and how they can be connected using ports and interfaces.

Once the system architecture is specified in the class- and component-diagrams, component, port and connection instances have to be created in the novel component-instance-diagram. The separation of definition and instantiation in this case has the big advantage of giving two clear points of view onto the system implementation. Furthermore, the new component-instance-diagrams give the opportunity to uniquely name and initialize the component and port instances. By the usage of e.g. OCL validators, it is guaranteed that names and field-value assignments are unique and that instantiated component connections are applicable (provider to processor only).

All the previously introduced novel GU-DSL language extensions in combination with standard GU-DSL features allow for a complete architectural system specification and instantiation. One possible CBSE system implementation, which maps to GU-DSL and all its features, will be shown in Sec. 11.4. An example system can be found in Sec. 11.5.

## 11.4 A CBSE System as an Exemplary Implementation of GU-DSL

This section introduces a highly complex, but effective and simple to use, novel Component-Based Software Engineering system using C++ in combination with OpenGL and OpenCL to guarantee fast data and image processing. It is an exemplary implementation of GU-DSL and is fully compatible to all its features. All types and objects are mapped to the C++ framework using Xtend [EEK$^+$12] as code generation framework. Furthermore, it proposes several new concepts and ideas as will be shown in the next sections.

The basis of the system and all other CBSE classes is the *IObject*-interface shown in Fig. 11.2. It provides some basic functionality required for dynamic object creation, which is necessary for further usage as will be shown in Sec. 11.4.4. The main component classes (Class, Source, Filter, Sink and Port) introduced in Sec. 11.3 can also be found in this figure as part of the CBSE system. In the following sections, the main important and novel features will be introduced.

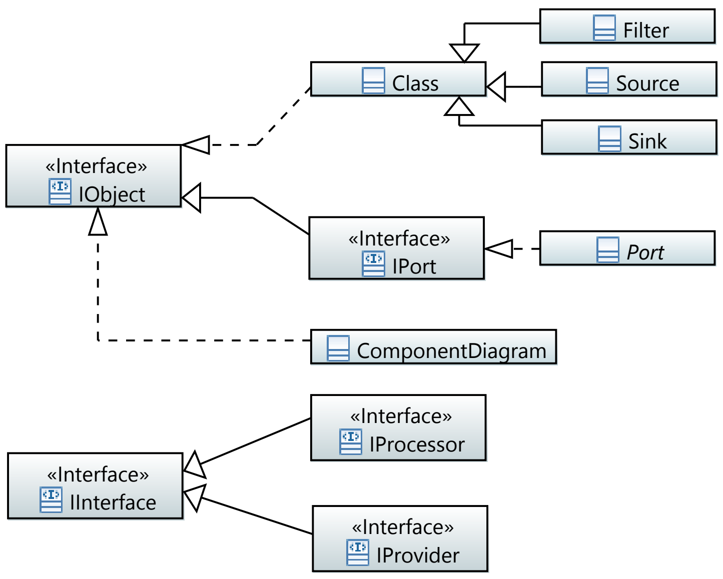


Figure 11.2: The CBSE schema. [HKK15]

### 11.4.1 Component Realization

Components are the basis of this CBSE framework. They are represented as C++ classes and provide predefined functionality. Fig. 11.3 shows the principle of component classes. Every component supports an *init()*, *process()* and *cleanup()*-method as mentioned in Sec. 11.3.5. While *init()* and *cleanup()* are mainly responsible for resource management (e.g. creating or releasing references to camera drivers, files or

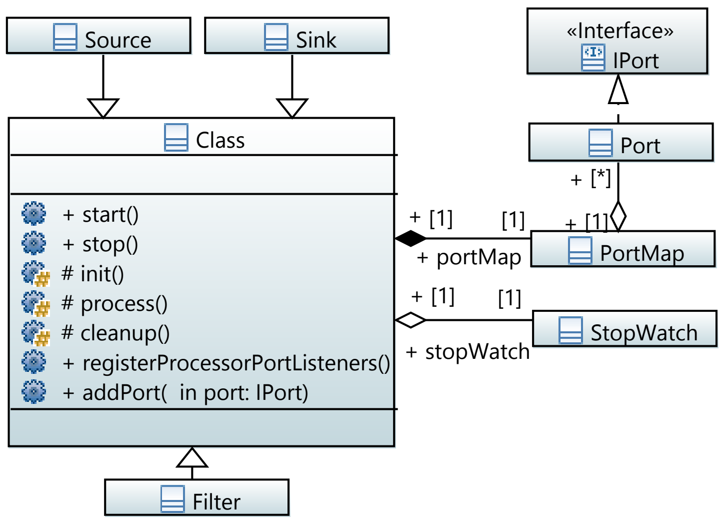


Figure 11.3: The basic component class. [HKK15]

databases), *process()* is the execution method of a component. In this CBSE system, all components are placed in a multithreaded environment, which means that the execution of every component is performed within its own thread. In general, this thread can sleep as long as there is no data processing required and it can wake up as soon new data arrives. This is called *data-driven mode*. The system realizes it for all kinds of components. A special *continuous-mode* is also provided. In this mode, the *process()*-method is continuously executed. This can be helpful e.g. for simulation purposes where internal data has periodically to be sampled or updated.

As proposed with GU-DSL, the full implementation of the *process()*-method can be realized using either activity-diagrams or sequential expressions and is in general also generated by the code-generator. But of course, it can also be coded manually, if the CBSE framework is to be used as standalone version without GU-DSL and its toolchain.

Furthermore, every component can contain an arbitrary number of named ports (*portMap*). Port naming allows the unambiguous identification of identical port interfaces. An example of using identical interfaces multiple times can be an image processing filter accepting two incoming images, one that has to be processed and another one that is used as mask image. The same interface types can be used for this purpose and the identification is then performed using the unique port names.

Besides the previously mentioned methods, two additional methods are added. The first method starts a component (*start()*), while the second method stops it (*stop()*). This is necessary and helpful for data acquisition or any kind of UI interaction allowing e.g. a manual start or stop of a data source, filter or sink. The *registerProcessorPortListeners()*-method functionality will be discussed in Sec. 11.4.3.

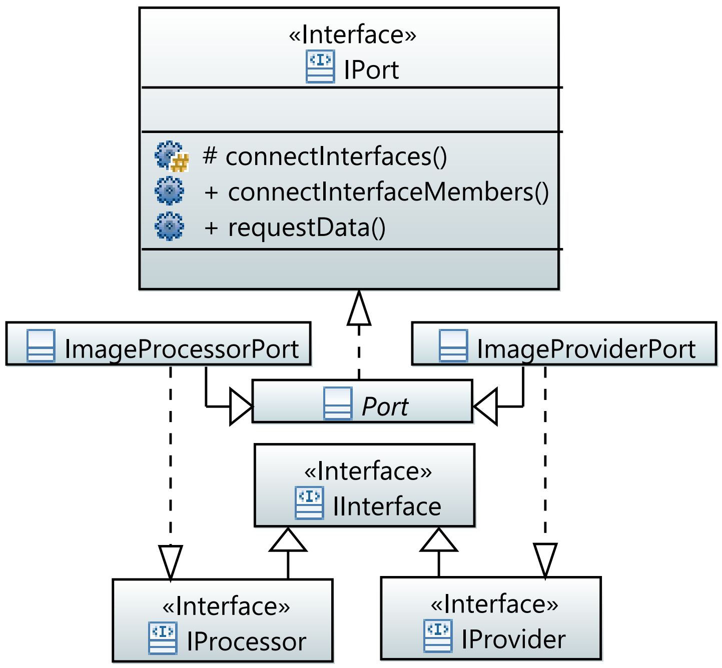


Figure 11.4: An exemplary port realization. [HKK15]

### 11.4.2 Port Realization

Ports are a substantive part of the newly proposed CBSE system and give components the possibility to connect their interfaces with the outer world. An exemplary realization of an image processor and provider port can be seen in Fig. 11.4.

Interface connections are established using the *connectInterfaces()*-method. It is called during the component instantiation whenever two interfaces have to be connected. By overwriting the *connectInterfaceMembers()*-method, every port can decide on its own which interface-methods of a provider and processor are connected. This allows ports to connect arbitrary method-signatures to act e.g. as converter functions. Using GU-DSL with its newly proposed component-based features, the connections are automatically generated using compatible method signatures (see also Sec. 11.3.1).

At first glance, a connection is just something like a method of how *interface A* calls the corresponding method of *interface B* (see also Fig. 11.5). However, this is not that easy if these calls are not supposed to be implemented manually in every port and something like reflection and type introspection (e.g. Java or C#) cannot be used. Since the framework is based on C++, reflection is not available in this form and would in most cases be slow anyway, which is why a signal based approach has been used. Thanks to modern libraries (in this case Boost.Signals [Boo16]), this is a simple and flexible, but also a fast method to connect all port interface members. Using macros, the whole signal and method declaration is hidden behind one single line of code (*MEMBER_SIGNAL_X(returntype, name, parameters ...)*). Type safety is achieved with the underlying framework's C++ template architecture. This makes the usage easy and minimizes sources of error. Another macro is provided to connect methods during the *connectInterfaceMembers()*-call (*MEMBER_SIGNAL_CONNECT_X(sourcemethodname, targetmethodname, parameters)*, with X as number of passed parameters for both macros). Using signals instead of hard coded connections slightly slows down the data transfer, but it is more flexible and fits better into the generic textual and graphical design concept of GU-DSL and the CBSE system.

Besides the generic concept of ports, another big advantage was introduced in Sec. 11.3.3 using the *async* and/or *buffered* keywords. Using ports, it is easy to add new kinds of functionality (e.g. data converters or buffering) even to existing components where only interfaces are available. Asynchronous and buffered connections are a big advantage for data processing purposes. The main goal of this new component-based data processing system is reducing design overhead as much as possible. Hence, to simplify the component development, data buffering is already added. Two kinds of buffering are possible:

1. A component is responsible for buffering, e.g. by using ring-buffers
2. A port is responsible for data buffering and asynchronous execution

Both methods have their advantages and are thus realized in this system, but it is the best choice to use the ring-buffers already embedded within the components. However, if realized components do not support buffering due to design issues, buffering ports are a good alternative. Therefore, a callback to all component-ports (*requestData()*) is performed after each successful execution of the component *process()*-method. This instructs a port to forward the next data-elements stored inside buffers to the defined component-interface-methods (see also Sec. 11.4.3). Besides passing the full data at once to a connected port, this also allows to provide data streaming solutions. The *requestData()*-method is then used to signal, that new streaming data is required. If the ports are generated by the code-generation framework of GU-DSL, the necessary buffers for all methods can automatically be created.

### 11.4.3 Component and Port Interaction

Once components and ports are designed and implemented or generated, it is important to know how the connection in-between can be realized. For that purpose, a signal based connection has been implemented.

The system initialization is a three-stage process which will be examined closer in Sec. 11.4.5. In the first stage, a component is created. A second stage creates all ports and adds them to the corresponding component using the *addPort()*-method of a component class. The third stage performs the interface connection using the *IPort* interface methods.

If a newly added port is a kind of processor, the component's *registerProcessorPortListeners()*-method is called to connect and register port interfaces (see Sec. 11.4.2).

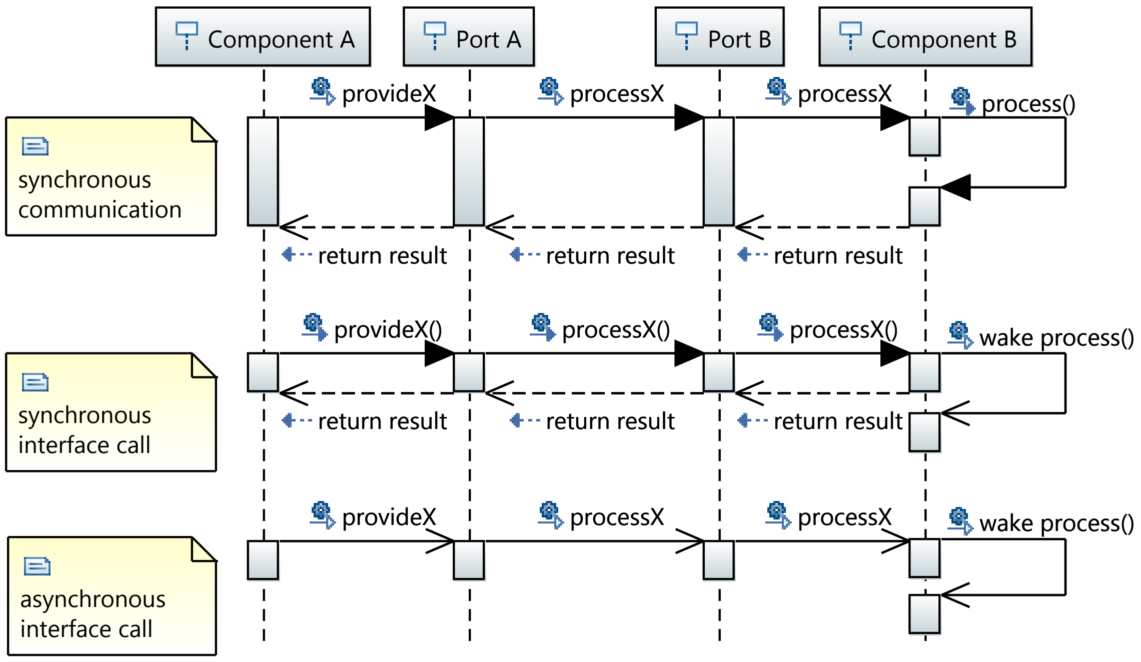


Figure 11.5: The three different kinds of communication. [HKK15]

As already mentioned, the connection from a port into a component is also realized using signals. This makes the usage simple, reduces errors and allows for buffered and/or asynchronous interface calls.

As can be seen in Fig. 11.5, three types of communication are provided:

1. Synchronous communication
2. Synchronous interface calls
3. Asynchronous interface calls

**Synchronous Communication** It is a completely synchronous, direct communication and also the fastest way for data processing in a system. However, in most cases it is not the best choice, because the execution of the calling component is blocked until the answer from the called interface-method arrives. This can take a long time if a large processing system is used. The big advantage is that it is easy to react to the return state of the interface call and the caller can be sure that the data is directly processed after the method call without doing some extra synchronization.

**Synchronous Interface Calls** This kind of communication is only partially synchronous. Interface-calls are executed synchronously up to the call/signal into the component. The receiving component thread is woken up and the data is stored in buffers until processing begins. The interface-call itself directly returns with a positive result back to the caller and the execution can immediately continue. It is guaranteed that all listening ports/components receive the new data. No additional info about a successful data processing is given back to the interface caller. The big advantage is that at least some state information can be received and that there is only a small time loss. This kind of communication is the best choice and is assumed to be the default.

**Asynchronous Interface Calls** It is a fully asynchronous kind of communication. Directly after the interface call from a component, the communication is delegated to the output ports performing a multithreaded, asynchronous method call. There is no guarantee on a non real-time system when the listening port will receive the new data. However, this has the advantage to immediately continue the component execution without losing nearly any time. This way of execution offers the highest performance.

Two types of component interface calls are proposed. The one and more sophisticated way is to manually find a corresponding port and call the assigned interface. The other and simpler way is using the provided macros *CALL_INTERFACE_X (interfaceType, parameters ...)* (all ports) and *CALL_INTERFACE_NAMED_X (interfaceType, portName, parameters ...)* (named port only).

Using signals has also another big advantage. In Sec. 11.3.2, nested components have been proposed. To hold on to the concept of components, generating the nested components as sequential code is not the recommended way in the novel CBSE system (but also a possible option). The presented implementation uses the same signaling concept as the previously introduced components. That means nested components are normally created and ports are added and finally connected. Until this state is reached, the nested components are hidden from the outer world. This is handled using ports, which are first added to the outer and the inner component and then connected using the proposed signaling concept.

### 11.4.4 Prototype-Factory Pattern - A Way for Dynamic Object Creation and Registration

The CBSE system is based on C++ and it is extendable using plugins (any kind of library, depending on the operating system, e.g. Dynamic Link Libraries on windows). Unfortunately, C++ neither supports reflection nor other types of dynamic object creation (e.g. by name). However, this drawback can be bypassed using special constructs and design patterns such as the newly proposed ***prototype-factory*** pattern. It is a combination of the well-known prototype-pattern and the factory-method-pattern [GHJV95]. The plugin idea of the CBSE framework is not that easy to realize in C++ if all components are supposed to interact, the types are not known during compile-time and the overhead should be kept minimal for developers. For dynamic registration, one instance of each object contained in the plugin is created during the initialization of the static/global library content (*the prototype*). Using anonymous structs in combination with an object creation and registration function (encapsulated in a C-macro *GENERIC_REGISTER_OBJECT*) reduces the overhead to a single line (see *REGISTER_OBJECT(PluginType, ObjectType)*, Listing 25).

```

#define CONSTRUCTOR_FUNCTION(FUNC)
namespace
{
  static const struct FUNC ## __ctor__
  {
      inline FUNC ## __ctor__() { FUNC(); }
  }
  FUNC ## _ctor_instance_;
}

#define GENERIC_REGISTER_OBJECT(PluginType, ObjectType, RegisterFunction)
namespace
{
  void RegisterFunction()
  {
    Core::Processing::Interfaces::ConstIObjectPtr sObject = ObjectType::createObject();
    PluginType::instance()->RegisterFunction(sObject);
  }
  CONSTRUCTOR_FUNCTION(RegisterFunction)
}

#define REGISTER_OBJECT(PluginType, ObjectType)
    GENERIC_REGISTER_OBJECT(PluginType, ObjectType, registerObject)

// Example usage
REGISTER_OBJECT(Processing::Plugin::ProcessingPlugin, ComponentClassDefinitions::Sources::Camera)
```

**Listing 25** Dynamic object creation and registration.

The *ObjectType* can be any kind of an *IObject* implementation proposed in Fig. 11.2 and its static method *createObject()* creates a new instance of this type (in this case the ***prototype*** *instance*). The fully qualified object name is automatically assigned during object creation using the stringification mechanism of the C++ preprocessor. This allows to register the object-prototype in a plugin specific, global name-object-map. Once the plugin is loaded, the global initialized map can be extracted and all the objects are added to the CBSE object factory.

Calling its *createObject()*-method and passing the unique object name, the factory can clone the prototype by calling the prototype's *IObject createObject(IObject instanceToClone)*-method. This newly created clone is then provided as a new instance to the CBSE system. The functionality is available for all objects inheriting from the abstract interface *IObject*. Using the new pattern in this way, it is quite simple to realize plugins also in C++. The necessity of this kind of object creation can be seen in Sec. 11.4.5.

### 11.4.5 Component Diagram Realization

In Sec. 11.3.4, the new concept of component-instance-diagrams has been introduced. The component-instance-diagram is used for the implementation of component instances and it is realized as a discrete object as shown in Fig. 11.2. Two different kinds of this diagram are proposed.

1. An XML-based version (see Sec. 11.4.5.1)

2. A fully generated, code-based version (see Sec. 11.4.5.2)

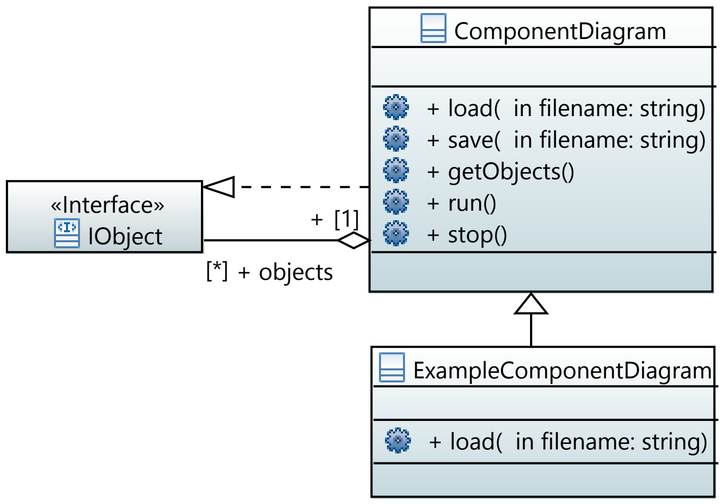


Figure 11.6: Component-instance-diagram realization. [HKK15]

Fig. 11.6 shows the realization of both diagram versions (*ComponentDiagram* and *ExampleComponentDiagram*). The basic *ComponentDiagram*-class is responsible for loading and saving the *XML-based* instance diagrams, while the *ExampleComponent-Diagram*-class is an exemplary, specialized implementation of a *code based* diagram. Both version use the prototype-factory to instantiate new components and ports.

### 11.4.5.1 XML based Component-Instance-Diagrams

The XML based version is the simplest but also the most flexible realization of the newly introduced component-instance-diagrams. It allows users to dynamically define full systems while only being restricted to the underlying architecture definition described in the previous sections. All objects, including components and ports, are stored with their initialization values (see also Sec. 11.3.4) and the fully qualified name as type specification. Additionally, the connections are stored as well. Using XML has several advantages. One big advantage is the readability. Small changes, such as modifications of property values, can simply be done using text editors, even if the design tools are not available. Furthermore, XML documents can be loaded (generic *load*-method), saved (generic *save*-method) or exchanged (*run()*- and *stop()*-methods) on the fly which is often required in productive environments. However, the biggest advantage is that the XML-based diagram version automatically supports all kinds of newly user defined components.

### 11.4.5.2 Code based Component-Instance-Diagrams

The code based version is represented by a sequential creation of all required components, ports and connections. For this purpose, a new class inherits from the *ComponentDiagram*-class and the *load()*-method is overloaded. The load mechanism is then replaced by a less generic, manual C++ implementation. In general, the initialization process is much faster compared to the XML version, because no XML-parsing is required. This can be helpful on small systems with slow hardware. The drawback of this, however, is the loss of flexibility of dynamic diagram reloading.

### 11.4.6 The Rich Client Platform

The proposed CBSE system is a powerful framework that can simplify development. To round off the CBSE implementation, it is embedded into a rich client platform (RCP), enabling programmers to build new and individual standalone applications. The implemented RCP supports the processing- and GUI-extension using the proposed CBSE system as basis. As is typical for RCPs (e.g. Eclipse [EC16]), the GUI is expandable using a plugin system in a similar way as proposed for the CBSE system (see Sec. 11.4.4). The current implementation supports OpenGL based image viewers, fast data processing using OpenCL, port implementations using network transfers and also the interaction between GUI and components using an event system. The RCP is just mentioned here for completeness and is not directly part of this thesis.

## 11.5 A Component-Based Modeling Example

In the next sections, a simple example of an image processing scenario extracted from Chapter 6 will be shown. A *Camera*-component (e.g. a color camera) acquires an image, which is filtered in the *MeanFilter*-component. Subsequently, the image is shown in the *Viewer*-component. It starts with the definition of the necessary classes and interfaces in Listing 26.

Until now, the three main component classes and the corresponding image processor and image provider interfaces are defined. The classes implement their required interfaces as shown in Sec. 11.3.2. Furthermore, they implement the required component *process*-method by calling the responsible activity diagrams (not part of this example; have a look at Chapter 10). Listing 27 shows the usage of the classes as basis for components. The components themselves define all valid ports (*CameraImageProviderPort*, *MeanImageProcessorPort*, *MeanImageProviderPort*, *ViewerImageProcessorPort*) and connections (*CameraToFilter*, *FilterToViewer*) describing the final software architecture. As stated in Sec. 11.3, this means that it is defined which components are connectable. In this example, the connection between a *Camera* and a processing *MeanFilter* and also the connection between a *MeanFilter* and a *Viewer* is valid.

```
ClassDiagram ComponentClassDefinitions
{
  public interface provider IImageProvider
  { // see also the corresponding processImage signature
    public void provideImage(ref Image image);
  }

  public interface processor IImageProcessor
  { // see also the corresponding provideImage signature
    public void processImage(ref Image image);
  }

  public source Camera implements IImageProvider
  {
    public int width;
    public int height;

    public bool process(ConstIObjectParametersPtr parameters)
    { // Call the activity-diagram responsible for
      // image acquisition
      call behavior AdAcquireImage;
      return true;
    }
  }

  public filter MeanFilter implements IImageProvider, IImageProcessor
  {
    public bool process(ConstIObjectParametersPtr parameters)
    { // Call the mean filter activity-diagram
      call behavior AdFilterMean;
      return true;
    }
  }

  public filter Viewer implements IImageProcessor
  {
    public bool process(ConstIObjectParametersPtr parameters)
    { // Call the activity-diagram to show a new image
      call behavior AdShowImage;
      return true;
    }
  }
}
```

**Listing 26** Component class example diagram. [HKK16]

```
ComponentDiagram ComponentExampleDiagram
{
  source ComponentClassDefinitions.Camera
  { // Define the output port
    port CameraImageProviderPort ( provider  CameraImageProviderPort :
              ComponentClassDefinitions.IImageProvider );
    // Define the port
    [CameraToFilter source CameraImageProviderPort.CameraImageProviderPort ==>
           target MeanImageProcessorPort.FilterImageProcessorPort ]
  }

  filter ComponentClassDefinitions.MeanFilter
  { // Define the input and output ports
    port MeanImageProcessorPort ( processor FilterImageProcessorPort :
              ComponentClassDefinitions.IImageProcessor );
    port MeanImageProviderPort ( provider FilterImageProviderPort :
              ComponentClassDefinitions.IImageProvider );

    // Define the port
    [FilterToViewer source MeanImageProviderPort.FilterImageProviderPort ==>
           target ViewerImageProcessorPort.ViewerImageProcessorPort ]
  }

  sink ComponentClassDefinitions.Viewer
  { // Define the input port
    port ViewerImageProcessorPort ( processor  ViewerImageProcessorPort :
                ComponentClassDefinitions.IImageProcessor );
  }
}
```

**Listing 27** Component example diagram. [HKK16]

As can be seen, the basic architecture definition is quite simple. While Listing 27 shows the required component definitions and thus the developed architectural definition, Listing 28 instantiates the final system using a component-instance-diagram (*ComponentInstanceExampleDiagram*). It creates two Camera instances (*Camera1*, *Camera2*), a MeanFilter (*MeanFilter1*) and also a Viewer (*Viewer1*). Property values are also assigned during creation. Furthermore, all required port instances (*Camera1ProviderPort*, *Camera2ProviderPort*, *MeanProcessorPort*, *MeanProviderPort*, *ViewerProcessorPort*) are created. Finally, the ports are connected defining the connection between a *source* and *target* port. As already mentioned, the instantiation of ports and also the connections are restricted to the fixed definitions in Listing 27.

```

ComponentInstanceDiagram ComponentInstanceExampleDiagram
{
  // Instantiate new component of type ComponentClassDefinitions.Camera
  Camera1 instantiates ComponentClassDefinitions.Camera
      // Assign property values
      (width = 640, height = 480)
  {
    // Create a port instance
    Camera1ProviderPort : CameraImageProviderPort (
      provider CameraImageProviderPort = IImageProvider() );

    // Connect the ports
    [ source Camera1ProviderPort.CameraImageProviderPort ==> target
      MeanFilter1.MeanProcessorPort.MeanImageProcessorPort ]
  }

  // Instantiate new component of type ComponentClassDefinitions.Camera
  Camera2 instantiates ComponentClassDefinitions.Camera ( )
  {
    // Create a port instance
    Camera2ProviderPort : CameraImageProviderPort (
      provider CameraImageProviderPort = IImageProvider() );

    // Connect the ports
    [ source Camera2ProviderPort.CameraImageProviderPort ==> target
      MeanFilter1.MeanProcessorPort.MeanImageProcessorPort ]
  }

  // Instantiate new component of type ComponentClassDefinitions.MeanFilter
  MeanFilter1 instantiates ComponentClassDefinitions.MeanFilter ( )
  {
    // Create the port instances
    MeanProcessorPort : MeanImageProcessorPort (
      provider MeanImageProcessorPort = IImageProcessor() );

    MeanProviderPort : MeanImageProviderPort (
      provider MeanImageProviderPort = IImageProcessor() );

    // Connect the ports
    [ source MeanProviderPort.MeanImageProviderPort ==> target
      Viewer1.ViewerProcessorPort.ViewerImageProcessorPort ]
  }

  // Instantiate new component of type ComponentClassDefinitions.Viewer
  Viewer1 instantiates ComponentClassDefinitions.Viewer ( )
  {
    ViewerProcessorPort : ViewerImageProcessorPort (
      provider ViewerImageProcessorPort = IImageProvider() );
  }
}
```

**Listing 28** Component instance example diagram. [HKK16]

As stated in Sec. 11.3.6, besides the textual DSL, also graphical editors are proposed with this approach. It rounds off the new CBSE system approach as can be seen in Fig. 11.7. The textual class-, component- and component-instance-diagrams are shown in their graphical version. While the textual form grows in size relatively fast, the graphical representations are more compressed and much simpler to understand, which can simplify design.

The listings and figures in this section show a small, representative example of the proposed GU-DSL CBSE features. Starting with the class- and interface-definitions (Listing 26), continuing with the component design (Listing 27) and finishing with the component instance definitions (Listing 28), the example creates a small image processing pipeline (two cameras -> mean-filter -> viewer) in a textual and graphical form. Using the proposed CBSE C++ infrastructure, the example can be compiled and run in the underlying RCP.

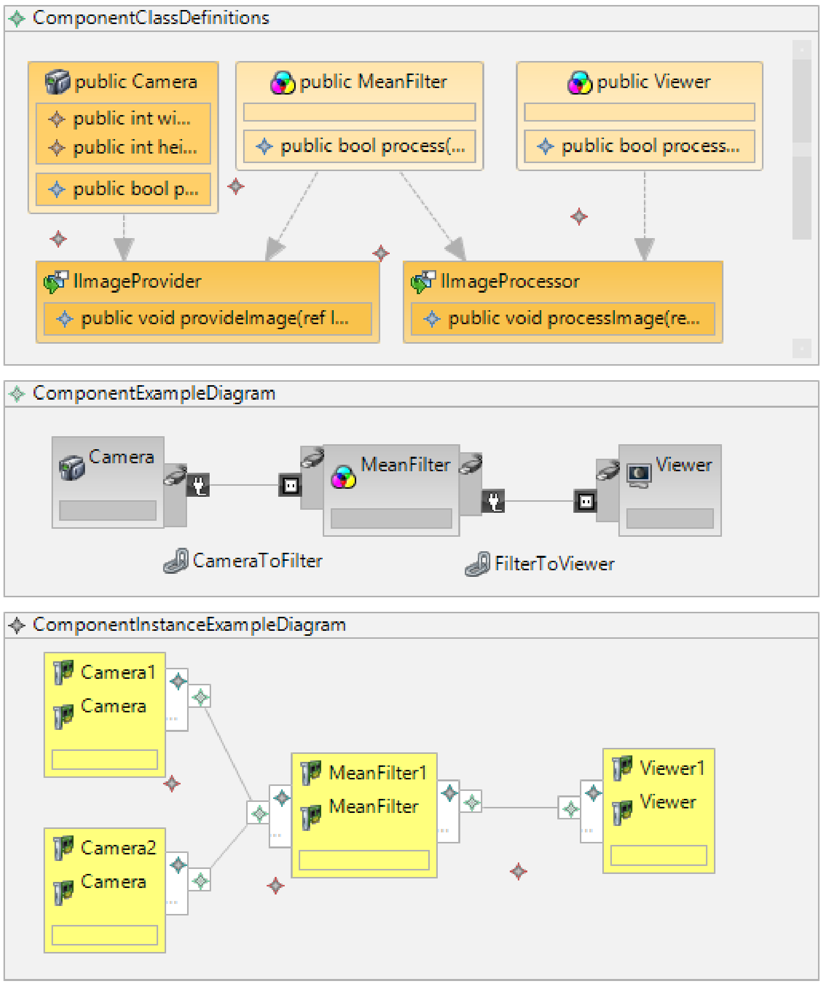


Figure 11.7: The graphical CBSE example. Top: the class diagram. Middle: the component-diagram. Bottom: the component-instance-diagram. [HKK16]

## 11.6 Summary

In this chapter the component-based extension of GU-DSL was presented. The novel features fully integrate into GU-DSL's design principles as classes and interfaces, but in general, the component-based extensions can also be used as a standalone language. All the novel features and concepts (provider and processor interfaces, components, ports, component-instances and also component- and component-instance-diagrams), that allow architecture definition and instantiation, have been discussed. Furthermore, their usage has been demonstrated in an image processing example. Besides the novel language features, a novel, exemplary design and implementation of a CBSE system architecture has been introduced, embedded into a rich client platform that can simply be extended using plugins.

A big advantage of the proposed work, compared to other concepts (such as UML), is the combination between graphical and textual modeling and the supported round-trip engineering. This means that both kinds of modeling are always synchronized in both directions. Starting with textual modeling does not prevent developers from switching over to graphical modeling and also back. So it is up to the modeler's and programmer's preferences which kind of technique is used. Experienced programmers can decide for textual programming, enjoying the full feature set necessary for component-based engineering with the advantage of much better guidance and support than generic programming languages provide.

The proposed language and CBSE framework is already used in several industrial projects. The first professional feedback has already been integrated into the language and infrastructure design. Even though the language and the editors are already suitable for productive development, especially the graphical editors have to be improved and simplified.

# 12 Model Driven Engineering Summary

The previous chapters of this part have shown how data can be processed using a more abstract way of algorithm development than shown in the first part. Introducing and using the novel domain specific language GU-DSL developed in this thesis, two improvements in model driven engineering have been presented. This chapter summarizes these results and the contributions of this thesis.

As a first contribution, the novel GU-DSL GPGPU language extension has been introduced in combination with graphical editors (see Chapter 10). It allows developers and modelers to design fully runnable GPGPU applications in a model driven way. The presented modeling toolchain is based on Eclipse-xText to design the novel DSL and its text editors. Eclipse-GMF is used to develop the corresponding graphical designers. In combination with the code generator and the high-level abstraction Heterogeneous Computing OpenCL framework, it is possible to easily develop fast image processing filters, e.g. a Bilateral Filter or even more complex processing methods (e.g. the *findIndex*-reduction). An import result is that too fine- or coarse-granular modeling of algorithms minimizes the advantages (e.g. clarity and maintainability) of using the proposed toolchain. A good mixture between expression grouping in a node and nodes that represent single operations has to be found. If a developer keeps this in mind, then this toolchain can significantly improve the development of complex image and data processing GPGPU systems. It structures the code that has to be written and can assist the developer in avoiding common mistakes.

The second main contribution in this part is a component-based extension of GU-DSL (see Chapter 11). The novel features fully integrate into GU-DSL's design principles such as classes and interfaces, but in general, the component-based extensions can also be used as a standalone language. All the novel features and concepts (provider and processor interfaces, components, ports, component-instances and also component- and component-instance-diagrams) have been discussed allowing the architecture definition and instantiation. Furthermore, their usage has been demonstrated in an image processing example. Besides the novel language features, a novel, exemplary design and implementation of a CBSE system architecture has been introduced, embedded into a rich client platform that can simply be extended using plugins. The big advantage of the proposed work, compared to other concepts (such as UML), is the combination between graphical and textual modeling and the supported round-trip engineering. This means that both kinds of modeling are always synchronized in both directions. Starting with textual modeling doesn't prevent developers from switching over to graphical modeling and also back. So it is up to the modeler's and programmer's preferences which kind of technique is used. Experienced programmers can decide for textual programming, enjoying the full feature set necessary for component-based engineering with the advantage of much better guidance and support than generic programming languages provide.

Taken together, the presented approaches and concepts provide a solid basis for future model driven development, also in the domain of data and image processing. It has been shown that it is also possible to use modern software concepts like Domain Specific Languages in a domain where real-time data processing is desired or even indispensable. Furthermore it is shown that using this kind of development opens a new wide area of opportunities (e.g. automated algorithm testing or documentation, etc.) and scientific challenges.

# 13 Summary And Conclusion

The growing demand towards industrial automation and autonomous systems requires more flexible technologies in different but interdependent domains of engineering. This thesis introduces and discusses two important areas: ToF camera data improvement (algorithm related) and related model driven engineering techniques (software related). The next paragraphs summarize the main constituents.

Range sensing techniques have been improved by the introduction of ToF range cameras like the PMD camera during the last years. Using this kind of sensor, 2.5D cameras like the PMD CamCube could be realized, allowing to acquire depth distance data for a whole scene in real-time. However, artifacts arise during range capturing. Depth data denoising has always been an important discipline in image processing. Methods for outlier removal and outlier correction have been developed to improve acquisition quality of such noisy data. But still many challenges remain. Compared to other industrial suited range sensing systems like laser-scanners, ToF cameras provide fully lateral 3D information at high frame rates, additional grayscale/intensity information and full eye safety. Furthermore, due to underlying Complementary Metal-Oxide-Semiconductor (CMOS) chip design, cheap industrial cameras, that can handle difficult environment conditions, can be developed.

Part I discusses the special challenges of data quality improvement on a very deep layer of ToF cameras. It deals with different challenges related to the working principle of this kind of sensor. A new method for a fast motion artifact compensation for ToF cameras has been presented. It is shown that the algorithm gives good results for simulated data (linear and non linear motion) as well as real data while providing real-time performance. Beside the motion artifact compensation, the second proposed algorithm deals with the automatic integration time estimation of ToF cameras. An online integration time adaption algorithm that works on a per-pixel basis and uses knowledge gained from an extensive analysis of the underlying inherent sensor behavior has been introduced and evaluated. Finally, an industrial real-time 3D car reconstruction example has been presented. It shows how the data of three PMD cameras has to be preprocessed, using an extensive depth data processing and filtering pipeline, to be able to combine this data.

Beside the range sensing techniques, model driven development techniques have passed great improvements in the past years. Several techniques have been established in software engineering but have been rarely used for image processing. The two most important approaches are graphical modeling and DSLs. Graphical modeling gives modelers the possibility to create simple, high-level, iterative graphical architecture designs, where all relevant parties (programmers and non-programmers as e.g. project leaders) can have an abstract graphical view on the system during development. DSLs are the second important approach. They have become more and more important in the past years for improving software development due to tools simplifying the language development.

Part II addresses these challenges of data related model driven software engineering. It introduces and contributes the new domain specific language GU-DSL and two data and image processing related extensions: GPGPU-programming and Component-Based Software Engineering principles in this domain. The presented GU-DSL GPGPU extension contributes a convenient combination and mixture of textual and graphical model- and dataflow-driven design. It proposes special new structures as conditional-, loop- and calculation-nodes, in combination with an expression language in activity-diagrams. This is necessary to fulfill the requirements for GPGPU-programming. Using a code generator, the GU-DSL code can be transformed into C++, compiled and be executed. All the GPU related features are encapsulated into a C++ Heterogeneous Computing framework (currently OpenCL support only). This simplifies code generation and allows to be independent either from OpenCL or CUDA in the future. The performance, the lines of code of GU-DSL and the generated code are compared against manually written OpenCL and C++ code. As a result, GU-DSL and the underlying code generator can compete against the manually written code. Additionally the more abstract GPU code can help to reduce common mistakes. The GU-DSL CBSE system introduces a concept for component based software engineering in the domain of data and image-processing. It proposes several new concepts for component- and component-instance-diagrams in combination with class- and activity-diagrams. Using a newly developed Rich Client Platform supporting a plugin based extension system, it shows how the GU-DSL CBSE concept can be realized and used in practice using C++. Exemplary a simple processing pipeline using cameras, filters and sinks has been implemented using class-, component- and component-instance-diagrams to demonstrate the new concepts.

Summarizing the main aspects of this thesis, the presented topics show the domain of depth-, image- and data-processing from two points of view. The first one is really low-level and deals with data improvement while the second one deals with high-level abstraction of algorithm and system development. It has been shown in the previous chapters of this thesis that both points of view are really important for current and future development. Furthermore, both sides are strongly linked and have to coexist. Especially the system and framework design should not be neglected by algorithm developers, because they profit from standardized and tested default frameworks. On the other side, framework and language designers have to closely work together with algorithm developers. They have to adopt the algorithms' special requirements such as low-level programming and performance.

Finally, the presented methods suggest new algorithms using prior ToF camera knowledge to achieve better and less noisy results. Furthermore this thesis shows how generic algorithm and system development using a Domain Specific Language can help to improve the design of data- and image-processing systems in the future. The results of both topics can help to advance the domain of data and image processing.

# Bibliography

[AG97] Robert Allen and David Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology, TOSEM*, 6(3):213–249, July 1997.

[AG16] Otto Christ AG. Roll-over wash units. http://www.christ-ag.com, October 2016.

[AMD16] AMD. OpenCL development. http://developer.amd.com, October 2016.

[Bes88] Paul J. Besl. Geometric modeling and computer vision. *Proceedings of the IEEE*, 76(8):936–958, October 1988.

[BETG08] Herbert Bay, Andreas Ess, Tinne Tuytelaars, and Luc Van Gool. Speeded-up robust features (surf). *Computer Vision and Image Understanding*, 110(3):346–359, June 2008.

[BM92] Paul J. Besl and Neil D. McKay. A method for registration of 3-d shapes. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 14(2):239–256, February 1992.

[Boo16] Boost. Boost. http://www.boost.org/, October 2016.

[Bro92] Lisa Gottesfeld Brown. A survey of image registration techniques. *ACM Computing Surveys, CSUR*, 24(4):325–376, December 1992.

[BSGL96] Robert Bergevin, Marc Soucy, Hervé Gagnon, and Denis Laurendeau. Towards a general multi-view registration technique. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 18(5):540–547, May 1996.

[CK05] Sen-Ching S. Cheung and Chandrika Kamath. Robust background subtraction with foreground validation for urban traffic video. *Journal on Applied Signal Processing, EURASIP*, 2005(1):2330–2340, January 2005.

[CL96] Brian Curless and Marc Levoy. A volumetric method for building complex models from range images. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 303–312, August 1996.

[CLSF10] Michael Calonder, Vincent Lepetit, Christoph Strecha, and Pascal Fua. Brief: binary robust independent elementary features. In *Proceedings of the 11th European conference on Computer vision: Part IV, ECCV*, pages 778–792. Springer-Verlag, September 2010.

[CM92] Yang Chen and Gérard Medioni. Object modelling by registration of multiple range images. *Image and Vision Computing*, 10(3):145–155, April 1992.

[CSC$^+$10] Yan Cui, Sebastian Schuon, Derek Chan, Sebastian Thrun, and Christian Theobalt. 3d shape scanning with a time-of-flight camera. In *Proceedings of IEEE Computer Society Conference on Computer Vision and Pattern Recognition, CVPR*, pages 1173–1180, June 2010.

[EC16] Eclipse-Community. Eclipse. http://www.eclipse.org, October 2016.

[EEK$^+$12] Sven Efftinge, Moritz Eysholdt, Jan Köhnlein, Sebastian Zarnekow, Robert von Massow, Wilhelm Hasselbring, and Michael Hanus. Xbase: Implementing domain-specific languages for java. In *Proceedings of the 11th International Conference on Generative Programming and Component Engineering, GPCE*, pages 112–121, March 2012.

[EEK$^+$16] Sven Efftinge, Moritz Eysholdt, Jan Köhnlein, Sebastian Zarnekow, and Contributors. Xtext 2.5 documentation. http://www.eclipse.org/Xtext/documentation/2.5.0/XtextOctober 2016.

[EHK14] Torsten Edeler, Stephan Hussmann, and Florian Knoll. Uncertainty analysis for optical time-of-flight sensors based on four-phase-shift range calculation. In *Proceedings of IEEE Sensors Applications Symposium, SAS*, pages 382–387, February 2014.

[EOHM10] Torsten Edeler, Kevin Ohliger, Stephan Hussmann, and Alfred Mertins. Time-of-flight depth image denoising using prior noise information. In *Proceedings of IEEE 10th International Conference on Signal Processing, ICSP*, pages 119–122, October 2010.

[EvdB10] Luc Engelen and Mark van den Brand. Integrating textual and graphical modelling languages. *Electronic Notes in Theoretical Computer Science, ENTCS*, 253(7):105–120, September 2010.

[FAT11] Sergi Foix, Guillem Alenya, and Carme Torras. Lock-in time-of-flight (tof) cameras: A survey. *Sensors Journal, IEEE*, 11(9):1917–1926, October 2011.

[FB07] Dragos Falie and Vasile Buzuloiu. Noise characteristics of 3d time-of-flight cameras. In *Proceedings of the International Symposium on Signals, Circuits and Systems, ISSCS*, volume 1, pages 1–4, July 2007.

[FLVC05] Peter H. Feiler, Bruce Lewis, Steve Vestal, and Ed Colbert. *An Overview of the SAE Architecture Analysis & Design Language (AADL) Standard: A Basis for Model-Based Architecture-Driven Embedded Systems Engineering*, pages 3–15. Springer US, 2005.

[Fow10] Martin Fowler. *Domain Specific Languages*. Addison-Wesley Professional, 1st edition, 2010.

[Fra16] Messe Frankfurt. Automechanika. http://www.automechanika.messe frankfurt.com, October 2016.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., 1995.

[GPT10] Pablo Gil, Jorge Pomares, and Fernando Torres. Analysis and adaptation of integration time in pmd camera for visual servoing. In *Proceedings of the 20th International Conference on Pattern Recognition, ICPR*, pages 311–315, October 2010.

[HA11] Tianyi David Han and Tarek S. Abdelrahman. hicuda: High-level gpgpu programming. *IEEE Transactions on Parallel and Distributed Systems*, 22(1):78–90, October 2011.

[HBK15] Thomas Hoegg, Christian Baiz, and Andreas Kolb. Online improvement of time-of-flight camera accuracy by automatic integration time adaption. In *Proceedings of IEEE International Symposium on Signal Processing and Information Technology, ISSPIT*, pages 613–618, December 2015.

[HFKK15] Thomas Hoegg, Guenther Fiedler, Christian Koehler, and Andreas Kolb. Gu-dsl – a generic domain-specific language for data- and image processing. Technical Report 1, University of Siegen, 2015. "".

[HFKK16] Thomas Hoegg, Guenther Fiedler, Christian Koehler, and Andreas Kolb. Flow driven gpgpu programming combining textual and graphical programming. In *Proceedings of the 7th International Workshop on Programming Models and Applications for Multicores and Manycores, PMAM*, pages 88–97. ACM, March 2016.

[HHE11] Stephan Hussmann, Alexander Hermanski, and Torsten Edeler. Real-time motion artifact suppression in tof camera systems. *IEEE Transactions on Instrumentation and Measurement*, 60(5):1682–1690, October 2011.

[HKK15] Thomas Hoegg, Christian Koehler, and Andreas Kolb. Component based data and image processing systems: A conceptual and practical approach. In *Proceedings of the 6th IEEE International Conference on Software Engineering and Service Science, ICSESS*, pages 66–69, September 2015.

[HKK16] Thomas Hoegg, Christian Koehler, and Andreas Kolb. Abstracting data and image processing systems using a component-based domain specific language. In *Proceedings of the 4th International Conference on Model-Driven Engineering and Software Development, MODELSWARD*, pages 292–300, February 2016.

[HLK13a] Thomas Hoegg, Damien Lefloch, and Andreas Kolb. *Real-Time Motion Artifact Compensation for PMD-ToF Images*, pages 273–288. Springer Berlin Heidelberg, 2013.

[HLK13b] Thomas Hoegg, Damien Lefloch, and Andreas Kolb. Time-of-flight camera based 3d point cloud reconstruction of a car. *Computers in Industry (3D Imaging)*, 64(9):1099–1144, December 2013.

[HS81] Berthold K.P. Horn and Brian G. Schunck. Determining optical flow. *Artificial Intelligence*, 17(1):185–203, August 1981.

[HS88] Chris Harris and Mike Stephens. A combined corner and edge detector. In *Proceedings of the 4th Alvey Vision Conference*, pages 147–151, August 1988.

[IFM16] IFM. O3d201 - ifm electronic gmbh. http://www.ifm.com, October 2016.

[IKH$^+$11] Shahram Izadi, David Kim, Otmar Hilliges, David Molyneaux, Richard Newcombe, Pushmeet Kohli, Jamie Shotton, Steve Hodges, Dustin Freeman, Andrew Davison, and Andrew Fitzgibbon. 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*, pages 559–568. ACM, October 2011.

[Ins16] National Instruments. Ni labview. http://www.ni.com/labview/, October 2016.

[JPP07] Ljubomir Jovanov, Aleksandra Pizurica, and Wilfried Philips. Wavelet based joint denoising of depth and luminance images. In *Proceedings of the 3DTV Conference*, pages 1–5, May 2007.

[JSHWY14] Kim Joong-Sik, Jo Hoon, and Kim Whoi-Yul. Adaptive noise reduction method using noise modeling for tof sensor. In *Proceedings of the 4th IEEE International Conference on Network Infrastructure and Digital Content, IC-NIDC*, pages 99–102, September 2014.

[KBKR12] Timo Kehrer, Stefan Berlik, Udo Kelter, and Michael Ritter. Modellbasierte entwicklung gpu-unterstützter applikationen. In *Proceedings of Modellierung*, volume 201, pages 139–154, March 2012.

[Khr16] Khronos. The khronos group - connecting software to silicon. https://www.khronos.org/, October 2016.

[KK09] Maik Keller and Andreas Kolb. Real-time simulation of time-of-flight sensors. *Simulation Modelling Practice and Theory*, 17(5):967–978, May 2009.

[LF96] Quan-Tuan Luong and Olivier D. Faugeras. The fundamental matrix: Theory, algorithms, and stability analysis. *International Journal of Computer Vision, Volume 17, Issue 1*, 171(1):43–75, January 1996.

[LHK13] Damien Lefloch, Thomas Hoegg, and Andreas Kolb. Real-time motion artifacts compensation of tof sensors data on gpu. In *Proceedings of SPIE - Three-Dimensional Imaging, Visualization, and Display,*, pages 87380U–87380U–7. Proc. SPIE 8738, May 2013.

[LHL12] Benjamin Langmann, Klaus Hartmann, and Otmar Loffeld. Depth camera technology comparison and performance evaluation. In *Proceedings of the 1st International Conference on Pattern Recognition Applications and Methods, ICPRAM*, pages 438–444, April 2012.

[LK81] Bruce D. Lucas and Takeo Kanade. An iterative image registration technique with an application to stereo vision. In *Proceedings of the 7th international joint conference on Artificial intelligence, IJCAI*, pages 674–679, August 1981.

[LK09] Marvin Lindner and Andreas Kolb. *Compensation of Motion Artifacts for Time-of-Flight Cameras*, pages 16–27. Springer Berlin Heidelberg, 2009.

[LKKK12] Seungkyu Lee, Byongmin Kang, James D.K. Kim, and Chang Yeong Kim. Motion blur-free time-of-flight range sensor. In *Proceedings of SPIE - The International Society for Optical Engineering*, volume 8298, pages 82980U–82980U–6, February 2012.

[LKS+13] Frank Lenzen, Kwang In Kim, Henrik Schäfer, Rahul Nair, Stephan Meister, Florian Becker, Christoph S. Garbe, and Christian Theobalt. Denoising strategies for time-of-flight data. In *Proceedings of Time-of-Flight Imaging: Algorithms, Sensors and Applications*, volume 8200, pages 24–45. Springer, November 2013. 1.

[LLK08] Marvin Lindner, Martin Lambers, and Andreas Kolb. Sub-pixel data fusion and edge-enhanced distance refinement for 2d/3d images. *International Journal of Intelligent Systems Technologies and Applications*, 5(3/4):344–354, November 2008.

[LNL$^+$13] Damien Lefloch, Rahul Nair, Frank Lenzen, Henrik Schäfer, Lee Streeter, Michael J. Cree, Reinhard Koch, and Andreas Kolb. *Time-of-Flight and Depth Imaging. Sensors, Algorithms, and Applications*, volume 8200, chapter Technical Foundation and Calibration Methods for Time-of-Flight Cameras, pages 3–24. Springer Berlin Heidelberg, October 2013.

[Low04a] Kok-Lim Low. Linear least-squares optimization for point-to-plane icp surface registration, technical report, tr04-004. Technical Report 1, Department of Computer Science, University of North Carolina at Chapel Hill., 2004. "".

[Low04b] David G. Lowe. Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, 60(2):91–110, November 2004.

[LSKK10] Marvin Lindner, Ingo Schiller, Andreas Kolb, and Reinhard Koch. Time-of-flight sensor calibration for accurate range sensing. *Computer Vision and Image Understanding*, 114(12):1318–1328, December 2010.

[McI00] Alan M. McIvor. Background subtraction techniques. In *Proceedings of the Image and Vision Computing Conference*, pages 1–6, November 2000.

[MDK92] Jeff Magee, Naranker Dulay, and Jeff Kramer. Structuring parallel and distributed programs. In *Proceedings of the International Workshop on Configurable Distributed Systems*, pages 102–117, May 1992.

[Mic16a] Microsoft. C++ amp : Language and programming model. http://www.microsoft.com, October 2016.

[Mic16b] Microsoft. Component object model (com). https://msdn.microsoft.com, October 2016.

[Mic16c] Microsoft. Kinect camera. www.microsoft.com, 2016.

[MIPG16] University of Kiel Multimedia Information Processing Group. Mip multi-camera calibration. http://www.mip.informatik.uni-kiel.de/tiki-index.php?page=Calibration, October 2016.

[MN16] Andreas Mülder and Alexander Nyßen. Tmf meets gmf, 2016.

[MRH$^+$16] Richard Membarth, Oliver Reichle, Frank Hannig, Jürgen Teich, Mario Korner, and Wieland Eckert. Hipacc: A domain-specific language and compiler for image processing. *IEEE Transactions on Parallel and Distributed Systems*, 27(991):210–224, January 2016.

[MWSP06] Stefan May, Björn Werner, Hartmut Surmann, and Kai Pervölz. 3d time-of-flight cameras for mobile robotics. In *Proceedings of the International Conference on Intelligent Robots and Systems*, pages 790–795, October 2006.

[NIH$^+$11] Richard Newcombe, Shahram Izadi, Otmar Hilliges, David Molyneaux, David Kim, Andrew J. Davison, Pushmeet Kohli, Jamie Shotton, Steve Hodges, and Andrew Fitzgibbon. Kinectfusion: Real-time dense surface mapping and tracking. In *Proceedings of the 10th IEEE International Symposium on Mixed and Augmented Reality, ISMAR*, pages 127–136, October 2011.

[NVI16] NVIDIA. www.nvidia.com. www.nvidia.com, October 2016.

[OMG16a] OMG. Corba. http://www.corba.org/, October 2016.

[OMG16b] OMG. Corba component model. http://www.omg.org/spec/CCM, October 2016.

[OMG16c] OMG. Metaobject facility (mof). http://www.omg.org/mof/, October 2016.

[OMG16d] OMG. Ocl. http://www.omg.org/spec/OCL/, October 2016.

[OMG16e] OMG. Uml. http://www.uml.org/, October 2016.

[Ope16] OpenCV. Opencv. http://opencv.org/, October 2016.

[PCL16] PCL. Pcl - point cloud library. http://pointclouds.org/http://opencv.org/, October 2016.

[PM90] Pietro Perona and Jitendra Malik. Scale-space and edge detection using anisotropic diffusion. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 12(7):629–639, July 1990.

[PMD16] PMD. Pmdtechnologies ag. www.pmdtec.com, October 2016.

[QT16] QT. Qt project. http://qt-project.org, October 2016.

[Rap07] Holger Rapp. Experimental and theoretical investigation of correlating tof-camera systems. Master's thesis, University of Heidelberg, Germany, 2007.

[RHG15] Mahesh Ravishankar, Justin Holewinski, and Vinod Grover. Forma: A dsl for image processing applications to target gpus and multi-core cpus. In *Proceedings of the 8th Workshop on General Purpose Processing Using GPUs*, pages 109–120, February 2015.

[RHHL02] Szymon Rusinkiewicz, Olaf Hall-Holt, and Marc Levoy. Real-time 3d model acquisition. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 438–446, July 2002.

[RKN00] Rolf-Jürgen Recknagel, Richard Kowarschik, and Gunther Notni. High-resolution defect detection and noise reduction using wavelet methods for surface measurement. *Journal of Optics A: Pure and Applied Optics*, 2(6):538–545, November 2000.

[RL01] Szymon Rusinkiewicz and Marc Levoy. Efficient variants of the icp algorithm. In *Proceedings of the Third International Conference on 3-D Digital Imaging and Modeling*, pages 145–152. IEEE, May 2001.

[SBK08] Ingo Schiller, Christian Beder, and Reinhard Koch. Calibration of a pmd-camera using a planar calibration pattern together with a multi-camera setup. In *Proceedings of the XXI International Society for Photogrammetry and Remote Sensing Congress, ISPRS*, pages 297–302, March 2008.

[Sch08] Markus Scheidgen. *Textual Modelling Embedded into Graphical Modelling*, pages 153–168. Springer Berlin Heidelberg, 2008.

[Sch11] Mirko Schmidt. *Analysis, Modeling and Dynamic Optimization of 3D Time-of-Flight Imaging Systems*. PhD thesis, IWR, Fakultät für Physik und Astronomie, Univ. Heidelberg, 2011.

[SHGR11] Frank Schumacher, Markus Holzer, Thomas Greiner, and Wolfgang Rosenstiel. Modeling and code generation of recursive algorithms with extended uml activity diagrams. In *Proceedings of the 21st International Conference Radioelektronika, RADIOELEKTRONIKA*, pages 1–4, April 2011.

[SHLS09] Christian Schlegel, Thomas Hassler, Alex Lotz, and Andreas Steck. Robotic software systems: From code-driven to model-driven designs. In *Proceedings of the International Conference on Advanced Robotics, ICAR*, pages 1–8, June 2009.

[Sim96] David Simon. *Fast and Accurate Shape-Based Registration*. PhD thesis, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, October 1996.

[SK10] Alexander Sabov and Jörg Krüger. Identification and correction of flying pixels in range camera data. In *Proceedings of the 24th Spring Conference on Computer Graphics*, pages 135–142, April 2010.

[SLB$^{+}$11] Arvind K. Sujeeth, Hyoukjoong Lee, Kevin J. Brown, Hassan Chafi, Michael Wu, Anand R. Atreya, Kunle Olukotun, Tiark Rompf, and Martin Odersky. Optiml: an implicitly parallel domainspecific language for machine learning. In *Proceedings of the 28th International Conference on Machine Learning, ICML*, pages 609–616, June 2011.

[SP09] Selo Sulistyo and Andreas Prinz. Recursive modeling for completed code generation. In *Proceedings of the 1st Workshop on Behaviour Modelling in Model-Driven Architecture*, pages 6:1–6:7, June 2009.

[ST94] Jianbo Shi and Carlo Tomasi. Good features to track. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition, CVPR*, pages 593–600, June 1994.

[SW99a] Christian Schlegel and Robert Wörz. *Der Softwarerahmen SMARTSOFT zur Implementierung sensomotorischer Systeme*, pages 208–217. Springer Berlin Heidelberg, November 1999.

[SW99b] Christian Schlegel and Robert Wörz. Interfacing different layers of a multilayer architecture for sensorimotor systems using the object-oriented framework smartsoft. In *Proceedings of the Third European Workshop on Advanced Mobile Robots, Eurobot*, pages 195–202, September 1999.

[TM98] Carlo Tomasi and Roberto Manduchi. Bilateral filtering for gray and color images. In *Proceedings of the Sixth International Conference on Computer Vision, ICCV*, pages 839–846, January 1998.

[USC16] USC. The "lena" test image. http://sipi.usc.edu/database/database.php, October 2016.

[WTP$^{+}$09] Manuel Werlberger, Werner Trobin, Thomas Pock, Andreas Wedel, Daniel Cremers, and Horst Bischof. Anisotropic huber-l1 optical flow. In *Proceedings of the British Machine Vision Conference (BMVC)*, London, UK, September 2009. to appear.

[ZF03] Barbara Zitova and Jan Flusser. Image registration methods: a survey. *Image and Vision Computing*, 21(11):977–1000, October 2003.

[Zha94] Zhengyou Zhang. Iterative point matching for registration of free-form curves and surfaces. *International Journal of Computer Vision*, 13(2):119–152, October 1994.

# List of Figures

# List of Tables

# List of Listings

# Abbreviations

**ADL** Architecture Definition Language 5, 124, 125, 127, 131, *Glossary:* Architecture Definition Language

**CASE** Computer Aided Software Engineering 114, 117, *Glossary:* Computer Aided Software Engineering

**CBSE** Component-Based Software Engineering 5, 123–127, 132–137, 139, 140, 142, 144–146, 148, 150, *Glossary:* Component-Based Software Engineering

**CMOS** Complementary Metal-Oxide-Semiconductor 149, *Glossary:* Complementary Metal-Oxide-Semiconductor

**CUDA** Compute Unified Device Architecture 1, 2, 4, 25, 57, 72, 101, 103, 104, 108, 122, 150, *Glossary:* Compute Unified Device Architecture

**DSL** Domain Specific Language 2–4, 6, 7, 17–20, 87, 88, 91, 92, 97, 99, 103–105, 108, 109, 114, 121–125, 144, 147, 148, 150, 151, 174, *Glossary:* Domain Specific Language

**EBNF** Extended Backus Naur Form 104, *Glossary:* Extended Backus Naur Form

**EMF** Eclipse Modeling Framework 174, *Glossary:* Eclipse Modeling Framework

**FOV** Field Of View 9, 14, 75, *Glossary:* Field Of View

**FPGA** Field Programmable Gate Array 4, 5, 24–26, 32, 40, 173, *Glossary:* Field Programmable Gate Array

**FPN** Fixed Pattern Noise 64, *Glossary:* Fixed Pattern Noise

**GMF** Graphical Modeling Framework 2, 18, 88, 91, 92, 98, 104, 121, 147, *Glossary:* Graphical Modeling Framework

**GPGPU** General Purpose Computation on Graphics Processing Unit 4–6, 88, 89, 101–106, 109, 111, 121, 122, 147, 150, 188, *Glossary:* General Purpose Computation on Graphics Processing Unit

**GPU** Graphic Processing Unit 1, 4, 5, 24, 25, 57, 72, 77, 80, 84, 88, 101–112, 114–117, 171–173, *Glossary:* Graphic Processing Unit

**HC** Heterogeneous Computing 101, 109, 110, 112, 120, 121, 126, 147, 150, *Glossary:* Heterogeneous Computing

**ICP** Iterative Closest Point 16, 59, 70–75, 77, 80, 84, *Glossary:* Iterative Closest Point

**ISO** International Organization for Standardization 2, 17, 87, 101, *Glossary:* International Organization for Standardization

**NIR** Near Infrared 64, *Glossary:* Near Infrared

**OCL** Object Contstraint Language 101, 132, 133, *Glossary:* Object Contstraint Language

**OMG** Object Management Group 2, 17, 87, 125, 127, 173, 174, *Glossary:* Object Management Group

**OpenCL** Open Computing Language 1, 2, 18, 87, 101, 102, 104, 108, 109, 111, 112, 114–121, 126, 134, 142, 147, 150, *Glossary:* Open Computing Language

**OpenGL** Open Graphics Library 18, 87, 109, 126, 134, 142, *Glossary:* Open Graphics Library

**PLC** Programmable Logic Controller 60, 80, 81, *Glossary:* Programmable Logic Controller

**PMD** Photonic Mixer Device 1, 3, 4, 7–12, 14–16, 23–25, 27, 29, 39, 41, 42, 46, 56–58, 63, 64, 67–69, 71–77, 80, 83, 84, 149, 174, *Glossary:* Photonic Mixer Device

**RCP** Rich Client Platform 124, 142, 145, 150, *Glossary:* Rich Client Platform

**ROI** Region Of Interest 102, 105, *Glossary:* Region Of Interest

**SBI** Suppression of Background Illumination 8, *Glossary:* Suppression of Background Illumination

**SDK** Software Development Kit 8–10, *Glossary:* Software Development Kit

**SNR** Signal-to-Noise Ratio 2, 7, 23, 42, 64, *Glossary:* Signal-to-Noise Ratio

**SQL** Structured Query Language 18, 87, *Glossary:* Structured Query Language

**ToF** Time-of-Flight 1–5, 7–9, 13, 23–25, 27, 39, 41, 42, 45, 47, 54–57, 60, 61, 65, 73, 76, 80, 83, 84, 149, 151, 173, *Glossary:* Time-of-Flight

**UML** Unified Modeling Language 2, 17, 87, 88, 91, 94–96, 98, 104, 124, 126, 131, 146, 148, 173, 175, *Glossary:* Unified Modeling Language

# Glossary

## A

**Architecture Definition Language** An ADL is a language to design and model (software) system architectures. 5, 124, 127, 169

## C

**Complementary Metal-Oxide-Semiconductor** It is a technology for constructing integrated circuits as microprocessors or other digital circuits. 149, 169

**Component-Based Software Engineering** CBSE is a method in the domain of software engineering to separate problem descriptions and functionality into reusable components. 5, 123, 124, 134, 150, 169

**Compute Unified Device Architecture** CUDA is a parallel computing platform (API) developed by NVIDIA enabling developers to use GPUs for computing. 1, 169

**Computer Aided Software Engineering** CASE is the usage of software tools to automatically create new software based on specialized descriptions. 114, 169

## D

**Domain Specific Language** DSLs are computer languages designed for special domains. 2, 4, 7, 17–19, 88, 91, 148, 151, 169, 174

## E

**Eclipse** Eclipse is an Open-Source software development environment originally developed for Java programming. 88, 91, 92, 99, 104, 121, 142, 147, 171, 172, 187, 188

**Eclipse Modeling Framework** EMF is an Eclipse Java framework for structured system modeling. 104, 169, 174

**Extended Backus Naur Form** The EBNF is a standardized extension of the Backus-Naur-Form (BNF) to visualize the syntax of programming languages. 104, 169

F

**Field Of View** The FOV is the observable world that can be seen at a moment. 9, 169

**Field Programmable Gate Array** A FPGA is an integrated circuit that can be programmed. Using a special hardware description language, designers are able to develop customized, high performance solutions in hardware. 4, 169

**Fixed Pattern Noise** FPN is a location based noise occurring due to underlying (hardware) structure that can mostly be corrected by an offset and gain for each pixel. 64, 169

G

**General Purpose Computation on Graphics Processing Unit** It is the usage of a Graphic Processing Unit for general purpose processing comparable to CPU processing. 4, 169

**Graphic Processing Unit** It is a processor specialized for graphic processing. 1, 169, 172

**Graphical Modeling Framework** GMF is a graphical modeling framework within the toolchain of Eclipse. 2, 169

H

**Heterogeneous Computing** Heterogeneous Computing covers systems using and supporting several different kind of processors as e.g. CPUs and GPUs at the same time. 101, 109, 110, 112, 121, 126, 147, 150, 170

I

**International Organization for Standardization** The ISO is an organization responsible for the development of international standards in all domains. 2, 170

**Iterative Closest Point** The ICP is an algorithm developed to find the error-minimized transformation between two point-clouds. 16, 170

N

**Near Infrared** NIR is an invisible radiant energy with longer wavelength compared to visible light. It lies is in a range between about 750nm and 1400nm. 64, 170

O

**Object Contstraint Language** The OCL is a language standardized by the OMG and part of UML. Using OCL it is possible to define conditions and restrictions during model design. 101, 132, 170

**Object Management Group** The OMG is an international technology standards consortium. 2, 170

**Open Computing Language** OpenCL is a parallel computing interface originally developed by Apple enabling developers the heterogeneous computing across platforms (CPU, GPU, FPGA. OpenCL is maintained by the Khronos Group. 1, 170

**Open Graphics Library** OpenGL is a cross platform API for rendering 2D and 3D vector graphics e.g. on a GPU. 18, 170

**P**

**Photonic Mixer Device** It is an optical sensor working on the Time-of-Flight principle. 1, 170, 174

**Programmable Logic Controller** A PLC is computer commonly used in industrial automation. Using Input/Output and controlling interface components a PLC can control assembly lines or robots. 60, 170

**R**

**Region Of Interest** A Region Of Interest is the range of interesting data points within a data set identified for a specific purpose. In the domain of image processing, a region of interest can e.g. be an area around a single pixel. 102, 170

**Rich Client Platform** A RCP is a platform that can be used as basic runnable software allowing to simply integrate new components. 124, 150, 170

**S**

**Signal-to-Noise Ratio** The SNR is a measure to compare the signal and the background noise. Ratios higher than 1:1 indicate that a signal is available. 2, 7, 42, 64, 170

**Software Development Kit** A SDK is a collection of tools that allows the development of applications based on it. 8, 170

**Structured Query Language** SQL is a language to query and manage data in relation databases. 18, 170

**Suppression of Background Illumination** The Suppression of Background Illumination of a Photonic Mixer Device is a mechanism to enlarge the ratio between the distance carrying signal and the uncorrelated extraneous light. It reduces unnecessary charge carriers within the individual pixel channels A and B (e.g. the steady component). 8, 170

**T**

**Time-of-Flight** It is a method to measure the distance between a light source and and a reflecting object. 1, 83, 170, 173

**U**

**Unified Modeling Language** UML is a graphical modeling language to design and visualize systems. It is standardized by the OMG. 2, 170

**X**

**Xbase** Xbase is a reusable javalike expression language implemented using xText. 94, 104

**Xtend** Xtend is a javalike programming language implemented using xText. 109, 134

**xText** xText is an Open-Source-Framework for the development of Domain Specific Languages and part of the Eclipse Modeling Framework (EMF) project. 2, 18, 87, 88, 91, 98–100, 104, 121, 123, 147, 174

# A GU-DSL Important Syntax Constructs

Using associations well-known from the graphical UML modeling allows the developer to define relations between objects. This is simply done by defining fields in classes, as can be seen here:

```
// Image contains 0 .. to n sub images
self aggregation [0 .. *] subImages : Image;

// At least one or more pixel belong to an image
self composition [1 .. *] pixel : Pixel;

// Bidirectional association: Image contains
// 0 to n pixel. One pixel belongs to one image
self [1] contains : Pixel [0 .. *];

// A uni-directional association: A pixel
// has a parent, but the parent does not know it
self parentImage : Image;
```

The GU-DSL expression grammar allows three types of variables declarations:

1. Mutable variables, defining variables that can be changed:

```
var int i = 0;
var Image image;

```

2. Constant variables, defining variables that cannot be changed:

```
const int j = 0;

```

3. Reference variables, as known from C/C++ as pointers, allowing direct access to memory addresses:

```
var ref int k = 0;
var ref Image image;

```

Reference variables are treated in a special way, different from the C/C++ dereferencing mechanism. Depending on the assigned value type, the kind of assignment is automatically chosen (see Listing 29).

```
var int i = 0;          // Declare variable i
const int j = 0;        // Declare constant j
var ref int k = ref i;  // Assigning i as
                        // reference to k
k = 100;                // Assigns 100 to k,
                        // and respectively to i
k = j;                  // Assigning the content
                        // of j == 0 to k
k = ref j;              // Assigning the j as
                        // new reference to j
```

**Listing 29** Reference assignment and automatic dereferencing.

Using this mechanism simplifies the language usage and improves maintainability. Additionally, through the support of references, it is possible to develop fast code, which is the most important requirement in image processing algorithm development.

**Loop Expressions** Loops are a central part of programming languages, allowing simple repetition of statements. Three kinds of control structures are supported:

1. Head-controlled loops:

```
loop(i < 100) { i = i +1; };

```

2. Tail-controlled loops:

```
do { i = i + 1; } loop(i <= 100);

```

3. Conditionally head-controlled loops:

```
for(i = 0 : 1 : i < 100) { };

```

**Conditional Expressions** Besides loops, two kinds of conditional expressions are supported:

1. If-Else-expressions:

```
if(i < 100) { k = 0; }
else{ k = 1; };

```

2. Switch-Case-expressions:

```
switch i:
{
case 0: {}
default: {}
};

```

# B GU-DSL Examples

```
ActivityDiagram Main()
{
  // ... Variable declarations

  swimlane P1
  {
    start S1
    {
      => initViewer;
    }

    action initViewer
    {
      viewer = new Viewer2D();
      => loadImage(imageFilename);
    }

    action loadImage(string asd)
    {
      image.load(asd);
      image = image.convertToFormat(QImageFormat::Format_ARGB32);

      => initGPUBuffer;
    }

    action initGPUBuffer()
    {
      imageContainer = new ImageContainer<unsigned byte>("bilateralImageIn", HCImageChannelOrder.RGBA);
      imageContainer.create(image.width(), image.height());

      outImageContainerBilateral = new ImageContainer<unsigned byte>("bilateralImageOutBilateral", HCImageChannelOrder.RGBA);
      outImageContainerBilateral.create(image.width(), image.height());

      outImageContainerMean = new ImageContainer<unsigned byte>("bilateralImageOutMean", HCImageChannelOrder.RGBA);
      outImageContainerMean.create(image.width(), image.height());

      buffer = new unsigned byte[ ( image.width() * image.height() ) * 4];

      var int iy;
      for(iy = 0 : 1 : iy < image.height())
      {
        var ref unsigned byte scanline = image.scanLine(iy);
        var ref unsigned byte lineAdr = buffer + iy * (4 * image.width());

        CPPMethods::memcpy(lineAdr, scanline, (4 * image.width()));
      };



      => initRandomReduceBuffer;
    }
```

**Listing 30** The full textual Main activity diagram of all examples. Part 1

```
    action initRandomReduceBuffer()
    {
      bufferAllElements = new unsigned int[max_elements];

      inFindMinElements = new BufferContainer<unsigned int>("inFindMinElements");
      inFindMinElements.create(max_elements, 1);

      outFindMinElements = new BufferContainer<int>("outFindMinElements");
      outFindMinElements.create(max_elements, 1);

      outFindLastElement = new BufferContainer<int>("outFindLastElement");
      outFindLastElement.create(1, 1);

      CPPMethods::srand(CPPMethods::time(0));

      var int i;
      for(i = 0 : 1 : i < max_elements)
      {
        bufferAllElements[i] = i;
      };

      for(i = 0 : 1 : i < max_elements)
      {
        var int index1 = CPPMethods::rand() % max_elements;
        var int index2 = CPPMethods::rand() % max_elements;

        var unsigned int tmpValue = bufferAllElements[index1];
        bufferAllElements[index1] = bufferAllElements[index2];
        bufferAllElements[index2] = tmpValue;
      };

      => syncToGPU;
    }

    action syncToGPU()
    {
      imageContainer.fill(buffer);
      imageContainer.syncToGPU();

      inFindMinElements.fill(bufferAllElements);
      inFindMinElements.syncToGPU();

      => addViewerImages;
    }

    loop addViewerImages(var int i = 0; i < 4; i = i + 1)
    {
      var int newImageIndex = viewer.addImageObject(0, 200, 200);
      viewer.setImage(image, newImageIndex, false);

      => showViewer(1000, 600);
    }

    action showViewer(int sizeX, int sizeY)
    {
      viewer.show();
      viewer.resize(sizeX, sizeY);

      => processImages;
    }

    action processImages()
    {
      imageIndex = imageIndex + 1;

      [imageIndex == 1] => filterBilateralCL(image.width(), image.height(), 0, imageIndex);
      [imageIndex == 2] => filterMean(image.width(), image.height(), 0, imageIndex);
      [imageIndex == 3] => startReduction();
      []=> f;
    }

    calc filterBilateralCL(int imageIndex)
    {
      call BilateralFilter::filterBilateral(imageContainer, outImageContainerBilateral, 5.0f, 10.0f, 50.0f);

      => showFilteredImage(outImageContainerBilateral);
    }

    calc filterMean(int imageIndex)
    {
      call MeanFilter::filterMean(imageContainer, outImageContainerMean, 10);

      => showFilteredImage(outImageContainerMean);
    }
```

**Listing 31** The full textual Main activity diagram of all examples. Part 2

```
action showFilteredImage(ref ImageContainer<unsigned byte> container)
{
  container.syncToCPU();
  viewer.setImage(container.hostPointer() as ref void, container.numberOfElements() * 4, 1, image.width(),
 image.height(), 5121, imageIndex, false, 3);

  => processImages;
}

action startReduction
{
  => estimateMinPow2ForInputSize(max_elements, nbWorkItemsNeeded);
}

action estimateMinPow2ForInputSize(int inputSize, out int value)
{
  value = -1;
  var int maxInt = 2147483647;

  var int i;
  for(i = 1 : i = (i * 2) : i < (maxInt / 2))
  {
    if(i >= inputSize)
    {
      value = i;
      return;
    }
  }

  return;

  => estimateWorkSizes(inFindMinElements.device(), max_elements);
}

action estimateWorkSizes(HCDevicePtr device,
             int problemSize)
{

  var int maxLocalWorkingSize = device.getMaxDeviceWorkGroupSize();

  var int maxNumberOfWorkgroups = maxLocalWorkingSize;
  var int localWorkingSize = maxLocalWorkingSize;

  // Calculated the work-groups needed
  nbWorkgroups = nbWorkItemsNeeded/localWorkingSize;

  // Limit the number of work-groups if necessary
  if(nbWorkgroups > maxNumberOfWorkgroups)
    nbWorkgroups = maxNumberOfWorkgroups;

  // Calculate the number of elements, each work-group has to process
  var int nbElementsPerWorkgroup = nbWorkItemsNeeded / nbWorkgroups;

  // Calculate the number of elements, each work-item has to process.
  numberOfElementsToProcessPerWorkItem = nbElementsPerWorkgroup / localWorkingSize;

  localRangeLarge = localWorkingSize;
  globalRangeLarge = nbWorkgroups*localWorkingSize;

  localRangeSmall = nbWorkgroups;
  globalRangeSmall = nbWorkgroups;

  => findIndexFirstStage(globalRangeLarge, 1, localRangeLarge, 1);
}

calcRange findIndexFirstStage()
{
  call FindIndex::findMinimumIndexFirstStage(inFindMinElements,
                       outFindMinElements,
                       max_elements,
                       localRangeLarge * 4,
                       localRangeLarge,
                       bufferAllElements[valueIndexToFind],
                       numberOfElementsToProcessPerWorkItem);

  => findIndexSecondStage(globalRangeSmall, 1, localRangeSmall, 1);
}

calcRange findIndexSecondStage()
{
  call FindIndex::findMinimumIndexSecondStage(inFindMinElements,
                       outFindMinElements,
                       outFindLastElement,
                       nbWorkgroups,
                       nbWorkgroups * 4,
                       nbWorkgroups,
                       bufferAllElements[valueIndexToFind]);

  => findIndexThirdStage;
}
```

**Listing 32** The full textual Main activity diagram of all examples. Part 3

```
    action findIndexThirdStage()
    {
      outFindLastElement.syncToCPU();
      var ref int indexPtr = outFindLastElement.hostPointer();
      var int index = indexPtr;

      Console::writeValue("Searched index: ");
      Console::writeValue(valueIndexToFind);
      Console::writeEOL();

      Console::writeValue("Found index: ");
      Console::writeValue(index);
      Console::writeEOL();

      => processImages;
    }

    final f
  }
}
```

**Listing 33** The full textual Main activity diagram of all examples. Part 4

```
ClassDiagram Reduction
{
  import OpenCL.GPU.*;

  public class FindIndex implements OpenCL
  {
    public global void findMinimumIndexFirstStage(global ref unsigned int data,
                   global ref int idxReduced,
                   const int nbElements,
                   shared ref unsigned int bufferLocal,
                   const int bufferSizeLocal,
                   const unsigned int searchElement,
                   const int nbElementsToPreprocessPerWorkItem)
    {
      call behavior FindMinimumIndexFirstStage(data, idxReduced, nbElements, bufferLocal, bufferSizeLocal,
      searchElement, nbElementsToPreprocessPerWorkItem);

    }

    public global void findMinimumIndexSecondStage( global ref unsigned int data,
                       global ref int idxReduced,
                       global ref int idxReducedSingleElement,
                       const int nbElements,
                       shared ref unsigned int bufferLocal,
                       const int bufferSizeLocal,
                       const int searchElement)
    {
      call behavior FindMinimumIndexSecondStage(data, idxReduced, idxReducedSingleElement, nbElements,
      bufferLocal, bufferSizeLocal, searchElement);
    }

    private int bindBetterValueAndReturnNumber(const unsigned int a, const unsigned int b, const unsigned int
      searchElement)
    {
      call behavior BindBetterValueAndReturnNumber(a, b, searchElement);
    }

    private int findBetterValueIdx(global ref unsigned int data, int idxA, int idxB, unsigned int searchElement
      )
    {
      call behavior FindBetterValueIdx(data, idxA, idxB, searchElement);
    }

    private void reduceFindFirstGreaterOrEqualIdx(global ref unsigned int data,
                           shared ref unsigned int bufferLocal,
                           int bufferSizeLocal,
                           int idxLocal,
                           const int searchElement)
    {
      call behavior reduceFindFirstGreaterOrEqualIdx(data, bufferLocal, bufferSizeLocal, idxLocal,
      searchElement);
    }

  }

}
```

**Listing 34** The class diagram of the Reduction activity diagram example from the GPGPU programming chapter.

```
ActivityDiagram BindBetterValueAndReturnNumber(const unsigned int a, const unsigned int b, const unsigned int
      searchElement)
{
  public int returnValue = -1;

  swimlane Swimlane1, owner Reduction.FindIndex
  {
    start S1
    {
      => checkElements;
    }

    action checkElements
    {
      [a >= searchElement && b >= searchElement] => handleBothValuesToLarge;
      [a >= searchElement && b < searchElement]  => handleAToLarge;
      [b >= searchElement && a < searchElement]  => handleBToLarge;
      =>f;
    }

    action handleBothValuesToLarge
    {
      if(a <= b)
      {
        returnValue = 0;
      }
      else
      {
        returnValue = 1;
      };

      => f;
    }

    action handleAToLarge
    {
      returnValue = 0;
      => f;
    }

    action handleBToLarge
    {
      returnValue = 1;
      => f;
    }

    final f return returnValue

  }
}
```

**Listing 35** Reduction activity diagram example: Comparison of two indices.

```
ActivityDiagram reduceFindFirstGreaterOrEqualIdx(global ref unsigned int data,
                         shared ref unsigned int bufferLocal,
                         int bufferSizeLocal,
                         int idxLocal,
                         const int searchElement)
{

  import OpenCL.GPU.*;

  swimlane Swimlane1, owner Reduction.FindIndex
  {
    start S1
    {
      => block_threads;
    }

    action block_threads
    {
      barrier(barrier_flags.CLK_LOCAL_MEM_FENCE);

      => loopBuffer;
    }

    loop loopBuffer(var int i = bufferSizeLocal/2; i > 0; i = i / 2)
    {
      if (idxLocal < i)
      {
        var int idxA = bufferLocal[idxLocal];
        var int idxB = bufferLocal[idxLocal + i];

        var int retval = findBetterValueIdx(data, idxA, idxB, searchElement);

        bufferLocal[idxLocal] = retval;

      };
      barrier(barrier_flags.CLK_LOCAL_MEM_FENCE);

      => f;
    }

    final f
  }

}
```

**Listing 36** Reduction activity diagram example: Try to find a good starting index

```
ActivityDiagram FindBetterValueIdx(int g, global ref unsigned int data, int idxA, int idxB, unsigned int
      searchElement)
{
  public int returnValue = -1;

  swimlane Swimlane1, owner Reduction.FindIndex
  {
    start S1
    {
      => evaluateIndex;
    }

    action evaluateIndex
    {
      [idxA != -1 && idxB != -1] => handleBothIndicesValid;
      [idxA != -1 && idxB == -1] => handleIndexAValid;
      [idxB != -1 && idxA == -1] => handleIndexBValid;
      =>f;
    }

    action handleBothIndicesValid
    {
      var unsigned int valA = data[idxA];
      var unsigned int valB = data[idxB];

      var int test = bindBetterValueAndReturnNumber(valA, valB, searchElement);


      switch (test):
      {
        case 0:
          returnValue = idxA
        case 1:
          returnValue = idxB
      };

      => f;
    }

    action handleIndexAValid
    {
      var int valA = data[idxA];

      if(valA >= searchElement)
      {
        returnValue = idxA;
      };

      => f;
    }

    action handleIndexBValid
    {
      var int valB = data[idxB];

      if(valB >= searchElement)
      {
        returnValue = idxB;
      };

      => f;
    }

    final f return returnValue
  }
}
```

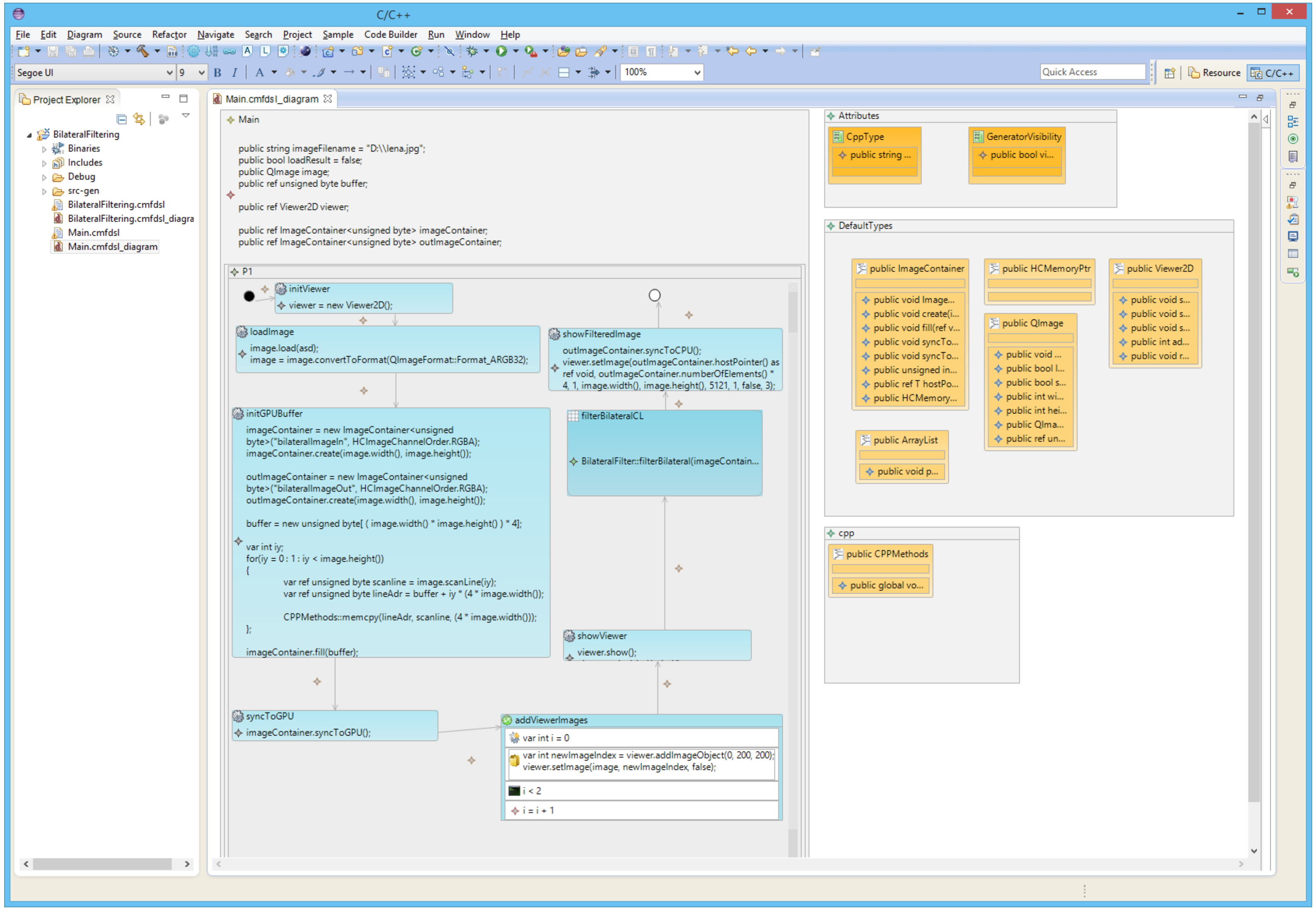
**Listing 37** Reduction activity diagram example: Best value search.

```
ActivityDiagram FindMinimumIndexFirstStage(global ref unsigned int data,
                       global ref int idxReduced,
                       const int nbElements,
                       shared ref int bufferLocal,
                       const int bufferSizeLocal,
                       const unsigned int searchElement,
                       const int nbElementsToPreprocessPerWorkItem)
{

  public int idxGlobal;
  public int idxGroup;
  public int idxLocal;
  public int globalSize;
  public unsigned int idxA;

  swimlane Swimlane1, owner Reduction.FindIndex
  {
    start S1
    {
      => init;
    }

    action init
    {
      idxGlobal = get_global_id(1) * get_global_size(0) + get_global_id(0);

      // If the global work-item-index exceeds the problem size, return.
      if(idxGlobal >= nbElements)
        return;

      // Calculate the local work-item-index
      idxGroup = get_group_id(1) * get_num_groups(0) + get_group_id(0);

      // Calculate the local work-item-index
      idxLocal = get_local_id(1) * get_local_size(0) + get_local_id(0);
      globalSize = get_global_size(0);

      idxA = idxGlobal;

      [nbElementsToPreprocessPerWorkItem > 1] => handleWorkItemElements;
      => reduceFirstElements;

    }

    loop handleWorkItemElements(var int i = 0; i < nbElementsToPreprocessPerWorkItem; i = i + 1)
    {
      var int idxB = idxGlobal + i*get_global_size(0);

      if(idxB< nbElements)
      {
        idxA = findBetterValueIdx(data, idxA, idxB, searchElement);
      };

      => reduceFirstElements;
    }


    action reduceFirstElements
    {
      bufferLocal[idxLocal] = idxA;
      reduceFindFirstGreaterOrEqualIdx(data, bufferLocal, bufferSizeLocal, idxLocal, searchElement);

      [idxLocal == 0] => reduceLastGroup;
      => f;
    }

    action reduceLastGroup
    {
      idxReduced[idxGroup] = bufferLocal[0];
      => f;
    }

    final f


  }
}
```

**Listing 38** Reduction activity diagram example: Index search, first stage.

```
ActivityDiagram FindMinimumIndexSecondStage( global ref unsigned int data,
                       global ref int idxReduced,
                       global ref int idxReducedSingleElement,
                       const int nbElements,
                       shared ref int bufferLocal,
                       const int bufferSizeLocal,
                       const int searchElement)
{
  public int idxGlobal;
  public int idxGroup;
  public int idxLocal;
  public int globalSize;
  public unsigned int idxA;

  swimlane Swimlane1, owner Reduction.FindIndex
  {
    start S1
    {
      => init;
    }

    action init
    {
      idxGlobal = get_global_id(1) * get_global_size(0) + get_global_id(0);

      // If the global work-item-index exceeds the problem size, return.
      if(idxGlobal >= nbElements)
        return;

      // Calculate the local work-item-index
      idxGroup = get_group_id(1) * get_num_groups(0) + get_group_id(0);

      // Calculate the local work-item-index
      idxLocal = get_local_id(1) * get_local_size(0) + get_local_id(0);

      => loadDataFromGlobal;
    }

    action loadDataFromGlobal
    {
      // Load data from global to local memory
      bufferLocal[idxLocal] = idxReduced[idxGlobal];

      => reduceFirstElements;
    }

    action reduceFirstElements
    {
      reduceFindFirstGreaterOrEqualIdx(data, bufferLocal, bufferSizeLocal, idxLocal, searchElement);

      [idxLocal == 0] => reduceLastGroup;
      => f;
    }

    action reduceLastGroup
    {
      idxReducedSingleElement[0] = bufferLocal[0];
      => f;
    }

    final f

  }
}
```

**Listing 39** Reduction activity diagram example: Index search, second stage.



```
public string imageFilename = "D:\\lena.jpg";
public bool loadResult = false;
public QImage image;
public ref unsigned byte buffer;

public ref Viewer2D viewer;

public ref ImageContainer<unsigned byte> imageContainer;
public ref ImageContainer<unsigned byte> outImageContainer;
```

```
initViewer
viewer = new Viewer2D();
```

```
loadImage
image.load(asd);
image = image.convertToFormat(QImageFormat::Format_ARGB32);
```

```
showFilteredImage
outImageContainer.syncToCPU();
viewer.setImage(outImageContainer.hostPointer() as
ref void, outImageContainer.numberOfElements() *
4, 1, image.width(), image.height(), 5121, 1, false, 3);
```

```
initGPUBuffer
imageContainer = new ImageContainer<unsigned
byte>("bilateralImageIn", HCImageChannelOrder.RGBA);
imageContainer.create(image.width(), image.height());

outImageContainer = new ImageContainer<unsigned
byte>("bilateralImageOut", HCImageChannelOrder.RGBA);
outImageContainer.create(image.width(), image.height());

buffer = new unsigned byte[ ( image.width() * image.height() ) * 4];

var int iy;
for(iy = 0 : 1 : iy < image.height())
{
          var ref unsigned byte scanline = image.scanLine(iy);
          var ref unsigned byte lineAdr = buffer + iy * (4 * image.width());

          CPPMethods::memcpy(lineAdr, scanline, (4 * image.width()));
};

imageContainer.fill(buffer);
```

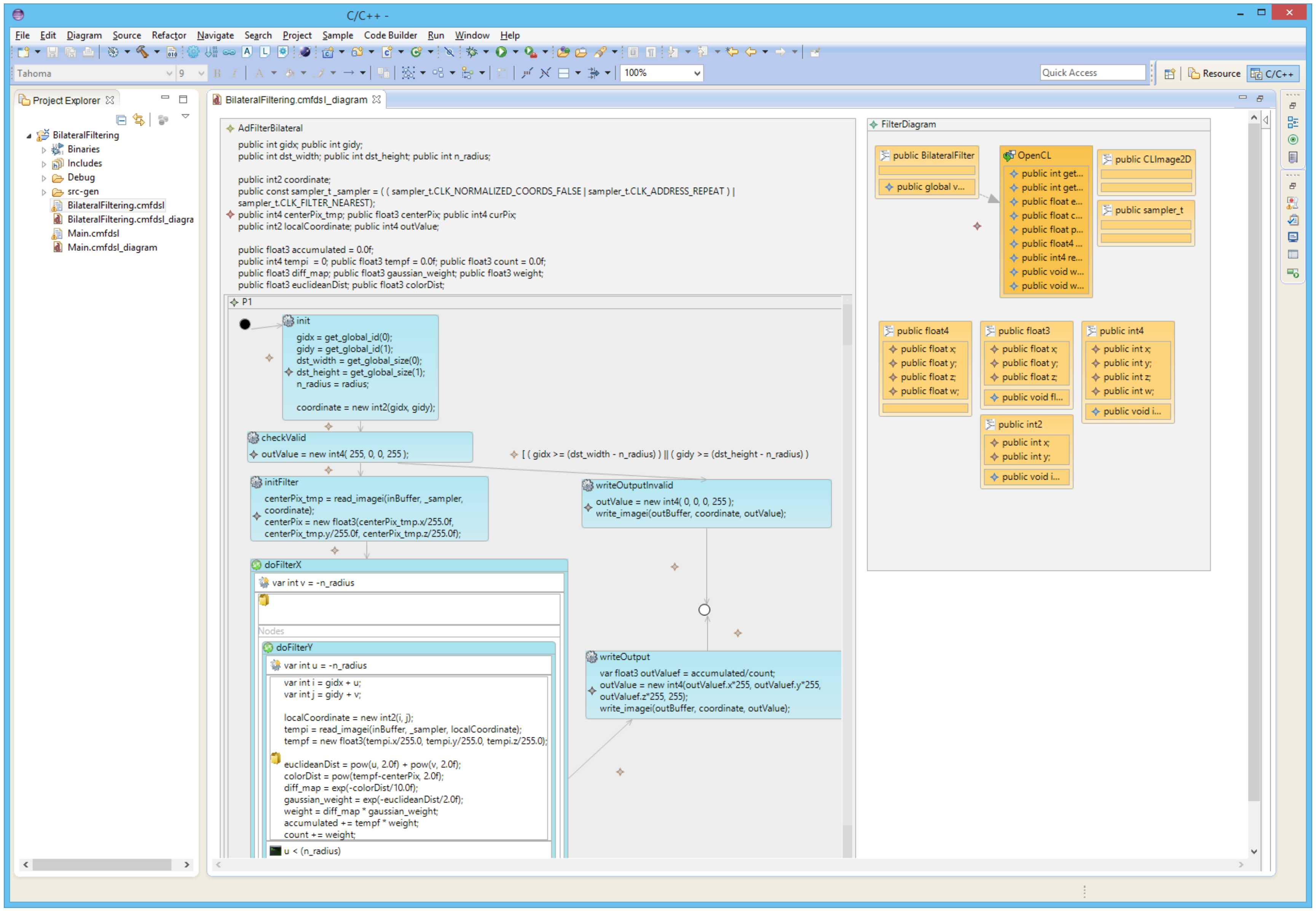
```
filterBilateralCL
BilateralFilter::filterBilateral(imageContain...
```

```
showViewer
viewer.show();
```

```
syncToGPU
imageContainer.syncToGPU();
```

```
addViewerImages
var int i = 0
var int newImageIndex = viewer.addImageObject(0, 200, 200);
viewer.setImage(image, newImageIndex, false);
i < 2
i = i + 1
```

Figure B.1: The Eclipse IDE showing the designed Main diagram and the corresponding classes.



```
public int gidx; public int gidy;
public int dst_width; public int dst_height; public int n_radius;

public int2 coordinate;
public const sampler_t _sampler = ( ( sampler_t.CLK_NORMALIZED_COORDS_FALSE | sampler_t.CLK_ADDRESS_REPEAT ) |
sampler_t.CLK_FILTER_NEAREST);
public int4 centerPix_tmp; public float3 centerPix; public int4 curPix;
public int2 localCoordinate; public int4 outValue;

public float3 accumulated = 0.0f;
public int4 tempi  = 0; public float3 tempf = 0.0f; public float3 count = 0.0f;
public float3 diff_map; public float3 gaussian_weight; public float3 weight;
public float3 euclideanDist; public float3 colorDist;
```

```
gidx = get_global_id(0);
gidy = get_global_id(1);
dst_width = get_global_size(0);
dst_height = get_global_size(1);
n_radius = radius;

coordinate = new int2(gidx, gidy);
```

```
outValue = new int4( 255, 0, 0, 255 );
```

```
[ ( gidx >= (dst_width - n_radius) ) || ( gidy >= (dst_height - n_radius) ) ]
```

```
centerPix_tmp = read_imagei(inBuffer, _sampler,
coordinate);
centerPix = new float3(centerPix_tmp.x/255.0f,
centerPix_tmp.y/255.0f, centerPix_tmp.z/255.0f);
```

```
outValue = new int4( 0, 0, 0, 255 );
write_imagei(outBuffer, coordinate, outValue);
```

```
var int v = -n_radius
```

```
var int u = -n_radius
```

```
var int i = gidx + u;
var int j = gidy + v;

localCoordinate = new int2(i, j);
tempi = read_imagei(inBuffer, _sampler, localCoordinate);
tempf = new float3(tempi.x/255.0, tempi.y/255.0, tempi.z/255.0);

euclideanDist = pow(u, 2.0f) + pow(v, 2.0f);
colorDist = pow(tempf-centerPix, 2.0f);
diff_map = exp(-colorDist/10.0f);
gaussian_weight = exp(-euclideanDist/2.0f);
weight = diff_map * gaussian_weight;
accumulated += tempf * weight;
count += weight;
```

```
u < (n_radius)
```

```
var float3 outValuef = accumulated/count;
outValue = new int4(outValuef.x*255, outValuef.y*255,
outValuef.z*255, 255);
write_imagei(outBuffer, coordinate, outValue);
```

Figure B.2: The Eclipse IDE showing the designed Bilateral GPGPU Filter diagram and the corresponding classes