Charon Suite Software Framework

modular algorithms for image processing

Jens-Malte Gottfried* Daniel Kondermann*

June 27, 2012

Abstract

Here, we describe an approach to implement algorithms in a modular way simplifying code-reusing and analysis of individual parts of algorithms. The Charon Suite software provides base classes and interfaces for data interchange and module collaboration arranged in so called workflows. This includes tools for easy module generation, graphical configuration and visualization as well as workflow execution. Additionally, we provide a large collection of modules for optical flow estimation, 3D reconstruction and general image processing tasks as well as example workflows as starting point for experimenting with re-arranging the algorithms.

1 History and Goals

We start with a short motivation (1.1) followed by the goals we wanted to achieve with our software (1.2) and the project history (1.3).

1.1 Motivation

First, we describe the widespread way of implementing algorithms. A straight-forward implementation consists of some large function or procedure, perhaps calling some subroutines representing the algorithm steps. The final result is one monolithic executable or some script which is highly tuned and performs well on some test sequences or images.

Assuming some experience, the resulting code is easy to understand, works efficiently and with high performance.

On the other hand, the resulting code is often very specialized and may not be reusable for other algorithms or may be difficult to adapt because of many implementation details or tweaks. At least it is very difficult to keep the algoritms generic and maintaining such a library causes much effort. Parameters for tuning the algorithms are often hard-coded so that one really has to read the code to figure out their values. Hence, the parameters are invisible from outside.

Or even worse, two different implementations of the same algorithm may yield different results, but it is not possible to compare them directly because of many design choices.

Many complex algorithms consist of several steps of data processing. Comparing similar algorithms, only some of the steps may differ. Implementing such a similar algorithm one has to change existing code and replace some function calls or even perform major changes to make the data processing fit to the second algorithm.

*Heidelberg Collaboratory for Image Processing, University of Heidelberg

1.2 Goals

To simplify this, we propose to divide complex algorithms into parts and to implement these parts separately from each other. We call these parts *modules*. Although this idea is not new, we provide a first open-source implementation of a generic framework for implementing such modules with a common interface and a well documented specification.

The modular solution has the disadvantage that there is some overhead of module communication and that there is no possibility for global optimization or tuning to reach the performance of the monolithic version.

Given one has implemented some modules, they are reusable in similar algorithms without changes of the module code. The common interface hides implementation details. All one has to do is to arrange a data processing pipeline where data is passed from module to module.

It is possible to create new algorithms by rearranging the processing pipeline and perhaps adding or replacing some of the modules by new ones. Therefore, the modular approach simplifies development of new algorithms.

Testing of parts of algorithms is easier than testing the whole complex algorithm itself. It is possible to write unit tests for single modules, simplifying debugging of the whole algorithm. For debugging purposes, one may replace some modules by other, already tested ones to identify which one still contains some bugs. This way, *white-box testing* of the algorithms may be realized. Using existing modules, it is less likely to have bugs in there because they have already been tested and working. Not having to touch working code assures that no new bugs are introduced. Replacing only parts of the algorithm assures that no effects are caused by implementation details in the common code base.

Using a common module interface simplifies idea and algorithm exchange. Modules may be published, others use them and play around implementing some changes or developing new variants of the published algorithm. In this case, the reusage of unchanged parts of the algorithm assures that the new effects are really caused by the changed modules and not by implementing the common parts in a different way.

Another advantage of this approach is that it's possible to store the algorithm configuration (i.e. the workflow configuration file) together with the experiment results making sure that the experiment keeps reproduceable.

Using scripts that replace the parameter values in the config files and run the workflow execution from command line, it is possible to run sweeps or parameter iterations, even parallelized.

1.3 History

This first Charon implementation was done by Kondermann (2009) in his PhD thesis. Although beeing written in a modular way, the modules were pieces of templated c++ code. These pieces have been compiled into one large monolithic executable resulting in huge compilation times even on small code changes. Module interaction was controlled by hand-written text files. This was the first running Charon implementation of a module framework for optical flow estimation.

As a next step, configuration has been simplified by providing the GUI configuration tool *Tuchulcha*. Additionally, the module base classes have been reworked to provide their meta informations to be used during configuration.

Subsequently, the object framework has been refactored, such that the modules are now represented as small dynamic libraries like a plug-in mechanism. This way, the modules became independent of the management and configuration software. The new `PluginManager` now acts as a factory for creating module instances out of the dynamic libraries.

In the time afterwards, large module collections have been developed. The most important of them are called *charon-utils*, *charon-flow* and *hekate*. They are described in detail in the following sections.

Recently, the possibility of *dynamic modules* (see below) has been added as a first step of implementing so called *supernodes*, i.e. modules that represent an own sub-workflow. These supernodes are currently in focus of development and will be available soon.

2 Content

Before describing the parts in detail, we give an overview of the building blocks that make up the Charon-Suite software framework (2.1), then proceed with the details (2.2) and show the implmented algorithms (2.3).

2.1 Overview

Figure 1 shows the main blocks of the Chaoron-Suite software framework. Each block represents a standalone software project and may be built individually. Dependencies are visualized using arrows.

Module collections are bundles of dynamic libraries that represent single modules, i.e. parts of the algorithms as described above. Each module classes is derived from the ParameteredObject class, perhaps using some intermediate classes (e.g. handling templates).

Block Description:

charon-core: module interface specification (i.e. the base class `ParameteredObject`, Parameters, Slots) and processing pipeline management (`PluginManager`)

tuchulcha: GUI workflow configuration application

template-generator: application for easy module generation

Additionally there are module (library) collections for optical flow estimation (**charon-flow**), 3D reconstruction (**hekate**) and general image processing tasks (**charon-utils**).

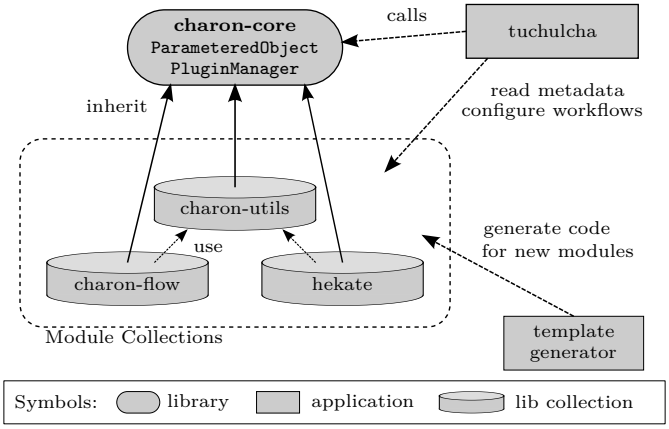

Figure 1: Main Building Blocks of the Charon-Suite software framework including libraries and applications

2.2 Detailed Description

We implemented a library called *charon*[1] (named after the ferryman in greek mythology). This library includes a common module interface and manages data-interchange as well as execution of the processing pipeline. There are many helper tools to simplify creation of modules, setup of processing pipelines and module configuration.

We tried to keep the restrictions as small as possible. Modules are not forced to use any library, the data to be passed may be arbitrary. Modules may even pass pointers of some interface types which makes ist possible to pass e.g. functions or other objects that are evaluated later (like e.g. interpolation schemes). This way, one may distinguish between data-like and object-like connections.

Even in this situation, wich may be confusing at first sight, side effects and module dependencies stay visible in the configuration graph. Unconnected modules cannot influence each other. This is another advantage compared to the monolithic approach where changing something in a function may cause effects somewhere else that are hard to track.

Many modules for image processing and some flow estimation algorithms are provided and usuable out-of-the-box e.g. using the also provided example workflows[2]. This includes well-known algorithms like global optical flow estimation by Horn & Schunck (1981) but also state-of-the-art methods like by Sun et al. (2008). More recent algorithms are currently under development.

One of the mentioned helper-tools is a wizard to help users to generate new modules, the so called *template-generator*. It creates the source code files needed to get the plugin working. Slots, parameters and documentation may be configured in this graphical interface. Later, one has to fill in the code into the execute method and compile the module. The CMake project files that are also generated, simplify this task so that even beginners may get their first modules working within a few hours.

The GUI configuration tool *Tuchulcha* (named after a helper-daemon of Charon) may be used to configure the module parameters, set up and visualize the processing pipeline. It shows the module documentation and is able to export the visualization into graphic files (svg, png, pdf).

[1] http://charon-suite.sourceforge.net
[2] http://charon-suite.sourceforge.net/doc/examples/

To execute the configured workflow, it calls the charon-core base classes in a seperate process (*thuchulcha-run*) which may also be used as a standalone command line application. In this process, the PluginManager is used to generate and set up the module instances and start the workflow.

2.3 Implemented Algorithms

Here we show, which algorithms are already implemented in the mentioned module collections and so usable out-of-the-box. Using the provided example configuration files, it is easy to get these agorithms working, have a look how they work and tune the algorithm parameters as needed.

charon-utils

Here, there are modules for data input/output based on Cimg and Hdf5. This way, the supported file formats range from png, jpeg, tiff to all formats supported by ImageMagic (if installed). Hdf5 handling is done via the Vigra library.

Additionally, there are basic image manipulation modules like blur, resize, threshold as well as linear filtering, derivative calculations etc. These basic modules are used in more complex workflows later, e.g. for data preprocessing.

charon-flow

The provided algorithms range from the early optical flow estimation methods like the one proposed by Horn & Schunck (1981) to recent state-of-the-art methods. This includes iterated and multiscale versions of the mentioned global method, the *learning flow* algorithm by Sun et al. (2008), with the provided gaussian-mixture penalty functions as well as with charbonnier penalty functions as proposed e.g. by Papenberg et al. (2006).

Recent development added the *classic+nonlocal* algorithm proposed by Sun et al. (2010). Additionally, there are modules to apply the global optical flow methods to depth images resulting in *range flow* aka *scene-flow* estimation (used in Gottfried et al., 2011).

hekate

With this collection, it is possible to reconstruct 3D scenes using a *Structure from Motion* approach (Wang & Wu, 2011; Hartley & Zisserman, 2005) using feature-detection (e.g. SIFT) and tracking. Outliers of feature correspondences may be eliminated as proposed by Hartley & Zisserman (2005); Li & Hu (2010). Meshes may be generated by Delaunay triangulation. Additionally, cameras may be auto-calibrated by image sequences and the results may be used for camera tracking.

3 Design Decisions

The software framework has been implemented as a C++ library including helper tools. Some of these tools provide GUIs that use the Qt[3] libraries. We decided to implement each module as a shared library that is loaded dynamically during workflow execution.

[3] http://qt.nokia.com

Data interchange is done similar to a signal/slot model (described by Dexin et al., 2006) as it is used by the Qt or boost libraries. This model implements the observer design pattern (description e.g. in Freeman et al., 2004; Shalloway & Trott, 2004). Each module defines input and output slots which may be connected to other modules. Modules may query which slots are connected and e.g. only perform needed calculations and skip generation of unneeded data.

To implement the module interface, new modules are derived from the `ParameteredObject` base classes and have to register their input/output slots and configurable parameters. Modules override the virtual `execute` method that contains the code to perform the data processing of the algorithm part the module represents.

Recent development added the possibility to let modules determine their interface based on their configuration, i.e. to add parameters or slots if needed (e.g. providing a parameter to adjust the number of input/output slots). Such modules are called *dynamic modules*.

Charon is platform independent and has been tested on Windows, Linux and MacOS. The compiled module libraries are platform-specific. To exchange modules in a platform independent way, their source code has to be provided.

For project build, we use CMake as platform-independent Makefile-generator (see Martin & Hoffman, 2010).

We provide binary distributions[4] for Windows (32/64 bit), Ubuntu packages (also work on Debian) as well as Gentoo and Arch linux build scripts. This way, users may experiment with the implemented modules without having to compile the whole framework themselves.

Charon is no framework providing matrix classes, solvers or image formats itself but may be integrated with arbitrary external frameworks. Currently, we use CImg[5] and the Vigra framework by Köthe (2000) to represent image data, Qt for graphical user interfaces and e.g. PETSc[6] to solve linear equation systems.

The whole software framework is published as open-source under the terms of the GNU Lesser General Public License (LGPL). This way, users may publish their modules under the terms of arbitrary licenses without violating the LGPL conditions. If modules contain GPL code or other restricted code, this has to be stated in the module documentation.

4 Usage and Examples

The Charon-Suite software framework is intended to be used by computer vision developers. Usually it is used by students during internships or during their bachelor, master or PhD thesis. This means that there are various levels of background knowledge within the target user group. Therefore it is possible for unexperienced users to use the provided workflows as a black-box but writing or extending the modules one has the full flexibility to adapt algorithms to the needs.

4.1 Simple Usage Example

A very simple workflow using the image manipulation modules from *charon-utils* is shown in Figure 2. An image file (penguin) is loaded by the `FileReader` module which passes the loaded

[4] http://sf.net/apps/trac/charon-suite/wiki/InstallationGuide
[5] http://cimg.sourceforge.net
[6] http://www.mcs.anl.gov/petsc

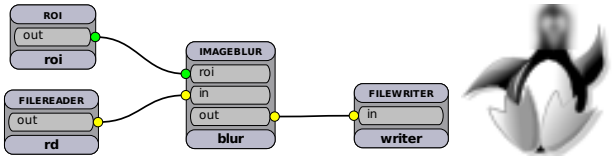

Figure 2: Simple workflow loading an image and blurring a selected region (penguin head). The resulting file is printed beside the workflow visualization.

data to a `ImageBlur` instance. The result is then written back to disk by the `FileWriter`. The blur region is selected by a `ROI` (region-of-interest) which has parameters for region coordinates (top-left, bottom-right, up to 5D).

4.2 Optical Flow Example

Figure 3 shows the module interaction of our implementation of the algorithm by Horn & Schunck (1981). The image derivatives are calculated by the plugin "diff2d", the data term "bcce" is assembled using a brightness model and a motion model which introduce the horizontal and vertical optical flow components u and v as unknowns. The plugin "l2norm" represents the spatial term. The solver "petscsolver" builds the matrix for solving global flow estimation and solves it numerically using the PETSc library.

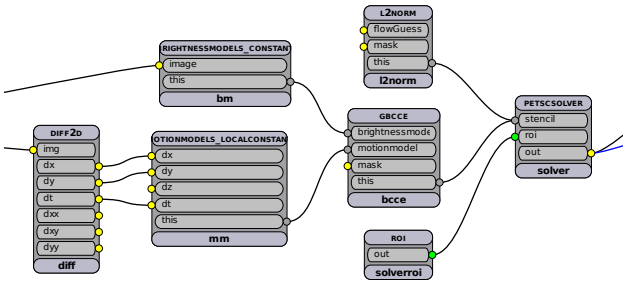

Figure 3: Example workflow showing the central part of our modular implementation of the algorithm by Horn & Schunck (1981). For simplicity, the modules reading the image data (at the left) and writing the flow result to a file (at the right) have been omitted. The visualization has been generated usint the *tuchulcha* configuration and visualization tool.

References

Xu Dexin, Tan Zhenfan and Gao Yanbin. Developing application and realizing multiplatform based on qt framework. *Journal of Northeast Agricultural University*, 3, 2006.

Elisabeth Freeman, Eric Freeman, Bert Bates and Kathy Sierra. *Head first design patterns*. O'Reilly & Associates, Inc., 2004. ISBN 0-596-00712-4.

Jens-Malte Gottfried, Janis Fehr and Christoph S. Garbe. Computing range flow from multi-modal kinect data. In George Bebis, Richard D. Boyle, Bahram Parvin, Darko Koracin, Song Wang, Kyungnam Kim, Bedrich Benes, Kenneth Moreland, Christoph W. Borst, Stephen DiVerdi, Yi-Jen Chiang and Jiang Ming, editors, *ISVC (1)*, volume 6938 of *Lecture Notes in Computer Science*, pages 758–767. Springer, 2011. ISBN 978-3-642-24027-0.

Richard Hartley and Andrew Zisserman. Multiple view geometry in computer vision. *Robotica*, 23(2):271, 2005.

Berthold K. P. Horn and Brian G. Schunck. Determining optical flow. *Artif. Intell.*, 17(1-3):185–203, 1981.

Daniel Kondermann. *Modular optical flow estimation with applications to fluid dynamics*. PhD thesis, University of Heidelberg, 2009. URL http://archiv.ub.uni-heidelberg.de/volltextserver/volltexte/2010/10184.

Ullrich Köthe. *Generische Programmierung für die Bildverarbeitung*. PhD thesis, Department of Informatics, Universität Hamburg, 2000.

Xiangru Li and Zhanyi Hu. Rejecting mismatches by correspondence function. *International Journal of Computer Vision*, 89(1):1–17, 2010.

Ken Martin and Bill Hoffman. *Mastering CMake*. Kitware, Inc., USA, 5 edition, 2010.

Nils Papenberg, Andrés Bruhn, Thomas Brox, Stephan Didas and Joachim Weickert. Highly accurate optic flow computation with theoretically justified warping. *International Journal of Computer Vision*, 67(2):141–158, 2006.

Alan Shalloway and James R. Trott. *Design patterns explained: a new perspective on object-oriented design*. Addison Wesley, 2 edition, 2004. ISBN 0-321-24714-0.

Deqing Sun, Stefan Roth, J. P. Lewis and Michael J. Black. Learning optical flow. In David A. Forsyth, Philip H. S. Torr and Andrew Zisserman, editors, *ECCV (3)*, volume 5304 of *Lecture Notes in Computer Science*, pages 83–97. Springer, 2008. ISBN 978-3-540-88689-1.

Deqing Sun, Stefan Roth and Michael J. Black. Secrets of optical flow estimation and their principles. In *CVPR*, pages 2432–2439. IEEE, 2010.

Guanghui Wang and Q.M. Jonathan Wu. *Guide to Three Dimensional Structure and Motion Factorization*. Springer Publishing Company, Incorporated, 2011.