Available online at www.sciencedirect.com
ScienceDirect
Journal of Structural Biology 157 (2007) 28–37
Journal of Structural Biology
www.elsevier.com/locate/yjsbi

Collaborative EM image processing with the IPLT image processing library and toolbox ☆

Ansgar Philippsen *, Andreas D. Schenk, Gian A. Signorell, Valerio Mariani, Simon Berneche, Andreas Engel

Maurice E. Müller Institute for Structural Biology, Biozentrum Basel, Switzerland

Received 12 April 2006; received in revised form 13 June 2006; accepted 15 June 2006
Available online 14 July 2006

Abstract

We present the Image Processing Library and Toolbox, IPLT, in the context of a collaborative electron microscopy processing effort, which has driven the evolution of our software architecture over the last years. The high-level interface design as well as the underlying implementations are described to demonstrate the flexibility of the IPLT framework. It aims to support the wide range of skills and interests of methodologically oriented scientists who wish to implement their ideas and algorithms as processing code.

Keywords: Image processing; Electron crystallography; Software engineering

1. Introduction

Structural investigations of biological samples with the electron microscope is a continuously growing and evolving field. The analysis of the measurements to extract the structural information is referred to as image processing, because a collection of images forms the majority of raw data. The 3D EM community has produced, and is producing, a large collection of sophisticated image processing algorithms, made available in several software packages. Here we describe one of these software packages, the Image Processing Library and Toolbox (IPLT), that was started in our group, with a special emphasis on the dynamic nature of the 3D EM research.

In our first paper on IPLT (Philippsen et al., 2003), we discussed the rational for creating a new image processing package from scratch, our intended goals, and the architectural design. We wish to provide a modular, integrated, collaborative, flexible, extendable, open-source, cross-platform framework for image processing of electron microscope images. Modular in the sense that it is build from clearly separated building blocks that may be used together as seen fit; integrated implying that it is a single package whose components have been made to work together; collaborative meaning that we do not want it to become an in-house software only but evolving by means of community involvement; flexible because it accommodates the various user requirements; extendable hinting at the intended ease of adding algorithms and procedures.

This manuscript continues pretty much where the previous one left off at, namely how the concepts and the first implementation of the IPLT framework have proven themselves within a, albeit small, collaborative environment, in particular describing the maturation of the architecture and the currently implemented image processing modules. This is preceded by a recapitulation of the major features as well as a discussion of the implementation.

To allow a more concise description of the current state of IPLT, the software engineering terms user, abstraction, interface, encapsulation, implementation, instance, class, and object are introduced. Since these are all coupled, a suitable definition is the generic statement: An interface offers an abstract interaction of some encapsulated functionality to a user, who does not need to know any details

☆ The IPLT homepage is found at http://www.iplt.org.

* Corresponding author. Fax: +41 61 267 2109.
E-mail address: ansgar.philippsen@unibas.ch (A. Philippsen).

1047-8477/$ - see front matter © 2006 Elsevier Inc. All rights reserved.
doi:10.1016/j.jsb.2006.06.009

of the actual implementation of this functionality. In terms of software development, an abstract concept—like an image—can be implemented as a set of classes in the context of a specific design; each class has a set of methods, which form the class interface; each method has its (hidden) implementation in the programming language, which in turn is an abstraction to the CPU instruction set and implemented in a compiler or interpreter. On the opposite end, an overall image processing scheme is implemented in terms of conceptual steps, each in turn leading to concrete use of the various components of a class library. And finally, the processing application may be completed by adding a graphical user interface.

In an object oriented language, the construct that offers this encapsulation scheme is usually referred to as *class*, and each instance, i.e. independent unit, of a class is usually referred to as *object*. The distinction between class and object may not seem obvious now, and there are certainly many contexts where the terms can be interchanged. Here, we use class when we are talking about a design construct, and object as the actual use of the class.

While the concepts of interface and implementation have been around implicitly since the dawn of programming, it is only with the advent of object oriented design that they have been explicitly formulated, in particular discussing the difficulties and challenges of designing a good interface, which is an art in itself. These concepts have proven to be especially powerful in an environment of larger projects, with evolving software and heterogeneous developers, exemplified by techniques such as refactoring/extension cycles (Fowler, 1999) or Extreme Programming (Beck and Andres, 2004).

In the following sections, we describe the interfaces available in IPLT, descend into some implementation details where appropriate, and explain the rational for this particular design. While this manuscript, like the last one, is still aimed at the interested developer, and not so much the regular user, one of the major claims contained herein is that the particular design of the IPLT empowers a casual user to become a contributor, due to the various framework layers that range from simple Python scripts to sophisticated C++ routines.

2. Technical description

Since the publication of the first paper on IPLT (Philippsen et al., 2003), the software has evolved considerably, not only in the actual processing capabilities, but also in the software engineering sense, mainly due to its direct involvement in a collaborative effort to establish a novel 2D electron crystallography processing suite (see below). As a consequence, several previously described concepts have been superseded, some design decisions were discarded and replaced with more sophisticated ones. These changes affect both the interface as well as the implementation level. Future changes, however, anticipate the integration of new ideas without affecting the interfaces to ensure a stable evolution of the system.

The current software architecture of IPLT is schematized in Fig. 1. It comprises C++ and Python components, where the basic constituents are written in C++ and consequently reflected into Python by means of a wrapper. Conceptually, the C++ level is meant to provide the basic building blocks, and the processing logic is then implemented on the Python level, utilizing these building blocks.

2.1. Images

The **`ImageHandle`** class represents the high-level, central image concept in IPLT. It combines several features and functionalities: First of all, it offers a unified interface to the underlying image state (explained below) and its properties, independent of the actual state used, thereby exempting the users from dealing with all possible state combinations and their instances. This unified interface allows access to the image values as well as retrieval of the essential image properties extent, pixel sampling, type, and domain. Second, the **`ImageHandle`** interface provides a set of **`Apply`** methods which facilitate the interaction with all algorithm objects; in a nutshell, each algorithm can be applied either in-place or out-of-place, independent of the actual algorithm object implementation. Third, in its nature as a handle, it removes all memory management responsibilities from the user: a handle only points to an automatically allocated image state instance, and copying handles will not duplicate this memory area, but it will be shared by all handles, and de-allocated when no longer referred to. This feature has the additional advantage that code in C++ and Python is almost identical, save for syntax differences (see Fig. 2). Fourth, it supports the unary or binary operators negation, addition, subtraction, multiplication, and division, which operate on a per-pixel level.

In view of all this comfort and sophistication, there is but a single drawback of the image handle, namely that reading and writing of pixel values is relatively slow compared to directly accessing the memory. As a consequence, a special mechanism has been implemented to give developers access to the particular image state of an image handle, by means of an image state algorithm—as explained further below.

The image state represents the combination of a particular value storage or pixel type, namely real or complex, with a particular domain, namely spatial, frequency, or half-frequency. It manages the actual pixel value memory, and allows direct, raw access to it. Readers familiar with such a scheme will immediately notice that the template feature of C++ is perfectly suited for implementing such a state ensemble, and this is indeed how it is done. A full description of the templated implementation is, however, beyond the scope of this paper. It must suffice to mention that the particularities of our architecture (mostly the hybrid implementation of C++ and Python as well as the strong modularity) render the use of templates complicated. For this reason, the complexity of dealing with templates has