Proceedings of the 5th WSEAS Int. Conf. on Software Engineering, Parallel and Distributed Systems, Madrid, Spain, February 15-17, 2006 (pp73-81)

A Modular Software Framework for Camera Calibration

STEPHAN RUPP
Image Processing And Medical Engineering Department
Fraunhofer-Institute for Integrated Circuits IIS
Am Wolfsmantel 33, 91058 Erlangen
GERMANY

http://www.iis.fraunhofer.de

Abstract: - Camera calibration is an indispensable step for augmented reality or image guided applications. In the context of three-dimensional machine vision, camera calibration is the process to determine the internal camera geometry and optical characteristics and/or the 3-D position and orientation of the camera frame relative to a certain world coordinate system. Usually, a camera calibration procedure exhibits typical steps, including feature point location in the acquired images, camera model fitting, correction of distortion introduced by the optics and finally an optimization of the model's parameters. Thus, the components required for designing new calibration procedures show a high degree of similarity, so that reuse of the processing steps plays an important role. For this reason, we present fundamental design issues of a component-based calibration framework, which guides and supports the researcher in writing reusable software components and therefore, improves the efficiency of his work. The modularity enables him to modify only aspects of a given calibration procedure allowing to deepen the understanding on how the different steps influence its overall performance.

Key-Words: - Component-based software framework, interface injection pattern, camera calibration

1 Introduction

Generically, calibration is the problem of estimating values for the unknown parameters in a sensor model in order to determine the exact mapping between sensor input and output. A wide range of computer vision applications exist which require an accurate calibration of the visual system. In these application, certain quantitative information is extracted from the 2-D images and overall performance depends on calibration accuracy [1].

According to [2], there are four main problems when designing a whole calibration procedure: control point location in the images, camera model fitting, image correction for radial and tangential distortion and estimating the errors originated in these stages. In many cases, in a fifth step, these errors are combined into a merit function that is subject to an optimization procedure yielding an improvement of the camera model's parameters. While many research has been devoted to model fitting, and some implementations are publicly available [3, 4, 5, 6], few works can be found in the literature about the other stages of the process and surprisingly no approaches have been published concerning software frameworks that support the scientist in concentrating on the different steps of calibration while leaving the others untouched.

Thus due to the structure of calibration, when investigating new calibration procedures, reuse plays a decisive role. Powerful tools such as software architecture and design patterns [7][8, 9] aiming at the development of reusable software (systems) have been evolved from the software engineering community, however they are very abstract due to their catholicity and require certain experience in order to be chosen and applied right.

On the other hand, the researchers' ambition is to find new approaches for solving the calibration problem. In order to attain this, they generally do not spend much interest in writing aesthetic, highly reusable code, nor are they willing to invest much time on secondary aspects like graphical user interface programming or other technical issues. Instead, they would like to concentrate on the functional aspect and be able to realize their work efficiently.

2 Background

2.1 Camera Calibration Theory

Capturing a scene with a camera implies a mapping of 3-D (world) coordinates to 2-D sensor coordinates. This mapping is usually described by means of a camera model which is characterized by several parameters that are commonly subdived into *extrinsic* and *intrinsic camera parameters*. Camera calibration is the process to determine parts or all of these parameters.

The most simple and commonly used camera model is that of a *pinhole camera*, incorporating the perspective projection of 3-D world coordinates onto the 2-D imaging plane which is in most cases equivalent to the CCD sensor array of a camera. In this model, the relationship between the 2-D pixel coordinates and 3-D world coordinates can be described by a 3×4 matrix $\tilde{\mathbf{P}}$, called projection matrix, which maps points from the projection space $\mathcal{P}^3$ to the projective plane $\mathcal{P}^2$, and can be decomposed uniquely:

$$\tilde{\mathbf{P}} = \lambda_{\mathrm{w}} \underbrace{\begin{pmatrix} \alpha_u & \gamma & u_0 \\ 0 & \alpha_v & v_0 \\ 0 & 0 & 1 \end{pmatrix}}_{\mathbf{A}} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \underbrace{\begin{pmatrix} \mathbf{R}_{\mathrm{w}} & \mathbf{t}_{\mathrm{w}} \\ \mathbf{0}_3^T & 1 \end{pmatrix}}_{\mathbf{D}_{\mathrm{w}}}$$

The 3×4 matrix $\mathbf{A}$, whose five entries are called *intrinsic parameters*, describes the change of the retinal coordinate system. The *angle of skew* γ between the two axes of the CCD sensor array is frequently set to zero, justified by the virtually orthogonal image axes in most modern CCD cameras. A camera is calibrated when the matrix $\mathbf{A}$ is known, so that one can use *normalized coordinates* which have an Euclidean meaning. The 4×4 displacement matrix $\mathbf{D}_{\mathrm{w}}$ describes the change of the world coordinate system (pose of the camera) called *extrinsic parameters*.

2.2 Camera Calibration Procedure

Camera calibration is usually performed by observing a special calibration object, which is in most cases a flat plate with a regular pattern marked on it using colors causing a high contrast between the marks and the background. The pattern is chosen such that the image coordinates of the projected reference points can be measured with high accuracy. The relationship between the 2-D image coordinates and 3-D world coordinates is given by the collinearity equations [10, 11] or rather the homography [2, 12, 13] by applying the projection matrix $\tilde{\mathbf{P}}$ along with projective coordinates for the image plane and the 3-D space:

$$\begin{pmatrix} u \\ v \\ 1 \end{pmatrix} = \tilde{\mathbf{P}} \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix} \tag{1}$$

with $(u, v)'$ denoting the image coordinates of a mark and $(X, Y, Z)'$ being the corresponding world coordinate. Using a great number F of marks, each one yielding an equation of the form equation (1), the perspective transformation matrix $\tilde{\mathbf{P}}$ of the visual system can be estimated. For stability reasons, usually $N = 6 \ldots 20$ images of the calibration target are captured from different positions and the identified correspondences are utilized to build up the equation system yielding $\tilde{\mathbf{P}}$.

In general, lens-based optical imaging systems exhibit geometric distortions in acquired images due to liabilities in lens manufacturing. Especially, when considering wide-angle lenses usually a strong barrel distortion is observable, which necessitates a correction. These distortions are typically modelled by *radial* and *tangential* distortion and introduced into the calibration process as distortion coefficients adjusting the image coordinates $(u, v)'$. They extend the set of camera parameters and are reasonably assigned to the intrinsic camera parameters, because they affect the image formation.

In many cases the camera model's parameters are adjusted within a subsequent (non-linear) optimization of the so-called *back-projection error*, yielding an improvement of the overall model fitting quality:

$$\epsilon = \sum_{i=1}^{N} \sum_{j=1}^{F} \epsilon_{ij}, \quad \epsilon_{ij} = \left\| \begin{pmatrix} u_{ij} \\ v_{ij} \\ 1 \end{pmatrix} - \tilde{\mathbf{P}} \cdot \begin{pmatrix} X_{ij} \\ Y_{ij} \\ Z_{ij} \\ 1 \end{pmatrix} \right\|_2$$

The back-projection error of a single calibration mark ϵ_{ij} can be basically understood as the Euclidean distance between its initially extracted image coordinates and the corresponding 3-D world coordinates being projected back to the image plane with the (current) camera model's parameters.

As a *merit function* of the optimization procedure the sum of all back-projection errors ϵ of all the marks in all the images is often considered and its minimization is persued.

3 Related Work

Application frameworks for special domains have been intensivly proposed in the past years. Such approaches support the user in writing applications for the specific domain they were designed for. As a representant, the *lgf3* framework is concerned with image based rendering, especially with light field appliances [14]. In order to speed-up research activities, a versatile software framework has been proposed, that encapsulates single tasks within Unix processes that consume and produce standardized datastructures. As a matter of fact, the approach is similar to our, as it encapsulates essential tasks of a light field application in computational entities. However, it differs in how these entities are models; within lgf3, an entity is modelled as a Unix process that is connected to subsequent steps via the Unix pipe mechanism.

In the scope of camera calibration, there exists surprisingly no dedicated application framework, that supports the user in developing or studying camera calibration schemes. However, several toolboxes and publicly available algorithms exist [3, 4, 5]. The most famous toolbox is called *Calibration Toolbox for Matlab* [6] and requires the Matlab environment. Unfortunately, extensability and modularity was not in the scope when the toolkit has been designed. Instead it provides a simple graphical user interface for locating calibration marks of a chessboard pattern. Internally, it makes use of the algorithm of Heikkilä and Silven's, published in [2, 15]. However, the high complexity of its Matlab code makes it hard to extend or customize the toolbox – even for unexperienced Matlab users.

Similarily, Zhang and Heikkilä provide implementations of their approaches as stand-alone application [5] or Matlab scripts [4]. Again, these solutions distract a scientist from a systematic analysis of the calibration steps or the development of new schemes on the basis of established algorithms since they require a not quite negligible effort for customization.

4 Contribution

In this contribution, we propose a framework based on a modular software architecture with distinctive calibration steps being represented by software components. The *software component* itself is a *computational unit* [7] and represents "*a coherent package of software that can be independently developed and delivered as a unit, and that offers interfaces by which it can be connected and exchanged with other components*" [16].

Our approach enforces modularity due to its design and supports the scientists in writing reusable software components with the algorithm of a certain calibration step being encapsulated in a *plugin*. This modularity enables the researcher to modify only aspects of a given setup when designing new calibration schemes. On the other hand, exchanging and modifying only parts of an existing technique allows to deepen its understanding and how the different steps influence the overall performance.

The framework is designed to be independent from any graphical user interface, but provides abstract interfaces in order to handle graphical user interface interaction such as mouse events or drawing facilities (i.e. for selection and visualization of calibration marks). This in conjunction with fundamental interfaces and base classes for calibration specific datastructures, scientists are enabled to focus on the functional aspect while realizing their work quickly and easily.

We introduce a design pattern, called *interface injection pattern*, that is applied in conjunction with the visitor pattern [8] in order to realize a *reflexion mechanism* that allows an automatic generation of configuration dialogs. Furthermore, we present an universal data exchange mechanism that requires no prior knowledge of the data's type, so that in general every data type can be used directly without writing any adaption code. In the context of a calibration tool, the mechanism is applied in order to exchange the plugins' data with a tool's graphical user interface. For this, we identify the typical input and output data of the different calibration steps and provide abstract base class implementations of these fundamental datatypes. By applying the well-known *adapter* design pattern [8], third-party datatypes can easily adopted to the fundamental type and thus used within the framework. The data exchange mechanism has been designed to be efficient regarding performance and the system's resources with the framework being responsible for memory managment and holding off (de)allocation code from the developer.

5 Methods

5.1 Framework Requirements

Many camera calibration techniques often only differ in a few aspects such as calibration mark location or the applied merit function. So the development of new or understanding of existing ones requires the exchange of only parts of the whole system (i.e. the merit function) and demands for a **modular design principle**, so that the **reuse of the remaining parts (algorithms)** is feasible.

In order to support the developer the framework should prevent him from being distracted from **concentrating on the functional part** of his work. This can be achieved by certain **automatisms** provided by the framework, i.e. for rendering configuration dialogs from the specification of the algorithms' parameters, GUI programming for capturing mouse movement, dragging and dropping of image area selections or hide component communication and resource control from the user.

In addition, the creation of new plugins should be quickly and easily feasible. This demands for a **reflexion mechanism** providing **meta information** and a facility that gathers information about the plugin's connection points, the algorithm's parameters and dependent third-party libraries in order to automatically generate all the necessary project files and code skeletons.

5.2 Design and Implementation Aspects

This section is concerned with fundamental design decision and implementation aspects in order to meet all the requirements discussed in section 5.1. Due to the limited space, we have to restrict the coverage of design and implementation details to the fundamental key concepts of our approach.

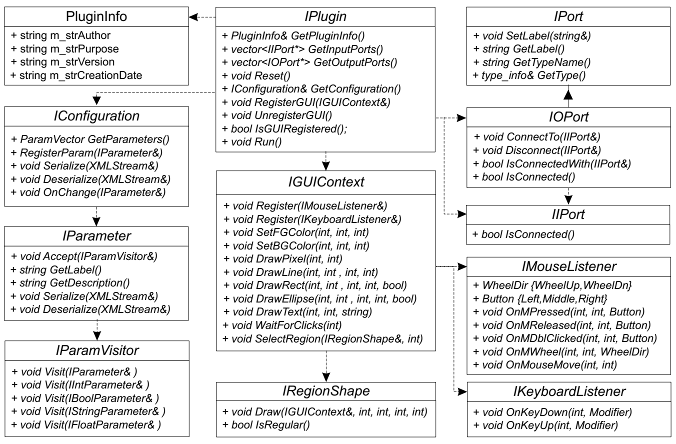

PluginInfo
+ string m_strAuthor
+ string m_strPurpose
+ string m_strVersion
+ string m_strCreationDate
IPlugin
+ PluginInfo& GetPluginInfo()
+ vector<IIPort*> GetInputPorts()
+ vector<IOPort*> GetOutputPorts()
+ void Reset()
+ IConfiguration& GetConfiguration()
+ void RegisterGUI(IGUIContext&)
+ void UnregisterGUI()
+ bool IsGUIRegistered();
+ void Run()
IPort
+ void SetLabel(string&)
+ string GetLabel()
+ string GetTypeName()
+ type_info& GetType()
IConfiguration
+ ParamVector GetParameters()
+ RegisterParam(IParameter&)
+ void Serialize(XMLStream&)
+ void Deserialize(XMLStream&)
+ void OnChange(IParameter&)
IOPort
+ void ConnectTo(IIPort&)
+ void Disconnect(IIPort&)
+ bool IsConnectedWith(IIPort&)
+ bool IsConnected()
IGUIContext
+ void Register(IMouseListener&)
+ void Register(IKeyboardListener&)
+ void SetFGColor(int, int, int)
+ void SetBGColor(int, int, int)
+ void DrawPixel(int, int)
+ void DrawLine(int, int , int, int)
+ void DrawRect(int, int , int, int, bool)
+ void DrawEllipse(int, int , int, int, bool)
+ void DrawText(int, int, string)
+ void WaitForClicks(int)
+ void SelectRegion(IRegionShape&, int)
IIPort
+ bool IsConnected()
IParameter
+ void Accept(IParamVisitor&)
+ string GetLabel()
+ string GetDescription()
+ void Serialize(XMLStream&)
+ void Deserialize(XMLStream&)
IMouseListener
+ WheelDir {WheelUp,WheelDn}
+ Button {Left,Middle,Right}
+ void OnMPressed(int, int, Button)
+ void OnMReleased(int, int, Button)
+ void OnMDblClicked(int, int, Button)
+ void OnMWheel(int, int, WheelDir)
+ void OnMouseMove(int, int)
IParamVisitor
+ void Visit(IParameter&)
+ void Visit(IIntParameter&)
+ void Visit(IBoolParameter&)
+ void Visit(IStringParameter&)
+ void Visit(IFloatParameter&)
IRegionShape
+ void Draw(IGUIContext&, int, int, int, int)
+ bool IsRegular()
IKeyboardListener
+ void OnKeyDown(int, Modifier)
+ void OnKeyUp(int, Modifier)

Figure 1: The framework is designed to be independent from any graphical toolkit which is attained by a set of interfaces defining fundamental services.

Regarding the implementation, we decided to use the C++ programming language due to its dominance in industrial and medical image processing projects, object-oriented character and its powerful template mechanism supporting generative programming techniques [17].

5.2.1 Graphical User Interface Abstraction

One of our major concerns when designing the framework was decoupling the frameworks business logic from any graphical user interface (GUI), so that everybody is able to use it in conjunction with his favorite graphical toolkit. In accordance with this, the framework is built on a set of interface classes, providing all the services an application may need to manage the plugins – this includes access to the parameters and user interactions such as capturing of mouse events (fig. 1).

▷ Graphic Contexts

In order to support the developer in focussing on calibration related issues, user interface details are encapsulated and hidden from the user. The framework provides a clear interface for drawing, picking and (image) region selection.

For example, if a plugin's algorithm requires a rectangular region within an image presented on the screen, the developer simply calls `GUI().SelectRegion()`. The `GUI()`-method is part of the plugin's interface and returns a reference to an implementation of the `IGUIContext` interface (i.e. a hypothetical calibration tool's image viewer widget). Whenever a plugin is loaded, the widget's implementation of the `IGUIContext` interface is automatically bound to the plugin, so that within the plugins code, a call to `GUI()` will always result in a valid context object.

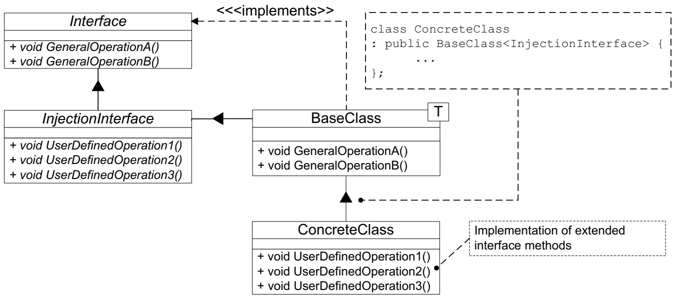
In order to decouple the graphical feedback of the region selection process from a specific implementation, an object can optionally be passed that implements the `IRegionShape` interface. The framework itself calls the `IRegionShape`'s paint method within the selection routine of the current `IGUIContext` implementation, so that a user can simply introduce customized shapes that are expressed by calls to the `IGUIContext`'s abstract drawing methods.

▷ User Interaction Abstraction

In addition to the former drawing context abstraction, the framework provides two interfaces, that are concerned with peripheral user interface devices. So, if a programmer is interested in the mouse's position or in a keyboard event, he only has to inherit form the corresponding listener interface (`IMouseListener` or `IKeyboardListener`, see fig. 1), implement its notification methods and announce his interest by registering this implementation in the GUI context by calling `GUI().RegisterListener(*this)`.

▷ Interface Injection Pattern

We introduce a design pattern, called *interface injection pattern*, that is based on a template mechanism in conjunction with subclassing technique. The design pattern is related to the *external polymorphism* pattern of [18] and concerned with extending


Interface
+ void GeneralOperationA()
+ void GeneralOperationB()
<<<implements>>
InjectionInterface
+ void UserDefinedOperation1()
+ void UserDefinedOperation2()
+ void UserDefinedOperation3()
BaseClass
T
+ void GeneralOperationA()
+ void GeneralOperationB()
ConcreteClass
+ void UserDefinedOperation1()
+ void UserDefinedOperation2()
+ void UserDefinedOperation3()
Implementation of extended interface methods

```
class ConcreteClass
: public BaseClass<InjectionInterface> {
      ...
};
```

Figure 2: The *interface injection pattern* allows to extend a common interface class (***Interface***) by introducing a customized interface (***InjectionInterface***) into a inheritance structure. Since the customization is passed as template argument of the generic base class (**BaseClass**), no multiple inheritance mechanism is required.

a class' interface by application specific methods. In contrast to other methods [8], this pattern does not require a multiple inheritance mechanism.

The pattern consists of an abstract class (***Interface***) defining the general interface of an application-specific aspect. This class resides on top of the inheritance structure and its implementation is realized in a templated class (**BaseClass**) that is **not** directly derived from the interface class. Instead, its super class is parameterized by the template argument. By this, customized interfaces (***InjectionInterface***) can easily extend the base interface by inheriting the common interface, declaring additional methods and passing this extended interface as template parameter to the base class.

When developing families of classes (**ConcreteClass**), that all exhibit a common interface, but still vary in some aspects (expressed by different additional methods), all the classes are derived from the base class and inject their special aspect as an extended interface by passing it as template parameter. Since the customization inherits the common interface, the base class provides implementation for the general methods and the classes definition only receives implementation for the additional methods.

In the following, we examplarily demonstrate the application of the *interface injection pattern* to the parameter reflexion mechanism of the framework.

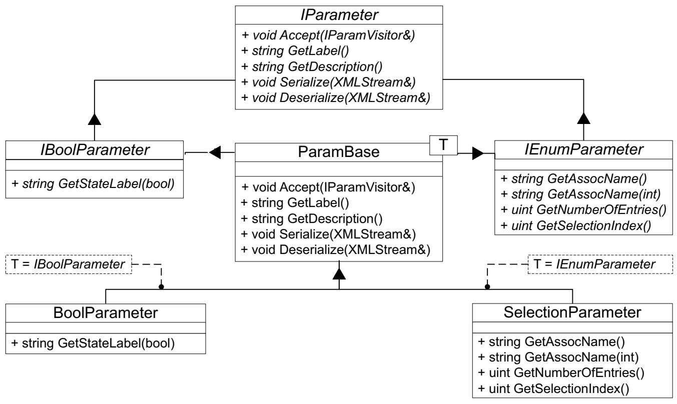

IParameter
+ void Accept(IParamVisitor&)
+ string GetLabel()
+ string GetDescription()
+ void Serialize(XMLStream&)
+ void Deserialize(XMLStream&)
IBoolParameter
+ string GetStateLabel(bool)
ParamBase
T
+ void Accept(IParamVisitor&)
+ string GetLabel()
+ string GetDescription()
+ void Serialize(XMLStream&)
+ void Deserialize(XMLStream&)
IEnumParameter
+ string GetAssocName()
+ string GetAssocName(int)
+ uint GetNumberOfEntries()
+ uint GetSelectionIndex()
T = IBoolParameter
T = IEnumParameter
BoolParameter
+ string GetStateLabel(bool)
SelectionParameter
+ string GetAssocName()
+ string GetAssocName(int)
+ uint GetNumberOfEntries()
+ uint GetSelectionIndex()

Figure 3: The class diagramm for a boolean parameter and a parameter that allows the selection of different alternatives. The interface injection pattern introduce type-specific methods and extends the common `IParameter` interface.

5.2.2 Reflexion

In order to focus on the essentials of an algorithm secondary aspects have to be automated. A reflexion mechanism gathers and provides relevant information about a plugin that is used by the framework to generate configuration dialogs or for performing semantic checks on connection of plugins.

▷ Parameters and Dialog Rendering

The parameter's interface class `IParameter` provides standardised services for getting and setting the parameter's value as well as methods for associating its type with a name and a description of its purpose. This information can be accessed by a GUI in order to automatically render the dialogs with mapping the types to GUI elements or utilizing the purpose description for displaying help.

All accesses are realized by type-safe method calls so that no parsing and interpretation of additional textual description is necessary. This is realized by designing a *double dispatch* mechanism, in order to select the appropriate rendering algorithm based on the parameters' types at runtime.

Double dispatching is a mechanism that dispatches a function call to different concrete functions depending on the runtime types of *multiple objects* involved in the call. In most object-oriented systems, the concrete function that is called from a function call in the code depends on the dynamic type of *a single object* and therefore they are known as *single dispatch* calls, or simply virtual function calls.

At a first glance, double dispatching appears to be a natural result of function overloading. Function overloading allows the function called to depend on the type of the argument as well as the class on which it is called, but calling an overloaded function goes through at most one virtual table, so dynamic dispatching is only based on the type of the calling object. The problem is that, while virtual functions are dispatched dynamically in C++, function overloading is done statically.

The problem described above is resolved with an application of the visitor pattern [8]. The parameter interface `IParameter` defines an accept method, that is generically implemented in the templated base class implemenation `ParamBase`:

```
template <class T>
class ParamBase : public T {
public:
  ...
  virtual void Accept(IParamVisitor& rVis) {
    rVis.Visit(*this);
  }
  ...
};
```

The argument requires an object that implements the `IParamVisitor` interface class, which provides overloaded `Visit()` methods, each for a certain type.

The rendering engine itself implements this interface and places the code for appropriate widget creation in the corresponding methods. For example, let a `IEnumParameter` require a widget that allows the user to select between several options whereas a `IBoolParameter` demands for a check box widget to determine its state. Accordingly, the `Visit(const IEnumParameter&)` method carries the code for the creation of a *combo box* GUI element whereas the `Visit(const IBoolParameter&)` contains code for a *check box* construction.

When rendering the configuration dialog, the GUI iterates through the plugin's list of parameters and calls their `Accept()` method while passing itself as visitor. The base class implementation of `Accept()` calls the visitor's `Visit()` method and provides a `this` pointer as argument. By this, the decision how a parameter object is rendered is resolved by the runtime system, which re-routes the request to one of the overloaded visitor methods that matches the concrete type of the provided parameter object.

According to the different parameter types, injection interfaces have been modelled that incorporate the special aspects of the different types. Each of these interfaces is derived from the common `IParameter` whereas the implementation of a concrete parameter type is derived from the templated base class with the injection interface as template parameter. Fig. 3 depicts the class diagramm for a boolean parameter and a parameter that allows the selection between different alternatives. As we can see `IBoolParameter` and `IEnumParameter` define type-specific methods that extend the common `IParameter` interface. Finally, the application of the interface injection pattern yields two concrete parameter classes, one representing a boolean parameter (`BoolParameter`) and the other one modelling an enumeration (`SelectionParameter`).

5.2.3 Data Exchange

As one of the requirements, data exchange should be hidden from the user, so that he does not have to concern himself with writing allocation or deallocation code in order to pass tokens. However, inter-component communication should still be efficient and flexible. In order to support the user, the framework takes responsibility for managing all the resources that are required in order to transport tokens from one plugin to another one.

For the sake of simplicity the standard data passing is done by value, therefore the data is copied. To gain performance, data passing by reference, along with automatic memory management, that eliminates copying wherever possible can then be reintroduced separately (see 5.2.4). Decoupling these two problems (data passing, and resource management) not only allows for simpler solutions to both problems, it also has the advantage that it is not mandatory to use the automatic resource management. This is advantageous in situations where very simple or even primitive datatypes are passed, since for those copying is more efficient than to perform extensive resource management.

The data exchange mechanism is implemented on two levels: One abstract, polymorphic level, that allows the application framework to connect arbitrary plugins, with arbitrary datatypes as input and output tokens, and one concrete, typed level which provides a simple and typesafe interface to the application programmer.

For the first level a purely abstract interface class exposes functions for connecting ports and extracting framework and GUI relevant data (fig. 1). The interfaces' methods allow the application core to handle data connections without treating different datatypes explicitly. Type safety for the connection of ports is given through a type check using the C++ builtin runtime type information (RTTI) system. If the datatypes of an input and an output port are not compatible an exception is thrown. The ports themselves are accessible through interface functions of the core plugin class, so that standardized entry points for dynamic link libraries are possible.

These interfaces are then realized by a templated set of classes that implement the interface functions, along with the typed functions intended for use by the application programmer.

```
template <class T>
class TypedIPort : public IIPort {
public:
  ...
  bool Get(T& data);
  ...
};

template <class T>
class TypedOPort : public IOPort {
public:
  ...
  void Put(T data);
  ...
};
```

Since all the functionality of these typed ports is implemented as inline code, it is possible to use the ports with any datatype, without the need to add any user code. The `Get` function only returns `false` in the case that the port will not provide any more data, and the plugin therefore can terminate its thread. Accordingly the main loop in the user code will usually look like this:

```
while (m_inPort.Get(data)) {
  // process data
  ...
  m_outPort.Put(data);
} // end while
```

5.2.4 Resource Management

As mentioned the data passing framework is only able to pass data by copying it. In cases where large datastructures like images, volume data or 3-D geometries are passed between plugins this would get very inefficient. Thus the framework provides a shared memory pointer class that eliminates all copying wherever possible, without introducing any memory access overheads. The basic idea is similiar to other shared memory pointers, with reference counting, as they are implemented for example in [19]. Every instance keeps track of a shared reference counter and the shared data. If the number of references drops to zero the shared data is deallocated.

They main difference of this implementation to others is that most shared memory pointers explicitly all point to the same object, and accordingly also all modify the same object, this behaviour would be completely unacceptable in a multithreaded environment. Especially plugins that work in parallel on the same token would cause severe run-conditions if they were allowed to modify the same shared data. What is acceptable however, are concurrent read accesses, as long as the data classes do not modify any internal data on a read access (no `mutable` statements etc. allowed).

```
template <class T>
class Token {
public:
  Token();
  Token(const Token<T>& ptr);
  explicit Token(T*& data);
  virtual ~Token();

  bool operator==(const Token<T>& ptr);
  bool operator!=(const Token<T>& ptr);

  const T& operator*();
  const T* operator->();
  T GetWriteAccess();

private:
  T*   m_pData;
  int  m_iCount;
};
```

The implementation of a shared memory pointer the framework provides therefore explicitly forbids any write accesses to the shared data (`operator*` and `operator->` only return constant references). If write accesses are needed for processing, then the data can be retrieved from the shared memory pointer, which either results in copying the data or, if the data actually was not shared (one reference only), in exclusive ownership of the once shared data. Retrieving a write access pointer from the shared memory pointer (`GetWriteAccess()`) invalidates it. This is neccessary since the shared memory pointer is not useful anymore, and keeping the reference around might hinder a quick deletion of the shared data. Similiarly, creating such a shared memory pointer invalidates the provided pointer, to minimze the risk that the user retains a not memory managed pointer to the now shared data.

5.2.5 Datastructures

In the context of a calibration tool, the prior mechanism is applied in order to exchange the plugins' data with a tool's graphical user interface. For this, we identify the typical input and output data of the different calibration steps and provide abstract base class implementations of these fundamental datatypes. By applying the well-known *adapter* design pattern [8], third-party datatypes can be easily adopted to the fundamental type and thus used within the framework.

6 Discussion And Conclusion

The presented approach proved its worth in numerous research activities at our department. As expected, the framework allowed to concentrate on the several foci of the research work and prevents from wasting time with implementation effort that does not contribute to the solutions.

In this paper, we presented a calibration framework based on a modular software architecture with typical calibration steps being represented by software components that can be quickly and easily exchanged and replaced by new approaches in order to create partially or totally new calibration schemes. Our approach enforces modularity due to its design and supports the scientist in writing reusable software components with a problem-specific algorithm being encapsulated in a plugin. The framework is designed to be independent from any graphical user interface. It provides abstract interfaces in order to allow user interaction and enables drawing and component configuration. By this, secondary aspects like UI programming and resource control for the plugin's communication are kept away from the user and are automatized by the framework.

This all speeds-up research activities in the field of camera calibration and allows computer vision scientists to concentrate on their work and allows them to deepen the understanding of camera calibration, which is an important prerequisite for the enhancement of camera calibration algorithms.

Acknowledgements: The research was supported by the *Research Training Group 244 (Graduiertenkolleg)* of the *German Research Foundation (DFG)*.

References

[1] Mateos G. A Camera Calibration Technique Using Targets Of Circular Features. *5th Ibero-America Symposium On Pattern Recognition (SIARP)*, 2000.

[2] Heikkilä J. and Silven O. Calibration Procedure for Short Focal Length Off-the-shelf CCD cameras. *Int. Conference on Pattern Recognition*, 1996.

[3] Intel. Open Source Computer Vision Library. `sourceforge.net/projects/opencvlibrary`.

[4] Heikkilä J. Camera calibration toolbox for Matlab. `www.ee.oulu.fi/~jth/calibr`.

[5] Zhang Z. Microsoft Easy Camera Calibration Tool. `research.microsoft.com/~zhang/Calib`.

[6] Bouguet J.Y. Camera Calibration toolbox for Matlab. `www.vision.caltech.edu/bouguetj`.

[7] Shaw M. and Garland D. *Software Architecture. Perspectives on an Emerging Discipline.* Prentice Hall, 1996.

[8] Gamma E., Helm R., Johnson R., and Vlissides J. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[9] Buschmann F., Meunier R., Sommerlad P., and Stahl M. *Pattern-Oriented Software Architecture, Vol.1 : A System of Patterns.* 1 ed. John Wiley and Sons, 1996.

[10] Chris McGlone E.M. *Manual of Photogrammetry.* 5 ed. ASPRS, 2004.

[11] Tsai R.Y. A versatile camera calibration technique for high-accuracy 3-D machine vision metrology using off-the-shelf TV cameras and lenses. *IEEE Trans. Robot. Automat.*, August 1987, pp. 323 – 344.

[12] Zhang Z. A Flexible New Technique for Camera Calibration. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 22, 2000.

[13] Sturm P. and Maybank S. On Plane-Based Camera Calibration: A General Algorithm, Singularities, Applications. In *IEEE Conference on Computer Vision and Pattern Recognition*, June 1999.

[14] Vogelgsang C., Scholz I., Greiner G., and Niemann H. lgf3 - A Versatile Framework for Vision and Image-Based Rendering Applications. In *Vision, Modeling, and Visualization*, 2002.

[15] Heikkilä J. Geometric Camera Calibration Using Circular Control Points. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 22(10), October 2000, pp. 1066 – 1077.

[16] D'Souza D.F. and Wills A.C. *Objects, Components and Frameworks with UML - the Catalysis Approach.* Addison-Wesley, 1998.

[17] Alexandrescu A. *Modern C++ Design.* Addison-Wesley, 2001.

[18] Cleeland C., Schmidt D.C., and Harrison T.H. External Polymorphism. *In Proc. of the 3rd Pattrn Languages of Programming Conference*, September 1996.

[19] Sutter H. and Alexandrescu A. Boost C++ Libraries. `www.boost.org`.