The CTK-Plugin Framework in CAS Applications - A Tutorial

Introduction

Increasing complexity and security requirements are major issues in the implementation of modern, clinically useful CAS applications. In order to address these problems, frameworks such as IGSTK [1], MITK [2], CISST [3] etc. are used. They offer reusable methods and code skeletons, which simplify development. Hence, complex re-implementation of standard methods, such as visualization [4], registration/segmentation [5] and navigation [1], can be avoided. Plugin frameworks [6] [7] [8] and state machine support [1] [6] also help to handle the level of complexity and increase the reliability.

The increasing scale of a framework may lead to a higher effort in initial training, even though only small parts may be used. In particular, this effects the development of prototypes and tech demos, which are time-sensitive due to their exploratory character. For this, lightweight frameworks with a short familiarization period, like the Common Toolkit [7], are well-suited.

CTK is an open-source toolkit for the biomedical field that provides a service-based plugin framework [7]. The design of the CTK framework enables the creation of modular CAS applications. So far, there have been only a few comprehensive tutorials of the CTK toolkit on the internet. Although CTK is used in the MITK framework [2], there are no open-source sample applications. This makes efficient development even more difficult.

In this tutorial, we take this problem into consideration and provide an introduction to the CTK plugin framework, minimal working examples of its use, as well as an open-source demo application that can be used as a template for standard navigation. In addition, the modules/plugins can be reused easily due to service/event-based loose coupling.

CTK-Overview

This section provides a brief overview of the Common Toolkit, as well as an introduction to the plugin framework and its use in the context of computer-assisted surgery. Thereby obtained knowledge builds the basis of the demo application that is presented in Section 3.

The CTK framework consists of a comprehensive DICOM loader [9], a DICOM application hosting system, a collection of Qt-Widgets, which are related to biomedical applications, a command line interface and a plugin framework. The individual elements are based on C++ and Qt-Libraries [6]. In the following, only the plugin framework is discussed since it allows a modular design of CAS applications and the demo application is built upon. Other parts of the framework can still be used without restriction to extend the demo application.

CTK plugins are modular units within an application that represent conventional-separated functions such as communication to a tracker, the preparation of GUI elements or reading data. Plugins are compiled separately and loaded by the plugin framework at runtime. Plugins are initialized by a plugin activator class, for parallel execution, a GUI- or background-thread needs to be started. Each plugin contains a pointer to

the plugin context, which provides an access point to all loaded plugins and services. Thus, the main program starts only the plugins and initializes a skeleton for the graphical user interface if required. The communication between plugins is provided through services. Services are objects that are registered by a plugin at the CTK service registry and are derived from the CTK service class. The CTK service registry serves as a central point for plugins that either provide services or access services. A service is uniquely defined by its interface (C++ class which usually contains only virtual methods) and its properties (C++ hash map which contains names and states of properties as standard data-types). Therefore, the interface must be included by the plugin that provides the service, as well as the plugins that access the service. A plugin provides a service by registering itself to the CTK service registry. Other plugins can hand over a tracker object (instance of a class that implements the abstract class *ctkServiceTracker*) by specifying its interfaces and its properties. The tracker object will be notified after the service of the CTK service registry is added, removed or the properties of the service are changed. The notification takes place synchronously, which is why plugins that propagates a change to the service object should not have locks. As an alternative, a plugin or the main program can access a service object directly from the service registry but a manipulation of the object by multiple plugins should be avoided due to memory safety. Figure 1a shows the service-based communication between plugins. Plugin A provides a service object. Plugin B accesses a service tracker object that is notified after a change. Plugin C and the main program access the service object directly.

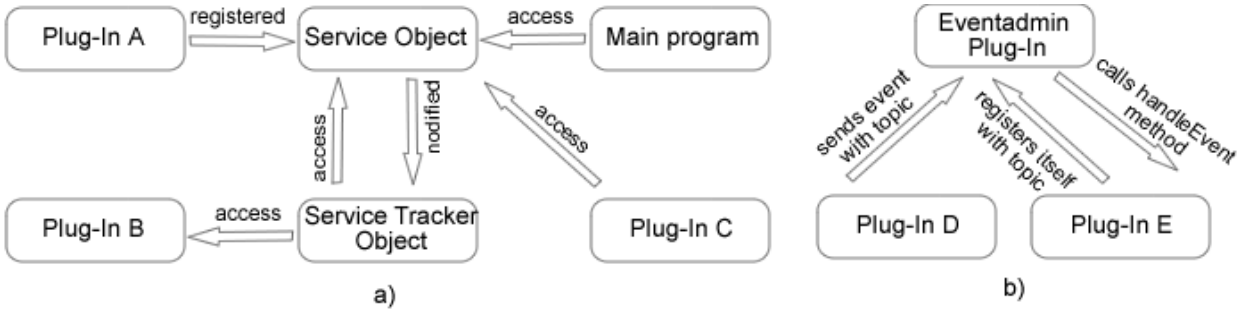


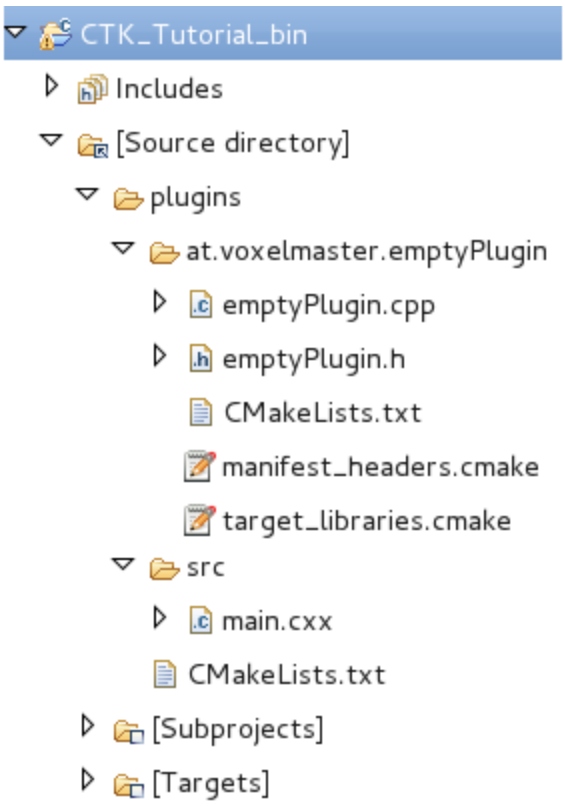
Figure 1: Types of service-based (a) and event-based (b) communication

Events are managed by CTKs own Eventadmin plugin, which provides the service ctkEventAdmin for sending and receiving events. This service allows plugins (consisting of properties analog to services) to publish events under a specific topic (of type QString). To receive events for a specific topic, plugins have to implement the interface *ctkEventHandler* and register itself at the service registry. Events can be sent synchronously or asynchronously. Figure 1b illustrates the relationship between the Eventadmin plugin and other plugins. Plugin D sends an object under a particular topic, the Eventadmin plugin calls the handleEvent method of plugin E (after it has been registered previously under the topic).

In the context of computer-assisted surgery, service-based communication is especially suitable for exchanging data that cannot be described by standard C++ data types, because they are not supported by events – for example the transfer of DICOM [9] data between plugins or from Qt GUI elements to the main program. In contrast, asynchronous transmitted events are used for small, frequently updated data such as tracker coordinates for real-time navigation.

The service/event-based nature of data exchange provides loose coupling between individual plugins, and therefore a modular design of CAS applications as presented in Section 3. Once created plugins can be reused easily for other programs. Each plugin acts as a standalone module whose status is defined by defined access points (GUI inputs, service updates, and events). Consequently, plugins can include their separate state-machine. Since CTK is based on Qt [6], it is appropriate to use Qts own state-machine implementation. Nevertheless, plugins can include any library, so the use of other implementations, such as IGSTK [1] is also possible.

Creating a new CTK Plugin



In this section we discuss in detail how to create and manage a project and a plugin. In a first step to create a project you need a top-level *CMakeList* (*src/CMakeList.txt*) to be able to compile the project and its plugins with CMake cross platform [11]. The possible project layout is shown in the adjacent figure. The project contains one application "*CTK_Tutorial_bin*" and one plugin *at.voxelmaster.emptyPlugin*.

The top-level *CMakeLists.txt* file is in charge of finding installed CTK and Qt4 libraries and manage the plugin creation. CTK Plugin integration [12] requires CTK and Qt4 libraries. To find CTK and Qt,4 you need to add following CMake commands in our *CmakeList.txt*:

```
FIND_PACKAGE(CTK)and FIND_PACKAGE(Qt4)
```

To be able to take advantage of CTKs sophisticated dependency checking system inside your own build system, you need to write a small CMake macro called GetMyTargetLibraries. It is used as a callback inside CTKs own CMake macros and functions to distinguish between targets being build in your project, and targets external to your project (e.g. targets coming from CTK). It works by taking a list of target names and filtering names that belong to your own project using regular expressions. Between *_tmp_list* and *OUTPUT_VARIABLE* the use of regular expressions is not limited. Note that underscores ("_") will be replaced by points (".").

```
macro(GetMyTargetLibraries all_target_libraries varname)
set(re_ctkplugin "^at_voxelmaster_[a-zA-Z0-9_]+$")
set(_tmp_list)
list(APPEND _tmp_list ${all_target_libraries})
ctkMacroListFilter(_tmp_list re_ctkplugin OUTPUT_VARIABLE ${varname})
endmacro()
```

Possible example plugin names for this macro are:

```
at.voxelmaster.testname
at.voxelmaster.newPlugin
at.voxelmaster.pluginName
at.voxelmaster.asdfjkasdf
```

In a next step you should create a list containing one entry for each plugin in the plugins directory with the CMake command *SET*. Each entry consists of the directory name containing the plugin followed by a

double point (":") and the default build option value (*ON/OFF*). This is the main macro to set up your CTK plugins inside your own CMake project.

```
SET(plugins
plugins/at.voxelmaster.emptyPlugin:ON
)
```

ctkMacroSetupPlugins macro takes care of validating the current set of plugin build options, enables and/or checks required plugins and handles all aspects of plugin dependencies. Additionally, it calls *add_subdirectory()* on each given plugin.

```
ctkMacroSetupPlugins(${plugins}
)
```

To integrate our application "*CTK_Tutorial*" in CmakeList.txt, which needs to call methods from the CTK Plugin Framework library, include the directories by "INCLUDE_DIRECTORIES" and link the application to the CTKPluginFramework [13]. Therefore, you need following CMake commands:

```
PROJECT(CTK_Tutorial)

SET(CTK_Tutorial_SRCS
src/main.cxx
)

INCLUDE_DIRECTORIES(
${CTK_Tutorial_SOURCE_DIR}
${CTK_Tutorial_BINARY_DIR}
${CTK_Tutorial_SOURCE_DIR}/src
${CMAKE_CURRENT_SOURCE_DIR}
${CMAKE_CURRENT_BINARY_DIR}
)
SET(my_libs
CTKPluginFramework
)

ADD_EXECUTABLE(CTK_Tutorial ${CTK_Tutorial_SRCS})
TARGET_LINK_LIBRARIES(CTK_Tutorial {QT_LIBRARIES} ${CTK_LIBRARIES} ${my_libs}
)
```

Hint 1: Find the example under this link:
https://sourceforge.net/p/ctkcas/code/ci/master/tree/CTK_Tutorial/CMakeLists.txt

To create your own plugin in your application, you need to create a plugin-directory "plugins" in your main project. In this directory you can place your plugin directories like for example *at.voxelmaster.emptyPlugin.* The Plugin should contains at least following files: *pluginName.cpp, pluginName.h, CmakeList.txt, manifest_headers.cmake* and *target_libraries.cmake*. To generate this plugin files, you can use the CTKs plugin generator application *CTKPluginGenerator* located in *../yourCTKBuildFolder/CTK-build/bin* or copy the "emptyPlugin" plugin

(https://sourceforge.net/p/ctkcas/code/ci/master/tree/CTK_Tutorial/plugins/at.voxelmaster.emptyPlugin/)

External plugins (plugins without a main application which can be used in different projects) are build exactly the same way as plugins within the CTK project itself. Hence the CmakeList of your plugin should have following CMake commands to integrate your plugin to your application (this is simply an example CMakeLists file):

```
PROJECT(at_voxelmaster_myNewPlugin)

SET(PLUGIN_export_directive "at_voxelmaster_myNewPlugin_EXPORT")

SET(PLUGIN_SRCS
emptyPlugin.cpp
)

SET(PLUGIN_MOC_SRCS
emptyPlugin.h
)

ctkFunctionGetTargetLibraries(PLUGIN_target_libraries)

ctkMacroBuildPlugin(
NAME ${PROJECT_NAME}
EXPORT_DIRECTIVE ${PLUGIN_export_directive}
SRCS ${PLUGIN_SRCS}
MOC_SRCS ${PLUGIN_MOC_SRCS}
UI_FORMS ${PLUGIN_UI_FORMS}
RESOURCES ${PLUGIN_resources}
TARGET_LIBRARIES ${PLUGIN_target_libraries}
)
```

Hint 2: Additional help for project and plugin creation can be found at:
http://www.commontk.org/index.php/Documentation/CTK_Plugin_Framework:_Setting_up_a_project

Hint 3: For some reason it is possible that you get an Segmentation fault or similar exceptions after you have changed the CMakeLists.txt of a plugin and rebuild it. In this case, delete everything in the build directory, make the directory with CMake and build the application again.

Hint 4: You can add additional libraries after `TARGET_LIBRARIES ${PLUGIN_target_libraries}`. Additionally we have discovered strange behavior after linking to a shared library (“first library”) which links itself to another shared library (“second library§). It can’t find the second shared library at runtime. Therefore you have to use the “QLibrary” class:

```
QLibrary myLib("first library.so");
library.setLoadHints(QLibrary::ExportExternalSymbolsHint);
library.load();
typedef void (*MyPrototype)();
MyPrototype myFunction = (MyPrototype) myLib.resolve("mysymbol");
```

```
if (myFunction)
    myFunction();
```

In the last step you need to find and start your created plugin in your main application.
Before you can load a plugin, you need to add the plugin directory path to the ctkPluginFrameworkLauncher

```
ctkPluginFrameworkLauncher::addSearchPath("plugins",true);
```

Afterwards you can start the plugin with with

```
ctkPluginFrameworkLauncher::start("name of plugin");
```

Plugin Communication

According to general inter-plugin communication mechanism, plugins can communicate with each other [14]. The communication conforms to the popular publish/subscribe paradigm and can be performed in a synchronous or asynchronous manner. The communication between plugins can be done in two modes: event-based or service-based.

The main components in a publish/subscribe communication are:
Event Sender: Sends events or messages related to a specific topic.
Event Receiver (or Handler): Expresses interest in one or more topics and receives all the messages belonging to such topics.

Event-based Communication

By event based communication [14], events are composed of two attributes:
A topic defining the nature of the event and a set of properties describing the event.
In the following, you can find an event with properties created and send using *eventAdmin* to the receiver.

// sendEventToReceiver located in
//plugins/at.voxelmaster.eventBasedCommunicationSenderPlugin/eventBasedCommunicationSenderPlugin.cpp

```
ctkServiceReference reference=
context->getServiceReference<ctkEventAdmin>();

ctkEventAdmin* eventAdmin=
context->getService<ctkEventAdmin>(reference);
ctkDictionary properties;
properties["sendText"]= sendText;
```

```
properties[sendText];
ctkEvent reportGeneratedEvent("SendEventUpdated", properties);
eventAdmin->sendEvent(reportGeneratedEvent); //for synchronous sending
eventAdmin->postEvent(reportGeneratedEvent); //for asynchronous sending
```

To receive event notifications, the receiver creates an event handler (an event handler is a class implementing the ctkEventHandler interface which is registered as a service object) and register it as a service at the ctk service registry. EVENT_TOPIC property describes the list of topics in which the event handler is interested.
//receiveEventFromSender located in //plugins/at.voxelmaster.eventBasedCommunicationReceiverPlugin/eventBasedCommunicationReceiverPlugin.cpp

```
ctkDictionary props;
props[ctkEventConstants::EVENT_TOPIC] = "SendEventUpdated";

context->registerService<ctkEventHandler>(instance, props);

receiverText= event.getProperty("sendText").toString();
```

Hint 5: Before you start the sender plugin, you should start the receiver plugin in both communication modes (event- or service-based) of plugins . Otherwise, the receiver plugin can't receive the events that the sender sent (before the receiver was active).

Hint 6: Find additional information for event-based communication at http://www.commontk.org/docs/html/PluginFramework_EventAdmin_Page.html

Service-based Communication

To create service-based communication, the sender needs to initialize an interface and register itself to the service with its properties.

//sendServiceToReceiver located in //plugins/at.voxelmaster.serviceBasedCommunicationSenderPlugin/serviceBasedCommunicationSenderPlugin.cpp

```
ctkDictionary properties;

interfaceInstance = new serviceInterface();
interfaceInstance->sendText= "Send Service Text ";
```

```
serviceRegistrationReference =
context->registerService<serviceInterface>(interfaceInstance,
properties);
properties["serviceEventID"]=0;
serviceRegistrationReference.setProperties(properties);
```

To set the service tracker as listener to all “serviceInterace” service, it can be initialized in this way:
//initializeServiceTracker located sendServiceToReceiver.cxx

```
serviceTracker* sTracker = new serviceTracker(context,this);

ctkServiceTracker<serviceInterface*,serviceInterface*>* tracker = new
ctkServiceTracker<serviceInterface*,serviceInterface*>(context,sTracker)
;
tracker->open();
```

The service tracker class is an instance of a class that implements the abstract class *ctkServiceTracker* by specifying its interfaces and its properties. The tracker object *serviceBasedCommunicationReceiverPluginObject* can be accessed by the service tracker. To receives notification, after the service is added, removed or modified at the CTK service registry.

//serviceTracker located in
//plugins/at.voxelmaster.serviceBasedCommunicationReceiverPlugin/serviceTracker.h

```
serviceInterface* addingService (const ctkServiceReference &reference)
{

modifiedService(reference,context->getService<serviceInterface>(referenc
e));
      return context->getService<serviceInterface>(reference);
}

void modifiedService (const ctkServiceReference &reference,
serviceInterface* service)
{
      serviceBasedCommunicationReceiverPluginObject->receiveServiceFromS
ender(service->sendText);
      qDebug()<< "Receive Service From Sender \n"<< sendText;
}
```

Hint 7: Usually, the “modifiedService” method is used for communication. To trigger a modification, just call the “setProperty” method from the service registration reference.

```
      serviceRegistrationReference =
      context->registerService<serviceInterface>(interfaceInstance,
```

```
        properties);
        serviceRegistrationReference.setProperties(properties);
```

```
//serviceInterface located in
//plugins/at.voxelmaster.serviceBasedCommunicationSenderPlugin/serviceInterface.h
//and
//plugins/at.voxelmaster.serviceBasedCommunicationReceiverPlugin/serviceInterface.h
class serviceInterface : public QObject
{
        Q_OBJECT
public:
        QString sendText;
};
```

CAS Demo Application

A demo application has been developed to validate the application of CTK in computer-assisted surgery, as well as to facilitate the familiarization with the framework. It provides plugins for loading DICOM [9] images, integration of tracking data, marker-based registration of CT data, navigation and the visualization of (stereo-) video streams.

The structure of the application can be divided into three subgroups:

Main program: The main program handles the loading of plugins and provides a basic Qt GUI skeleton (Figure 2a). The GUI is loaded dynamically from service objects that were registered by visualization plugins. The right frame contains the visualization of the currently selected plugin, the left one the controls for all plugins. By clicking on the plugin title an active plugin can be selected.

Visualization-Plugins: Visualization plugins provide GUI elements to the main program by services. These consist of a visualization element (VE, Figure 2a, red), which handles the actual graphical representation, and a control element (KE, Figure 2a, blue). For the demo applications three visualization plugins were created:

Dicom Loader: The KE is used for loading CT DICOM data [9]. In the VE, data is presented in a coronal sagittal, axial and a combined. Therefore, the IGSTK *DicomReader* [1] was used, the GUI elements were created with the QT Designer [6]. A pointer on the DICOM data is stored in a service object, which notifies the navigation plugin after a modification.

Navigation: The navigation plugin implements the marker-based registration of DICOM [9] data, its visualization and navigation. The visualization is performed analogously to the DICOM Loader plugin (Figure2a). The tracking data is received via asynchronous events from the tracker-communications plugin. Due to loose coupling, it is possible to replace the tracker-communication plugin at will.

Stereo-Microscope-Visualization: Intended as a tech demo, the microscope plugin serves no benefit for the navigation. The KE contains no control elements, the VE shows two video streams of

a stereo-camera setups. The stream is received as a GUI element by a service of the microscope communication plugin. In order to enable stereo-microscope navigation, it is possible to receive events from the tracker communications plugin, and reuse the navigation GUI elements. For simplicity, this feature has not been implemented.

Communication-Plugins: Communication plugins provide information for visualization plugins by services and events. They serve as an interface to the hardware. These plugins can be replaced any time (e.g. to communicate with other trackers) as long as they implement the same service interface or send the same events.

Tracker-Communication: The Tracker-communication plugin establishes a connection to a specific tracker. In the demo application, IGSTK [1] marker coordinates, from an Optotrak Certus Tracker (NDI, Germany), are read sent by events (via the Eventadmin plugin) to the Navigation plugin. To integrate other trackers (e.g. by IGSTK [1] or OpenIGTLink [10]), it is sufficient to create a new Tracker-Communication plugin and send events to the Eventadmin plugin.

Microscope-Communication: This plugin implements the connection to a stereo-camera setup, which represents the cameras of a stereo-microscope. The video-stream is provided as Qt GUI element by a service to the Stereo-Microscope-Visualization plugin. This plugin provides all parameters of the microscope (zoom, focus, position, etc.).

Figure 2b shows the communication between the main program, Visualization plugins, Communication plugins and the Eventadmin plugin, as well as their services. The main program starts all plugins. This can be done in any order, because each plugin has its own state machine. As long as required services haven't been initialized, each plugin waits in a "wait" status. The main program loads the GUI elements in the GUI skeleton after starting the visualization plugins. Then, DICOM data can be loaded by the DICOM loader, which will notify the service object tracker of the Navigation plugin. If the Navigation plugin receives tracker coordinate events (i.e. if the tracker communication has been started successfully), registration and subsequent navigation can be initialized. Independent from these processes, the Stereo-Microscope-Visualization plugin renders a video stream as a GUI element, once the microscope-communication has been loaded.

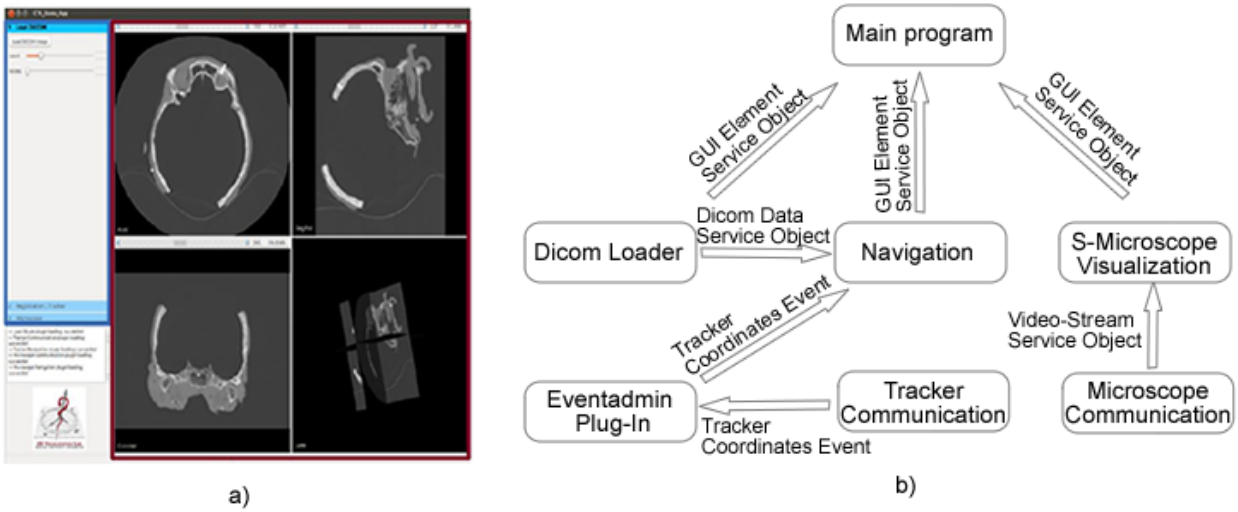


Figure 2: Screenshot of demo application with marked control element (blue) and visualization element (red), as well as a schematic representation of the plug-in communication

Discussion

Compared to comprehensive toolkits in the CAS area, the Common Toolkit provides a short familiarization period. For example, the use of MITK [2], although it is based on CTK, is associated with high effort for its extensive range of functions. In contrast, the CTK plugin framework enables effective development of prototypes and demos. Plugins may be easily replaced or reused by service-based communication. Any additionally required functions (e.g. segmentation/registration, state machines) can be integrated by using well-known libraries such as ITK [5] or QT [6].

The introduced demo application represents a simple example of standard navigation using the Common Toolkit. The implementation demonstrates, that CTK can be used in the field of CAS. Due to the public source code, the demo offers a help for the development with the CTK plugin framework.

Summary

In this article, we take increasing complexity and security requirements of CAS applications into consideration and provide an introduction to the CTK plugin framework.

This allows the creation of modular applications and the integration of external libraries to extend the functionality. Therefore, the usage of extensive frameworks can be avoided. In Section 2, the creation of plugins has been described, as well as their communication by services between plugins.
A demo application, which is described in Section 3, has been developed to validate the application of CTK in computer-assisted surgery, as well as to facilitate the familiarization with the framework.

Additional Information

The source code of the demo application, minimal examples of plugins and a tutorial are freely available on https://sourceforge.net/projects/ctkcas/ and http://www.wopsys.com

If you use this tutorial or application for your work, please cite our paper:
"The Common Toolkit CTK in computer-assisted surgery - A demo application", F. Ganglberger, Y. Özbek, W. Freysinger, CURAC 12th Annual Conference, 2013

References

[1] Cleary K, Cheng P, IGSTK: The Book, Insight (2007)
[2] Nolden M, Zelzer S, Seitel A, Wald D, Müller M, Franz AM, Maleike D, Fangerau M, Baumhauer M, Maier-Hein L, Maier-Hein KH, Meinzer HP and Wolf I, The Medical Imaging Interaction Toolkit: challenges and advances, International Journal of Computer Assisted Radiology and Surgery (2013)
[3] Deguet A, Kumar R, Taylor R, Kazanzides P, The CISST libraries for computer assisted intervention systems,Insight 1-8 (2008)
[4] Schroeder WJ, Martin K, Lorensen WE, The Visualization Toolkit: An Object-Oriented Approach to 3D Graphics, Third edition, ISBN 1-930934-07-6, Kitware, Inc. (formerly Prentice-Hall) (2003)
[5] L. Ibanez and W. Schroeder. The ITK Software Guide. Kitware, Inc. ISBN 1-930934-10-6, http://www.itk.org/ItkSoftwareGuide.pdf (2003)
[6] Dalheimer M, Programming with QT, Second edition, ISBN 978-0-596-00064-6, O'Reilly Media (2002)
[7] http://www.commontk.org
[8] Pieper S, Halle M, Kikinis R, 3D SLICER. Proceedings of the 1st IEEE International Symposium on Biomedical Imaging: From Nano to Macro 632-635 (2004)
[9] Bidgood WD, Horii, SC Introduction to the ACR-NEMA DICOM standard. RadioGraphics 12, 345-355 (1992)
[10] Tokuda J, Fischer GS, Papademetris X, Yaniv Z, Ibanez L, Cheng P, Liu H, et al., OpenIGTLink: an open network protocol for image-guided therapy environment. The international journal of medical robotics computer assisted surgery MRCAS 5, 423-434 (2009)
[11] CMake 2.8.4 Documentation, http://www.cmake.org/cmake/help/v2.8.4/cmake.html
[12] Documentation/CTK Plugin Framework: Setting up a project, http://www.commontk.org/index.php/Documentation/CTK_Plugin_Framework:_Setting_up_a_project, 18 November 2011
[13] Documentation/CTK Plugin Framework: Introduction, http://www.commontk.org/index.php/Documentation/CTK_Plugin_Framework:_Introduction, 31 January 2011
[14] CTK Event Admin Service, http://www.commontk.org/docs/html/PluginFramework_EventAdmin_Page.html