


MITK-OpenIGTLink for combining open-source toolkits in real-time computer-assisted interventions

Martin Klemm¹  · Thomas Kirchner² · Janek Gröhl² · Dominique Cheray² · Marco Nolden² · Alexander Seitel² · Harald Hoppe¹ · Lena Maier-Hein² · Alfred M. Franz²

Received: 17 May 2016 / Accepted: 8 September 2016 / Published online: 29 September 2016
© CARS 2016

Abstract

Purpose Due to rapid developments in the research areas of medical imaging, medical image processing and robotics, computer-assisted interventions (CAI) are becoming an integral part of modern patient care. From a software engineering point of view, these systems are highly complex and research can benefit greatly from reusing software components. This is supported by a number of open-source toolkits for medical imaging and CAI such as the medical imaging interaction toolkit (MITK), the public software library for ultrasound imaging research (PLUS) and 3D Slicer. An independent inter-toolkit communication such as the open image-guided therapy link (OpenIGTLink) can be used to combine the advantages of these toolkits and enable an easier realization of a clinical CAI workflow.

Methods MITK-OpenIGTLink is presented as a network interface within MITK that allows easy to use, asynchronous two-way messaging between MITK and clinical devices or other toolkits. Performance and interoperability tests with MITK-OpenIGTLink were carried out considering the whole CAI workflow from data acquisition over processing to visualization.

Results We present how MITK-OpenIGTLink can be applied in different usage scenarios. In performance tests, tracking data were transmitted with a frame rate of up to 1000 Hz and a latency of 2.81 ms. Transmission of images

with typical ultrasound (US) and greyscale high-definition (HD) resolutions of 640×480 and 1920×1080 is possible at up to 512 and 128 Hz, respectively.

Conclusion With the integration of OpenIGTLink into MITK, this protocol is now supported by all established open-source toolkits in the field. This eases interoperability between MITK and toolkits such as PLUS or 3D Slicer and facilitates cross-toolkit research collaborations. MITK and its submodule MITK-OpenIGTLink are provided open source under a BSD-style licence (<http://mitk.org>).

Keywords Computer-assisted interventions · Image-guided therapy · Ultrasound · Interoperability · OpenIGTLink · MITK

Introduction

Technological advances have opened many new doors in medical imaging, medical image processing and robotics research. Computer assistance is no longer restricted to diagnostics and surgical planning but has been expanded to surgical and radiological interventions. From a software engineering point of view, these extensions bring new requirements, e.g. support for complex clinical workflows, integration of different kinds of hardware and data, real-time data processing, and the need for high levels of robustness. In this context, research can greatly benefit from reusing software components and the community has gradually become aware of the importance of publishing source code to ensure reproducibility of results [11]. As a consequence, open-source software for medical imaging applications has gained popularity in the scientific community in recent years. Instead of reinventing the wheel, researchers can now concentrate on their preferred research and other groups can easily reproduce their results [5].

✉ Martin Klemm
martin.klemm@hs-offenburg.de

¹ Laboratory for Computer-Assisted Medicine, Department of Electrical Engineering and Information Technology, Offenburg University, Badstraße 24, 77652 Offenburg, Germany

² German Cancer Research Center (DKFZ), Im Neuenheimer Feld 280, 69120 Heidelberg, Germany

A number of open-source toolkits have been published in recent years, all with their degree of differentiation. Some are highly specialized in a certain field, e.g. PLUS [14] and the medical ultrasound imaging and intervention collaboration (MUSiiC) [26]. Others provide necessary infrastructure, e.g. the insight segmentation and registration toolkit (ITK) [18], OpenIGTLink [29], and yet others combine the functionality and the infrastructure into one application, e.g. 3D Slicer [23] and MITK [21, 30].

Starting off as stand-alone software components, the mentioned toolkits developed towards intra-operable solutions especially with the introduction of OpenIGTLink.¹ A good example is the combination of PLUS and 3D Slicer [14]. While PLUS is acquiring and processing the data, 3D Slicer is used to visualize it. OpenIGTLink is an open-source network protocol originally developed for IGT environments [29] and is the de facto standard in the medical research field. It has proved its functionality in many different applications, such as MRI-guided robotic prostate interventions for communication between scanner, workstation and robot [29] or neurosurgery for communication between a commercial navigation system and 3D Slicer [29]. A wide range of toolkits such as 3D Slicer [8], IGSTK [22], MUSiiC [12], MeVis-Lab [7], PLUS [14] and NifTK [4] already support this protocol.

MITK provides dedicated modules for IGT (MITK-IGT, [9, 20]), range imaging (MITK-TOF, [25]) and ultrasound (MITK-US, [16]) and is used as a basic toolkit by other open- and closed-source software such as NifTK [4]. These modules have been used for a wide variety of CAI applications including a needle-based navigation system for CT-guided radiofrequency ablation of the liver [15], mobile augmented reality (AR) systems for nephrolithotomy and forensic medicine [13, 19], a navigation system based on a compact electromagnetic field generator integrated with an US probe [17] and markerless navigation for percutaneous needle insertions [24].

We introduce the module MITK-OpenIGTLink that extends MITK with a network interface to allow cross-application, cross-toolkit and cross-platform communication within CAI systems. This module implements the OpenIGTLink protocol version 2, including full support for the querying mechanism: Data can be requested and replies are sent if the data is available. We highlight different usage scenarios and evaluate the modules performance with respect to frame rate and latency.

Methods

MITK-OpenIGTLink is implemented as a module within the MITK toolkit providing standardized independent commu-

nication across toolkits and medical devices. The implementation complies with the MITK software process [21] using a continuous integration,² a database for tracking changes³ and a dedicated release process with manual tests at the application level.

This section starts with the requirement analysis, followed by the architectural overview, and finishes with the methodology for validating the developed architecture.

Requirements

The following requirements were identified for MITK-OpenIGTLink as a communication layer for MITK:

Extensibility The new module must easily integrate into the pipeline structure of MITK and its modules to allow for interchangeability of e.g. data processing methods. Using new and customized OpenIGTLink message types must be possible.

Flexibility The data transmission and its processing inside the MITK pipelines have to be connected in a flexible way in order to easily exchange the processing steps.

Performance High frame rates and low latency are necessary for real-time applications. The US image data transfer shall run with 30Hz since typical real-time US devices run with such a frame rate. Tracking data shall be transmitted with up to 1000Hz in order to cover robotic applications [29]. The latency for tracking data caused by MITK shall be one order of magnitude smaller than the latency caused by the tracking device itself which is the time from when the tracking device is sampled until tracking data is available in the PC.

Application-wide availability The module should provide an application-wide availability of all filters and devices that are necessary for an OpenIGTLink connection. This makes multiple configurations of the same component unnecessary.

Portability MITK-OpenIGTLink should be implemented in C++ and run on Windows, Linux and Mac.

Robustness Messages must not be discarded as long as the user wants to process all of them. It must be configurable to keep only the latest message or all.

Usability The module must be easy to integrate for developers and the resulting application or plugin should be easy to use for the end-user.

Architecture

An architectural overview of MITK-OpenIGTLink is given in Figs. 1 and 2. MITK-OpenIGTLink is structured in the following three layers:

² See <http://cdash.mitk.org/>.

³ See <http://bugs.mitk.org/>.

¹ See <http://openigtlink.org/>.

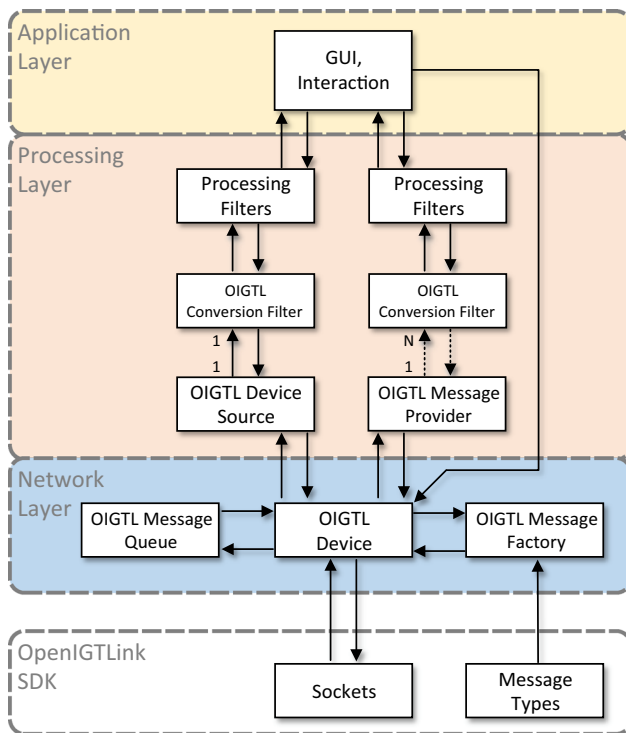


Fig. 1 The MITK-OpenIGTLink layered architecture. The network layer wraps the OpenIGTLink classes and manages the communication. The processing layer connects the OpenIGTLink device to the processing pipeline. The OIGTLDeviceSource is used for 1 to 1 connections, while the OIGTLMessageProvider can handle several conversion filters and supports streams. The OIGTLDeviceSource is statically coupled with the conversion filters, while the OIGTLMessageProvider is only loosely coupled with them. Both versions can run in parallel, but normally only one of them is used

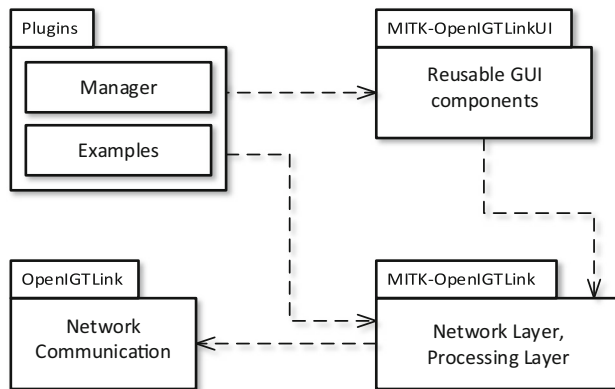


Fig. 2 Module structure of MITK-OpenIGTLink

- Network layer: handles the communication with the OpenIGTLink SDK
- Processing layer: handles the processing of incoming and outgoing messages
- Application layer: handles the management of connections

Figure 1 shows the classes used to connect MITK pipelines to other OpenIGTLink devices. A pipeline is a concatenation

of processing filters, where each filter does a particular job and sends the result to the next stage of the pipeline. This approach is based on ITK [10]. In ITK, pipelines are implemented as pull pipelines, meaning that the processing is triggered on demand by any filter inside the pipeline (in general the last one). This stands in contrast to a push pipeline in which the processing is started from the first pipeline component.

Figure 2 shows the module structure. The source code for the network and the processing layer is contained in MITK-OpenIGTLink which depends on OpenIGTLink. MITK-OpenIGTLinkUI provides reusable GUI components. On top of all these components, there is a manager plugin to configure the connection. Additionally, different examples are provided.

OpenIGTLink SDK

The OpenIGTLink SDK⁴ consists of two parts: a low-level C library and a higher-level C++ library. OpenIGTLink is designed to run on top of the transmission control protocol (TCP) stack. As an alternative to TCP, the user datagram protocol (UDP) is supported. There is no session management, which is the reason why an OpenIGTLink message contains all necessary information (data type, etc.) for the receiver to interpret it. This simplifies the protocol but also increases the overhead of each message. Besides the standard messages for exchanging tracking data, images, control and monitor information, custom message types can be defined. The protocol version 2 also specifies a querying mechanism used to request single messages or streams of a given message type.

Network layer

The network layer interfaces with the OpenIGTLink protocol and encapsulates its implementation as provided in the SDK. It contains all classes for establishing and managing OpenIGTLink communications and messages.

Client-server architecture The central class in the network layer is the OIGTLDevice. An OIGTLDevice is responsible for the communication with other toolkits or devices supporting OpenIGTLink. For sending and receiving messages, it uses the OpenIGTLink sockets. The device runs three different threads to continuously check for new connections, receive messages and send messages. This allows the server to accept new client connections while it is already communicating with other clients. These threads are continuously put to sleep for 1 ms to reduce the CPU usage of the threads.

The OpenIGTLink client-server architecture is realized by two specializations of the OIGTLDevice: OIGTLClient

⁴ See <http://openigtlink.org/library.html>.

and `OIGTLServer`. You can decide to run a client or a server or both at the same time. This stands in contrast to other implementations as in `PLUS` in which there is only a server available.

In `OpenIGTLink`, a server can connect to an arbitrary number of clients but each client only to one server. Server and clients are classified by their role during the connection establishment and not during the connection itself. The client is the device that requests the connection with the server. During the connection, both devices (client and server) can request or send data.

Messages Incoming and outgoing messages are stored in an `OIGTLMessageQueue`. These queues can be configured in two different modes. Depending on the application, it might be necessary to process all incoming messages or only the latest one. The outgoing message buffer is used when messages are created faster than they can be sent.

An additional command queue stores the incoming commands. In the standard there are four different kinds of commands defined that can be used for the query functionality:

`GET_MSGTYPE`: Used to request a single message with the type `MSGTYPE`. The device answers with a message of type `MSGTYPE` or `RTS_MSGTYPE`.

`RTS_MSGTYPE`: Used to inform the requesting device that `MSGTYPE` messages are not available.

`STT_MSGTYPE`: Used to request a stream of messages with the type `MSGTYPE`. The device answers with a stream or an `RTS` message.

`STP_MSGTYPE`: Used to stop the stream of type `MSGTYPE`. The device has to stop the stream and answers with an `RTS` message.

The query mechanism that results from these command messages can be used for a two-way communication, e.g. to send control commands to an US machine and receive image data. This query mechanism is not used by all existing `OpenIGTLink` implementations. In these cases, the stream automatically starts as soon as client and server are connected to each other. To be compatible with such implementations, it is possible to configure `MITK-OpenIGTLink` to also send messages upon connection. The commands are received and sent from the `OIGTLDevice`; however, the handling of these commands is done in the processing layer.

Custom message types can be created and have to be added to the `OIGTLMessageFactory` before they can be used. This can be done at compile and at run time. In order to add custom types to the factory, two things are necessary:

1. A correct name according to the protocol: A data message is called by its data, e.g. `CUSTOMDATA`,

whereas a command starts with the command type, e.g. `GET_CUSTOMDATA`.

2. A method that allocates this message.

Standard types are automatically added to the factory at compile time.

The factory is registered as a Micro Service and therefore is available system-wide. C++ Micro Services⁵ are a low-level mechanism for a service-oriented modular system. The goal of this architecture is to hide complex tasks behind a simple service interface and to make it available to other components during run time. After registering a service in one module, it is available to other modules. The selection of services is based on properties and priorities and is managed by the so-called module context. In this way, functionality can be easily extended by registering a new service with a higher priority [21].

Processing layer

The processing layer holds the components establishing the connection between ITK style pipeline filters and the `OpenIGTLink` devices in the Network Layer. This conversion is important since existing `MITK-IGT` components are mainly implemented as such filters. The first step, a so-called conversion filter, converts either `MITK` data types into `OpenIGTLink` messages or vice versa. Custom conversion filters can be easily integrated; however, for the most common data types used in `MITK`, conversion filters are already available. In the second step, the message has to be sent or received. There are two different configurations to achieve this:

1. *Using the `OIGTLDeviceSource`*

The `OIGTLDeviceSource` manages an `OIGTLDevice` and can be statically connected to a conversion filter (left side of Fig. 1). This configuration is used in applications where all components are known at compile time.

2. *Using the `OIGTLMessageProvider`*

The `OIGTLMessageProvider` inherits from the `OIGTLDeviceSource` and therefore also manages an `OIGTLDevice`. The difference is that the provider is only loosely connected to the conversion filter. At start-up, it is not connected to any filter. It waits for an incoming command message and subsequently checks if it can provide the requested data. To find these data, the provider looks for conversion filters that are registered as a Micro Service and that are able to provide the appropriate data output. Once it finds an appropriate filter, it connects to it automatically and sends the data. It supports single `GET`

⁵ See <http://cppmicroservices.org/>.

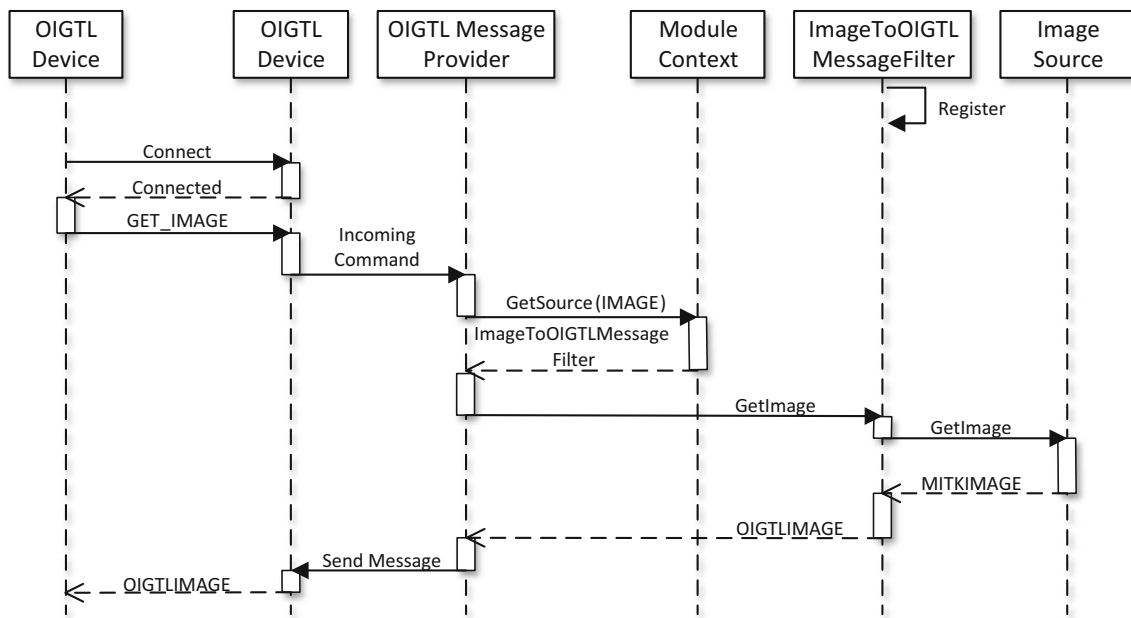


Fig. 3 Sequence diagram for the query of an image. An OIGTL device is requesting an image, the message provider looks for available image sources inside the module context, connects itself to the fitting conver-

sion filter (ImageToOIGTLMessageFilter) and requests an image, and subsequently, the image is provided, converted and sent to the requesting device

commands as well as streaming commands. This configuration is used in applications in which the set-up is defined at runtime. Figure 3 shows an example for the request of an image, but other types can be handled in the same way.

Alternatively, MITK-OpenIGTLink can also be used without the pipeline by directly sending OpenIGTLink messages to the OIGTLDevice as indicated in Fig. 1 by the connection from the application layer to the OIGTLDevice. This might be useful if the pipelining concept is not used.

Application layer

The application layer consists of ready-to-use MITK plugins and the MITK-OpenIGTLinkUI module. The latter contains several Qt⁶ GUI widgets that allow the developer to easily integrate the new module into his application. The plugins include the following views:

- OpenIGTLinkManager: The OpenIGTLink manager view is not mandatory but can be used to manage the OpenIGTLink devices registered as Micro Services. Additionally, it also finds all registered conversion filters and allows to manually start the streaming of these sources. Figure 4 shows this plugin with a running example in which tracking data is streamed with 500 Hz.

- OpenIGTLinkExample: This simple example connects itself to an OpenIGTLink device source and visualizes the received tracking data.
- OpenIGTLinkProviderExample: The provider example constructs an IGT-Pipeline that uses previously recorded navigation data to simulate a tracking device. These data is transformed into OpenIGTLink messages and streamed on request (receiving of a STT_TDATA command).

In addition to these new plugins, existing plugins for IGT and US applications were updated in such a way that, instead of hardware devices, these modules can also connect directly to OpenIGTLink network devices.

Performance analysis

In order to assess the performance of latency and frame rate parameters in realistic and reproducible environments, two recently set-up high-end computers⁷ were used. To neglect any falsification caused by the network, the two PCs were directly connected to each other.

The experiments were performed on Linux. Time synchronization between both computers was achieved by using the precision time protocol (PTP) [6] and its implementation, the

⁶ See <http://qt.io>.

⁷ CPU: Core i7-5960X 3.5 GHz 8 cores, RAM: 32 GB, Storage: SSD, GPU: Geforce GTX970 4 GB PCI-E x16, OS: Ubuntu 14.04.

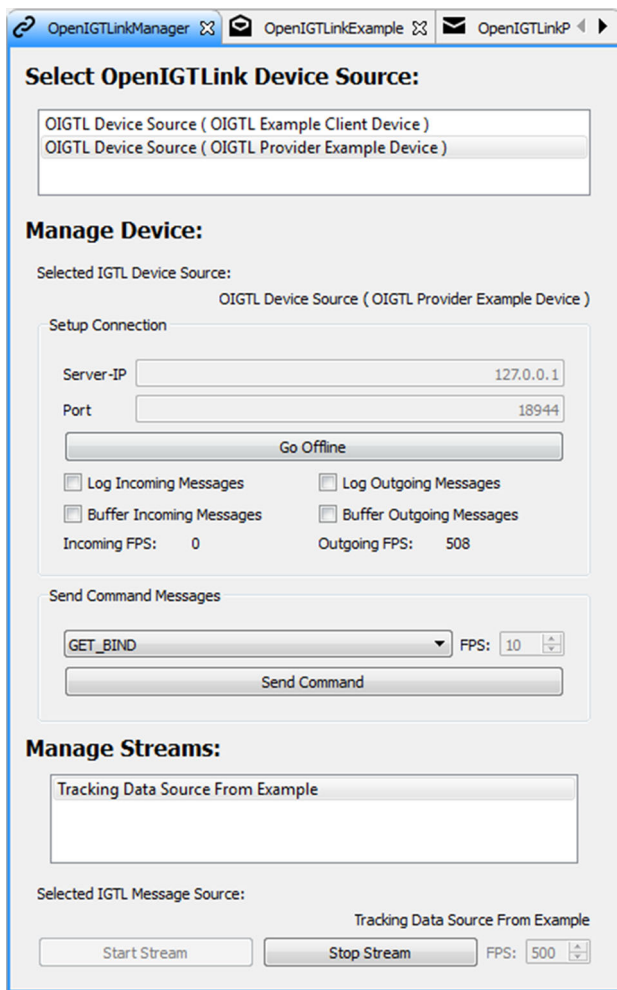


Fig. 4 The user interface of the manager plugin. In the top list view, the device source can be selected. In this example, the provider was selected. The lower part of the interface is used to set up a connection, send commands and manage streams

PTP daemon (PTPd).⁸ PTPd is open source and only available on Linux.

Previous experiments [4,29] with the OpenIGTLink protocol mainly focused on the network performance. Clarkson et al. and Tokuda et al. tested the latency from the generation of the OpenIGTLink message to the receiving in a second PC. Our analysis covers the whole pipeline, from the data generation in MITK to the rendering of the data in the other MITK instance.

In order to evaluate the latency of all steps of the pipeline presented before, we defined six measurement points (MPs) as illustrated in Fig. 5. In each MP, the current timestamp and the index of the current message were recorded. The rendering process was considered in the tests by performing the tests one time with rendering and one time without. However, in both cases the messages are received, converted and

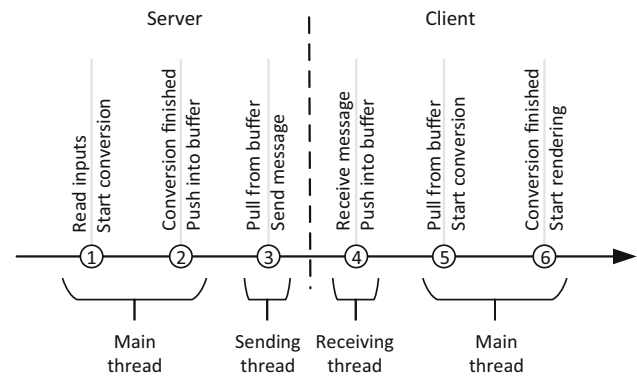


Fig. 5 Measurement points used for the performance analysis

processed in the pipeline. There is no MP inside the rendering since it normally runs slower than the processing. For certain processes, it might be necessary to run with high frame rates (e.g. 500 Hz), whereas the rendering is limited by the refresh rate of the monitor (typically 60 Hz). For this analysis, the rendering was set to 30 Hz.

Experiment 1: transmission of tracking data

We transmitted 10,000 messages containing tracking data in 16 channels with four frame rates of 128, 256, 512 and 1000 Hz. The tracking data were previously generated and read from file. During this experiment, the rendering was turned off since it is application specific. In the easiest case, every channel could be rendered as a single point. However, on a modern computer this would not have an influence on the test results.

Experiment 2: transmission of image data

We simulated an US stream by transmitting 1000 greyscale image messages with an US typical resolution of 640×480 with varying frame rates of 16, 32, 64, 128, 256 and 512 Hz. The images were taken from a standard USB webcam. All measurements were taken two times, with rendering enabled and with rendering disabled.

Experiment 3: transmission of HD image data

We transmitted 1000 greyscale image messages with a full HD resolution of 1920×1080 with frame rates of 16, 32, 64 and 128 Hz. All measurements were taken two times, with rendering enabled and with rendering disabled.

Results

The first part of this section depicts usage scenarios that become available with the MITK-OpenIGTLink module. The second part shows the results of the tests described above.

⁸ See <http://ptpd.sourceforge.net/>.

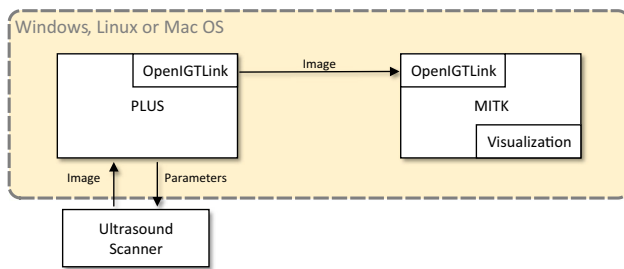


Fig. 6 An example for a cross-toolkit application. Both toolkits run on the same platform. PLUS communicates with the US scanner, performs its algorithms and sends the result to MITK where it is visualized subsequently

Usage scenarios

The integration of the OpenIGTLink protocol into MITK allows several interoperability usage scenarios. They range from intra-toolkit communication on the same computer to intra-toolkit communication on different platforms and communication with medical devices and robotic systems.

Interfacing with other toolkits

Through MITK-OpenIGTLink, MITK can now be easily connected with other toolkits to exchange data and other information. We illustrate this by interfacing MITK to two common open-source toolkits in the medical domain, PLUS and 3D Slicer. Both scenarios were tested on Linux and Windows and worked well with our implementation.

A usage scenario of interfacing between PLUS and MITK is depicted in Fig. 6. This can e.g. be used to acquire data from an US scanner and make use of the algorithmic functionality of PLUS such as volume reconstruction, or spatial and temporal calibration. The acquired and processed data can be recorded and streamed over the network via OpenIGTLink to be used within MITK [14].

Furthermore, the new OpenIGTLink module allows collaboration with other end-user GUI applications, such as 3D Slicer. Hence, research groups can easily exchange and share data and results directly from their favourite frameworks. 3D Slicer, for example, could benefit from the range imaging module of MITK (MITK-TOF, [25]) or from research in progress that is not yet available as open source but provided as binaries.

Figure 7 shows this usage scenario. 3D Slicer contains a plugin called OpenIGTLink Remote that allows querying data and sending commands to a remote OpenIGTLink device. In this case, MITK reacts to the request and answers if the data is available. Moreover, MITK could be configured to run without a GUI just responding to the requests of another end-user GUI application.

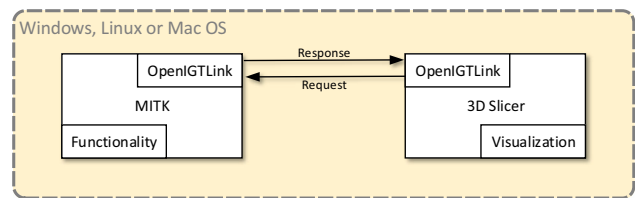


Fig. 7 An example of a collaboration between two toolkits in which MITK is used by another end-user GUI application

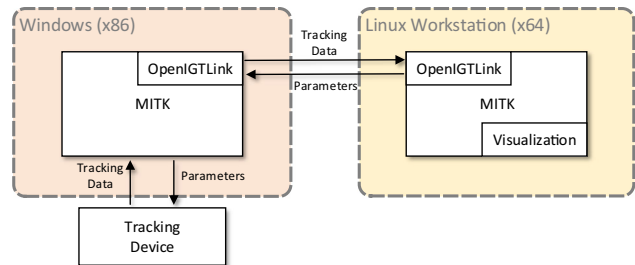


Fig. 8 An example of a cross-platform device integration. The visualization runs on a windows workstation with 64-bit architecture, while the tracking device is connected to a MS Windows PC with x86 architecture due to missing drivers for 64bits

Cross-platform communication

The use case depicted in Fig. 8 allows communication between systems/applications that are bound to run on different platforms (e.g. Windows and Linux) and/or on a different instruction set architecture of the CPU (x86 or x64). With MITK-OpenIGTLink, it is now possible to e.g. interface with proprietary applications that are only available for one specific platform and instruction set without having to port the application code to this (often outdated) system architecture. An example could be a tracking device only accessible on a certain system configuration. In this case, a simple console application that streams the tracking data through OpenIGTLink would be enough. We tested this by streaming tracking data received from a tracking device that only runs on a Windows PC to an application running on Linux.

Interfacing with clinical devices and robotic systems

Recently, the first clinical devices that support OpenIGTLink natively, e.g. the Brainlab navigation system, were introduced. By using this interface, these clinical devices can be integrated in a faster way. Moreover, OpenIGTLink is often used in robotic applications [1–3, 29] which makes the integration of robotics much easier. Another common use case is the data retrieval from an MRI or CT scanner as described in [27]. Figure 9 shows such a set-up.

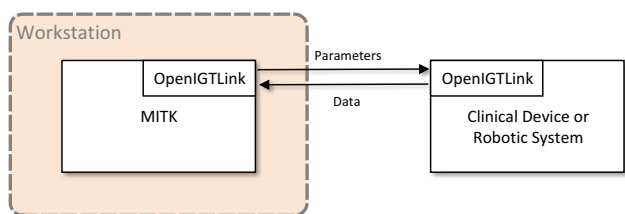


Fig. 9 Interfacing with clinical devices or robotic systems

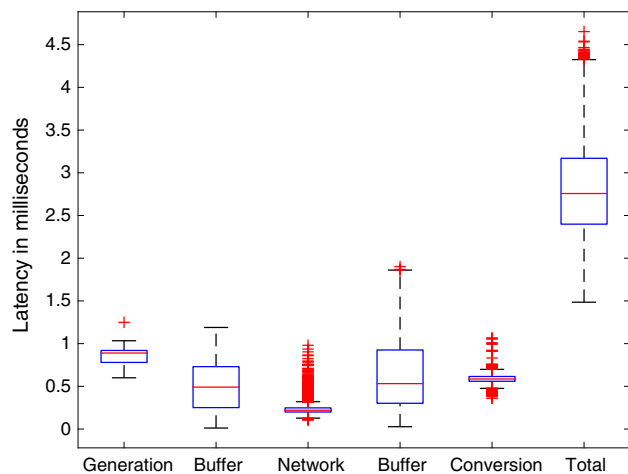


Fig. 10 Experiment 1: latency in transmission of tracking data. 10,000 messages with tracking data of 16 channels were sent at 1000Hz. The average total latency was 2.81 ms

Performance analysis

In the following sections, the results of the performance tests are shown for the tracking data transfer (Sect. 3.2.1), the image transfer (Sect. 3.2.2) and the HD image transfer (Sect. 3.2.3). For each of those individual evaluations, the results are presented in two ways. First, a boxplot diagram is given showing the latency produced by the components of the pipeline. Since these values do not change essentially from one test run to another, only one diagram per experiment is shown. Each test run uses a different frame rate and calculates the mean values for transmitting 10,000 tracking or 1000 image messages. The central marks of a boxplot indicate the median, the edges of the box are the 25th and 75th percentiles, the whiskers extend to the most extreme data points not considered outliers, and outliers are plotted individually as crosses. Second, a table showing the average, median, minimal and maximal values of the test runs is given.

Tracking data transfer

Figure 10 shows a boxplot diagram of Experiment 1. The first boxplot shows the time required for data generation. The second and fourth boxplots show the time a message lies in a buffer. As explained above, this is not processing time

Table 1 Measurement result for tracking data with different frame rates and 10,000 recorded messages

Frame rate (Hz)	Average (ms)	Median (ms)	Min (ms)	Max (ms)
128	7.00	8.38	1.89	10.65
256	3.13	3.08	1.95	6.19
512	2.90	2.84	1.64	4.81
1000	2.81	2.76	1.48	4.65

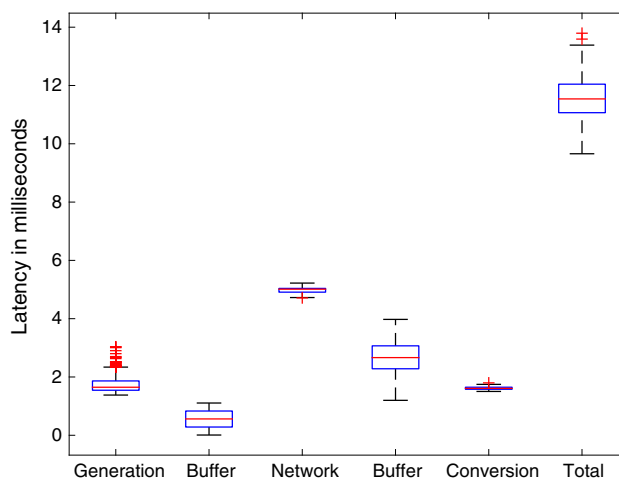


Fig. 11 Experiment 2: latency in transmission of image data (640 × 480 pixel). 1000 messages were sent at 128 Hz with disabled rendering. The data transmission is the most time-consuming step

but idle time and has a major influence on the total result. The third boxplot is the time the message is sent from one socket to the next one. The fifth one is the time necessary to convert the incoming OpenIGTLink message into an MITK data type, and the sixth boxplot shows the total latency from generation till conversion.

As shown in Fig. 10, the two buffers produce the highest latencies in the pipeline. The higher the frame rate, the lower the buffer idle time and, thus, the lower the total latency. This behaviour is given in Table 1. The average latencies, depending on the frame rate, lie between 2.81 and 7 ms.

Image transfer

Figure 11 shows the result for Experiment 2 in which image data with a fixed resolution of 640 × 480 pixel were sent over the network. Compared to Experiment 1, the major part of the latency is not produced by the buffers but by the network itself. The average latency, depending on the frame rate and if rendering is enabled, lies between 10.5 and 21 ms (see Tables 2 and 3). Until 128 Hz the rendering only causes minor latencies. However, for higher frame rates it causes a significant increase in latency of almost 100 %.

Table 2 Measurement result for image data with different frame rates and 1000 recorded messages (rendering disabled)

Frame rate (Hz)	Average (ms)	Median (ms)	Min (ms)	Max (ms)
16	13.77	13.75	12.44	15.20
32	13.01	13.00	11.57	14.96
64	12.57	12.58	10.65	14.42
128	11.55	11.54	9.66	13.79
256	10.55	10.31	8.54	23.01
512	11.20	10.52	8.36	25.43

Table 3 Measurement result for image data with different frame rates and 1000 recorded messages (rendering enabled)

Frame rate (Hz)	Average (ms)	Median (ms)	Min (ms)	Max (ms)
16	14.17	14.11	11.12	18.68
32	13.66	13.57	10.69	19.09
64	12.60	12.61	8.99	17.80
128	12.64	12.48	9.56	20.57
256	18.36	18.84	8.87	30.86
512	21.71	14.28	8.47	53.44

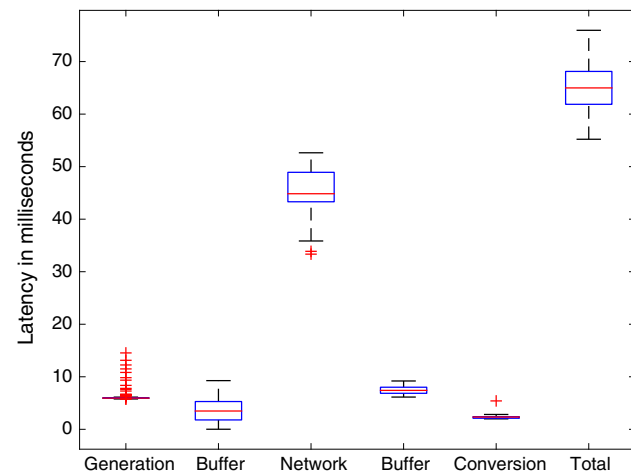


Fig. 12 Experiment 3: latency in transmission of HD image data. 1000 messages were sent at 128 Hz with disabled rendering. The data transmission is by far the most time-consuming step

HD image transfer

Figure 12 shows the latencies for Experiment 3 in which HD greyscale image data with a fixed resolution of 1920 × 1080 pixel were sent over the network. Due to the increased message size, the major part of the latency is produced in the network itself. The average latency, depending on the frame rate, lies between 65 and 69.5 ms (see Table 4). Unlike

Table 4 Measurement result for HD image data with different frame rates and 1000 recorded messages

Frame rate (Hz)	Average (ms)	Median (ms)	Min (ms)	Max (ms)
16	69.45	69.57	60.75	74.72
32	60.86	60.86	51.07	71.13
64	68.90	68.96	50.29	84.47
128	65.09	64.99	55.21	75.95

the results in Experiment 2, these results are only slightly influenced by the rendering.

Discussion

The experiments were carried out under Linux utilizing the PTPd implementation. Due to a missing open-source alternative on MS Windows and missing hardware for OS X, tests were not performed on these platforms.

The missing implementation of an asynchronous connect and receive method in OpenIGTLink v2.0 makes it necessary to have individual threads for polling the socket. On the one hand, putting the threads to sleep for 1 ms is necessary to reduce the CPU usage. On the other hand, it also reduces the upper frame rate limit to 1000 Hz. If there were to be an asynchronous receive available in future versions of OpenIGTLink, these threads would not be necessary anymore.

Tokuda et al. [29] state that the frame rate of tracking devices is in the range of 40–375 Hz and the one for robotic applications in the order of kHz. The experiments show that tracking data can be sent with more than 1000 Hz. The highest measurable frame rate in our test set-up, 1000 Hz, resulted in a latency of 2.81 ms on average. In contrast, the lowest measured frame rate was 128 Hz and resulted in a latency of 7 ms. According to Teather et al., an NDI Polaris tracking system has a latency of approximately 75 ms [28]. Wu and Taylor state that electromagnetic tracking systems have an even higher latency [31]. Considering these tracking latencies, an additional latency of 7 ms is acceptable.

The fact that the latency is decreasing with increasing frame rates is due to the implemented buffers that cause a big part of the total latency. This means that the faster the buffers are polled, the lower the latency will be. Therefore, the latency can be decreased by running the “consuming” pipeline with a higher frame rate than the transmission, e.g. the transmission runs with 128 Hz and the pipeline with 512 Hz. Another way to improve this behaviour could be to couple the message reception with the pipeline by triggering the pipeline update once a message is received.

According to [29], around 32 Hz are sufficient for real-time US imaging. Our implementation measures up to 512 Hz

with enabled rendering and a median latency of 14 ms. HD greyscale images were processed with 128 Hz and a latency of 66 ms. Theoretically, HD RGB images could be sent with up to 43 Hz ($128/3 \approx 43$) which is still more than the recommended 32 Hz. Therefore, the presented implementation is able to cover most applications utilizing image messages.

Enabling the rendering of transmitted images only showed differences in Experiment 2 but not in Experiment 3. In Experiment 2, US image data were transmitted with up to 512 Hz and a difference occurred for frame rates higher than 128 Hz. In Experiment 3, HD image data were transmitted with up to 128 Hz. We assume that the PCs are able to transmit and process images up to 128 Hz, independent of the image size, but that they are partly overloaded for higher frame rates. This is based on the fact that the increase in processing time does not correlate with the increase in image size when comparing US with HD images. In MITK, the handling of US and HD images is exactly the same and mainly management, the part that has to differentiate between the image sizes for rendering purposes is running on the GPU and does not influence the measurement.

A direct comparison between the presented results and the previously published experiments in [4, 29] cannot be made since they concentrated on the network performance and their OpenIGTLink implementation. The presented analysis, on the contrary, covers the whole pipeline, from the data conversion in MITK to the rendering of the data in the other MITK instance. [4] and [29] state for a tracking data transfer with 128 Hz a latency of around 0.3 ms, respectively, 0.36 ms, whereas the presented results show a latency of 7 ms. [4] and [29] measure the time from creating an OpenIGTLink message at the sender host until the end of the deserialization at the receiver host. These measurements are very interesting concerning the OpenIGTLink interface but do not give information about the performance of an application. They do not include visualization nor management of messages. In the presented tests, we measure the application-specific latency: we generate real data (as MITK data types), convert it into messages, send them and vice versa on the receiving side. Moreover, we implemented buffers to easily integrate OpenIGTLink into the pipelined structure of MITK. The most time-consuming component in the tests by [4] and [29] is the networking. In our 128 Hz tracking data test, we had an average network latency of 0.19 ms. Creation, serialization and deserialization can be considered as less time-consuming as the networking which means that our implementation performs similarly to [4] and [29].

For image data transfers with frame rates higher than 512 Hz, the network load was so high that the PTP synchronization packages arrived delayed. In this scenario, two separate network connections might be necessary, one for the data and one for the synchronization.

The usage scenarios show that MITK can communicate with other toolkits such as 3D Slicer or PLUS. These toolkits can also run on different platforms, e.g. Linux $\times 64$ and Window $\times 32$.

Conclusion

In this paper, we presented a new software module which enables OpenIGTLink support in MITK. We carried out performance tests and described new usage scenarios. MITK-OpenIGTLink was released as open source together with the MITK toolkit release 2016-03. The new module enables real-time communication of different types of CAI data, such as tracking data, US data and HD greyscale image data which was demonstrated by the performance analysis. MITK can now be combined with other toolkits in a plug-and-play manner which was shown for 3D Slicer and PLUS. Tutorials and manuals on how to use MITK-OpenIGTLink were published together with the release.

Acknowledgements The authors would like to acknowledge support from the European Union through the ERC starting Grant COM-BIOSCOPY under the New Horizon Framework Programme Grant Agreement ERC-2015-StG-37960.

Compliance with ethical standards

Conflict of interest The authors declare that there are no known conflicts of interest associated with this publication.

Ethical approval This article does not contain any studies with human participants or animals performed by any of the authors.

Informed consent This articles does not contain patient data.

References

1. Arata J, Kenmotsu H, Takagi M, Hori T, Miyagi T, Fujimoto H, Kajita Y, Hayashi Y, Chinzei K, Hashizume M (2013) Surgical bedside master console for neurosurgical robotic system. *Int J Comput Assist Radiol Surg* 8(1):75–86
2. Arata J, Kozuka H, Kim HW, Takesue N, Vladimirov B, Sakaguchi M, Tokuda J, Hata N, Chinzei K, Fujimoto H (2010) Open core control software for surgical robots. *Int J Comput Assist Radiol Surg* 5(3):211–220
3. Arata J, Tada Y, Kozuka H, Wada T, Saito Y, Ikedo N, Hayashi Y, Fujii M, Kajita Y, Mizuno M, Wakabayashi T, Yoshida J, Fujimoto H (2011) Neurosurgical robotic system for brain tumor removal. *Int J Comput Assist Radiol Surg* 6(3):375–385
4. Clarkson MJ, Zombori G, Thompson S, Totz J, Song Y, Espak M, Johnsen S, Hawkes D, Ourselin S (2014) The NifTK software platform for image-guided interventions: platform overview and NiftyLink messaging. *Int J Comput Assist Radiol Surg* 10(3):301–316
5. Cleary K, Peters TM (2010) Image-guided interventions: technology review and clinical applications. *Annu Rev Biomed Eng* 12(1):119–142

6. Correll K, Barendt N, Branicky M (2005) Design considerations for software only implementations of the IEEE 1588 precision time protocol. In: Conference on IEEE, vol 1588, pp 10–12
7. Egger J, Tokuda J, Chauvin L, Freisleben B, Nimsky C, Kapur T, Wells W (2012) Integration of the OpenIGTLink network protocol for image-guided therapy with the medical platform MeVisLab. *Int J Med Robot Comput Assist Surg* 8(3):282–290
8. Fedorov A, Beichel R, Kalpathy-Cramer J, Finet J, Fillion-Robin JC, Pujol S, Bauer C, Jennings D, Fennessy F, Sonka M, Buatti J, Aylward S, Miller JV, Pieper S, Kikinis R (2012) 3D Slicer as an image computing platform for the quantitative imaging network. *Magn Reson Imaging* 30(9):1323–1341
9. Franz AM, Seitel A, Servatius M, Zollner C, Gergel I, Wegner I, Neuhaus J, Zelzer S, Nolden M, Gaa J, Mercea P, Yung K, Sommer CM, Radeleff BA, Schlemmer HP, Kauczor HU, Meinzer HP, Maier-Hein L (2012) Simplified development of image-guided therapy software with MITK-IGT. In: Proceedings of SPIE, medical imaging 2012: image-guided procedures, robotic interventions, and modeling, vol 8316
10. Ibanez L, Schroeder W, Ng L, Cates J (2005) The ITK software guide. The ITK software guide 804
11. Ince DC, Hatten L, Graham-Cumming J (2012) The case for open computer programs. *Nature* 482:485–488
12. Kang HJ, Stolka PJ, Boctor E (2011) OpenIGTLinkMUSiC : a standard communications protocol for advanced ultrasound research. *MIDAS J* 1–12
13. Kilgus T, Heim E, Haase S, Prüfer S, Müller M, Seitel A, Fangerau M, Wiebe T, Iszatt J, Schlemmer HP, Hornegger J, Yen K, Maier-Hein L (2015) Mobile markerless augmented reality and its application in forensic medicine. *Int J Comput Assist Radiol Surg* 5(1):573–586
14. Lasso A, Heffter T, Rankin A, Pinter C, Ungi T, Fichtinger G (2014) PLUS: open-source toolkit for ultrasound-guided intervention systems. *IEEE Trans Biomed Eng* 61:1–11
15. Maier-Hein L, Tekbas A, Seitel A, Pianka F, Muller SA, Satz S, Schawo S, Radeleff B, Tetzlaff R, Franz AM, Muller-Stich BP, Wolf I, Kauczor HU, Schmied BM, Meinzer HP (2008) In vivo accuracy assessment of a needle-based navigation system for CT-guided radiofrequency ablation of the liver. *Med Phys* 35(12):5385
16. März K, Franz AM, Seitel A, Winterstein A, Bendl R, Zelzer S, Nolden M, Meinzer HP, Maier-Hein L (2014) MITK-US: real-time ultrasound support within MITK. *Int J Comput Assist Radiol Surg* 9(3):411–420
17. März K, Franz AM, Seitel A, Winterstein A, Hafezi M, Saffari A, Bendl R, Stieltjes B, Meinzer HP, Mehrabi A, Maier-Hein L (2014) Interventional real-time ultrasound imaging with an integrated electromagnetic field generator. *Int J Comput Assist Radiol Surg* 9(5):759–768
18. McCormick M, Liu X, Jomier J, Marion C, Ibanez L (2014) ITK: enabling reproducible research and open science. *Front Neuroinf* 8(February):13
19. Müller M, Rassweiler MC, Klein J, Seitel A, Gondan M, Baumhauer M, Teber D, Rassweiler JJ, Meinzer HP, Maier-Hein L (2013) Mobile augmented reality for computer-assisted percutaneous nephrolithotomy. *Int J Comput Assist Radiol Surg* 8(4):663–675
20. Neuhaus J, Wegner I, Käst J, Baumhauer M, Seitel A, Gergel I, Nolden M, Maleike D, Wolf I, Meinzer H (2009) MITK-IGT: eine navigationskomponente für das medical imaging interaction toolkit. *Bildverarbeitung die Med* 2009:454–458
21. 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, Wolf I (2013) The medical imaging interaction toolkit: challenges and advances: 10 years of open-source development. *Int J Comput Assist Radiol Surg* 8(4):607–620
22. Ordas S, Yaniv Z, Cheng P, Tokuda J, Liu H, Hata N, Cleary K (2009) Interfacing proprietary hardware with the image-guided surgery toolkit (IGSTK): a case for the OpenIGTLink protocol. In: proceedings of SPIE, vol 7264, pp 72640F–72640F–7
23. Pieper S, Halle M, Kikinis R (2004) 3D Slicer. In: 2004 2nd IEEE international symposium on biomedical imaging: nano to macro (IEEE Cat No. 04EX821)
24. Seitel A, Bellemann N, Hafezi M, Franz AM, Servatius M, Saffari A, Kilgus T, Schlemmer HP, Mehrabi A, Radeleff BA, Maier-Hein L (2015) Towards markerless navigation for percutaneous needle insertions. *Int J Comput Assist Radiol Surg* 11:107–117
25. Seitel A, Yung K, Mersmann S, Kilgus T, Groch A, Dos Santos TR, Franz AM, Nolden M, Meinzer HP, Maier-Hein L (2012) MITK-ToF-range data within MITK. *Int J Comput Assist Radiol Surg* 7(1):87–96
26. Stolka PJ, Kang HJ, Boctor E (2010) The MUSiC toolkit: modular real-time toolkit for advanced ultrasound research. *MIDAS J Comput Assist Interv* 1–11
27. Su H, Shang W, Member S, Cole G, Li G, Member S, Harrington K, Camilo A, Tokuda J, Tempany CM, Hata N, Fischer GS (2014) Piezoelectrically actuated robotic system for MRI-guided prostate percutaneous therapy. *IEEE/ASME Trans Mech* 1:1–13
28. Teather RJ, Pavlovych A, Stuerzlinger W, MacKenzie IS (2009) Effects of tracking technology, latency, and spatial jitter on object movement. In: Proceedings of 3DUI— IEEE symposium on 3D user interfaces 2009, pp 43–50
29. Tokuda J, Fischer GS, Papademetris X, Yaniv Z, Ibanez L, Cheng P, Liu H, Blevins J, Arata J, Golby AJ, Kapur T, Pieper S, Burdette EC, Fichtinger G, Tempany CM, Hata N, Alexandra J, Kapur T, Pieper S, Burdette EC, Fichtinger G, Clare M, Hata N (2009) OpenIGTLink: an open network protocol for image-guided therapy environment. *Int J Med Robot Comput Assist Surg* 5(4):423–434
30. Wolf I, Vetter M, Wegner I, Nolden M, Bottger T, Hastenteufel M, Schobinger M, Kunert T, Meinzer HP (2004) The medical imaging interaction toolkit (MITK) a toolkit facilitating the creation of interactive software by extending VTK and ITK. *Med Imaging* 2004:16–27
31. Wu X, Taylor RH (2003) A framework for calibration of electromagnetic surgical navigation systems. In: Proceedings 2003 IEEE/RSJ international conference on intelligent robots and systems (IROS 2003), vol 1, pp 547–552