CoCo - A Framework for Multicore Visuo-Haptics in Mixed Reality

Emanuele Ruffaldi(✉) and Filippo Brizzi

TeCiP, Scuola Superiore Sant'Anna, Pisa, Italy
e.ruffaldi@sssup.it
http://www.percro.org

Abstract. Mixed Reality applications involve the integration of RGB-D streams with virtual entities potentially extended with force feedback. Increasing complexity of the applications pushes the limits of traditional computing structures, not keeping up with the increased computing power of multicore platform. This paper presents the CoCo framework, a component based, multicore system designed for tackling the challenges of visuo-haptics in mixed reality environment, with structural reconfiguration. Special care has been also given to the management of transformation between reference frames for easing registration, calibration and integration of robotic systems. The framework is described together with a description of two relevant case studies.

1 Introduction

Advancements in sensing, computing and display technologies is expanding the possibility of uses of Virtual, Augmented and broader Mixed Reality (MR) applications[1], in local or networked situation, involving robots or multiple users. Such variable Mixed Reality applications, in general, require the integration of sensing components, simulation and visuo-haptic feedback. The development of such applications is typically based on a computer graphics oriented framework, typically structured around a scene graph, and many of such frameworks do exist spanning from commercial ones to community or research developed. The scene graph is then paired with some form of application development based on explicit programming or visual programming following a data flow paradigm.

The abstraction level of a MR development tool allows the developer to easily create a prototype, but, due to the nature of high-latency, high-throughput nature of sophisticated MR applications it can soon impact into performance issues associated to frame rate and latency. The aspect of processing and output rate is even more relevant when the MR application needs to provide haptic feedback or timely feedback to a robot. In the end, as it is well known, a multimodal MR application is organized around multiple components exchanging data at different rates. The point is how to take advantage of recent multi-core systems for reasonably interesting MR solutions.

[1] We use the general term MR for all the applications in the Mixed Reality continuum.

L.T. De Paolis and A. Mongelli (Eds.): AVR 2016, Part I, LNCS 9768, pp. 339–357, 2016.
DOI: 10.1007/978-3-319-40621-3_24

This paper discusses the design choices and the implementation of the Compact Components (CoCo) framework that aims at providing abstractions concepts and specific components for addressing MR applications with modern multi-core systems. CoCo has been so far employed in the context of VR and MR applications with haptic feedback [1,2] or interfaced with robots [3]. CoCo is founded over three pillars that can be considered fundamental in such applications: integration, computing abstraction and transformations. The first pillar is provided by the component based approach, the second allows to control the use of resources after the development of components, and the third is an important mechanism that supports reference frames flexibility, registration and sensor fusion. Conceptually CoCo is based on three orthogonal graphs: *computational graph* that support the data flow between components (a DAG), a *scheduling graph* that supports the partitioning of components' execution in processes and threads (a tree), and a *transformation graph* (a general graph).

The following section provides the research context on the topic for better understanding the proposed innovation. Section 3 presents the concepts behind the design of CoCo, Sect. 4 describes the functionalities provided for supporting visuo-haptic applications. Section 5 discusses how Transformations between frames are addressed. Then Sect. 6 provides a case study followed by conclusions.

2 Background

There is a long history of frameworks for 3D graphics applications and their natural extension to Virtual Reality setup. Since the beginning the VR and in general MR applications have posed computing challenges, due to the requirement of high and regular output frame rates with minimal latency. In VR applications the main issues arise from integration with physics simulation and interaction devices. When moving to AR or MR applications the sensing component involves higher computing requirements due to the closed loop with imaging devices. Finally when moving to visuo-haptic devices there is the additional requirement of providing timely haptic feedback that, for rigid objects, requires update rates in the order of 1kHz or more.

Two main approaches are found in literature and in the implementation for structuring the application: scene graph-based and flow-based approaches. In scene-graph approaches the developer creates a hierarchy of entities spanning across the different modalities, then the framework organizes the processing in loops (graphics, physics, collision) with some flexibility for the developer. This approach is found in systems such as OpenSceneGraph [4], XVR [5] or Unity. Conversely flow-based approaches allow the developer to describe the application in terms of streams of data and events connected in a flow structure. The latter approach can be found in the X3D standard and derivatives like InstantReality [6] and other systems such as InTml [7] or FlowVR [8]. FlowVR is a notable example because it provides distributed computing capabilities for VR, introducing a very flexible mechanism for synchronization among modules that allows the implementation of several communication patterns. In general the use of

declarative approaches, in comparison to immediate programming, allows for the underlying framework to perform certain optimizations based on the underlying platform, or for extending the rendering from single output to stereo or more.

When dealing with visuo-haptics the existing frameworks follow both patterns like scene-graph in CHAI3D [9,10], or flow-graphs derived from X3D in H3DAPI [11]. CHAI3D leaves to the developer the burden of organizing the loop, while internally performing traversal and computing, while H3DAPI and derived follow the event-based approach of X3D. In the above examples the control of execution loops is hidden in the framework or explicitly managed by the developer at low-level, with hard work for scalability or profiling.

CoCo contributes to the field by proposing a component based approach in which each component has an independent execution loop that is not directly controlled by the component developer. Instead the framework allows for structuring the scheduling of components at run-time providing space for reconfiguration depending on the users needs. This approach takes inspiration from the OROCOS robotic framework [12] introducing specific aspects aimed at visuo-haptic MR applications.

As discussed in the introduction the third pillar of CoCo is the transformation systems, an aspect that is fundamental in any MR or robotics application: how transformation between reference frames are generated and used by the components. In scene-graph based frameworks the scene-graph itself provides the structure of the transformations forcing a tree-like structure, while in data flow based approaches there is the need to stream the transformation along the structure.

CoCo provides a general declarative approach of transformations that is orthogonal to the information flow between components and based on a Spatial-Relationship Graph (SRG). Differently from the ROS [13] TF2 system, CoCo allows to associate semantics to transformations at run-time or at build-time, providing, in this way, the support for sensor fusion and moreover generalized registration, two aspects that are fundamental for AR/MR [14].

3 Concepts

The foundation of the CoCo framework is a component-based system in which each node is an independent unit of execution exchanging data, invoking operations or triggering the execution of other components. CoCo has been developed in C++11 with the objective of being lightweight and multi-platform. Components are loosely coupled to increase modularity and reduce development dependencies: in terms of C++ this means that the only common element between components is the data exchanged. Components are stored in dynamic or static libraries and they can be instantiated at run-time by name. A CoCo application is typically launched by providing an XML file that configures the components and connects them.

In general a CoCo component comprises the following elements:

– Callbacks, in particular with the loop callback *onUpdate*.
– Input and output data ports that are used for the main exchange of data.
– Declarative attributes that are configured via XML or at run-time.
– Operators that can be invoked in a thread-safe manner.

3.1 Components Lifetime

Components can be instantiated at any time although it is more typical to have main instantiations at application startup time. Two functions are used to perform initialization and configuration (*init*, *onConfig*) based on the parameters received via the configuration file. After initialization a component receives an execution request in the *onUpdate* function that, due to the execution abstraction of CoCo, can be periodic or not. In any case the *onUpdate* implementation should not block the execution.

3.2 Ports

Ports are the key mechanism for data exchange between components inside CoCo and they have been designed to support different patterns of exchange and moreover, the exchange of large entities such as images or pointclouds.

Each component can declare a set of input and output named ports which are templated against a C++ native type. Thanks to C++11 capabilities introspection of ports and their types is straightforward but at the moment no serialization capability between processes has been introduced. A port marked as event port is then used by the scheduling system to trigger aperiodic components.

The connection between ports is many-to-many meaning that a single element written to an output port can be received by multiple recipients. When an input port has multiple sources they are processed in a round robin fashion, although in a future timestamps could be used for chronological ordering.

An important principle of CoCo is that ports never block nor in reading or writing, because components should never block inside their looping step.

Two aspects, controlled in the XML file, specify the nature of the connection: buffering and synchronization policy. Three types of buffering are supported corresponding to recurring patterns in MR applications: for example a pipeline with different generation-consumption rates, or a window of the last valid values.

– **DATA**: The connection has a buffer of length 1 and new incoming data always overrides existing data even if it has not been read.
– **BUFFER**: The connection has a buffer of length as specified in the configuration file. If the buffer is full new incoming data is discarded without blocking.
– **CIRCULAR**: The connection has a circular FIFO buffer of length as specified in the configuration file. If the buffer is full new incoming data overrides the oldest one. A DATA buffering is equivalent to a CIRCULAR with length 1 except much more efficient.

The connections support two synchronization policies: **LOCKED** and **UNSYNC**. In the first case data access is regulated by mutexes while in the second case there is no resource access control policy. A lockless policy could be added providing high-efficient data access. The unsync policy applies for the connections between components inside the same activity.

CoCo ports and connections operate via value-copy of the received content. This solution is efficient for small sized types, and it allows to manage large entities via shared pointer solutions. The issue with shared pointer solution is that when the last user of the object releases the object this is destroyed. This is not the optimal choice for large objects produced at high rates for the effect on the memory manager. CoCo provides a pooled channel mechanism for supporting efficiently the exchange of large entities such as images, point clouds or meshes. Every slot of the pooled buffer has four states logically ordered: free, writing, ready, reading. This means that when a writer needs to write an entity, first it receives it from the pool, writes it and then it makes the entity available for ready. Similarly a reader receives the entity and needs to notify when it has finished using the content.

3.3 Scheduling

Execution of components in CoCo, and in a MR application in general, is periodic with fixed rate or aperiodic being triggered by some event, being it internal or external to the framework.

Whatever the nature of the component the execution takes place inside a container called *activity* that can hold multiple components. An activity corresponds, in practice, to an OS thread and, for this reason, it can be associated to system priority and processor affinity. At every step of the activity all the components are executed sequentially. Periodicity or triggering are specified at the level of activity: a triggered activity is activated when any of the contained components receive some data in a triggerable input port.

The XML configuration file of an application is organized per-activity each with the contained components. The activity configuration comprises the periodic nature as period in milliseconds or triggering. In addition one of the activity can be marked as "main" for being associated to the main thread of the CoCo application.

The component-activity separation allows the developer to reconfigure the flow of execution at run-time, without the need to customize or recompile the components. An improvement of the model discussed above relies on increasing the granularity of components activation, that is supporting multiple rates inside an activity or controlling the triggering.

3.4 Operations

In addition to the flow-based data exchange components can invoke operations of other components with the guarantee that the invoked operation is executed in a thread-safe manner: that is if the two components belong to the same thread

no overhead is introduced otherwise a messaging system deals with the delivery of the function call.

Each component can bind any of its C++ methods to an operation. An operation is identified by a name and by the signature of the function it embeds. A task's operations can be called by any other task or can be enqueued in the task pending operations list. Every time an activity resumes its execution, either because the period timer expires or it is triggered by data reception, before executing the main loop function *onUpate*, it execute all the pending operations. The task invoking the function to enqueue an operation on another task can add to the call a function to get the return value of the operation.

3.5 Peers

There are some situations in which the activity-component hierarchy of execution is not enough, for this reason it is possible to nest components inside other components. CoCo calls them peers. Peers are used to extend the functionalities of a component preserving code encapsulation and reusability as they can be instantiated multiple times for different components and the binding is decided at run-time. Peers are components by themselves inheriting all the functionalities such as ports, attributes and operations. The main difference relies in the fact that peers execution is controlled by the owner component, and typically they are used via operations. Thanks to operations there is no limits in the number of peers that can be associated to a component or to another peer giving the possibility to create a tree of peers with any desired depth or width.

3.6 Patterns

From the scenarios of MR applications it is possible to identify several patterns of components usage. A common pattern is the one of sensor source that produces a stream of data triggered by an external source such as a socket, an external API or a USB file descriptor. A lightweight filtering component that needs to use the sensor source data, can be placed in the same activity to reduce latency, while heavier processing is better to be moved in a separate one. Another pattern is the one of state holders with fast query such as KD-Tree in which updates are expensive but queries are fast. In this case the update comes via an input port, while queries are realized with an operation.

3.7 Profiling

The CoCo libraries provides also an utility to easily calculate execution time of blocks of code. This functionality can be used by the user inside its components to quickly evaluate the computational load and it has been inserted inside the core of the library to obtain precise statistics on the components performance. The component profiling is activated passing a specific flag to the launcher and statistics of the execution are provided with a certain time interval. The measurements include, for each component, the number of executions and the total

execution time; mean and variance of the average computation and service time. This last value is used to evaluate the feasibility of the application scheduling because it represents the mean time between two activations of a component and should be equal to the period for periodic components.

4 Visuo-Haptics for Mixed Reality

The CoCo framework has been used as the infrastructure for the realization of a set of libraries, called CoCo Mixed Reakity (CoCoMR), targeting the creation of visuo-haptics applications for MR scenarios. One of the complexity in this kind of applications stands in the different rates at which each component executes. A standard visuo-haptics application can be composed of a module reading frames from a camera at 30 Hz or more, the graphics renderer module that runs between 60 to 120 Hz depending on the visualization nature and a module controlling the haptic device running at least at 1 kHz. Three modules sets (Vision, Haptic and Display) have been developed to target each specific scenario and thanks to the CoCo features they can be combined and customized as desired at run-time. CoCoMR contains also common utilities and a shared interface to allow different components to exchange data through the CoCo ports. An overview of the core modules is shown in Fig. 1.

Each module set's components can be divided into two main categories: the ones that interacts with the external world, either devices or other applications, and the ones doing the internal computations. The components in the first group are all executed periodically and their period can be adjusted at run-time to synchronize it with a specific device or an external software (sources in the

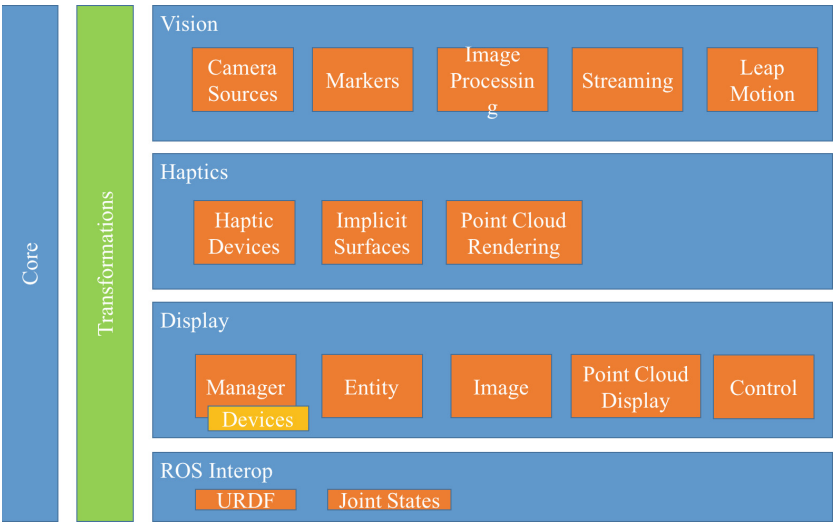

Fig. 1. Overall view of modules and components

computational graph). The second group, instead, contains the components that perform the internal calculations and their execution is usually triggered by the data received from the components in the first group.

The section continues with the features of each module that allow to understand the span of applications covered by CoCo and the approach described in the previous Concepts section.

4.1 Vision Module

The vision module provides the services for the computer vision part of a MR application, that is the acquisition of image sources, the tracking of features or fiducial markers, and, for the case of tele-presence applications, the streaming.

Components in the vision module exchange data structures that correspond to images, RGB-D images, and camera parameters (intrinsics matrix and distortion). In particular color images are encoded with the possibility of using several color formats (grayscale, RGBA, YUYV and YUV420) with the aim of limiting the conversion from sensors (typically producing images in YUYV) and toward computer vision algorithms that employ often grayscale images. The YUV420 is instead the layout of video compressors. GB-D data is stored as a combination of the color image and a depth image (float or signed int16).

Source components are the following:

- *CameraReader*: using gstreamer or OpenCV captures camera frames and share them with the other CoCo components.
- *RgbdCameraReader*: same as *CameraReader*, but captures also the depth buffer. The component supports multiple cameras such as Kinect 360 or One, Asus Xtion and Intel R200 directly through libfreenect, OpenNI or libfreesense. For each camera there is a specific CoCo peer embedding the different API for each vendor. If provided by the driver cameras are associated with the intrinsics.
- *LeapReader*: it interfaces the LeapMotion API with CoCo providing to the other components position, orientation and fingers pose of the hands.
- *StreamingReceiver*: it is used to receive via TCP and decodes image and depth streams in case of applications for tele-operation.

Filtering and sink components are:

- *MarkerTracker*: receives a camera buffer and produces in output the pose of eventual markers present in the image.
- *MeshReconstructor*: receives the image and the depth buffers from *RgbdCameraReader* and creates a mesh interpolating the missing points.
- *StreamingServer*: receives an image and/or a depth buffer and sends it via TCP. Must be coupled with *StreamingReceiver* at the client side.

4.2 Haptic Module

The haptic module provides *haptic rendering* that is the generation of force feedback following the convention of computer graphics rendering [15]. There are several challenges in haptic rendering tasks mainly related to the high rate of the update loop, 1 kHz, and especially when the interacting surface is dynamically updated as coming from a RGB-D camera. CoCo provides internally the support for the haptic rendering of implicit surfaces or point clouds (used in [2]). Other techniques can be integrated such as volumetric voxel models [16] or the 3DOF spherical Proxy algorithm [17] for triangular meshes to external libraries such a CHAI3D.

The rendering of implicit surfaces is based on the Salisbury algorithm [18] that updates a contact proxy based on an implicit surface described as a distance to the surface and the local gradient. CoCo supports the creation of procedural implicit surfaces expressed on constructive solid geometry over building blocks such as cylinders, planes and spheres. The *HapticImplicitSurf* component is configured over a functional description of the surface, and then it tracks the proxy over the surface, generating force feedback with friction parameters.

Contact rendering of live point clouds, or point-sampled meshes, is based on the identification of the points around the proxy via KD-Tree and the creation of a local surface from such points. The *KDTreeBuilder* component is responsible for the creation of the KD-Tree either from a point cloud or the vertices of a meshes provided by the adapter *MeshReconstruction* component. The *HapticCloud* component provides haptic rendering over these points using the KD-Tree emitted by the *KDTreeBuilder*. This is an example of the large resource management of CoCo: the KDTreeBuilder has an input port with the new point cloud and inside the loop it performs the slow update, while, at the same time, the last value of the KD-Tree is available over the output port. The port mechanism allows the reuse of the values in the ports, much like happens in a classical front-back buffer, but only in a more general way.

For a complex scene with multiple surfaces (or layered materials) it is possible to coordinate different rendering components, or aggregate them inside a single component, called *HapticRenderer*, that invokes the various renderers using the peer mechanism. As discussed, anyway, the peers share the same activity, meaning the same OS thread.

4.3 Display Module

The Display module has been developed to provide a reasonably good visualization capability for MR applications with the main idea of displaying images or point clouds produced by set of cameras looking more at performance than display visual effects. There is no intention to replicate features found in more sophisticated 3D engines, such as shadows or large complex models.

The display module is based on OpenGL 3.3 and it is responsible for the rendering of 3D objects and the eventual images obtained from cameras.

It is composed of a single component and several peers, mainly due to the single-thread nature of the OpenGL API. Multi-threaded OpenGL could be an option but it is known to impact the performance of the overall 3D rendering, and multi-threading can be exploited only for the memory transfer between CPU and GPU buffers, e.g. for uploading point cloud or texture data. The recent introduction of the Vulkan API has opened the way for multi-threading with GPU and it could be an interesting enhancement for the graphics part of CoCo. The component (*GLManager*) is the graphics manager and it is in charge of initializing the OpenGL context and the rendering window across a variety of devices. The OpenGL camera and every element that has to be rendered are associated to a peer. *GLManager* queries the camera peer for the projection/view matrices and then iterates over all the other peers calling their rendering function.

When instantiating the *GLManager* component it is possible to specify the frame rate, setting the desired period in the activity containing it, the window resolution and the visualization type covering 2D, 3D stereo or Oculus Rift DK2. CoCoMR supports the creation of multiple visualization windows in Linux by instantiating at run-time one *GLManager* component for each desired display. Furthermore *GLManager* can render on texture and produce the result through a port allowing the streaming of the visualization scene or using CoCoMR as the input for some computer vision algorithms that requires the synthetic rendering of the estimated entity (e.g. hand's pose).

Camera. The *GLCameraManager* peer, one per *GLManager* at the moment, is in charge of managing the OpenGL camera, specifying the initial position of the camera through CoCo attributes and the type of camera through additional peers. The camera system supports oblique projections because they emerge in the common situation of head-tracking systems with precise co-location as in the encountered system work [2], or multi screen systems [19]. Camera controllers are also expressed as peers: first person shooter style camera (*FPSCamera* peer) and arcball camera (*ArcBallCamera* peer). Camera can be moved either using mouse and keyboard or by sending the desired position to the *GLCameraManager* dedicated port.

Camera Images. To render images provided by cameras two peers are available. *GLImage*: renders a 2D image in the background of the virtual world. The image is scaled to fit the resolution of the window. *GLRGBDImage* renders the 3D scene obtained from the *MeshReconstructor* component. In case of very noisy meshes it supports the possibility to average the position of the mesh points among multiple sequential frames. Furthermore with the support of a geometric shader it is possible to clean the scene removing the big triangles that connect objects far from each others.

3D Objects. The *GLEntity* peer is used to render any mesh into the 3D world. It supports all the standard formats and provides a set of basic shaders to support textures and lights. *GLEntity* exposes several attributes to set the object

initial pose and scale, the eventual color if not present in the mesh file and the possibility to run a subdivision algorithm on the object surface. *GLEntity* can be further specialized associating custom peers to it. A peer, to be supported, has to expose an operation named *preRender* that takes in input a pointer to the *GLEntity* object. The operation is called by the component before the OpenGL draw function and can be used to modify the standard behavior of the virtual object. For example it could be possible to alter its color according to external information, to change the mesh shape or to modify the pose.

The display module also supports the rendering of Universal Robot Description Format (URDF) objects from ROS through the *GLUrdf* peer. This feature is very useful when performing robot teleoperation because it allows to easily check that the camera mounted on the robot and the robot itself are correctly registered. Details of registration are provided in the following section. Furthermore it allows to have a clear idea of the robot pose when the camera only focuses the end-effectors. It can also be used to simulate a robot in a pure virtual environment.

4.4 ROS Interface

The integration and interoperability with ROS is a mandatory requirement when developing robotic applications. ROS has become the de facto standard in robotics and many vendors provides the control software of their devices directly as ROS nodes. Thanks to the simplicity and versatility of CoCo it is very straightforward to transform a component so that it can be used as a bridge between ROS and CoCo. To do so the user has to simply create a CoCo component inside a standard ROS package and compile it as a library. In this way the component can declare a *ros::NodeHandle* object and use it to register or publish in topics. Received data can be transformed to be exchanged trough CoCo ports. When an application contains a ROS component a different launcher has to be used, that is a ROS node embedding the same functionalities and the same behavior of the standard launcher.

5 Transformations

Transformation between reference frames are a fundamental element of any MR application in particular when multiple image sources are used together with other tracking devices. CoCo approaches the problem by providing a general **transformation graph** that is orthogonal to the other graphs of the CoCo structure (components and scheduling). The nodes of the graphs are reference frames that are connected via edges that express relative poses.

The components publish transformations between pairs of frames and these are used to update the internal graph. The transformation query mechanism is based on a path resolution over the graph: the transformation between two frames corresponds to one of the paths between them in the transformation graph. The path associated to each query is cached for future requests avoiding

repetition of the path resolution and it is updated every time one of the edges change.

The transformation information flows inside the computational graphs via the CoCo ports as long as the rest of the data. The *TransformationInterface* is a single component in the computational graph that receives all the transformations between frames, updates the internal graph, and propagates information to the other components. Two type of propagation are supported, via callback or operations. Each component can register a callback function to a query, that is, when any of the edges in the query's path changes and after the computation of the new transformation all the registered callbacks are called receiving the new result. In addition *TransformationInterface* exposes an operation that can be called by any other component to retrieve the transformation between two frames. Internally a readers-writer lock is implemented to avoid concurrency problems.

In contrast to ROS TF2 that is fully dynamic, CoCo allows to declare in advance the structure of the graph, in particular describing the nature of the edges that can be static, or dynamic, or projected from the 6DOF space to a single axis. The declarative feature is useful during development for controlling the different transformations, and, at the same time, at run-time, to optimize the execution of the internal processing of the graph.

5.1 Robotics Support

Single axis is specifically useful for robotics or human tracking in which the update value is the single joint value and the 6DOF transformation is the resulting application of a Denavit-Hartenberg transformation or equivalent. URDF provides a hierarchical, joint based, description of the robot and it can be directly loaded into the transformation graph.

5.2 Registration

The declarative approach allows for the automatic support for the registration between two disjoint frames (A,B). The registration is obtained by the introduction of a new temporary path between A and B that is produced by an external source, e.g. a chessboard or a fiducial marker that is not present during the execution. Multiple measures of the transformation between A and B are accumulated during the registration phase, and then the final value is obtained by 6DOF averaging.

This approach has been successfully used for registering the camera of a robotic head wrt the rest of the robot body by placing a fiducial marker over the robot arm and moving the arm (Fig. 2). In practice the approach is quite flexible and can take into account multiple temporary paths.

5.3 Diagnostics

A fundamental aspect of the CoCo management of transformation is diagnostics. First it is possible to serialize the graph over JSON at any time and to generate

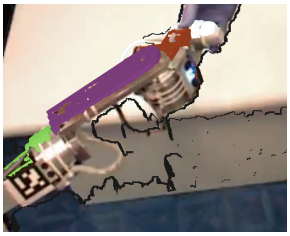

Fig. 2. Result of the registration of the Kinect with the robot kinematics. The image part is a rendering of the Kinect point cloud, while the colored parts are meshes from the URDF controlled by robot joints. Finally in the lower left the calibration fiducial marker is visible.

a graphical representation of the graph using graphviz. Secondly it is possible to access the graph via a Web based REST interface in which all frames and the graph in general are accessible. The interface supports also Websockets for continuous streaming of transformations.

This is specifically useful for interfacing CoCo with other frameworks such as WebGL based frameworks or Unity.

6 Case Study

The CoCoMR libraries have been used for several applications both involving haptic devices and robot tele-operation. In the following two examples will be provided, showing the component structures and the performance measurements.

6.1 Visuo-Haptic Application

This section will describe how CoCoMR has been used in an application [2] for virtual remote palpation examination. Figure 3 shows the different hardware components involved in the setup. The whole system comprehends a Kinect 360 streaming via *StreamingServer* the RGB-D image of a mannequin laying on a table and representing a patient; a 3-DOF haptic device; the Leap Motion sensor to track the position of the hand and a 3D screen to visualize the remote scene. The user moves the hand below the screen and his movements are captured by the Leap Motion. The hand's pose is used to display on screen the 3D model of an hand superimposing it on the remote scene obtained through TCP (ZeroMQ) from the Kinect. The position of the hand is also sent to *HapticDeviceInterface*, an ad hoc component used to interact with the haptic device. *HapticDeviceInterface* communicates with a Simulink module performing the low level control of the device. The module uses the hand position to place the end-effector exactly below the user hand; in this way when the user lowers the hand and touches the virtual surface it will find the end-effector of the device providing force feedback based on the indentation with the virtual remote scene. To improve the

visual feedback the virtual hand gradually shifts its color towards a red shade the deeper the end-effector is inside the surface; this is performed thanks to the *UpdateColor* peer assigned to the *GLEntity* component responsible for the virtual hand.

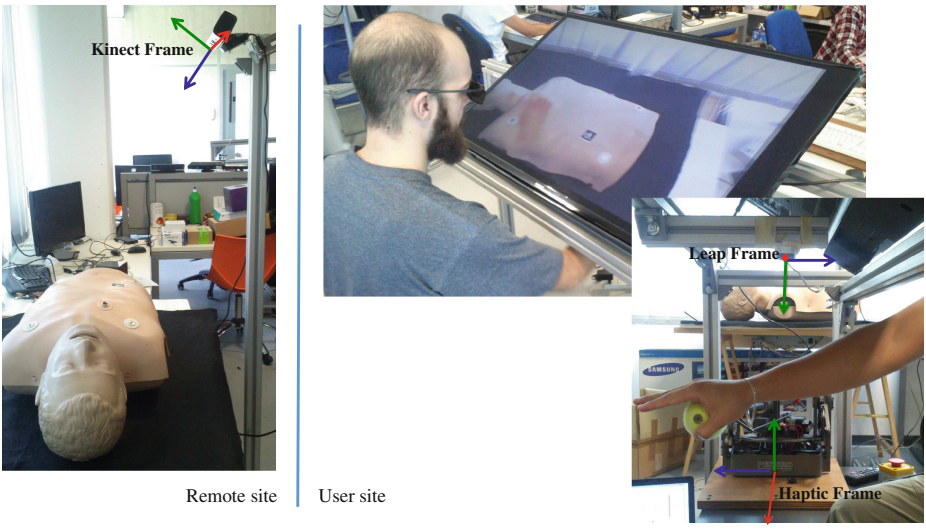

Fig. 3. The virtual palpation system. The hardware components are shown along with their reference frames.

The system is composed of multiple components each one running at different rates: 60 Hz the graphics, 30 Hz for reading images from the Kinect streaming, 100 Hz the LeapMotion and 1 kHz the haptic interface. As shown in Fig. 3 each device produces its data in its own reference system; to allow the various component to communicate the pose informations are all gathered by the *TransformationInterface* component that provides to the other components the position information in the desired reference frames. *TransformationInterface* is also in charge of performing the calibration between the different reference frames, in particular between the remote virtual scene, the hand pose and the device end-effector pose. This is done by putting a marker on the belly of the remote mannequin and its pose is associated with the one of the hand from the Leap Motion, the user has also to grasp the haptic device end-effector so that it is possible to assume that the three reference systems are aligned.

Figure 4 shows all the components and peers involved in the application. The components are divided per activity and the scheduling policy is shown. Given they high number of connections the lines have not been drawn but the assumption is that connection with the same name are linked. This apart from the pose transformations data that all flow trough *TransformationInterface*.

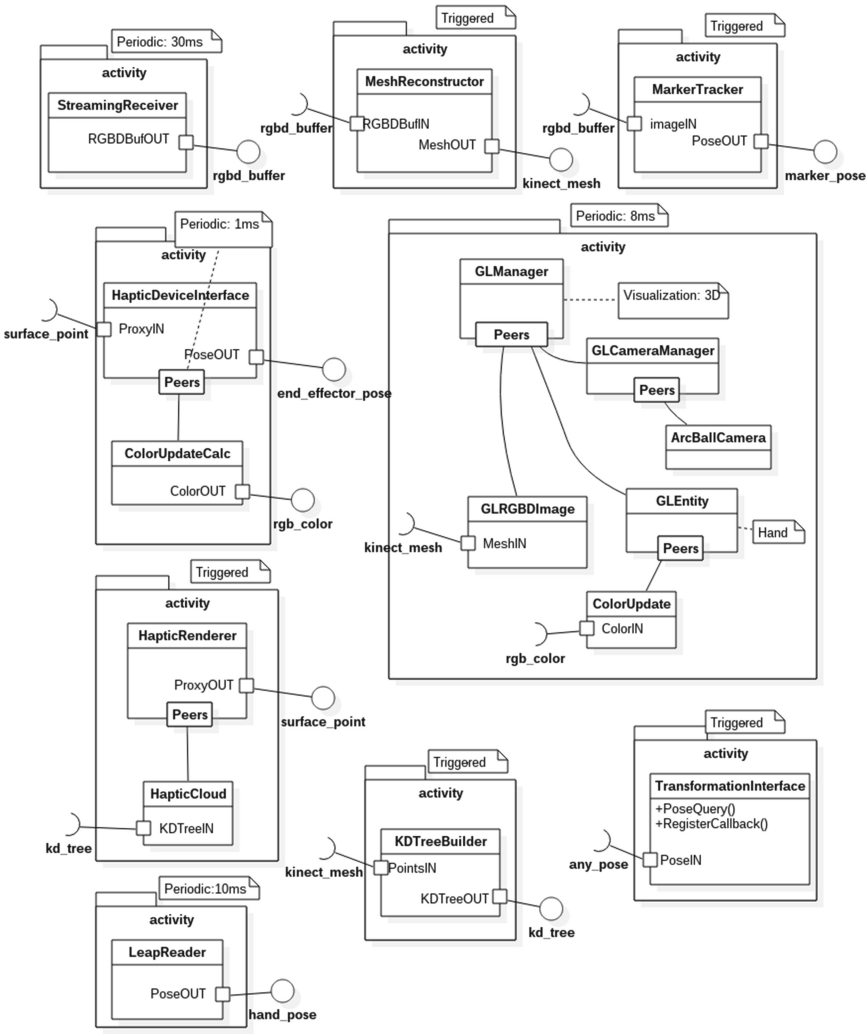

Fig. 4. CoCoMR components involved in the visuo-haptic application. Port with the same name type are connected to each other. Every poses pass through *TransformationInterface* to be transformed in the correct reference system.

Performance. The application run on an Intel PC (Core i7 4770R 3.2 GHz, 8 GB RAM, embedded GPU) running Ubuntu Linux 14.04. The estimated sensor to display average latency is 75 ms, computed after synchronizing the robot and graphics computers with the Precision Time Protocol (PTP). The *HapticRender* component was able to calculate the surface proxy position at 1 kHz while some optimization were required to manage the kd-tree update at 30 Hz. The *KDTreeBuild* component performed a filtering on the received points based

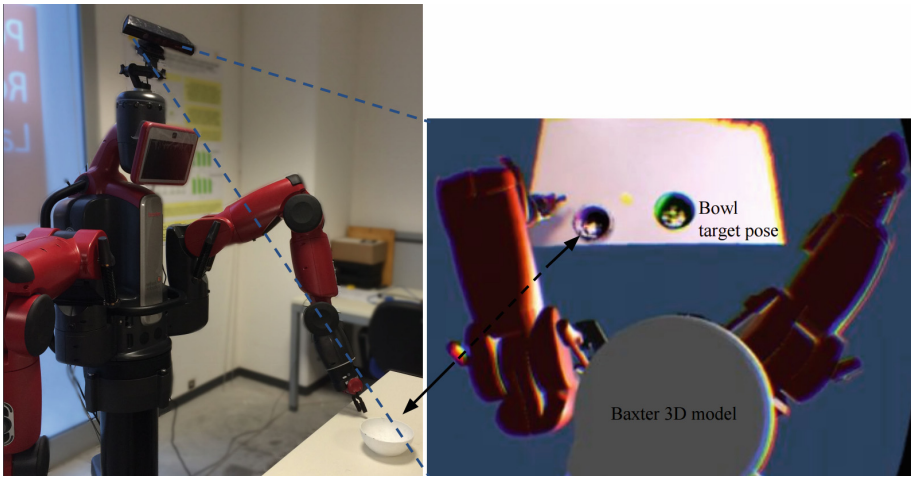

Fig. 5. The Baxter robot, on the left, with the Kinect mounted on the head. On the right the image displayed in the Oculus Rift

on the distance from the camera to reduce the mesh size and be able to run in less than 33 ms. Another possible solution to speed up the computation is to use a GPU-based algorithm.

6.2 AR for Teleoperation Application

This section discusses how the CoCoMR components have been combined to create an augmented visual feedback during teleoperation of a remote robotic device. The setup that will be described is part of an application [3] aimed at providing a set of tools to teleoperate a robot in industrial tasks. In this application a Baxter robot's arm is controlled by an operator's movements captured with a wearable device. The operator wears an Oculus Rift DK2 showing the remote scene, captured by a Kinect 360 placed on top of the Baxter head, plus virtual objects to help him in the tasks completion, see Fig. 5.

The task to be performed by the operator is to grasp a bowl and move it to a target position. The bowl to be picked is identified by a colored mesh that overlay the real bowl in the rendered image. The virtual mesh helps the user to identify the object which is of the same color of the table; in addition the mesh color changes the closer the robot end-effector is to the bowl easing the task. The target position, where to move the object, is indicated by another mesh of the bowl allowing the operator to be more precise in the placement. The visual feedback is augmented also by the 3D model of the Baxter robot, allowing the user to know the position of the end-effector even when it exits from the Kinect field of view. In addition it helps the operator predicting the remote robot arm movements and its pose in the environment given the high movement latency of the robot arm.

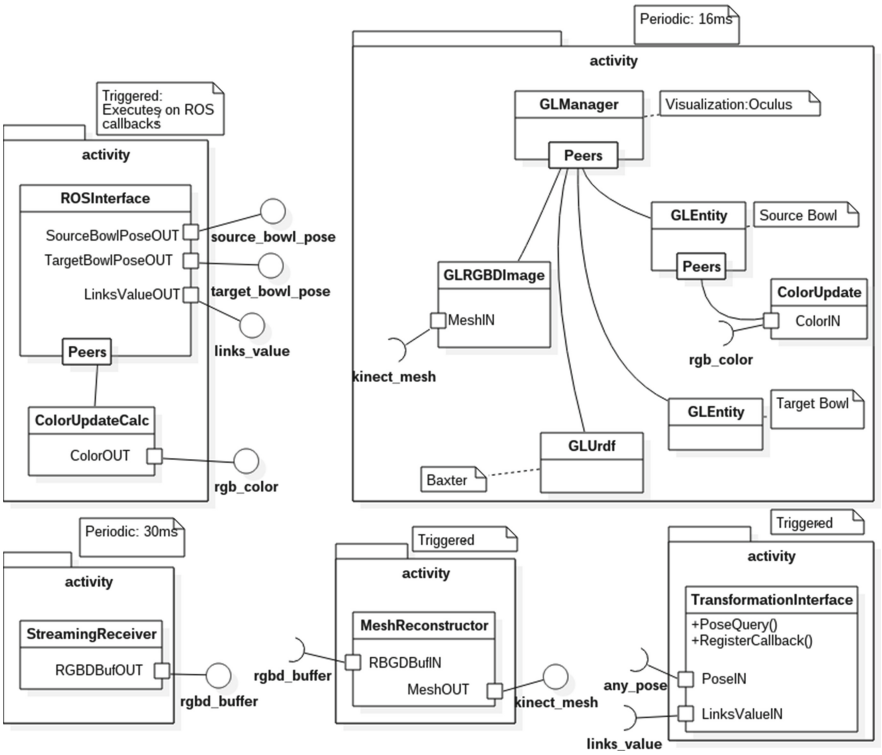

Fig. 6. CoCoMR components involved in the teleoperation application. Port with the same name type are connected to each other. Every pose passes through *TransformationInterface* to be transformed in the correct reference system.

The components structure is showed in Fig. 6. Several of the components are the same of the ones used for the application described before, proving the reusability capability of the framework. The *ROSInterface* component collects from the ROS topics the information regarding the Baxter joints position and the bowl location, obtained from the *object_recognition_tabletop* package[2], and delivers them to the *TransformInterface* component. The *UpdateColorCalc* peer calculates the distance between the robot end-effector and the bowl to correctly update its color. There are two *GLEntity* peers, one for each rendered bowl and a *GLUrdf* peer to display the Baxter 3D model.

Performance. The application runs on quad-core Intel i7 CPU (2.3 GHz) and a NVIDIA GeForce GT 650 MacBook Pro. The estimated sensor to display average latency is 89 ms, computed after synchronizing the robot and graphics computers with the Precision Time Protocol (PTP). Compared with the previous application the computational load is lower and the execution is smooth also on a laptop.

[2] http://wg-perception.github.io/tabletop/.

7 Conclusions

The diffusion of multicore platform is requiring new approaches for organizing computation in particular in demanding tasks such as visuo-haptic mixed reality applications. The paper has presented the CoCo framework as an approach for addressing these challenges. The organization based on the three graphs allows to tackle these challenges by providing flexibility and developer control while hiding several low-level aspects.

There are several aspects that can be investigated starting from the present work. One aspect is related to the analysis and optimization of the scheduling resulting from the data flow and the user-defined schedule, this could give space for the identification and reuse of common patterns and adaptation to a new machine with different number of cores.

The second aspect is instead related to the support of GPUs in the data flows. The most promising solution is based on CUDA mainly due to the number of libraries in the vision and simulation world that provide optimization for such library. The port mechanism could be easily extended for supporting CUDA pointers, but it is needed to introduce an automatic mechanism for transferring the content from/to the GPU when a connection is created between a GPU-bound port and a CPU-bound port.

This proposed solution, currently under investigation, is clearly limited to single-process architecture and it is also subject, in terms of scheduling, to the policies of the CUDA driver.

Incidentally it is worth discussing that the extension of CoCo to multiprocess is viable, provided an efficient mechanism for data exchange between components in different processes is found. ZeroMQ is an optimal candidate, while reduced or no-serialization could be employed for efficient data exchange.

Acknolwedgments. This work has been carried out within the framework of the European Project REMEDI, grant number 610902, and the Tuscany Regional Project TAUM.

References

1. Ruffaldi, E., Brizzi, F., Filippeschi, A., Avizzano, C.A.: Co-located haptic interaction for virtual usg exploration. In: Proceedings of IEEE EMBC, pp. 1548–1551 (2015)
2. Filippeschi, A., Brizzi, F., Ruffaldi, E., Jacinto, J.M., Avizzano, C.A.: Encountered-type haptic interface for virtual interaction with real objects based on implicit surface haptic rendering for remote palpation. In: 2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), pp. 5904–5909. IEEE (2015)
3. Peppoloni, L., Brizzi, F., Ruffaldi, E., Avizzano, C.A.: Augmented reality-aided tele-presence system for robot manipulation in industrial manufacturing. In: Proceedings of the 21st ACM Symposium on Virtual Reality Software and Technology, pp. 237–240. ACM (2015)

4. Burns, D., Osfield, R.: Tutorial: open scene graph a: introduction tutorial: open scene graph b: examples and applications. In: Proceedings of the IEEE Virtual Reality, p. 265 (2004)
5. Carrozzino, M., Tecchia, F., Bacinelli, S., Cappelletti, C., Bergamasco, M.: Lowering the development time of multimodal interactive application: the real-life experience of the XVR project. In: Proceedings of the 2005 ACM SIGCHI International Conference on Advances in Computer Entertainment Technology, pp. 270–273. ACM (2005)
6. Behr, J., Bockholt, U., Fellner, D.: Instantreality – a framework for industrial augmented and virtual reality applications. In: Ma, D., Fan, X., Gausemeier, J., Grafe, M. (eds.) Virtual Reality & Augmented Reality in Industry, pp. 91–99. Springer, Heidelberg (2011)
7. Figueroa, P., Bischof, W., Boulanger, P., Hoover, H., Taylor, R.: InTml: a dataflow oriented development system for virtual reality applications. Presence **17**(5), 492–511 (2008)
8. Allard, J., Gouranton, V., Lecointre, L., Limet, S., Melin, E., Raffin, B., Robert, S.: FlowVR: a middleware for large scale virtual reality applications. In: Danelutto, M., Vanneschi, M., Laforenza, D. (eds.) Euro-Par 2004. LNCS, vol. 3149, pp. 497–505. Springer, Heidelberg (2004)
9. Conti, F., Morris, D., Barbagli, F., Sewell, C.: Chai 3d (2006). http://www.chai3d.org
10. Ruffaldi, E., Frisoli, A., Gottlieb, C., Tecchia, F., Bergamasco, M.: A haptic toolkit for the development of immersive and web enabled games. In: ACM Symposium on Virtual Reality Software and Technology (VRST), pp. 320–323. ACM (2006)
11. Eck, U., Sandor, C.: Harp: a framework for visuo-haptic augmented reality. In: 2013 IEEE Virtual Reality (VR), pp. 145–146, March 2013
12. Bruyninckx, H.: Open robot control software: the orocos project. In: Proceedings of the 2001 IEEE International Conference on Robotics and Automation, ICRA 2001, vol. 3, pp. 2523–2528. IEEE (2001)
13. Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., Wheeler, R., Ng, A.Y.: ROS: an open-source robot operating system. In: ICRA Workshop on Open Source Software, vol. 3(3.2), p. 5 (2009)
14. Seichter, H., Looser, J., Billinghurst, M.: Composar: an intuitive tool for authoring AR applications. In: Proceedings of the 7th IEEE/ACM International Symposium on Mixed and Augmented Reality, pp. 177–178. IEEE Computer Society (2008)
15. Salisbury, K., Conti, F., Barbagli, F.: Haptic rendering: introductory concepts. IEEE Comput. Graph. Appl. **24**(2), 24–32 (2004)
16. Ruffaldi, E., Morris, D., Barbagli, F., Salisbury, K., Bergamasco, M.: Voxel-based haptic rendering using implicit sphere trees. In: Symposium on Haptic Interfaces for Virtual Environment and Teleoperator Systems, Haptics 2008, pp. 319–325, March 2008
17. Ruspini, D.C., Kolarov, K., Khatib, O.: The haptic display of complex graphical environments. In: Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques, pp. 345–352. ACM Press/Addison-Wesley Publishing Co. (1997)
18. Salisbury, K., Tarr, C.: Haptic rendering of surfaces defined by implicit functions. ASME Dyn. Syst. Control Div. **61**, 61–67 (1997)
19. Cruz-Neira, C., Sandin, D.J., DeFanti, T.A.: Surround-screen projection-based virtual reality: the design and implementation of the cave. In: Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques, pp. 135–142. ACM (1993)