

CamiTK: A Modular Framework Integrating Visualization, Image Processing and Biomechanical Modeling

Céline Fouard, Aurélien Deram, Yannick Keraval
and Emmanuel Promayon

Abstract In this paper, we present CamiTK, a specific modular framework that helps researchers and clinicians to collaborate in order to prototype Computer Assisted Medical Intervention (CAMI) applications by using the best knowledge and know-how during all the required steps. CamiTK is an open-source, cross-platform generic tool, written in C++, which can handle medical images, surgical navigations and biomechanical simulations. This paper first gives an overview of CamiTK core architecture and how it can be extended to fit particular scientific needs. The MML extension is then presented: it is an environment for comparing and evaluating soft-tissue simulation models and algorithms. Specifically designed as a soft-tissue simulation benchmark and a reference database for validation, it can compare models and algorithms built from different modeling techniques or biomechanical software. This article demonstrates the use of CamiTK on a textbook but complete example, where the medical image and MML extensions are collaborating in order to process and analyze MR brain images, reconstruct a patient-specific mesh of the brain, and simulate a basic brain-shift with different biomechanical models from ANSYS, SOFA and ArtiSynth.

C. Fouard (✉) · A. Deram · Y. Keraval · E. Promayon
UJF-Grenoble 1/CNRS/TIMC-IMAG, UMR 5525, 38041 Grenoble, France
e-mail: Celine.Fouard@imag.fr

A. Deram
e-mail: Aurelien.Deram@imag.fr

E. Promayon
e-mail: Emmanuel.Promayon@imag.fr

Stud Mechanobiol Tissue Eng Biomater
DOI: 10.1007/8415_2012_118
© Springer-Verlag Berlin Heidelberg 2012

1 Introduction

Soft-tissue biomechanical modeling in the context of Computer Assisted Medical Intervention (*CAMI* thereafter) is a complex and multi-disciplinary field. The general objective of intra-operative simulation requires a patient specific model that can be built only from medical image analysis and reconstruction, image or mesh registration and finally efficient and trustworthy simulations. CAMI research requires the collaboration of experts in several fields as various as medicine, computer science, mathematics, instrumentation, signal processing, mechanics, modeling, automatics, or optics. As shown in Fig. 1, CAMI research can be divided into three steps requiring very different expertise:

- *Perception* to acquire data on the patient, requiring sensor development and calibration, signal processing and analyzing, 2D/3D or even 4D images reconstruction, data processing (e.g., image segmentation) and analysis;
- *Reasoning* to extract and fuse information from several sensors, acquisition times, patients, between patients and models, etc. and which implies, among others, anatomic, biomechanical and functional modeling of patients as well as surgical intervention modeling, results analysis and prediction;
- *Action* to help procedures carried out on the patient, possibly using image-guided and/or surgical robotic technologies.

Even though this book mainly focuses on the *reasoning* part, soft tissue modeling is a key element to a bigger picture and needs other fields to be efficiently developed, used and validated. For research to be efficient, scientists must share not only their knowledge but also their know-how. The first is generally performed through publications, conferences and meetings. However, know-how is difficult to transmit outside lectures or tutorials. In addition, the use of specific computer software for each field can become a hindrance to share data and know-how. Data have to be converted several times, specific parameters are not visible by other specialists and prototyping a new concept or algorithm requiring several fields often becomes an obstacle race.

CamiTK offers a common framework for many fields of CAMI so that scientists from one field can develop their own expertise and use the tools developed by other experts without changing application or data structure. This is achieved through extensibility and modularity of the framework which allows every user to develop their own piece of software and use any other developed pieces thanks to the CamiTK abstraction layer.

1.1 Related Work

Different types of applications, libraries, and more generally frameworks are today available for scientists, most of them are open-source or based on collaborative projects. The lists given here are in no particular order and are not exhaustive but

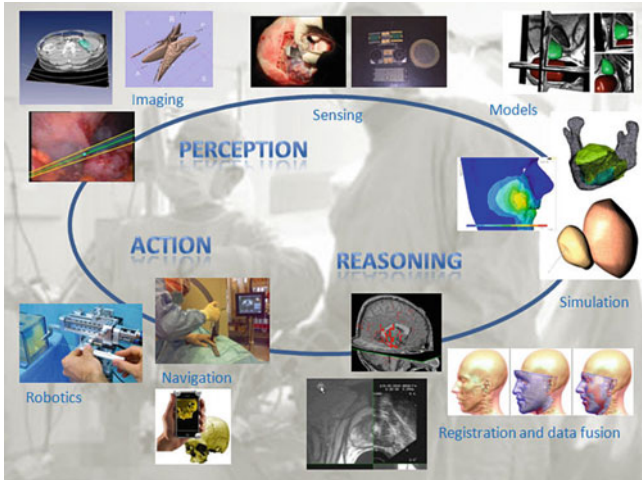


Fig. 1 CAMI research domains includes perception, reasoning and action. This figure shows some examples of research projects lead by members of our team. Courtesy of Jocelyne Troccaz

try to give the most popular non-defunct choices. The software websites can easily be found using any search engine. They can be categorized into different area, depending of their main functionality: data visualization (VTK, Paraview, voren), medical image analyzing or surgical planning (ITK, Analyze, Medical Imaging Interaction Toolkit, MeVisLab, DeVIDE, SciRun, amira, 3D Slicer, GIMIAS, MedINRIA/vtkINRIA3D, CreaTools), surgical navigation (IGSTK), biomechanical simulation (ANSYS, GiPSi, SOFA, ArtiSynth, see also the corresponding chapters in this book), generic viewers and user interaction (Common Toolkit).

Most of the open-source medical image analysis software are based on three key open-source component libraries: VTK for data visualization, ITK for medical image analysis, and Qt for the user interaction. These software systems are developed to fulfill needs in their main domain and offer a highly specialized vision of a specific CAMI scientific field. This specialization is a big advantage when the targeted application depends on one or two fields (e.g., image segmentation and organ surface reconstruction, or biomechanical simulation). Most of them are also workflow oriented, i.e., the data are transformed step by step through processing or visualization filters. The workflow may be conveniently displayed and modified interactively. This is a very efficient way of organizing medical image processing application. However, when trying to solve a problem that includes many different facets of CAMI research, both can be a disadvantage. Interoperability between very specific data structures in highly specialized libraries or frameworks is difficult to achieve. Static workflows are not always possible: algorithms might have to cooperate in a more complex way. CamiTK proposes to address these issues by introducing some abstraction of the software components, viewers and algorithms in order to ease the full collaboration between CAMI fields.

Table 1 CamiTK factsheet

URL	http://camitk.imag.fr
License	The core and distributed extensions are available under the terms of the LGPL v3
Language	C++
Size	≈ 200 classes, 35,000 Single Lines Of Code
Some Metrics	McCabbe cyclomatic complexity <12 (moderate risk), SLOC/Comments ≈ 1.5
Components	6 component extensions are distributed: DICOM, Vtk image, Vtk mesh, Itk image, PML, MML
Documentation	API documentation, user tutorials, developer tutorials, mailing lists
Dependencies	Qt, VTK, optional: libxml2, xerces-c, ITK, IGSTK
Download	Windows setup, Ubuntu package, source package, svn (through the collaborative forge)

1.2 Contributions

A common environment for CAMI requires data management, algorithms on data, Human-Machine Interface or Graphical User Interface (GUI) and scientific data visualization. As it would be problematic to impose one operating system to research scientists of various fields and area of competences, the choice has been made for CamiTK to be multi-platform. This requires to choose basic libraries to also be multi-platform and multi-compilers. For those reasons, we choose to base CamiTK on two multi-platform open source libraries: Qt for GUI and Vtk for scientific data visualization. Those libraries are mature, rapidly evolving and benefit from a large community of developers. CamiTK is therefore mainly implemented in C++, a very efficient object-oriented language. Object-Oriented programming paradigms, including inheritance and polymorphism, data abstraction, encapsulation and interface and numerous design patterns are used in CamiTK in order to achieve a clean and extensible architecture. Other technical details and how to get your version of CamiTK are given in Table 1. CamiTK is composed by CamiTK core and CamiTK extensions (see Fig. 2). CamiTK core includes an abstract layer, useful interaction widgets and viewers, as well as some default data component implementations (see Sect. 2). CamiTK plugins or extensions hold concrete implementations for specific CAMI fields' data or algorithms. While CamiTK core is distributed under the LGPL open-source license, the license of each extension can be freely chosen by the developer.

CamiTK extensively uses Kitware CMake, CPack, and CTest cross-platform suite in order to facily manage the build process on all targeted platforms, being Linux or Microsoft Windows, gcc or MS Visual C++ (a CamiTK MacOS version can also be build but is not yet supported by our development team).

CamiTK's general design is inspired by component-based software engineering (CBSE). Instead of building an application by adding features on existing code, CBSE intends to build a software by integrating, arranging or assembling pre-built software components. CamiTK shares the CBSE guiding principles:

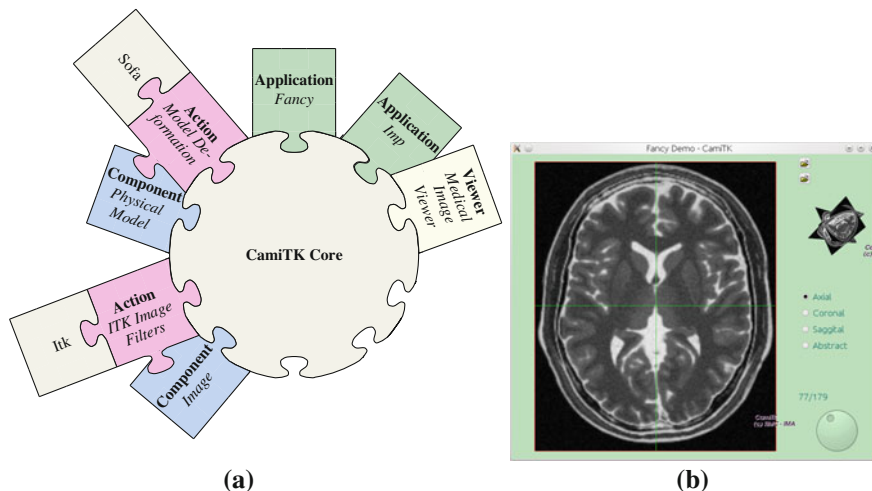


Fig. 2 CamiTK architecture: Several existing and collaborating modules. **a** CamiTK core viewed as a collaboration platform. **b** An example of a provided application extension

- reuse and do not reinvent the wheel,
- assemble pre-built modules and components instead of adding codes where needed.

This can be very efficient when component standard exists. Unfortunately no such standard exists in CAMI field. CamiTK core therefore introduces simple light standards for software modules that can be found in CAMI application. These light standards are not specific to any sub-domain such as imaging, robotics, surgical navigation or simulation. To conform to these standards, a software component simply has to inherit from one of the four CamiTK extensions. The four possible types of extension are *component* (data management), *viewer* (data visualization), *action* (algorithms), and *application* (the GUI). Figure 2 shows some examples of existing extensions. CamiTK source code includes a list of commonly required extensions. Provided *component* extensions include support for several formats of volume image and mesh management (e.g., DICOM/Itk/Vtk/Analyze images and Vtk/VRML/Obj/PML/SOFA scenes and meshes). Provided *viewer* extensions include a classical medical image viewer with four zones: one for axial plane display, one for coronal one for sagittal plane display and the last zone for 3D view. Among existing *action* extensions, we developed volume image filtering algorithms using ITK, model deformation using SOFA, ArtiSynth or ANSYS, and 3D mesh reconstruction from stereoscopic video. Some basic *application* extensions are also provided with CamiTK.

Section 2 details CamiTK framework and extension principles, Section 3 focuses on biomechanical modeling in CamiTK. Section 4 shows a whole CAMI textbook example that target the simulation of the brain shift, from volume image visualization to model construction, biomechanical simulation and data fusion.

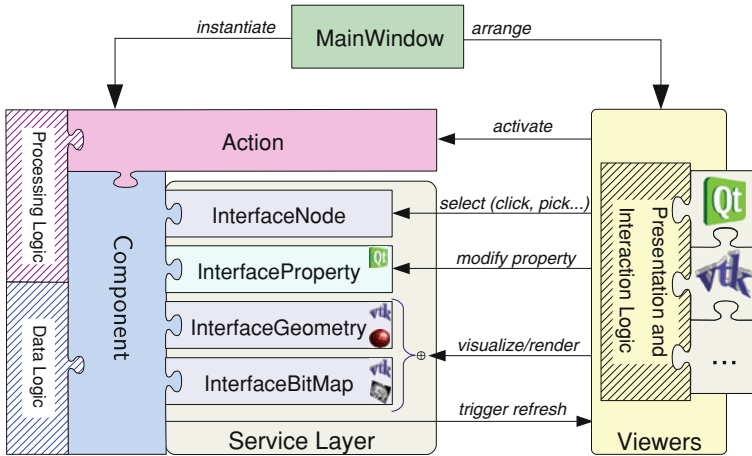


Fig. 3 CamiTK architecture overview

2 CamiTK Architecture and Key Concepts

2.1 Overview

CamiTK core architecture follows the *Model-View-Presenter* design pattern (MVP thereafter), a generalization of the *Model-View-Controller* design pattern where the presenter acts more like a supervising controller.

In classical software architecture, MVP design pattern helps to divide the logic in two: domain logic and application logic. Domain logic, the Model, is the part of the data structures and algorithms that is directly linked with the research field or sub-domain. Application logic is the part generally used to build a GUI to present and interact with the domain logic data and algorithms. Application logic generally aggregates the View and Presenter parts of MVP. CamiTK core lets the researcher concentrate on the domain logic and eases the development of the application logic by following CBSE principles. To do so, a software based on CamiTK assembles what we called *viewers* (application logic) than are used to present and interact with *components* (i.e., the data, the data logic), and *actions* (i.e., the algorithms, the processing logic). *Viewers* and *components* are bound together by a *main window* to build an interactive application where actions on the components can be triggered. Figure 3 presents an overview of the CamiTK architecture.

CamiTK furthermore distinguishes the domain logic in data logic (i.e., the management of dynamic or static data structures) and the processing logic (i.e., the modification and transformation of data). This is an essential distinction as it facilitates software testing (unit and integration testing) and improves the segmentation of tasks (it is easy to define precise and well segmented software development tasks). The data logic is handled by what we called *components*

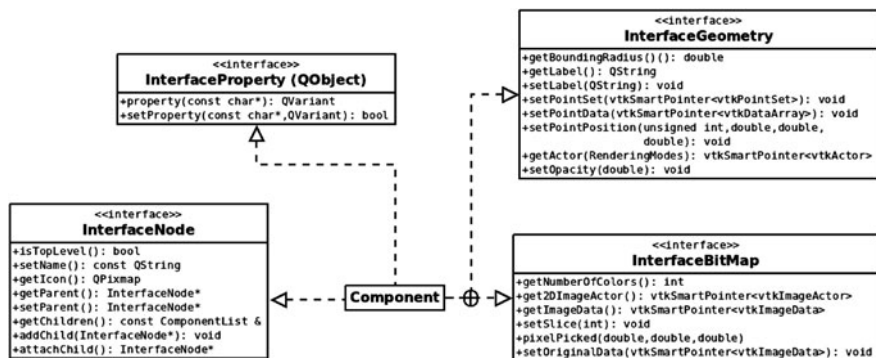


Fig. 4 CamiTK service layer overview

(generalized by the `Component` class), while the processing logic is handled by what we called *actions* (generalized by the `Action` class). In order to easily glue together viewers, components and actions, CamiTK uses a Service Layer Pattern [6]. The service layer is classically linked to the viewers by an Observer Pattern.

The service layer simplifies the creation of new components and actions by generalizing concepts and predefining generic behaviors. The service layer is an overlay of the `Component` class (see Fig. 3). The `Component` class implements the four interfaces: `InterfaceNode`, `InterfaceProperty`, `InterfaceGeometry` and `InterfaceBitMap`, see Fig. 4. Each interface describes the capabilities of a component without committing it to use a specific implementation. It can also be thought as contracts where the component is the provider of four services. On the other side of the MVP, the viewers are consumers of the services. The viewers interact with the components without knowing exactly which kind of data is managed by the component or how it reacts to a specific call.

Each interface lists the methods to be implemented and describes the corresponding capabilities of components:

- `InterfaceNode` guarantees that each component has a name and can be associated with other components in a tree-like hierarchical structure or a dependency graph.
- `InterfaceProperty` guarantees that components are associated with a list of properties and can be identified in the class hierarchy of component families.
- `InterfaceGeometry` defines the ability of a component to provide a 3D geometric representation of some kind (mutually exclusive with `InterfaceBitMap`). This is a convenient wrapper and simplifier for data that can be represented by a `vtkPointSet`.
- `InterfaceBitMap` defines the ability of a component to provide a representation as a bit map of a given thickness (mutually exclusive with `InterfaceGeometry`). This is a convenient wrapper and simplifier for data that can be represented by a `vtkImageData`.

A component has a concrete implementation of either an `InterfaceGeometry` or an `InterfaceBitmap` or none.

The `Action` class generalizes the notion of component processing. An action applies a given algorithm to a given component or list of components. Generally, viewers trigger actions on the currently selected components.

CamiTK viewers handle all the presentation and interaction logic (the `View` and `Presenter` part of MVP). The CamiTK viewers are using either Qt widgets or VTK renderers, but are not limited to that. Viewers manage the user interactions and map them into calls to the interface methods of the components. For example, viewers are aware of the current selection, that is the list of currently selected data component, and can be used to add or remove components to the selection (e.g., in the explorer viewer clicking on component while pressing the `Ctrl` key adds it to the selection). A specific interaction in a specific viewer is translated to a specific call of one of the method of the service layer interface. This method is implemented in the `Component` class or specialized in one of its subclass.

In order to build a CamiTK application, the developer may subclass `MainWindow`. `MainWindow` is a kind of super `Presenter`: it instantiates and organizes the viewers in an application GUI. It also sets up menus or widgets that allow the user to open, close, save or export components from files or connected devices.

Two extensions mechanisms are possible in CamiTK: software plugins (for component and action extensions) and simple inheritance (for viewer and main window). Adding an extension mainly consists in writing a new C++ class derived from a CamiTK core class. This can generally be done in 10–50 Single Lines Of Code. Core extension classes already define a lot of default behaviors, and each of them can be redefined by just overriding the corresponding method.

CamiTK also follows some rules of test-driven development by providing a default unit test architecture for its components. An automated quality software process is currently being set up for CamiTK thanks to Kitware CDash.

Sections 2.2–2.5 describe the four types of extensions in details for a developer or technical reader. Section 2.6 presents use-cases for the different types of extensions.

2.2 Viewer

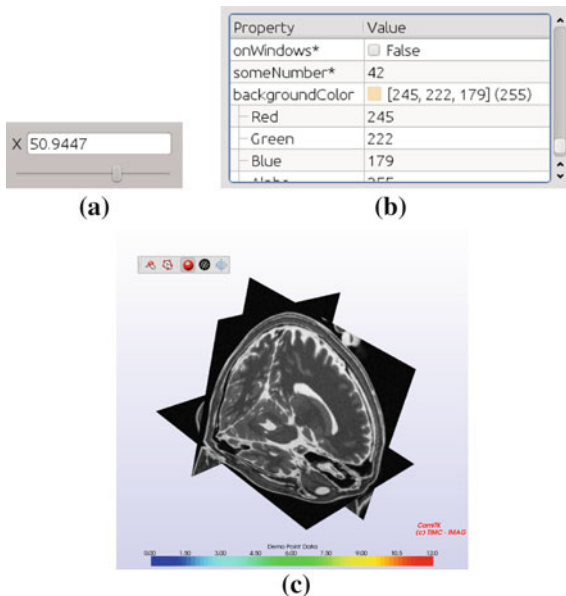
CamiTK viewers are automatically aware of instantiated data components, they can be considered as content managers that present the data and allow the end users to trigger actions.

Similarly to `CommonTK`¹ CamiTK defines low-level software components that can be used to build viewers (see Fig. 5). Such low-level components include pure Qt code (e.g., a synchronized slider and text input or a programmer-friendly Qt Property

¹ <http://www.common.tk.org>

Fig. 5 Some pure Qt and pure VTK low-level software components available for building CamiTK viewers.

a Synchronized slider and text Qt widget.
b ObjectController features display and interaction of any Component or Viewer.
c Full featured VTK 2D/3D renderer



controller) as well as pure VTK code (e.g., a 2D/3D full featured renderer that support in particular mouse interaction, camera, lightning, 3D axes, node/facet picking, and screenshots, similar to what can be found in MedINRIA, [15] or Slicer 3D).

CamiTK Viewers offer a specific way of displaying a CamiTK component. The same component can be displayed by any number of viewers, each showing a different representation or facet of the component (see Fig. 6).

CamiTK viewers can have a menu, a widget, a toolbar and some properties. Each of them can be assembled to build a CamiTK `MainWindow`, for example properties can be automatically added in the application settings dialog, toolbar in the main toolbar and menu in the application `View` menu.

Viewers can also be assembled to build other viewers (e.g., the medical image viewer), see Fig. 6e, is composed of three 2D interactive viewers and a 3D interactive viewer.

A specific viewer is the action viewer, which displays the list of possible actions for the currently selected component and triggers the action chosen by the user amongst tagged actions or families of actions.

Another useful viewer is the `Explorer` viewers, see Fig. 6a, which displays the `Components` as a hierarchical tree view.

2.3 Component

The component extension is very important in the CamiTK architecture. A component wraps the domain objects (specific data) so that they can be inserted into

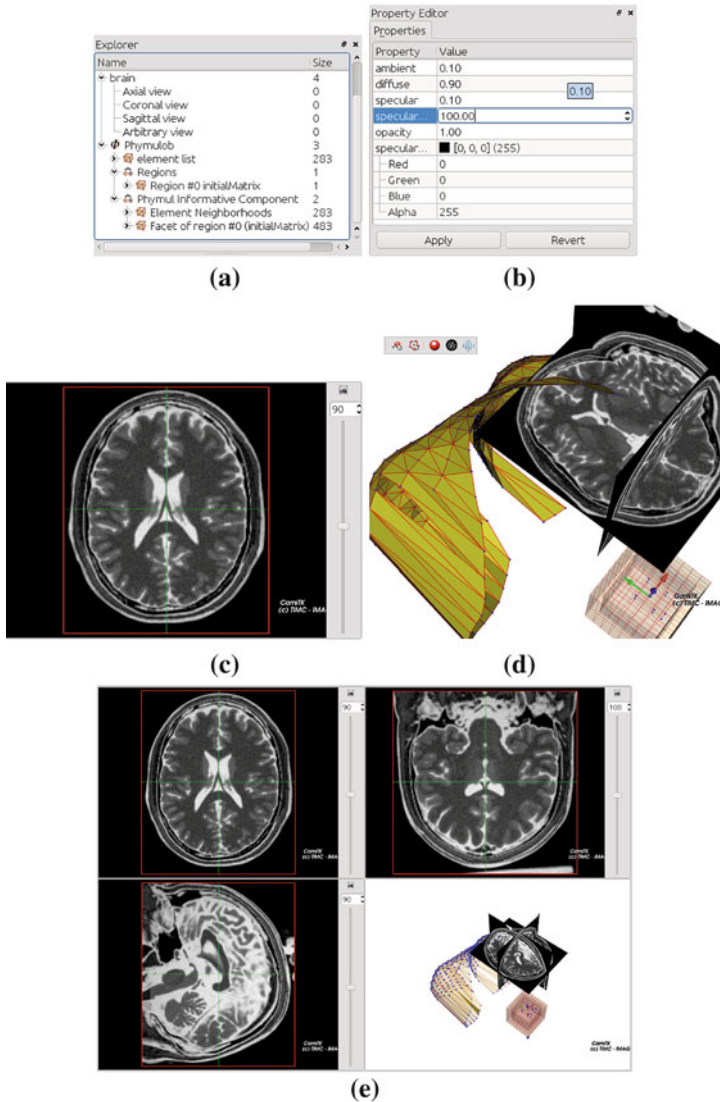


Fig. 6 Main viewers available to build a CamiTK MainWindow. **a** Explorer (data hierarchical tree). **b** Property explorer. **c** Axial, sagittal, or coronal viewer (block in 2D). **d** 3D interactive viewer. **e** Medical image viewer

the viewers and manipulated by the user. CamiTK core class `Component` inherits from all the service layer interfaces (`InterfaceNode`, `InterfaceProperty`, `InterfaceGeometry` and `InterfaceBitMap`). The `Component` class also provides a default implementation for most of the service layer methods for visualization and interaction, as detailed in the remaining of this section.

2.3.1 Node

A component is designed as a simplified *Composite* design pattern: an instance can be an individual component or a composition of components. For example, faces and nodes can be managed by one single component or may be managed individually by dedicated components. Therefore component instances can contain other components, and instances can form a hierarchical tree, each component being either the root or a node. The root of a component tree is specific and is called the *top-level component*. It is the master component that ensures the building of the whole tree. `InterfaceNode` ensures that a component provides an implementation of root and node behaviors. `InterfaceNode` also forces each component to have a name and other optional features like an icon pixmap.

Thanks to the service layer, the `InterfaceNode` may be used in a textual viewer, for example to display the component hierarchical tree (see Fig. 6a), like in a desktop file manager. If the user clicks on the textual representation of the component in a viewer, the latter calls the `setSelected(bool)` method of the component.

Examples of methods already implemented by default include `getParent():InterfaceNode*`, `addChild(InterfaceNode*)`, `isTopLevel():bool`.

2.3.2 Property

Each component can have a list of properties and methods for class introspection. It allows components to have static properties (declared in headers) as well as dynamic properties. Each property has a name and a value. Value types can be anything compatible with Qt class `QVariant`: int, double, boolean, string, 3D or 2D vector, color, char, date, time, rectangle, font... `InterfaceProperty` is directly managed by a `QObject` in order to benefit from the very efficient Qt Meta-Object system.

Thanks to the service layer, the `PropertyInterface` is used in the property viewer to let the user interact with some of the encapsulated data of the component, for example to view a medical image's dimensions or to change the color of a mesh. A specific viewer, called `PropertyExplorer`, is available in CamiTK to display and interact with any defined component properties as shown in Fig. 6b.

Examples of methods already implemented by default include `property(const char*):QVariant` and `setProperty(const char*,QVariant):bool`.

2.3.3 Graphical Representation

A component can have no graphical representation or can have one of the following two: a 3D geometry or a bit map (a flat image of a given thickness).

Under the hood, 3D geometry are managed by a `vtkPointSet` and bit map are managed by a `vtkImageData`. A component handling a dataset that explicitly uses 2D or 3D point arrays, with or without any topological elements like scattered points, lines, polygons, and tetrahedra, should therefore implement the `InterfaceGeometry`. A component handling a dataset that has a topological and geometrical regular array of points such as volumes (voxel data), slice in a volume, and pixmaps, should therefore implement the `InterfaceBitMap`.

In order to facilitate the development of new components, a lot of default behaviors are already programmed. Examples of methods already implemented by default include `getBoundingRadius():double` for `InterfaceGeometry` and `getNumberOfColors():int` for `InterfaceBitMap`. To satisfy code interdependency and modularity, graphical representation implementation is handled by two separate classes. Therefore components do not directly handle their geometrical or image representation. An *Object Adapter* design pattern (also known as *Delegate* pattern) is used by the component to delegate all `InterfaceGeometry` and `InterfaceBitMap` to an instance of respectively the `Geometry` class and the `Slice` class. In the considered design pattern, the Component class is the adapter and the `Geometry` and `Slice` classes are the adaptee classes.

`Geometry` and `Slice` are convenient overlays of VTK basic representation (respectively `vtkPointSet` and `vtkImageData`). They provide all the concrete implementation of the 2D/3D visualization service and define a lot of default behavior. A 3D viewer for example can easily access the surface representation of a component implementing `InterfaceGeometry` by calling `getActor(RenderingModes::Surface):vtkActor*`. `Geometry` and `Slice` can also be used to directly manipulate the low-level VTK objects.

One of the main advantages of the *Object Adapter* pattern is to simplify the construction of the 2D/3D representation. A component only needs to build the low-level encapsulated VTK structure corresponding to the chosen adaptee. The service layer implementation is therefore automatically added without any intervention the developer.

2.3.4 Component Extension

CamiTK core provides two generic classes: `MeshComponent` and `ImageComponent`. Both classes allow for quick integration of user data that are based on geometries and images respectively. They provide basic behavior for importing, loading and interacting with such data.

A component extension is managed by a class inheriting from the `ComponentExtension` class. It defines the extension name, the supported file suffix for opening and exporting and the dynamic or shared library itself. `ImageComponentExtension` and `MeshComponentExtension` fasten the development of user data plugins to provide image-based or mesh-based support in CamiTK. For example, `ImageComponentExtension` is the base class for

DICOM and JPEG plugins, while `MeshComponentExtension` is the base class for the VRML and PML plugins.

CamiTK extension wizard generates the extension class header and code, the component class skeleton, as well as the CMake configuration file: integration of a new component in the project, compilation, linking and configuration are therefore completely automated.

2.4 Action

CamiTK actions derive from the core class `Action` and are used to implement any kind of algorithms on components (transformation or computation). Actions are specific to a component or a family of components, they must therefore specify the targeted component classes, which could be one of the generic `MeshComponent` and `ImageComponent` class or a more specialized user-defined component class. Actions are also grouped by family and can be tagged for quick retrieval.

Building a new action requires subclassing the `Action` class and overriding the `apply(ComponentList)` method. Actions can be grouped in an `ActionExtension`, similarly to the component extension. The CamiTK extension wizard can be used to generate all skeleton code and configuration files that are needed for a new action.

Actions can be triggered many different ways, such as by the context menu of a component in the explorer viewer or by selecting the desired action in the action viewer (see [Sect. 2.2](#)). The context menu only displays the available action of the selected component.

The CamiTK action extension helps to concentrate on building or improving a given algorithm. This could be very useful to define a specific task for a limited amount of time and resource. The generalization of an action is currently being extended to define a scripting framework for CamiTK applications.

Available actions include image visualization (e.g., LUT), image processing (e.g., ITK filters, marching cube), mesh visualization (e.g., modifying the rendering mode or color) and processing (e.g., rigid transformation, tetrahedralization).

2.5 Application

Building a CamiTK application requires assembling viewers or other Qt-based software components together. CamiTK core `MainWindow` extension allows for rapid development of a CAMI application GUI. It can be done by subclassing `MainWindow` or by adding elements to the default application main window. It is also possible to build pure console or text-based applications, although, considering

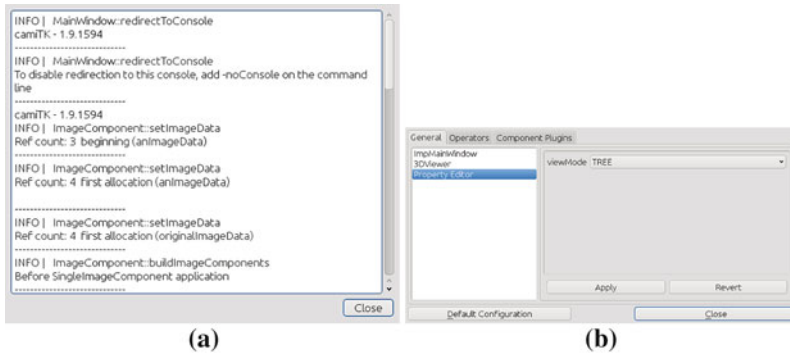


Fig. 7 Low-level GUI elements available to build a CamiTK `MainWindow`. **a** Console for standard output and error redirection. **b** Application setting widget showing all the included viewers preferences, and loaded component extension plugins

the goal of CamiTK, it should be a rare need. Subclassing CamiTK `MainWindow` also provides a way to add menus, application settings and preferences.

Thanks to the framework specific architecture, there is no need for a main controller or presenter as each viewer is managing its own Presenter/View interaction. Therefore there are no connections to be made between the viewers and the component instances. As soon as a CamiTK component is instantiated, all the viewers are aware of its existence. Some viewers may automatically display the component (e.g., the explorer viewer), and other may follow the component policy (e.g., the medical image viewer).

We defined some basic main window components that can be used to build `MainWindow` extensions: a text console where all output and errors can be automatically redirected, application logs (to be saved in a file or displayed on the output stream), and application settings (see Fig. 7).

Figure 8 shows a very basic example code where the axial 2D interactive viewer is added to the default CamiTK `MainWindow` and a test UNC Meta Image (`mha`) is loaded. Note that there is no need to tell which component extension is to be used to load the MHA file format and no need to tell the axial viewer to display the image.

Three example applications are distributed with CamiTK (see Fig. 9): the default *imp* that is used daily at TIMC-IMAG lab, a very *simple* application with only a medical image viewer, and *fancy* that demonstrates that CamiTK can also generate non-traditional GUI.

2.6 Use-cases and Tutorials

Table 2 describes CamiTK extension use cases, that is a list of situations where you should consider using CamiTK and how it can be done in CamiTK. It gives an overview of CamiTK possibilities from the developer point of view. For each use

```
#include <Application.h>
#include <InteractiveViewer.h>

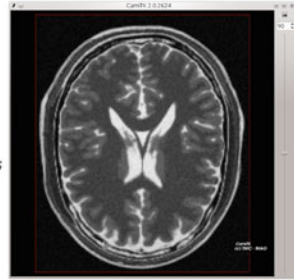
using namespace camitk;

int main ( int argc, char *argv[] ) {
    // create a camitk application
    Application a(argc, argv);
    a.init();

    // add an axial viewer to the main window
    a.getMainWindow()->setCentralViewer(InteractiveViewer::getAxialViewer() );

    // open an image volume
    a.getMainWindow()->open ( Core::getTestDataDir() + "/brain.mha" );

    // let's go!
    return a.exec();
}
```



(a)

(b)

Fig. 8 Very basic example to add an axial slice viewer in the default `MainWindow`. **a** Basic application code. **b** Basic application GUI

case, the situation is described, examples are given and a quick *how-to* program it using CamiTK is briefly described. Each use case refers to a type of CamiTK extension or component customization. Tutorials are available for each extension on the CamiTK website.

3 MML: An Environment for Physical Model Comparison and Evaluation

Soft tissue simulation is one of the main application fields of CamiTK. Using the CamiTK framework we built an environment for evaluation and comparison of soft tissue models. This section describes the main objectives of this environment and its integration into CamiTK.

3.1 Comparison and Evaluation of Soft Tissue Modeling

The evolution of modeling techniques and the improvement of computer hardware have lead to an increasing number of new algorithms which try to describe the mechanical motion of soft tissues. Lots of heterogeneous modeling techniques have been developed since the adaptation of traditional mechanical engineering techniques to medical simulation: soft tissue models using continuous mechanics equations and a numerical resolution method (e.g., Finite Element Method, thereafter FEM), discrete models (e.g., mass-spring networks) or meshless models. Furthermore, numerous implementations of these models have been developed in-house, by communities of scientists (e.g., SOFA [5] or ArtiSynth [8]) or by commercial software developers (e.g., ANSYS, Comsol, Abaqus). In the remaining, such software and libraries where soft tissue models are implemented

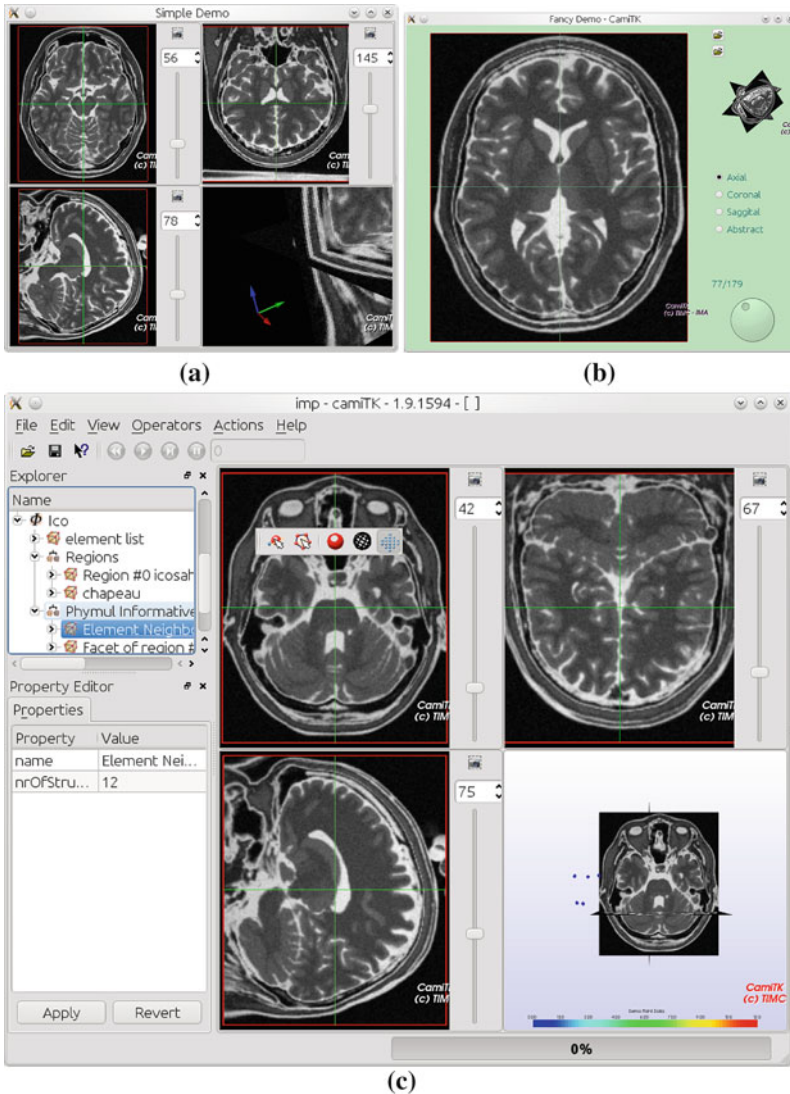


Fig. 9 Three examples of CamiTK MainWindow extension. **a** simple. **b** fancy. **c** imp

are generically named “simulation engine”. The heterogeneity of modeling techniques and the number of existing implementations lead to a wide variety of available algorithms. It is very challenging to compare them because models may not have the same structure (e.g., discrete, continuous, meshless) and may be implemented in different simulation engines. However, comparisons between algorithms are crucial to determine their relevance for a particular medical application depending of the targeted level of compromise between computational efficiency and calculation accuracy.

Table 2 CamiTK extension use cases

Description	Examples	Solutions
Viewer extension		
New type of interaction with the data or a new way of visualizing the data	Volume rendering	Program a new viewer extension
	Fly camera in 3D using a path and timer	Or inherit from an existing one and add/redefine a behavior
	Take regular screenshots to create a movie	
Component extension		
View a new mesh, graph scene or image format	Import.obj format	Program a new component extension
	View 3D Ultrasound images	Or combine existing components in a new one
	View a SOFA scene	
Connect a new device to view its data, control its parameter/action	Connect a medical robot or sensor	Program a new component extension
	Connect a motion tracker device to interact with the 3D viewer	Or a new generic component
Component or action parameters		
Interact with a specific parameter of an algorithm or a data	Numbers (int, double...), boolean	Choose which component/action is concerned and
	2D/3D Position, Rectangle	Program a new property (10–20 lines of code)
	Color, Font, Time, Date, Key sequence...	Optionally add a new user settings (5 lines of code) to keep the value between two runs
Action extension		
New algorithm to process some data	Image filter, volume reconstruction	Choose which component is concerned and
	Biomechanical simulation using a specific method	Program a new action extension
	Export a data to a specific format	
	Mesh/image registration	
MainWindow extension.		
You need a specific application with specific elements	No 2D/3D interaction, just data explorer and property viewer	Program a new MainWindow extension
	Specific application to test on a group of user (including log/trace)	Use the default CamiTK empty Main Window and add only the needed elements

Evaluation is another challenge in soft tissue modeling. If these algorithms are to be used for clinical purpose, they have to prove that their predictions are correct. These evidences can be gathered only by performing a large amount of validation tests. Validation is the process to determine the level to which a verified computational model accurately represent the reality of interest [12]. In soft tissue simulation, validation is often performed by comparing simulation results with in-vitro or in-vivo experiments. These experiments are quite difficult to perform because boundary conditions are difficult to control, in particular during in-vivo experiments. Therefore, validation studies often rely on a limited amount of tests which only implies partial validation.

We propose an environment, called MML framework, to ease the communication between different simulation engines and to allow comparisons between algorithms built from different modeling techniques. The second main objective is to propose a generic and sharable description of validation experiments in order to fill the lack of validation references. This framework is implemented as an external and independent library, and is introduced in CamiTK as a new type of component and actions (see Sect. 3.6).

3.2 Environment Overview

The main concepts of the MML framework are (see Fig. 10):

1. Constitutive elements of a simulation and simulation results are described by a set of generic languages.
2. Simulation calculations are computed by different simulation engines thanks to a plugin mechanism.
3. A set of quantitative metrics can be automatically computed.
4. A database stores the simulation descriptions, as well as simulation and real experiment metric values for further comparisons.

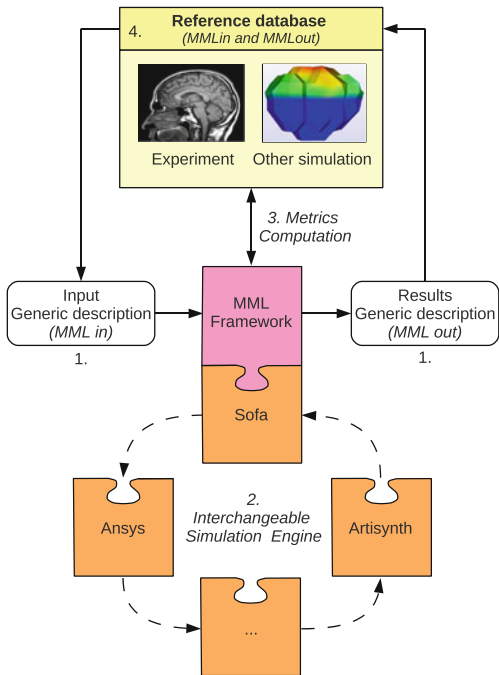
These key concepts are discussed in the next sections.

3.3 Generic Description

In order to achieve interoperability between different simulation engines and different modeling techniques, we use a generic XML description of a simulation. Other scientific domains tried to define standard formats for sharing models using XML description. This kind of XML description is for example widely used in biological system modeling (e.g., CellML [7], SBML [3] and many other [14]).

In our framework we use the two languages defined in [2]: PML and LML. PML stands for Physical Model markup Language. It aims at describing the geometry and physical structure independently of the modeling technique used

Fig. 10 MML framework overview



(discrete or continuous). PML includes an easy way to freely label any individual or set of nodes and degrees of freedom, any geometric or logical structures. LML stands for Load Markup Language. It aims at describing constraints and boundary conditions applied to a physical model.

We extended these descriptions with a third language called MML which stands for Monitoring Markup Language. MML description is divided in two parts: *MMLin* and *MMLout*.

MMLin describes a reproducible simulation by integrating PML and LML description together with the list of data to monitor, simulation parameters (e.g., integration step) and stability criteria. Monitored data can be standard measurements (e.g., positions, velocities, stress) or comparison metrics (see Sect. 3.5). The use of a stability criterion to decide when a simulation is considered finished is discussed in Sect. 3.5. In Sect. 4, an example of an *MMLin* description for a simple simulation is given. Having a simulation described by a *MMLin* document allows one to reproduce the same exact simulation on different simulation engines so that comparisons can be made between them.

MMLout is a description of both simulation and real experiment results. *MMLout* describes monitored data and stability criterion values for each step of the simulation. *MMLout* can be stored after simulation in order to be used as a comparison reference for other simulations. *MMLout* can also be created from real experiments in order to test the accuracy of simulations and make validation assessments. In this case, the MML framework takes advantage of the integration

into CamiTK where medical imaging taken from real experiments can be processed (e.g., segmentation, 3D reconstruction) and converted into MML description (see Sect. 4 for an example of the workflow from medical images to MML description using CamiTK). Once stored, MMLout documents can be used for automatic comparison with future simulations of the same problem using the corresponding MMLin document as input. The use of this kind of experiment description can help to reduce the lack of reference for validation by facilitating sharing of experimental results between research teams.

3.4 Simulation Engines

Our framework aims to help scientists to compare algorithms from different simulation engines by wrapping a comparison layer on top of them. A simulation engine plugin describes the interface between the simulation engine and the MML framework. Simulation engine plugins are simply named “simulator” in this section. Their roles are:

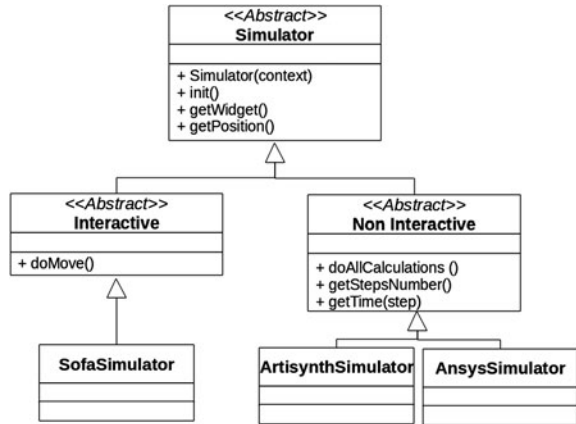
- to export an MML description into a specific simulation engine,
- to drive the simulation engine in order to compute the simulation steps, and
- to import back the computed data into the framework in order to compute the metrics.

Simulation generic parameters (e.g., integration step) are directly stored in the MMLin document. For simulation engine specific parameters, default configuration is used by the plugin. These specific parameters can also be manually changed before or during simulation.

There is two kinds of simulators: interactive and non-interactive. Interactive simulators are based on pluggable C++ simulation libraries (e.g., SOFA) which allows our framework to have a complete control during each step of the simulation. Non-interactive simulators are designed for other simulation engines run directly in a standalone application (e.g., ANSYS). In non-interactive simulators, all calculations are generally made in one big step by the simulation engine and post-processed afterward.

Figure 11 shows a partial UML class diagram for simulators. Simulation engine specific simulation is generated from simulation context containing the MMLin description. An optional `QWidget` item can be specified to provide user interaction. For example a specific widget acts as a SOFA .scn editor that let the user interactively modify parameters and apply changes to the current simulation. Interactive simulators have a `doMove()` method which makes the simulation engine compute one step of simulation. This method is replaced by the `doAllCalculations()` method in non-interactive simulators which launch the calculations that can be post-processed. A set of methods (only `getPosition()` method is represented in Fig. 11) allows to retrieve computed data for a given step from simulation engine.

Fig. 11 Partial UML class diagram of the simulation engine plugins structure



In order to simplify the development of new plugins we try to keep the interaction with the simulation engines the easiest possible, so that no extra code has to be written within the simulation engine. To achieve this goal, all metrics are computed inside the MML framework and only basic data are extracted from the simulation engine. For example only positions are needed to compute velocity-based or distance-based metrics. Therefore, only the `getPosition()` method has to be implemented in a plugin in order to compute a large amount of metrics, as described in the Sect. 3.5. Other metrics based on different data (e.g., force or stress) can be also implemented.

We have currently developed three simulators for SOFA, ANSYS and ArtiSynth. The SOFA plugin is an interactive simulator and is the most complete simulation engine developed so far. It can not only launch simulations computed by SOFA from MML description but also import SOFA `.scn` documents into MML compatible descriptions. While in ArtiSynth the user can interact with the simulation at each step using Controllers or Monitors, in the current version of MML, the Artisynt plugin is considered as a non-interactive simulator because we did not have immediate access to a C++ API. ANSYS and ArtiSynth non-interactive plugins, even though not as complete as SOFA one, can still be used for simple interactions with those applications.

3.5 Metrics

Comparison metrics are used to quantitatively compare models against themselves or against references. As explained before, soft tissue models used for a CAMI application have to be tested in terms of accuracy (Sect. 3.5.1) and computational efficiency (Sect. 3.5.2).

Metrics are computed from basic data taken from simulation context (e.g., current time), from simulator (e.g., nodes position) and from references if needed

(e.g., reference's nodes position for Relative Energy Norm computation, see [Sect. 3.5.2](#)). We developed a first set of metrics which are mostly position based. New metrics based on other extracted data can be defined. The metrics can be a simple output of values calculated in a simulator (e.g., velocities, stress and strain values) or a more or less complex combination of these data. As the overall performance during validation is not an issue, the best practice is to compute all values again externally in the MML library from basic extracted data (e.g., positions and forces). This avoids to maintain a complex interaction with the simulator and simplifies simulation plugins, as explained in [Sect. 3.4](#). For example velocities can be computed from node positions and time step. This section illustrates this principle and gives some position-based metric examples.

3.5.1 Accuracy Metrics

Accuracy can be assessed by comparing the simulation results with a reference. Reference values can be measured on real experiments or calculated by other simulation engines. There are only few papers which deal with the problem of the choice of metrics [1, 10]. Soft tissue models are often a collection of 3D nodes (degrees of freedom) linked together in a mesh. Depending on the structure of the simulated model and the reference used, different types of metrics are available.

If models and reference have the same structure (a node of the model corresponds to a node of the reference) node to node comparisons can be computed. This is for example the case when a simulation is compared to a previous version of the same model in order to test the influence of a parameter. Node to node metrics include for example, distances or Relative Energy Norm (REN) [1].

If the models and reference do not have the same structure (e.g., the reference has more nodes than the model) node set comparisons can be used. This is the case with references obtained from experiments which are often composed by sets of 3D points generated by surface segmentation. Node set comparisons include for example, point to set distance or Hausdorff distance.

If the model and reference are made of a mesh, volume and surface comparisons are possible.

All the metrics given here as examples only need the node positions of the model at a given time and then can be computed if the `getPosition` method is implemented.

3.5.2 Computational Efficiency Metrics

A classical way of evaluating the computational efficiency is to measure the number of frames per second (FPS) the simulation engine can compute.

Another interesting measurement, especially for CAMI, is the stability criterion. Indeed, even if computations are fast (i.e., if the FPS is high), a model cannot

be used in an interactive simulation if it requires a large number of iterations to reach equilibrium.

By stability criterion we do not mean a specific measurement of an algorithm convergence, but the global dynamic behavior of the simulation, that is the definition of the state of dynamic equilibrium. The stability criterion is defined by a boolean expression using tests on basic data extracted from simulation (e.g., positions) or computed externally from these data (e.g., velocities or kinetic energy). This boolean expression can be a complex combination of tests (e.g., a decreasing global kinetic energy and all nodal kinetic energy under a given threshold value). The stability criterion is defined in the MMLin document so that the same criterion can be used for all the simulations based on the same experiment.

3.6 Implementation and CamiTK Integration

3.6.1 Independent Library

The XML structure of MML documents is described using the WC3 standard XML Schema (XSD).² XSD documents are formal representations of the XML document format, including type definitions, possible values and content model organization. The open source software `xsdcxx` from CodeSynthesis³ is used to validate MML documents and to generate a C++ library for unmarshalling and marshalling MML documents. Using automatic C++ code generation simplifies the extension of MML description, for example when a new monitor or stability criteria is added.

The MML library is built on top of the generated I/O library. It supports simulation management and metrics calculation. This library can be used independently of CamiTK to build standalone applications, for example to build an optimization application that can use the computed metrics in order to optimize any model parameters (including mesh or rheology), or a sensitivity analysis application that can automatically explore the parameter spaces. In [Sect. 4](#), a specific standalone application is used to optimize the stiffness of a mass-spring model.

3.6.2 CamiTK Integration

A *component* extension is dedicated to interface the MML library into CamiTK. It can load any MMLin document in CamiTK Core. The MML component reuses the

² <http://www.w3.org/XML/Schema>

³ <http://www.codesynthesis.com/products/xsd>

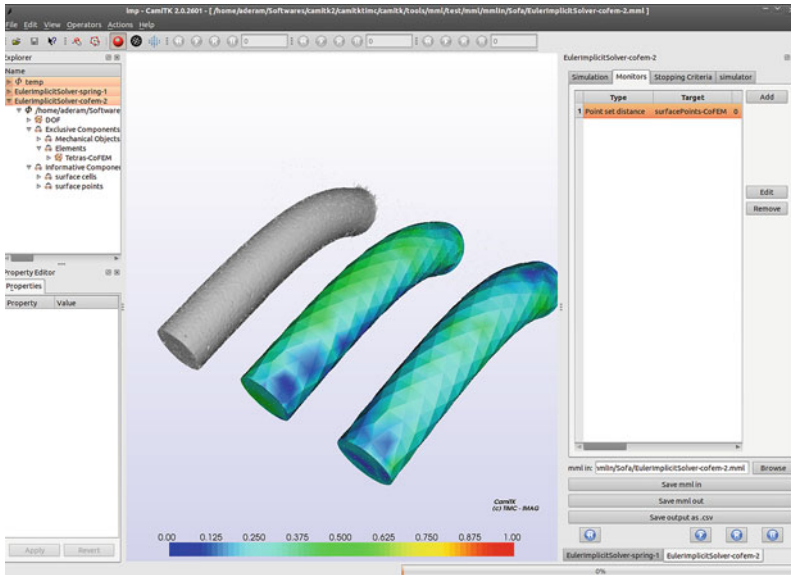


Fig. 12 MML framework into CamiTK. The specific action widget is shown on the right of the 3D interactive viewer. Its simulation tab can be used to interact with the simulation parameters (integration step, PML and LML documents,...). Other tabs are used to display, modify and add monitors and stability criteria. The “simulator“ tab is used to display the simulation engine plugin specific widget (if available). This screenshot shows the comparisons of two SOFA simulations presented in [10]

PML component which can load and manipulate PML documents and display their 3D representation in the interactive viewer. A specific action was added to interact with the simulation engines and display computed metrics as color code or vectors in the interactive viewer (see Fig. 12).

4 Example: Brain-Shift Modeling, From the Images to the Simulation

This section presents an example of a CAMI application which can be performed with CamiTK from image volume opening and display to deformation model comparison.⁴

The chosen application is the brain shift phenomenon. During neurosurgeries, when the surgeon opens the patient’s skull, the brain deforms due to physical changes among which are the loss of cerebrospinal fluid, surgical tool interactions and resections, as well as physiological changes due for example to inflammation, drugs and anesthesia.

⁴ Videos of the different steps are available as supplementary material on <http://camitk.imag.fr>

Fig. 13 Visualization of a MR Image in a CamiTK application (here *imp*)



As the tissues have moved or deformed, the brain structures during the operation are not where they can be seen on the preoperative images. It is especially true when one considers the targeted tissues or zones: a tumor region for example may be displaced or the cortex may be compressed compared to pre-operative images. A soft tissue simulation may therefore be used to estimate the deformations due to the brain shift.

4.1 Image Analysis and Mesh Generation

To obtain a patient-specific model of the brain deformation, lets start from a patient-specific MR Image.

4.1.1 Image Volume Visualization

Volume medical image I/O are handled by component extension inheriting from ImageComponent. This component implements the InterfaceBitmap interface. A specific ImageComponentExtension is dedicated to manage the DICOM format. Loading any kind of MR Image is therefore possible in CamiTK (see Fig. 13).

On the initial MR Image (see Fig. 13), the voxels belonging to the brain can be extracted in order to be define a biomechanical mesh. Once the volume image is loaded in the CamiTK application, any *action* extension designed to work on ImageComponent can be applied (independently of the specific original file format). The Look Up Table action can therefore be used to enhance the luminosity and the contrast of an image, as illustrated Fig. 14. Using this action shows that the choice of a unique threshold to only segment all the voxels belonging to the brain is impossible: voxels of the same gray level are present in the brain as well as in other tissues.

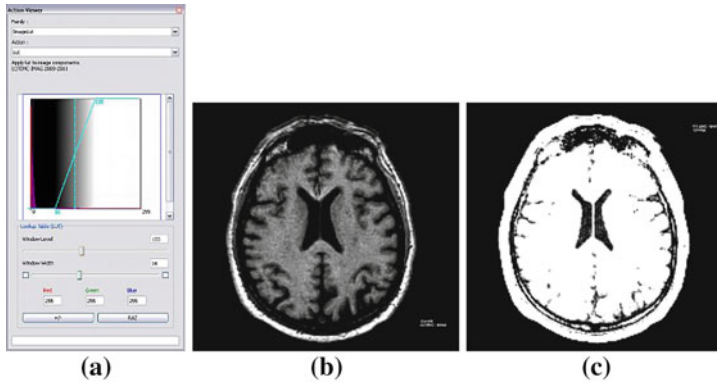


Fig. 14 Use of the Look Up Table action to visualize an image component and choose between several luminosities and contrasts. **a** Look Up Table action widget. **b** Axial view with a rather large LUT window. **c** Axial view with a small LUT window

4.1.2 Segmentation of the Brain

In this textbook example, a simple but yet efficient way can be used to segment the brain with ITK algorithms available as CamiTK actions. This example is freely inspired from [9]. Of course, one can imagine more precise, robust or efficient ways to segment this tissue thanks to ITK or other algorithms. The integration of such algorithms in CamiTK framework would only need a few lines of code and only requires one to interface the algorithm with `vtkImageVolume`. Table 3 illustrates the different segmentation steps.

4.1.3 Mesh Generation

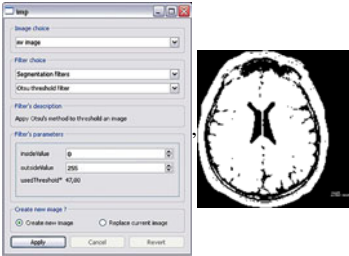


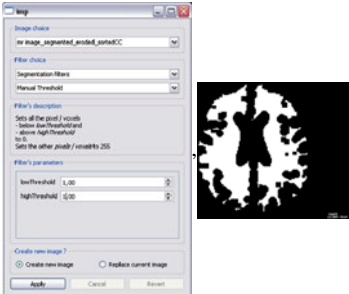
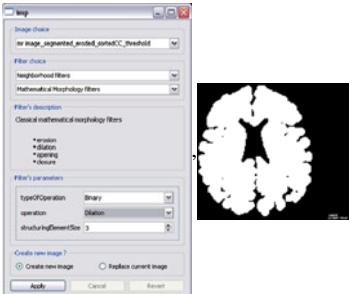
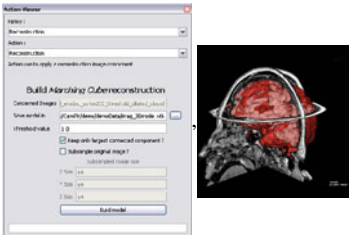
The triangular mesh obtained by marching cube reconstruction needs some cleaning. It can be done automatically by merging nearby nodes and eliminating resulting invalid triangles. Tetrahedralization is achieved thanks to a CamiTK action that encapsulates TetGen⁵ mesh modeler. The resulting mesh is described in a PML document and has 336 nodes and 1.184 tetrahedrons, see Fig. 15a. Note that this is a very rough representation, only suitable for this textbook example.

4.2 Model Comparisons

In this section, comparison tests between different modeling techniques from different simulation engines are performed using the MML framework. Using the brain tetrahedral mesh obtained after segmentation and reconstruction in the

⁵ <http://tetgen.berlios.de>

Table 3 Automatic brain segmentation from MR Image in six steps

Step 1	Step 2
<p>Preprocessing: automatic threshold using the Otsu method.</p> 	<p>A binary erosion is applied to <i>disconnect</i> voxels of the brain from those of other tissues.</p> 
<p>Labeling of the connected components</p> 	<p>Selection of the largest connected component.</p> 
<p>Dilatation to retrieve the brain original size, mirroring the erosion of Step 2.</p> 	<p>Creation of a Mesh Component by applying the marching cubes VTK implementation integrated as a CamiTK action extension.</p> 

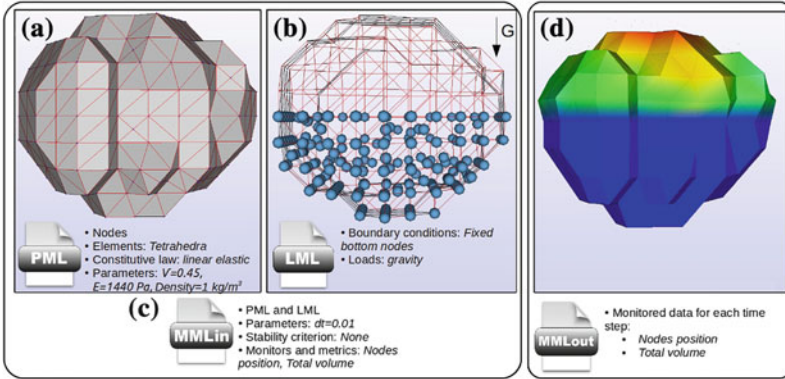


Fig. 15 Simplified visualization of the MML description of the ANSYS reference simulation (MMLin) and its results (MMLout)

previous section, a set of brain shift simulations is performed. To clarify, we do not aim to build an accurate simulation of the phenomenon but to show an example of how our framework can help scientists to compare different approaches when modeling soft tissue.

4.2.1 Generic Description Set-Up

In order to make a comparison study, we need:

- a PML description of the geometry, as obtained in [Sect. 4.1](#),
- a LML description of the loads and boundary conditions,
- a MMLin description which encapsulates PML and LML description and adds simulation parameters, stability criterion and monitored data.

For the brain shift simulation the chosen boundary conditions are composed by a set of fixed nodes at the bottom of the brain (opposite to the skull opening) and a gravity field. We described these loads in a LML document, see [Fig. 15b](#).

We choose to use an ANSYS simulation as reference for our simulations. ANSYS is a commercial FEM software widely used as reference for continuum mechanics simulation because of its well established verification procedure. Therefore, the ANSYS simulation is performed first, in order to obtain a reference. In order to fit the brain material, FEM parameters are a young modulus of 1,440 Pa, a Poisson ratio of 0.45 and a density of 1 kg/m^3 [13]. The chosen integration step is 0.01 s. The specific ANSYS parameters for this simulation are shown in [Table 4 \(e\)](#). The monitored data are the brain total volume and node positions. The MMLin document uses the previously introduced PML and LML document alongside the monitored data ([Fig. 15c](#)). Stability criterion cannot be specified for ANSYS simulation as the equilibrium is determined inside the software. Once

Table 4 Models and specific parameters used for comparisons

	Engine	Model	Reference	Solver	Specific parameters
(a)	SOFA	Corotational FEM	[11]	Euler implicit	Large deformation
(b)	SOFA	Tensor-Mass	[4]	Euler implicit	
(c)	SOFA	Mass-Spring		Euler implicit	Spring stiffness: 430
(d)	ArtiSynth	Linear elasticity FEM	[8]	Euler implicit	Large deformation
(e)	ANSYS	Linear elasticity FEM		Default	Solid285 element and large deformation

ANSYS simulation are performed, the MMLout description is saved and considered as the reference (Fig. 15d).

4.2.2 Tested Models

We perform a set of simulations using different modeling techniques. The same MMLin document is used (same integration step and material parameters), except that the distance between nodes and reference nodes, and REN errors against ANSYS are added to the monitored data and a stability criterion is specified. We choose a threshold on nodal displacement of 0.1 *mm* as the stability criterion. Models, simulation engines and specific parameters are summarized in Table 4. For the mass-spring model, a spring is placed at each edge of the tetrahedral mesh. The best stiffness for the springs is determined by a small standalone optimization program based on the MML library. This standalone program repeatedly computes the average distance between the ANSYS reference and the simulated positions obtained by SOFA. It automatically creates and runs a SOFA simulation in order to explore a given stiffness interval and to minimize the distance metric.

4.2.3 Results

Results of the simulations and monitoring are shown in Fig. 16 and Table 5. The time to reach stability represents the time to reach the stability criterion defined in Sect. 3.5.2. The computation time is the total time needed by the simulation engine to compute every steps of the simulation. REN and distance refer to ANSYS reference. Volume ratio is the ratio between final and initial brain volume. Let us emphasize that the results shown in Table 5 are not a comparison with a gold standard or ground truth, but with the ANSYS reference, including all the probable errors and approximations induced by the simple assumptions presented in Table 4. The metrics are given here to illustrates how scientists can use our framework to compare different models from different simulation engines.

Results Table 5 shows that both FEM based model (from SOFA and ArtiSynth) are closer to reference. These two simulations are quite similar even if SOFA is slightly closer to ANSYS. However, ArtiSynth has a better maximum distance which can be an important criterion to consider when a critical value can not be exceeded.

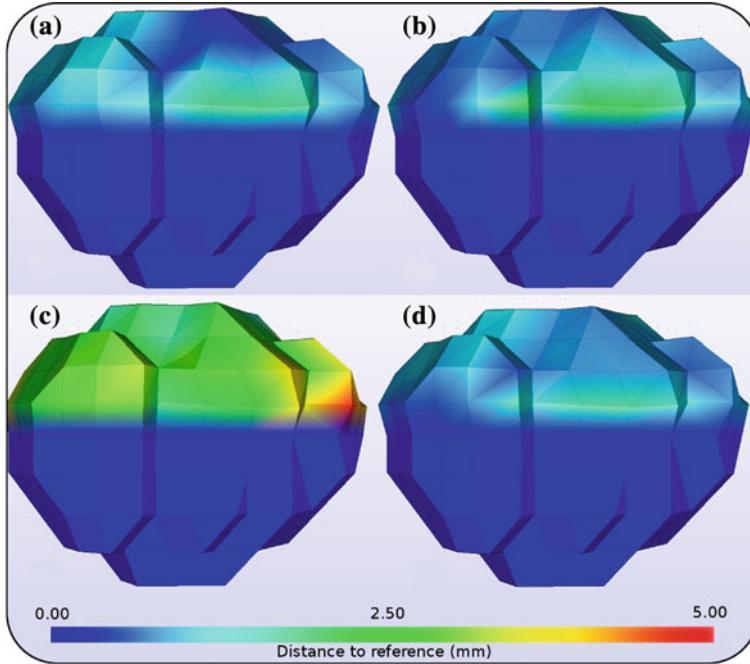


Fig. 16 3d colored shape displaying distance between the four models and ANSYS simulation

Table 5 Summary of the quantitative values of the computed metrics

Model	FPS	Time to reach stability (s)	Computation time (s)	REN (%)			Distance (mm)			Volume ratio (%)
				min.	max.	avr.	min.	max.	avr.	
(a)	96	57.46	58.44	1.97	32.85	12.76	0.21	1.86	0.78	96.7
(b)	143	55.86	38.54	3.70	32.60	13.49	0.17	2.15	0.87	96.4
(c)	947	70.21	7.25	7.14	105.42	44.06	0.64	4.76	2.60	94.5
(d)	n/a	n/a	n/a	5.13	27.73	12.98	0.24	1.78	0.83	96.2
(e)	n/a	n/a	764	n/a	n/a	n/a	n/a	n/a	n/a	96.7

Mass-spring model is the fastest (almost ten times higher FPS than FEM) but has the highest distance from reference: average distance to reference is three times more important than SOFA FEM, and maximum is near 5 mm, which would be a critical problem for neurosurgery if we consider the ANSYS simulation close to reality. This model also needs the longest time to reach stability.

The Tensor-Mass model shows a compromise between FEM models and mass-spring. This model succeeded to 50% faster than the FEM model but is nearly as close to reference with an average 0.1 mm.

For a more realistic simulation, a mesh convergence study should be performed to select the proper mesh refinement; constitutive law and parameters should also be studied more intensively. For example, a non-linear model could be used as brain is often considered non-linear [13].

5 Conclusion

Our framework could be seen as a linking module that can fill the gap between highly specialized software covering different CAMI research fields.

The first public version of CamiTK was published in October 2011 (version 2.0). Anybody wanting to contribute and join the effort is welcome. The current work is focusing on simplifying the scripting of actions and on software quality (test, packaging, continuous integration).

We are interested in participating to any workgroup in order to increase interoperability between CAMI software.

Acknowledgments The authors wish to acknowledge the support of the French ministry of research (PhD grant) and of ECCAMI (Excellence Center for Computer Assisted Medical Interventions, <http://www.eccami.com>), a community of practice bringing together clinicians, researchers and manufacturers. CamiTK was build over many years and we wish to thank all the students, PhD students, researchers and engineers for their work.

References

1. Alterovitz, R., Goldberg, K.: Comparing algorithms for soft tissue deformation: accuracy metrics and benchmarks. Technical report, Alpha Lab, UC Berkeley (2002)
2. Chabanas, M., Promayon, E.: Physical model language: towards a unified representation for continuous and discrete models. *LNCS* **3078**, 256–266 (2004)
3. Cornish-Bowden, A., Hunter, P., Cuellar, A., Mjolsness, E., Juty, N., Dronov, S., Takahashi, K., Nakayama, Y., Gilles, E., Kasberger, J. et al.: The systems biology markup language (sbml): A medium for representation and exchange of biochemical network models. *Bioinformatics* **19**(4), 524–531 (2003)
4. Cotin, S., Delingette, H., Ayache, N.: A hybrid elastic model for real-time cutting, deformations, and force feedback for surgery training and simulation. *Vis. Comput.* **16**(8), 437–452 (2000)
5. Faure, F., Duriez, C., Delingette, H., Allard, J., Gilles, B., Marchesseau, S., Talbot, H., Courtecuisse, H., Bousquet, G., Peterlik, I., Cotin, S.: SOFA, a Multi-model framework for interactive physical simulation, chap. 11. Springer (2012)
6. Fowler, M.: Patterns of enterprise application architecture. Addison-Wesley Professional (2003)
7. Hedley, W., Nelson, M., Bellivant, D., Nielsen, P.: A short introduction to cellml. *Philosophical Transactions of the Royal Society of London. Series A: Mathematical, Physical and Engineering Sciences* **359**(1783), 1073–1089 (2001)
8. Lloyd, J.E., Stavness, I., Fels, S.: ArtiSynth: A fast interactive biomechanical modeling toolkit combining multibody and finite element simulation, chap. 13. Springer (2012)
9. Malandain, G.: Filtrage, topologie et mise en correspondance d'images médicales multidimensionnelles. Thèse de sciences, Ecole Centrale de Paris (1992)
10. Marchal, M., Allard, J., Duriez, C., Cotin, S.: Towards a framework for assessing deformable models in medical simulation. *LNCS* **5104**, 176–184 (2008)
11. Nesme, M., Marchal, M., Promayon, E., Chabanas, M., Payan, Y., Faure, F.: Physically realistic interactive simulation for biological soft tissues. *Recent Res. Dev. Biomech.* (2) (2005)
12. Oberkampf, W., Trucano, T., Hirsch, C.: Verification, validation, and predictive capability in computational engineering and physics. *Appl. Mech. Rev.* **57**, 345 (2004)

13. Schiavone, P., Chassat, F., Boudou, T., Promayon, E., Valdivia, F., Payan, Y.: In vivo measurement of human brain elasticity using a light aspiration device. *Medical Image Anal.* **13**(4), 673–678 (2009)
14. Stromback, L., Hall, D., Lambrix, P.: A review of standards for data exchange within systems biology. *Proteomics* **7**(6), 857–867 (2007)
15. Toussaint, N., Sermesant, M., Fillard, P.: vtkinria3d: A vtk extension for spatiotemporal data synchronization. In: *Proceeding of Workshop on Open Source and Open Data for MICCAI* (2007)