CAMPVis – A Game Engine-inspired Research Framework for Medical Imaging and Visualization

Christian Schulte zu Berge, Artur Grunau, Hossain Mahmud, and Nassir Navab

Chair for Computer Aided Medical Procedures, Technische Universität München, Munich, Germany
christian.szb@in.tum.de

Abstract Intra-operative visualization becomes more and more an essential part of medical interventions and provides the surgeon with powerful support for his work. However, a potential deployment in the operation room yields various challenges for the developers. Therefore, companies usually offer highly specialized solutions with limited maintainability and extensibility As novel approach, the CAMPVis software framework implements a modern video game engine architecture to provide an infrastructure that can be used for both research purposes and end-user solutions. It is focused on real-time processing and fusion of multi-modal medical data and its visualization. The fusion of multiple modalities (such as CT, MRI, ultrasound, etc.) is the essential step to effectively improve and support the clinician's workflow. Furthermore, CAMPVis provides a library of various algorithms for preprocessing and visualization that can be used and combined in a straight-forward way. This boosts cooperation and synergy effects between different developers and the reusability of developed solutions.

1 Introduction

Recent advances in the field of medical image computing provides today's clinicians with a large collection of imaging modalities and algorithms for automatic image analysis. However, translating innovations from research into the daily workflow of clinicians is a difficult and time-consuming task since the deployment into a clinical setup poses various challenges. Developing solutions for medical imaging and visualization beyond mere image viewers does usually not yield small and self-contained algorithms but rather an aggregation of many libraries and algorithms. This is mainly due to the visualization pipeline being rather long and requiring many preprocessing steps (e.g. image retrieval, registration, filtering, etc.) before one can start with the actual work on the visualization aspects. Thus, this is an extensive field of research and brings together experts and researchers of different disciplines who are working on various aspects of the visualization pipeline.

However, at the same time everybody needs a more or less complete implementation of the entire pipeline in order to implement and evaluate their work: Researchers working on novel processing methods need sophisticated visualization techniques in order to evaluate their results. At the same, time visualization researchers also require advanced preprocessing algorithms in order to yield high quality input data for their renderings. This mutual dependency offers a large potential for synergy effects when researchers work within the same software framework.

Unfortunately, such highly interdisciplinary work often runs into problems when it comes to sharing a common code base or integrating the work from multiple working groups into a single solution. In particular in environments with limited funding and high employee fluctuation, such as universities, once created software libraries are often abandoned after finishing the project and are hardly designed to be reused by others.

With this motivation in mind, we identified the following list of requirements and design goals focusing on the usage in heterogeneous academic environments:

- *Modern software architecture:* Usage of platform-independent and standard-compliant state-of-the-art techniques. Start mostly from scratch and avoid deprecated interfaces due to forced backwards-compatibility.
- *Bridging the gap between development and deployment:* Focus on research-usage supporting rapid software prototyping, but at the same time allowing for easy transformation of implemented algorithms to end-user products.
- *Sandbox-like environments:* While using the same code base, multiple developers can implement code independently from each other without forcing each other to meet one's code dependencies.
- *Distributed/decentralized computing:* Allow CAMPVis to be run on different devices and support communication between them in order to share computational resources.

In order to meet these requirements, we use the software architecture design of modern video game engines as reference and inspiration. 3D video games have a long tradition in simulating complex environments and providing real-time visualizations. They usually run on a wide range of hardware and massively multiplayer online games (MMOGs) even manage to synchronize the game state over thousands of computers. Hence, video game engines provide a promising approach to integrate the handling of a large amount of data, real-time graphics, interactivity and network computing into a uniform, extensible infrastructure.

2 Related Work

As of today, there is a plethora of different software platforms and applications for medical imaging and visualization available. Categorizing them in a structured manner is a challenging task as one can differentiate them along various dimensions.

The specific focus of the software platforms is one dimension to differentiate. Some libraries such as ITK [13] target only the image processing part while others like VTK [14], Voreen [18], Inviwo [24] or ImageVis3D [9], only focus on the visualization part. Most software frameworks, however, try to combine both aspects into a single platform. While 3D Slicer [8] and MITK [29] emphasize the application domain, other frameworks, such as MeVisLab [20], DeVIDE [4] or XIP [25], use the concept of a data flow network to better support rapid prototyping development.

In 2007, Bitter et al. compared four freely available frameworks for medical imaging and visualization based on ITK [3]. The survey paper of Caban et al. focuses more on the rapid development aspect of such libraries [5].

2.1 Entity Component System Architecture

Many modern game engines exploit data-driven programming [2, 15] and implement the *Entity Component System* (ECS) paradigm as main software architecture. While there is no official definition of this paradigm, most approaches show strong similarities in their central design. The main intention of ECS is to yield a cleaner software architecture than classic object-oriented programming (OOP) approaches. Game engines have a large number of different game objects, each of them being formed of multiple aspects, such as physics, player interaction, graphical representation and automation. Trying to model such a complex setup by a traditional OOP class hierarchy will yield a very complicated inheritance graph that is very hard to maintain and extend [17, 27, 28].

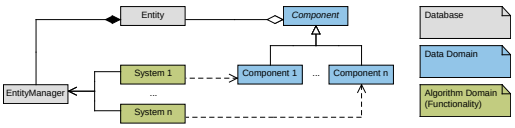

Figure 1: UML diagram illustrating the Entity Component System software architecture, which is separated into three parts: The data domain storing the system state, the algorithm domain storing the functionality and the EntityManager as database for storing all entities.

An alternative is to follow the common "favor composition over object-inheritance" paradigm [10]. Therefore, the central idea of ECS is to decouple objects from their state and their functionality as illustrated in Figure 1. This is achieved through introducing three concepts [16, 27]:

Entity The entity is the single general purpose object that stores neither data nor functionality (i.e. methods). Its sole purpose is to provide a tag for each game object.

Component Components are attached to entities and store the raw data but no functionality. Their purpose is to define a certain aspect of the object and how it interacts with the world. Attaching a component to an entity labels the entity to possess this particular aspect. An entity can have multiple components attached and each component can be attached to multiple entities.

System The system defines the actual functionality. Usually, there is one system for each component (aspect) that models and implements the global interaction and functionality of the game.

This concept allows for a very flexible game design where usually many objects of different type share parts of their aspects. Squeezing this into a classic OOP architecture would require a highly fragmented and complex inheritance graph of a large number of very tiny classes and interfaces or a bunch of quite large and partly redundant classes [17].

3 CAMPVis Software Architecture

The core of our software framework consists of four main components that interact together in order to generate output (cf. Figure 2).

- *DataContainers* act as central database storing and managing all non-temporary data that occurs during execution.
- A *Pipeline* defines what computations are actually performed, handles user interaction and provides the output render data.
- *Processors* act as building blocks implementing specific algorithms. Although one could certainly implement all algorithms fully inside a pipeline, we do not encourage this. Instead, encapsulating single algorithms in processors makes it easier to re-use them in other projects.
- *Properties* are used for configuring implemented algorithms.

In the application domain, these components are then exposed through an *OpenGL canvas* that takes care of bringing the corresponding pipeline's output onto the screen, as well as through *PropertyWidgets* that wrap around properties to automatically generate a graphical user interface (GUI).

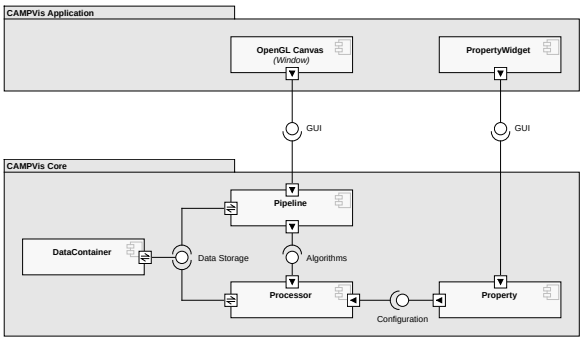

Figure 2: UML component diagram providing an overview of the main concepts of CAMPVis and how they interact with each other.

In this section, we will describe how these components interact with each other in order to implement a variant of the ECS paradigm. Furthermore, we will present architectural software design decisions and discuss how they relate to the initial set of requirements.

3.1 Build System

To provide a uniform build process across multiple platforms and compilers, CAMPVis uses the CMake build system [12] where the build process is defined using a scripting language. This allows us to effectively manage and configure the various build options, as well as to scan the file system for available modules to implement our module architecture (cf. Section 3.5). In a separate step, the build instructions are then transformed to project and make files for the specific target architecture.

3.2 Package Architecture

As introduced in Section 2.1, the ECS paradigm encourages the separation of data domain and algorithm domain. We transfer this concept to the main CAMPVis package architecture as illustrated in Figure 3. Driven by our goal to minimize the gap between development and deployment and inspired by Voreen architecture [18], we furthermore separate all GUI toolkit dependent code into a separate package. This allows to easily switch out the rapid-prototyping user interface with an end-user interface suited for clinical setups.

Core Package The core package wraps the data domain and provides the infrastructure for the core system of CAMPVis. These are basic data structures, base classes for processors, pipelines, properties, etc., common tool classes such as type traits or string utilities, as well as common GLSL headers. This package has as few external dependencies as possible and in particular no GUI dependencies.

Modules Package This package wraps the algorithm domain and contains the main functionality of CAMPVis. Modules contain individual processors that implement concrete algorithms, as well as individual pipelines that implement solutions and workflows for concrete applications. This package is also not dependent on GUI libraries. An important aspect of the modules package is that the package actually is a collection of modules, which can individually be selected to be included into the build or not. This allows to easily manage multiple independent modules maintained by different people. If one of them is faulty it can be excluded from the build so that CAMPVis is still compilable (cf. Section 3.5).

Application Package The application package provides the tools to build an actual application based on CAMPVis. It ties together the core and modules package and provides the user with a default GUI. This GUI is intended as research interface and hence exposes all internal parameters (i.e. properties) of the individual processors and pipelines. Furthermore, it comes with a convenient debug interface to inspect the contents of the DataContainer. However, the application package is fully optional and can be replaced by own implementations, for instance when integrating CAMPVis into an existing application.

The number of external libraries required for the core package is kept as small as possible. Besides OpenGL 3.3 and a small OpenGL wrapper library (CAMP Graphics Toolkit, cgt), the single other mandatory external library is Intel TBB [11] providing clean interfaces to support multithreading and concurrent algorithms. All other potential dependecies are part of individual CAMPVis modules and thus only required when the corresponding module is enabled.

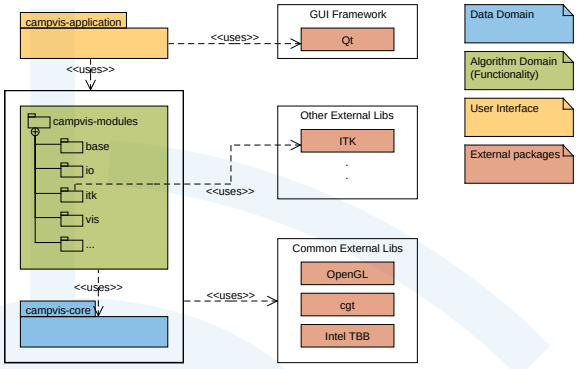

Figure 3: UML diagram of the CAMPVis package structure, which features a separation into data domain (core module), algorithm domain (modules package) and user interface (application package).

3.3 The Entitiy Component System for CAMPVis

We use the Entity Component System paradigm as basic architecture for the CAMPVis framework. Its main benefit is the very strict and clear separation of data and algorithms that allows for a great amount of flexibility. However, a software for medical imaging and visualization has a significantly smaller amount of alive objects during runtime than video games and not all of its systems are of fully global nature. Therefore, we applied some modifications to the classic ECS approach presented in Section 2.1 and developed with the design shown in Figure 4.

On the data domain, we implement the classic concept of entities, which act as general purpose object and store neither data nor functionality. Each entity has a unique identifier (for which

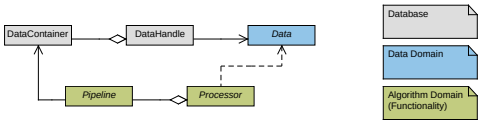

Figure 4: UML diagram illustrating the adapted Entity Component System software architecture for CAMPVis, which also separates the database, the data domain and the algorithm domain from each other.

we use strings in order to facilitate the handling for the user) and is stored and managed by the *DataContainer*, an entity database implemented as map. Therefore, we call our entities *DataHandle*. In the current implementation, every DataHandle just points to a single data aspect (e.g. transformation, image, geometry, etc.). However, for the future it is planned to have DataHandles aggregating multiple data aspects (components) in order to move this design closer toward the classic ECS architecture.

Regarding the algorithm domain, there is no fixed collection of systems in CAMPVis because of its nature of being a research framework. The necessary systems are rather dependent on the actual project implemented in the CAMPVis platform. Therefore, we decided to adapt the classic ECS model to a platform featuring a library of systems, which behave like building blocks and can be easily assembled together for each individual application. We call these building blocks of systems *processors* as they encapsulate specific algorithms. This approach is very similar to modules in MeVisLab and processors in Voreen. It encourages the developers to break their problems into sub-problems, which facilitates reusing code and provides an excellent rapid-prototyping environment.

However, based on our experience, there are certain limits in terms of generality. Forcing developers to design the processors as generic as possible either leads to a flood of very tiny and specific processors or to massive blobs that do everything alone and try to handle every corner case. Since we consider both these extremes as not desirable, we want to provide as much freedom as possible when combining the processors. Therefore, contrary to MeVisLab or Voreen, CAMPVis does not impose a fixed a-priori structure, such as a data flow network. Instead, CAMPVis offers the very generic concept of a *Pipeline* that coordinates the data domain (DataContainer) with the algorithm domain (processors). Every pipeline works on a single DataContainer and can aggregate multiple processors (Figure 5). The evaluation logic can be automated (e.g. to simulate a data flow network) as well as a custom pipeline-specific implementation. This provides maximum freedom to developers by allowing them to implement pipeline-specific code directly in the pipeline, instead of forcing them to extend existing processors or writing new ones for one-time tasks. In ECS terminology, a pipeline represents a system and pipelines are either executed in a continuous (render) loop or event-based on user interaction.

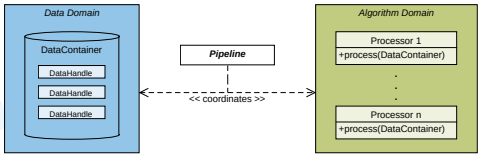

Figure 5: Illustration of the CAMPVis pipeline-processor concept. CAMPVis pipelines take care of coordinating the algorithm domain (processors) with the data domain (DataContainer).

CAMPVis processors are designed to be very loosely coupled: Following the ECS paradigm, a processor only implements

functionality and is inherently state-less regarding the data domain. Hence, it neither is aware of other processors it might be collaborating with nor does it know a-priori on which data to work on. Instead, this information is provided when executing the processor by passing a reference to the DataContainer holding the entities. This is also the indented way of coordinating the processors within a pipeline (cf. Figure 5). During execution, the processor queries the DataContainer for entities with certain components and performs its computations on them. The results are then pushed back into the DataContainer so that they can be used by other processors.

3.4 Properties

In order to support the rapid prototyping design goal, CAMPVis also offers a property system to configure processors and pipelines, similar to the one found in ITK, Voreen and Inviwo. Instead of directly configuring a class through primitive local member variables, one should use *Properties*, which wrap around data types and offer various benefits:

- Providing an observer-like behavior
- Providing automatic getter/setter methods
- Taking care of thread-safety
- The application package can provide an automatically generated GUI

3.5 Module Architecture

One of the initial requirements was to provide sandbox environments for developers. Since CAMPVis is intended to be used as research platform in a heterogeneous environment, there may be a large number of developers working on different projects at the same time.

Therefore, we developed a module system using CMake build scripts. It parses the file system for available modules and allows the user to select for each module whether it should be included into the build or not. Thereby, each module can be considered as a separate, independent sandbox minimizing possible side effects when having multiple projects sharing the same code base. For instance, if one module requires an external library as dependency, other independent modules are still able to compile and run without it. At the same time however, it is possible to define module dependencies so that developers can easily reuse code.

4 Framework Features

The CAMPVis software framework offers various features to programmers to implement their objective. We will present the most important ones in this section.

4.1 Signal Manager

For a large-scale software framework, inter-object communication becomes an important issue. Game engines often feature an event system for this purpose, where objects notify potential listeners by sending generic events through a central event manager instance [17]. Since sender and receiver of events do not know each other, the sender object simply sends an event message of certain type and the event manager then takes care of delivering the message to all listeners that registered themselves for this event type. The major advantage of such a system is its simplicity in design, easy decoupling of sending and receiving events (asynchronous messaging) and the fact that all communication runs through a central place, which makes tracking, monitoring and debugging of communication easy.

However, for our targeted software platform for medical imaging, such a system has one disadvantage: The mapping between sender and receiver of a message is solely based on the event type. Since the inheritance graph of our software framework is broader than it is deep (i.e. many communicating classes inherit from the same base class), effective filtering and routing of messages becomes an issue: One solution would be declaring a distinct event type for each subclass. However, since the semantic nature of all those events is the same (only the sender type changes), this approach would not follow clean object-oriented software design. The other solution would be to define the event type in the common base class, so that all child objects send messages using the same type. In such a case, however, the receiving object would receive all messages of all objects and thus needs to filter out the relevant messages.

Therefore, we decided to use the signal-slot pattern for our software framework and enhance it with the central manager part of event systems. As in the traditional signal-slot pattern presented by Qt [19], relationships between senders (signals) and receivers (slots) are defined through connections. Thus, the sender does know which objects would like to receive its messages. The actual processing of the communication is however done by the signal manager, which acts as a central singleton and takes care of the dispatching of messages. This way, we achieve the flexibility of signals and slots in combination with easy tracking and monitoring, as well as with implicit asynchronous messaging.

Our signal-slot API allows for emitting signals in three different ways:

Direct/blocking call Using `signal::triggerSignal()`, the signal will be processed directly in the emitting thread, the call will block until all signal processing has finished.

Asynchroneous call Using `signal::queueSignal()`, the signal will be put into the signal manager queue and processed asynchroneously in the signal manager thread. Hence, the call will immediately return.

Default call Using `signal::emitSignal()`, the signal will be queued by default, unless the calling thread is also the signal handling thread (i.e. in case of cascading signals). This ensures that cascading signals are processed in a single batch.

Implementation Details We designed a special data structure to store the per-signal connection information. Our `concurrent_pointer_list<T>` data structure is a list-like container that allows for thread-safe bidirectional iteration, insertion and removal of elements. Instead of removing deleted items from the data structure, we mark them internally as `nullptr`. Since connections are stored as pointers to slots and pointer types support atomic operations, this is an effective solution to avoid per-signal mutexes.

Furthermore, we use a memory pool for creating the signal handles, which are relatively tiny objects and created with high frequency from different threads. Relying on the default system allocator here would introduce a performance penalty as it usually employs critical sections around each allocation and deallocation. Implementing a pre-allocated memory pool as custom allocator elegantly cirvumcents this issue. For minimal efforts, we use `tbb::memory_pool` in CAMPVis, which yields a speedup of almost 20% compared to the standard memory allocator.

To facilitate the debugging of sent messages, we implemented a transparent debug layer into our signaling API. When built with the debug flag enabled, all emitted signals will automatically store information on the calling function, file and line together with the sent message (Listing 1). This compensates for the incomplete call stack information in multi-threaded, asynchronous messages.

```
class signal0 {
  // [...]
  void triggerSignal();
}

#ifdef CAMPVIS_DEBUG
struct signal0_debug_decorator : public signal0 {
    // overload method to store debug information and return this
    signal0& triggerSignal(string caller, string file, int line) {
    storeDebugInfo(caller, file, line);
    return *this;
  }
};

// redefine all necessary symbols
#undef signal0
#define signal0 signal0_debug_decorator
#undef triggerSignal
#define triggerSignal triggerSignal(__FUNCTION__, \
  __FILE__, __LINE__).triggerSignal
#endif
```

Listing 1: Code excerpt of the debugging hooks for our signal-slot API. Using C macros, we can redefine the signal symbol to a debug implementation that stores information on the calling function, file and line for debugging purposes.

4.2 Factory Registration

Another notable feature of the CAMPVis software platform is dynamic module registration. This lets modules register their classes with object factories, so that the core code can generically access their functionality without actually knowing of them at build time. With CAMPVis, this counts for instance for the dynamic registration of image converters, pipelines and property widgets.

CAMPVis combines the C++ static registration idiom with the factory pattern. Using C++ templates and static member variables, we achieve a non-intrusive solution for automatic module registration at static initialization time. Its central piece is the templated Registrar class with two essential parts:

1. a static function to create an object of that type (as it knows the object type through the template),
2. a static integer member variable.

The initialization of the integer member is performed by a function call to the factory singleton's registration method (cf. Listing 2), which returns an integer to store in the static field (since the actual value does not matter, our implementation returns the sequential number of the just registered object).

```
// The Registrar class takes care of the actual registration.
// Template parameter is the type of the registree
template<typename T>
class Registrar {
  // Static factory method for creating the object of type T.
  static AbstractRegistree* create() {
    return new T(dc);
  }

  // static field stores the result of the registration function
  static const size_t _factoryId;
};

// Static initialization performs the function call to register
// the registree with the factory.
template<typename T>
const size_t Registrar<T>::_factoryId =
  Factory::register<T>(&Registrar<T>::create);

// In a particular module, define a concrete class to register:
class ConcreteRegistree : public AbstractRegistree {
  // [...]
}

// Explicit template instantiation: instantiate the registrar,
// which performs the registration at static initialization time
template class Registrar<ConcreteRegistree>;
```

Listing 2: Code excerpt showing the C++ static registration idiom we use for factory registration.

4.3 Scripting Layer

CAMPVis has an optional scripting layer, which can be enabled in order to add support for runtime-scripting with the Lua scripting language [6]. This serves various purposes: On the one hand, it offers a scripting console allowing the user to inspect and modify the data model at runtime through Lua commands. Similar to MATLAB or many game engines, this provides developers with straight-forward interaction with their software at runtime without the need to explicitly program a graphical user interface. On the other hand, the scripting layer allows defining entire pipelines in Lua scripts (cf. Listing 3). This further accelerates the rapid prototyping nature of CAMPVis since changes to pipelines no longer relate to a recompilation of C++ code but only to updating the script and loading it at runtime.

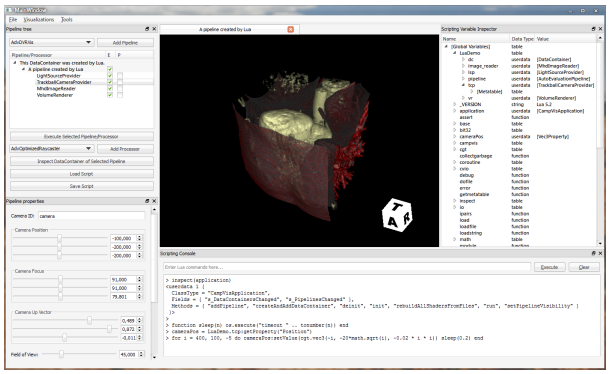

Figure 6: Screenshot of CAMPVis with enabled Lua scripting functionality. The scripting console in the bottom dock allows for accessing and manipulating the internal CAMPVis data model at runtime, for instance to modify the properties of processors in a systematic fashion. Furthermore, the widget in the dock on the right-hand side allows to inspect all variables and tables that are present in the Lua virtual machine.

Furthermore, we exploit the scripting support as persistence mechanism: To save the current program state in a file, CAMPVis writes a Lua script holding the necessary code to recreate the pipeline and its state. The program state can later be recovered by simply executing the Lua script.

Implementation Details C++ classes are exposed to the Lua virtual machine by creating appropriate bindings for them. The scripting layer uses SWIG [1] to wrap selected CAMPVis modules and core packages, and generate Lua modules that make them available to scripted pipelines. The process of generating bindings is not completely automated - SWIG must be told what classes to wrap, which of their members to expose, and how to deal with advanced C++ features such as templates. This information is encoded in interface files which are passed to SWIG along with the C++ code to produce Lua modules (cf. Figure 7).

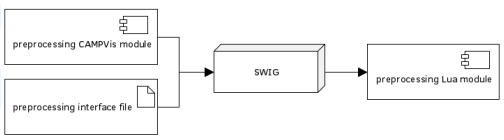

Figure 7: Illustration of the Lua binding generation process using SWIG. In order to expose CAMPVis functionality to the Lua virtual machine, SWIG uses interface files to create Lua bindings for the existing CAMPVis code.

```
require("application")
require("cvio")
require("vis")

-- All created CAMPVis objects have to be kept alive and must not
-- be garbage-collected by Lua. Thus, we create a global table for
-- this script, where everything resides in
LuaDemo = {}

-- create DataContainer and Pipeline
LuaDemo.dc = application:createAndAddDataContainer("DC Name")
LuaDemo.pipeline = campvis.AutoEvaluationPipeline(
  LuaDemo.dc, "Pipeline Name")
local pipeline = LuaDemo.pipeline;

-- create the processors we need
local canvas_size = pipeline:getProperty("CanvasSize")
LuaDemo.lsp = base.LightSourceProvider()
LuaDemo.tcp = base.TrackballCameraProvider(canvas_size)
LuaDemo.image_reader = cvio.MhdImageReader()
LuaDemo.vr = vis.VolumeRenderer(canvas_size)

-- register the processors with the pipeline
pipeline:addProcessor(LuaDemo.lsp)
pipeline:addProcessor(LuaDemo.tcp)
pipeline:addProcessor(LuaDemo.image_reader)
pipeline:addProcessor(LuaDemo.vr)

-- setup event listener and register it with the pipeline
LuaDemo.tcp:addLqModeProcessor(LuaDemo.vr)
pipeline:addEventListenerToBack(LuaDemo.tcp)

-- create an init callback function, so that the following code
-- gets called when the  pipeline gets initialized by CAMPVis.
local initCallback = function()
  -- set up the processors' properties
  LuaDemo.vr.p_outputImage:setValue("combine")
  pipeline:getProperty("renderTargetID"):setValue("combine")

  LuaDemo.image_reader.p_url:setValue(campvis.SOURCE_DIR
    .. "/modules/vis/sampledata/smallHeart.mhd")
  LuaDemo.image_reader.p_targetImageID:setValue("reader.output")
  LuaDemo.image_reader.p_targetImageID:addSharedProperty(
    LuaDemo.vr.p_inputVolume)

  -- automatically adjust the camera to the data
  LuaDemo.image_reader.p_targetImageID:addSharedProperty(
    LuaDemo.tcp.p_image)
end
-- register the callback with the s_init signal
pipeline.s_init:connect(initCallback)

-- Finished creating our LuaDemo - register it with the application
application:addPipeline(pipeline)
```

Listing 3: Example Lua script for creating a fully functional volume rendering visualization with CAMPVis.

4.4 Network Communication

One of the initial requirements during CAMPVis development was a good support for network communication, which has several use cases. Multi-modal image fusion often has to deal with multiple devices, which have an continuous exchange of data such as images, tracking information or control commands. In cases where mobile devices (e.g. tablet computers) lack the necessary computation power, distributed computing can furthermore provide a solution where the actual computations are performed on a stationary workstation and only the results are streamed to the mobile device.

The original plan was to integrate such a network communication stack directly into the CAMPVis core, as it is done with many video game engines. However, since there are already various established solutions available for streaming medical imaging data over network, we decided against implementing another protocol. Instead, networking support is enabled through CAMPVis modules. The current implementation features wrappers for both CAMPCom [21] and OpenIGTLink [26], two state-of-the-art libraries for real-time streaming of medical imaging data.

5 Clinical Application

As discussed in Section 1, it is important to keep the gap between research/development and deployment small. This is the case in particular in the context of medical imaging and visualization where the ultimate goal is always to improve the work of the clinician and/or the outcome for the patient. Since this can only be evaluated by the domain experts themselves, one needs to implement a working prototype that suffices clinical requirements. To demonstrate the capabilities of CAMPVis in this regard, we will present two case studies as examples where the CAMPVis platform was used for both development and deployment.

5.1 Real-time Uncertainty Visualization for 2D B-mode Ultrasound

As first example, we discuss a system for real-time uncertainty visualization for 2D B-mode ultrasound, which was presented in [22].

The system setup consists of an ultrasound machine and a standard workstation, which are connected through wired network (cf. Figure 8). The ultrasound system acquires the raw echo data and performs the initial processing into a B-mode image, which is then sent via OpenIGTLink to the workstation. There, the CAMPVis application receives the image through the `OpenIGTLinkClient` processor and performs some further filtering using the corresponding processors. The `CudaConfidenceMapsSolver` processor computes the Confidence Map, which is eventually fused into the final visualization by the `AdvancedUsFusion` processor. This rendering is then routed back to the display of the ultrasound machine. Since the new visualization is shown at exactly the same place as a normal ultrasound image, and the user only interacts with familiar hardware, our system has a minimal impact on the evaluation results and we can safely assume that they really represent the added value of the visualization technique itself and are not biased by the quality of the system integration.

Figure 8: Deployment of CAMPVis into a clinical setup for real-time B-mode ultrasound uncertainty visualization. The image is acquired by the ultrasound machine (left) and sent to a workstation running CAMPVis (right) through a network connection. CAMPVis interactively performs the processing of the incoming images and routes the resulting images back to the ultrasound machine. This way, our system is minimally intrusive since the domain expert only has to deal with the familiar ultrasound device.

Since the presented CAMPVis processing pipeline is entirely composed of processors as building blocks, each of the processing steps can be easily reused in a different project to maximize synergy effects (cf. Section 1). Furthermore, the asynchronous implementation in the CAMPVis framework allows for a smooth and interactive visualization even if single ultrasound frames should drop (e.g. due to network issues).

5.2 Multi-modal Image-guided Prostate Biopsy

As second example, we present an integrated system for multi-modal image-guided prostate biopsy, which was recently devel-

oped in our lab and improves the specificity of biopsy results for prostate cancer diagnosis [23, 30]. For this application, the current gold standard is a random biopsy of 10 to 12 samples under trans-rectal ultrasound (TRUS) guidance. However, this approach is prone to a high number of false negative results due to TRUS hardly highlighting suspicious regions. We improve this workflow by additionally acquiring a combined pre-operative PET-MR image of the anatomy using a ^{68}Ga-labeled PSMA radioactive tracer exhibiting a very high specificity for prostate cancer [7]. After registering the PET-MR images with the tracked ultrasound, our system uses multi-modal visualization for guiding the biopsy toward the highly suspicious regions.

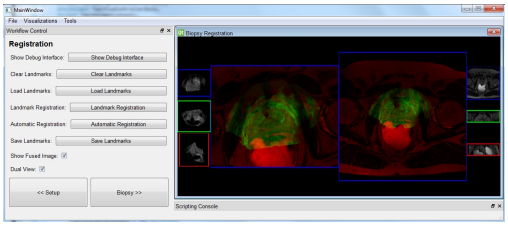

Figure 9: Screenshot of the developed multi-modal image-guided prostate biopsy framework [23, 30] implemented in CAMPVis. This image shows the semi-automatic registration step for aligning the tracked ultrasound images with the MR volume, for which the clinician defines a set of corresponding landmarks in each image.

Since the clinical workflow consists of multiple steps, we implemented a custom user interface for the clinician. As a first step, the system is initialized, a fully-automatic machine learning-based registration is performed and all necessary data is read from the disk. In the second step, CAMPVis presents a slice-based multi-modal visualization to the clinician, in which they evaluate the result of the automatic MR-TRUS registration. If the images are not aligned well, the clinician can define a set of anatomical landmarks in each modality in order to yield a better registration. The final step then uses OpenIGTLink to stream the ultrasound image and tracking information to CAMPVis, similar as with the presented ultrasound uncertainty visualization system. A real-time multi-modal visualization of the data the guides the clinician. One window shows the live B-mode ultrasound image and a second window shows the corresponding MPR in the PET-MR volumes, both featuring a needle guide to target suspicious regions appearing with high intensity in the PET data.

Since most functionality is encapsulated in processors, many parts of this project's code can easily be reused. For instance, the `OpenIGTLinkClient` processor is the same as the one used for the uncertainty visualization system and the `BiopsyMprRenderer` processor of the third workflow step is also used in a similar fashion in other CAMPVis-based projects. As shown in [30], our system does not only yield significantly improved clinical results in terms of biopsy specificity but also got very good feedback from our clinical partners in terms of usability and effectiveness.

6 Conclusion

We presented a new software framework targeted for both development of medical imaging and visualization techniques in heterogeneous research groups as well as for deployment into a clinical setup. As novel approach, the system architecture and design was inspired by modern video game engines, which are very effective in handling large amounts of data in an interactive real-time environment.

Future work includes the portation of CAMPVis onto mobile tablet computers as well as some adaption of the data model in order to move even closer to the classic Entity Component System architecture. CAMPVis will be published under the Apache License, Version 2.0.

Acknowledgements

The development of CAMPVis is funded by the the "SoftwareCampus" program of the German Federal Ministry of Education and Research (BMBF, Förderkennzeichen 01IS12057) .

References

[1] Beazley, D. Simplified Wrapper and Interface Generator (SWIG), 2015.

[2] Bilas, Scott. A Data-Driven Game Object System, 2002.

[3] I. Bitter, R. Van Uitert, I. Wolf, L. Ibanez, and J.-M. Kuhnigk. Comparison of four freely available frameworks for image processing and visualization that use ITK. *Visualization and Computer Graphics, IEEE Transactions on*, 13(3):483–493, 2007.

[4] C. P. Botha and F. H. Post. Hybrid scheduling in the devide dataflow visualisation environment. In *SimVis*, pages 309–322, 2008.

[5] J. J. Caban, A. Joshi, and P. Nagy. Rapid development of medical imaging tools with open-source libraries. *Journal of Digital Imaging*, 20(1):83–93, 2007.

[6] Celes, W. and de Figueiredo, L. H. Lua (programming language), 2015.

[7] M. Eiber, S. G. Nekolla, T. Maurer, G. Weirich, H.-J. Wester, and M. Schwaiger. 68ga-psma pet/mr with multimodality image analysis for primary prostate cancer. *Abdominal imaging*, pages 1–3, 2014.

[8] A. Fedorov, R. Beichel, J. Kalpathy-Cramer, J. Finet, J.-C. Fillion-Robin, S. Pujol, C. Bauer, D. Jennings, F. Fennessy, M. Sonka, et al. 3D slicer as an image computing platform for the quantitative imaging network. *Magnetic resonance imaging*, 30(9):1323–1341, 2012.

[9] T. Fogal and J. Krüger. Tuvok, an architecture for large scale volume rendering. In *Proceedings of the 15th International Workshop on Vision, Modeling, and Visualization*, pages 139–146, 2010.

[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1994.

[11] Intel Corporation. Threading Building Blocks.

[12] Kitware, Inc. Cmake: Cross platform make, 2015.

[13] Kitware, Inc. Insight segmentation and registration toolkit (itk), 2015.

[14] Kitware, Inc. Visualization toolkit (vtk), 2015.

[15] E. Lengyel. *Game Engine Gems 2*. CRC Press, 2011.

[16] Martin, Adam. Entity Systems are the future of MMOG development, 2007.

[17] M. McShaffry. *Game coding complete*. Cengage Learning, 2012.

[18] J. Meyer-Spradow, T. Ropinski, J. Mensmann, and K. Hinrichs. Voreen: A rapid-prototyping environment for ray-casting-based volume visualizations. *Computer Graphics and Applications, IEEE*, 29(6):6–13, 2009.

[19] Qt Company Ltd. Signals & Slots in Qt.

[20] F. Ritter, T. Boskamp, A. Homeyer, H. Laue, M. Schwier, F. Link, and H.-O. Peitgen. Medical image analysis. *Pulse, IEEE*, 2(6):60–70, 2011.

[21] A. Schoch, B. Fuerst, F. Achilles, S. Demirci, and N. Navab. A Lightweight and Portable Communication Framework for Multimodal Image-Guided Therapy. 2013.

[22] C. Schulte zu Berge, D. Declara, C. Hennersperger, M. Baust, and N. Navab. Real-time uncertainty visualization for b-mode ultrasound. *Scientific Visualization 2015, Proceedings on*, 2015.

[23] A. Shah, O. Zettinig, T. Maurer, C. Precup, C. Schulte zu Berge, J. Weiss, B. Frisch, and N. Navab. An open source multimodal image-guided prostate biopsy framework. In *Clinical Image-Based Procedures. Translational Research in Medical Imaging*, pages 1–8. Springer, 2014.

[24] E. Sundén, P. Steneteg, S. Kottravel, D. Jönsson, R. Englund, M. Falk, and T. Ropinski. Inviwo – an extensible, multi-purpose visualization framework. 2015.

[25] The Open XIP Project. Extensible imaging platform (xip), 2015.

[26] J. Tokuda, G. S. Fischer, X. Papademetris, Z. Yaniv, L. Ibanez, P. Cheng, H. Liu, J. Blevins, J. Arata, A. J. Golby, et al. OpenIGTLink: an open network protocol for image-guided therapy environment. *The International Journal of Medical Robotics and Computer Assisted Surgery*, 5(4):423–434, 2009.

[27] M. West. Evolve Your Hierarchy: Refactoring Game Entities with Components, 2007.

[28] K. Wilson. Game Object Structure: Inheritance vs. Aggregation, 2002.

[29] I. Wolf, M. Vetter, I. Wegner, T. Böttger, M. Nolden, M. Schöbinger, M. Hastenteufel, T. Kunert, and H.-P. Meinzer. The medical imaging interaction toolkit. *Medical image analysis*, 9(6):594–604, 2005.

[30] O. Zettinig, A. Shah, C. Hennersperger, M. Eiber, C. Kroll, H. Kübler, T. Maurer, F. Milletarì, J. Rackerseder, C. S. zu Berge, et al. Multimodal image-guided prostate fusion biopsy based on automatic deformable registration. *International journal of computer assisted radiology and surgery*, 10(12):1997–2007, 2015.