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

Christian Schulte zu Berge, Artur Grunau, Hosseinz Mahmoud, 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 straightforward way. This boosts cooperation and synergy effects between different developers and the reusability of developed solutions.

1 Introduction

The recent advances in the field of medical image computing provides today’s clinicians with an astonishing collection of imaging modalities and algorithms for automatic image analysis. Thus, intra-operative visualization has become an essential part of medical interventions and provides the surgeon with powerful support for his work. Since every imaging modality has its individual strengths and weaknesses the key to a successful visualization is the fusion of all available data so that clinicians get exactly the information they need. However, bringing innovations from research into the daily workflow of clinicians is a difficult and time-consuming task since the deployment in the operation room yields various challenges for the developers.

Medical imaging and visualization is an extensive field of research and brings together experts and researchers of many different disciplines. Research groups often split the task into sub-problems and domain specialists tackle them in individual projects looking for the optimal solution. Companies then try to implement these methods into (often highly specialized) end-products, which eventually are to be used by the clinicians. 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, created software libraries are often abandoned after finishing the project and are hardly designed to be reused by others.

Our CAMPVis software platform is designed to fill exactly this gap. Targeting medical image computing and visualization and in particular multi-modal image fusion for intra-operative usage, it provides not only essential data structures and algorithms but also an extensive infrastructure to facilitate the collaboration in heterogeneous environments. With the above motivation, CAMPVis was developed with the following main design goals:

- Platform-independence and standard compliance
- 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 the other to meet one’s code dependencies
- Support of distributed/decentralized computing

Further, we target a modern and clean software design with a slim, modular architecture and as few mandatory external dependencies as possible. Since the CAMPVis project starts mostly from scratch, we want to only use state-of-the-art technologies and avoid deprecated interfaces due to forced backwards compatibility at all costs.

In order to meet these requirements, we use the software 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. Massively multiplayer online games (MMOGs) manage to synchronize the game state over thousands of computers. Hence, video game engines provide a promising approach to combine the handling of a large amount of data, real-time graphics, interactivity and network computing into a uniform, extensible infrastructure.

This technical report introduces the CAMPVis research framework for medical imaging and visualization, which will be released under a permissive open source license. Section 2 provides an overview over similar software platforms and introduces the Entity Component System architecture, which serves as main inspiration for the CAMPVis software design. The general software architecture and design decisions are discussed in Section 3, while important features of the CAMPVis framework are introduced in Section 4. Section 5 presents two example applications in medical imaging and visualization that were realized using the CAMPVis software platform.

2 Related Work

Today, there is a plethora of different software platforms and applications for medical imaging and visualization available, many of them being open source. Some of the platforms, such as ITK [12] focus only on the image processing part while others, such as VTK [23], Voreen [16] 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 [24] focus more on the application domain, other frameworks, such as MeVisLab [17], DeVIDE [9] or XIP [1], 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 [4], while the survey paper of Caban et al. more focuses on the rapid
development aspect of such libraries [6].

2.1 Entity Component System Architecture

Many modern game engines exploit data-driven programming and implement the Entity Component System (ECS) paradigm as main software architecture. While there is no official definition of such a system, most approaches show strong similarities in their central design. The main intention of an ECS is to break the complex inheritance graphs, which arise when using classic object-oriented programming (OOP) paradigms to implement a game engine with its large number of different game objects each often having multiple aspects [3,15].

![UML diagram illustrating the Entity Component System software architecture](image)

**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.

The central idea of ECS is to decouple objects from their state and their functionality (Figure 1). This is achieved through introducing three concepts [14]:

**Entity:** The entity is the single general purpose object that stores neither data nor functionality (i.e. methods). It’s 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. When attaching a component to an entity this 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 very flexible game design were 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 [15].

3 Software Architecture

3.1 Build System

To provide a uniform build process across multiple platforms and compilers, CAMPVis uses the CMake build system [11], which uses a scripting language to define the build process. 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.

![UML diagram of the CAMPVis package structure](image)

**Figure 2:** 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 Entity Component System for CAMPVis

Since the CAMPVis software design is inspired by modern game architectures we use the Entity Component System paradigm as basic architecture. 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 different live object during runtime than common video games and not all of its systems are of fully global nature. Therefore, we applied some modifications to the classic ECS approach (Figure 3).

On the data domain, we implement the classic concept of entities, which act as general purpose object and stores neither data nor functionality. Each entity has a unique identifier (for which 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 towards 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 are the central object to 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, from 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 globs 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 4). 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 the developer by allowing to implement pipeline-specific code directly into the pipeline, instead of forcing him 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 3: 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.

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 intended way of coordinating the processors within a pipeline (cf. Figure 4). 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

Inspired by ITK and Voreen, CAMPVis also offers a property system to configure processors and pipelines. Instead of directly configuring a class through primitive local member variables, one should use Properties, which wrap around a 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 provided GUI

3.5 Module Architecture

One of the initial requirements was to provide a sandbox-like environment for developers. Since CAMPVis is mainly intended as research platform, there may be a large number of developers that are working on different very heterogeneous projects at the same time. Therefore, we developed a module system as sandbox environment into our 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 in the build or not. This minimizes 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. In combination with our factory registration system, the CAMPVis module architecture further allows modules to register their classes with object factories at static initialization time (cf. Section 4.2).
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 [15]. Sender and receiver of events do not know each other and all communication is passed via the event manager. The sender object simply sends an event message of a 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 more broad than 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 need 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 which acts as a central singleton and takes care of 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.
- **Asynchronous call**: Using signal::queueSignal(), the signal will be put into the signal manager queue and processed asynchronously 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.

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.

```cpp
#define triggerSignal triggerSignal(__FUNCTION__, __FILE__, __LINE__).triggerSignal
#endif

// redefine all necessary symbols
#define signal0 signal0_debug_decorator
#define triggerSignal signal0_debug_decorator

class signal0 {
    // overload method to store debug information and return this
    signal0& triggerSignal(string caller, string file, int line) {
        storeDebugInfo(caller, file, line);
        return *this;
    }
}
```

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 introduces a performance penalty, as it usually employs critical sections around each allocation and deallocation. Implementing a pre-allocated memory pool as custom allocator elegantly circumvents this issue. For minimal efforts, we use ttb::memory_pool in CAMPVis, which yields a speedup of almost 20% compared to the standard memory allocator.

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. The 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.

![Figure 5: UML diagram illustrating our static factory registration architecture.](http://campar.in.tum.de/Main/CAMPVis)
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 [7]. This serves various purposes: On the one hand it offers a scripting console, which allows to inspect and modify the data model at runtime through Lua commands. Similar to MATLAB [21] or many game engines, this allows developers 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 to define entire pipelines in Lua scripts. This accelerates the rapid prototyping nature of CAMPVis even further since changes to pipelines no longer relate to a recompilation of C++ code but only to updating the script and loading it at runtime.

Furthermore, we plan to exploit the scripting support as persistence mechanism: In order to save the current program state in a file, CAMPVis will write a Lua script holding the necessary code to recreate the pipeline and its state.

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

```cpp
// The Registrar class takes care for 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 pipeline to register:
class ConcreteRegistree : public AbstractRegistree {
  // [...]
} // Explicit template instantiation instantiate the registrar,
// which performs the registration at static initialization time
template class Registrar<ConcreteRegistree>;
```

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 such as 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 [18] and OpenIGTLink [22], two state-of-the-art libraries for real-time streaming of medical imaging data.

5 Applications

Besides proof-of-concept implementations and simple demos, we used CAMPVis already for two large projects. In cooperation with the urology department of our university hospital we developed a multimodal image-guided prostate biopsy framework (Figure 8). It features the acquisition of prostate 3D freehand ultrasound, which is registered to a pre-operative MR image. Streaming the tracking information into CAMPVis using the OpenIGTLink protocol (cf. section 4.4), we can provide a real-time visualization of the MR image in the same view as the ultrasound plane to support the clinician’s navigation during the biopsy. The initial feedback from the clinicians were very positive [20].

Furthermore, we used the CAMPVis research framework to develop a predicate-based focus-and-context volume rendering technique (Figure 9). We introduce point predicates as a novel generic formulation for integrating the evaluation arbitrary information into the classification process, which allows us to successfully filter clinically relevant from non-relevant information. We complement our approach with the predicate histogram as an effective and intuitive user interface showing an overview over the parameter space and providing with interaction metaphors to interactively manipulate the visualization result in real-time. In combination with a scribble technique we obtain an intuitive workflow, which allows also non-expert users to obtain insightful visualizations [19].

6 Conclusion

We presented a new software framework targeted for medical imaging and visualization and in particular multi-modal image fusion for intra-operative usage. 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. First applications show that our research platform can be successfully used as basis for concrete projects. 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 “Software-Campus” program of the German Federal Ministry of Education and Research (BMBF, Förderkennzeichen 01IS12057).

References


