ARCS: A framework with extended software integration capabilities to build Augmented Reality applications

Conference Paper · March 2012

DOI: 10.1109/SEARIS.2012.6231170

CITATIONS
2

READS
4,592

4 authors, including:

Jean-Yves Didier
Université d'Évry-Val-d'Essonne
67 PUBLICATIONS **383** CITATIONS

SEE PROFILE

Malik Mallem
Université d'Évry-Val-d'Essonne
178 PUBLICATIONS **1,084** CITATIONS

SEE PROFILE

Samir Otmane
Université d'Évry-Val-d'Essonne
121 PUBLICATIONS **611** CITATIONS

SEE PROFILE

ARCS : A Framework with Extended Software Integration Capabilities to Build Augmented Reality Applications

Jean-Yves Didier* Mehdi Chouiten† Malik Mallem‡

laboratoire IBISC
Université d'Evry Val d'Essonne
FRANCE

ABSTRACT

Nowadays, many Augmented Reality (AR) frameworks are using a component based approach. It ensures that the non functional requirements of AR such as extensibility, configurability and modularity are handled.

One of the challenge the AR application designer have to face is the problem of the integration of multiple technologies that could be provided by different component systems that do not cover the same issues. In this paper, we will see how our AR component based framework, named ARCS (Augmented Reality Component System), is intended to integrate other components from exogenous component systems and how it could be integrated into other software. At the same time, we also tackle the issues that may arise in multi-threaded environments and explain how we implement the networking non functional requirement. Then, we discuss and study the impact such genericity has on performances of our system.

Finally, our approach will be exposed and validated through different examples of different sizes.

Index Terms: D.2.11 [SOFTWARE ENGINEERING]: Software architectures—Domain specific architectures D.2.13 [SOFTWARE ENGINEERING]: Reusable software—Reusable libraries D.2.m [SOFTWARE ENGINEERING]: Miscellaneous—Rapid prototyping H.5.1 [INFORMATION INTERFACES AND PRESENTATION]: Multimedia Information Systems—Artificial, augmented, and virtual realities

1 INTRODUCTION

Nowadays, Augmented Reality (AR) tends to go upstream. Many applications are made available to the general public in fields such as entertainment or commercial prospect.

However, such applications, in order to spread more and more, should be as easy as possible to develop. From the developer view, to write an AR application from scratch is very challenging and requires many skills in different domains of programming such as image processing, 3D graphics, data acquisition from different types of devices, networking and so on.

This is where component-based architectures are interesting because they offer solutions to non-functional issues of AR applications such as genericity, modularity and reusability. Each component can also address issues from functional requirements such as tracking, image processing and so on. Therefore, the work of a developer on such kind of framework is easier: he can reuse some old software parts and link them with new components he has developed. Therefore, writing AR application becomes more and more a software integration problem.

In this paper, we will examine to which extent we will process the software integration problem in order to ease the development of AR applications. Our proposed solutions are implemented in our framework named ARCS (which stands for Augmented Reality Component System). We will address specifically two issues of software integration. The first one is the problem of offering capabilities to integrate other component systems in our framework. The second one is the reverse problem, that is to say the integration of our engine (which is a part of our framework) as a third party software component inside an application. We will also tackle common non functional requirements such as multi-threading and networking.

The remaining is organized as follows: first, we will introduce some related works about AR component-based software framework. Then, we will examine the two software integration problem: the first one about the integration of components provided by other component systems and the second one is the integration of a component-based software as a third party in another application. At the same time, we will also address the issues of multi-threading and networking. We will also have a brief discussion on the performance issue that arises when such solutions are set up. Finally, we will present different applications we developed using our framework in order to validate it.

2 RELATED WORKS

Modular software architectures and frameworks for MR have been addressed during these past years by almost thirty different projects of frameworks. The survey written by Endres et al. [5] is quite exhaustive. Therefore, we chose to cite the most prominent component based systems.

Amongst the most remarkable ones, we can mention the Studier-Stube led by the Technical Universities of Vienna [6]. Its main principle is that each user has its own workspace described by a distributed scene-graph which parts may be common with other users. Workspaces are able to discover each other using a centralized session manager. In order to manage sensors as well as input devices, this project integrates OpenTracker [15], a subsystem relying on an XML description of data flows. Developers can program using scripts and predefined nodes as well as develop their own C++ nodes. One can notice this framework is well suited for collaborative and distributed AR applications.

Started in year 2000 at the Technical University of München, the DWARF [1] (Distributed Wearable Augmented Reality Framework) project uses decentralized and distributed services. This framework relies on CORBA [10] (Common Object Request Broker Architecture) to provide services. As an example, each tracker becomes a service broadcasting data to other services (that could be filters, rendering loops,...). Each service is described by XML data and can be dynamically linked to the current application, which is seen as a distributed data flow. At start-up, applications only need one component manager. This last one discovers step by step other

*e-mail: jean-yves.didier@ibisc.fr
†e-mail: mehdi.chouiten@ibisc.fr
‡e-mail: malik.mallem@ibisc.fr

services in order to integrate them at execution. DWARF is one of the most accomplished distributed AR framework nowadays, however the core use of CORBA and the large component granularity might make it not suitable for small AR applications. Some other frameworks are following the same guiding precepts such as the AMIRE (Authoring Mixed Reality) project [4] or MORGAN [11].

ImageTclAR [12] aims at providing a rapid prototyping environment to test and design MR applications. People can use proposed components or develop their own ones in C++ whereas the whole logical glue between components is written using Tcl interpreted scripts. ImageTclAR contains scripts that make it possible to create new component skeletons as well as a graphical editor in order to link visually components and generate the corresponding logical glue. The scripting paradigm is also used in the commercial software D-Fusion[19].

Tinmith [13] is a library written to develop mobile AR systems. This framework is based on data-flow description and is a library of hierarchical objects. It manages data-flow from sensors, many data filters and rendering components. Objects written in C++ rely on a callback system and data are serialized using XML. Communications between objects are managed through a data-flow graph. An application written with Tinmith is organized in different layers that modify the data-flow from data acquisition to final scene rendering. This architecture, development oriented, is suitable for mobile AR systems, especially embedded ones due to the effort made to optimize elementary system components. Custom developments of such tools has also been proposed in VHD++ [14], MRSS [8] (Mixed Reality Software Suite) or Avango NG [9].

An AR framework must cope with works in progress and future works built on the present ones. It should be able to integrate technologies of tomorrow in terms of new devices and algorithms. Works in progress would then require flexibility, future works extensibility and future works built on top of current ones reusability. One of the classical answers is to use a component based software architecture because it allows to separate an application into several components, that are, according to Szyperski [18], *"a coherent package of software that can be independently developed and delivered as a unit, and that offers interfaces by which it can be connected, unchanged, with other components to compose a larger system"*. Another benefit associated to reusability is the ability to rapidly prototype applications. However, these systems address partially the problem of software integration. They usually do not offer capabilities that would ease the integration of other component systems and force the framework developer to extend his framework to work with other component systems on a per need basis. At the same time, components usually developed with such frameworks are usually not usable by other framework that would need to integrate them. We will now see how those two software integration problems are solved in our framework.

3 Integrating exogenous component systems

As intended when it was started in 2005, ARCS is a component based framework [3]. Since then, it has gone through a major revision in order to integrate multi-threading in its application model and introduces a more generic component model in order to integrate exogenous component systems. Those results are exposed in this paper.

The framework contains a library storing a lightweight engine written in C++ that parses and runs applications according to an XML description of them. In order to achieve this, it loads components from dynamic libraries at runtime, instantiate and configure them. The framework is completed with some other tools in order to ease the generation of component libraries and the editing of application descriptions.

As a framework, ARCS introduces several models that should be followed in order to build an AR application. We will first briefly introduce its application model. Then, we will deeply detail its component model and show how it can support integration of components coming from other component system (which we call *exogenous* component systems since they are not included in the engine by opposition to *native* ARCS component system). Then we will also tackle the multi-threading and networking issues that may arise.

3.1 Application model

The ARCS application model describes applications as a set of process (actually a set of threads). Each one of them is controlled by a finite state machine. When the internal state of the statemachine changes, it triggers changes in how components are connected to each other. A set of connections as well as a set of component invocations (in order to initialize them and launch data processing) is called a *sheet*. According to this, each process is, at a given time, in a given state that corresponds to an active sheet. Sheets are sharing the same components instantiated from a component pool. Therefore, components do not belong to any thread in particular but slots may be invoked by different threads.

This model allow us to describe the multi-threading aspects of applications directly inside the framework and relieves the component developer from implementing specific methods related to multi-threaded concerns inside components.

As we said before, each process maintains one active sheet at a time. A sheet activation cycle follows the following steps:

- The controller (statemachine) receives a token (from a component in the active sheet) that will trigger one of its transitions and then change the controller state;
- All components from the current sheet will be disconnected from each other;
- The controller new state corresponds to another sheet. First, some invocations, called *pre-connection invocations*, are performed on the components to properly initialize them;
- Then connections are established according to the new sheet description;
- Post-connection invocations are finally performed in order to launch the actual processing of data.

3.2 Component model

The component model is important in ARCS because it allows to integrate components from other component systems. ARCS describes components as entities having signals (outputs) and slots (inputs). The signal/slot mechanism is well known since it is commonly used in graphical user interface libraries such as Qt for instance. It is also deriving from the observer design pattern. Thus, the communication through a signal/slot connection is synchronous. Composition of components can be performed in two different ways: the first one is through *connection composition*, where a component emits a signal that is processed by a slot from another component. The second one is through *invocation composition*: a component is passed as a parameter to a slot of another component.

Amongst its specificities, ARCS proposes an abstract component model that allows to introduce new component types or new component behavior as long as they respect the signal/slot scheme. Therefore, a class represents in ARCS *AbstractComponents*. To make it recognize by the ARCS engine a new type of component coming from another component system, one must subclass the *AbstractComponent* class.

Therefore, subclassing it should define the following functionalities:

- Instantiation and destruction of the actual component: *AbstractComponent* is an interface that acts as a wrapper around

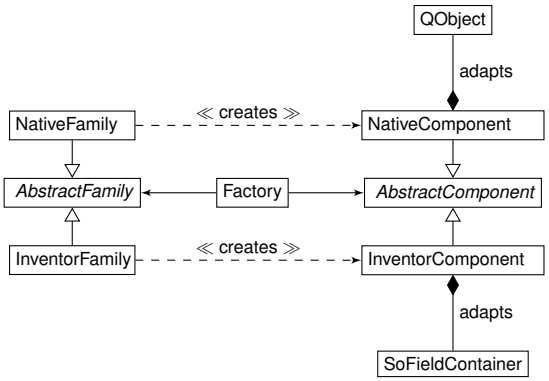


Figure 1: UML class diagram of ARCS engine's component factory.

the actual component and should at the same time manage references to it ;

- Signal/slot management: inputs and outputs of the actual component should be redefined and made available to the ARCS engine as signals and slots;
- Connection management: wrapping is known to introduce some computation time overhead. Here, connection management helps to improve performances on connections between components coming from the same component system. In the case the two components do not belong to the same component system, the connection management should also implement a fallback mechanism in order to make the two components work together;
- Serialization/Deserialization: this is mainly used at the instantiation of the actual components. Deserialization allows to configure the component according to string contents passed through an XML description of the application.

Since the idea is to make it possible to extend ARCS by interfacing other component systems with it, the ARCS engine contains a factory that could be extended by introducing new *Families* of components, each family being related to a different component system. The component *Factory* in the ARCS engine can then use several families deriving from an *AbstractFamily* that is the interface in order to instantiate components. Its main functionalities are the management of component factories as well as the instantiation of components.

Figure 1 shows the class diagram of a part of the ARCS engine where the factory is operating. It relies on the abstract factory design pattern [7] in order to provide components. In the figure two concrete component families are introduced: the *NativeFamily* which instantiates *NativeComponents* which are the privileged components handled by the engine and the *InventorFamily* which instantiates *InventorComponents* that will be presented longer in section 4.1. As stated before, *AbstractComponents* act as wrappers to actual components. In the case of the *NativeComponents*, it is *QObject* objects, a base class of Qt library which is implementing the signal/slot mechanism and written in C++. In the case of the *InventorComponents*, it is objects of *SoFieldContainer* class, which is the base class for OpenInventor nodes and engines. Other families of components have been developed in order to manage distribution of an AR application through a network (see section 3.3.2 for more details) as well as scripting components in order to ease the rapid prototyping of logical glue components.

In order to complete the integration of other component families, the framework has also abstract type factories that parse XML descriptions and instantiate directly objects of the considered type that can be used as parameters of components invocations in order to initialize the latter ones.

These functionalities are also exposed in the dynamic libraries the ARCS engine can load at runtime. They usually contain:

- Type factories, in order to extend ARCS with new types when needed and serialize them;
- Native component factories that instantiate components handled by the previous version of the engine and that are the privileged components in the framework;
- Family component factories in order to make ARCS compatible with other component systems.

As a first example, we will explain how the inventor family is implemented inside a dynamic library loadable by the ARCS engine.

3.3 Handling other non-functional requirements

In ARCS, we decided to develop an engine that is as lightweight as possible. Therefore, the engine in itself does not implements all the non-functional requirements that would be necessary to make all kind of applications. Nevertheless, it is possible to explore other ways to add non-functional requirements when they are really needed. We will briefly see how it possible by examining how the concurrency management through multi-threading and how networking, that are two non-functional requirements, are handled into ARCS.

3.3.1 Multi-threading

As we stated before, in ARCS, we can express an application as a set of threads working and synchronizing together. Slots of a component may be invoked by different threads. Therefore, components are shared across threads and prone to concurrency problems that arise in multi-threading context. In order to solve most of the problems, one solution is to use monitors in order to make component slots accessible by only one thread at a time. Its consists in locking mutual exclusions (called mutexes) once a thread invokes a slot and unlocking it once the slot have been invoked. There are at least three different ways to implement it :

1/ Each component owns a mutual exclusion object that is locked when slots are called and released when slots have been executed. The problem is then that every component is monitored even if it is not necessary. Therefore, it results in supplementary costs in terms of computation time and memory consumption;

2/ The developer exactly knows which component is shared across threads and implement mutual exclusions on them. However, a component developer can not plan every situation in which his components are reused;

3/ The third solution is to rely on a smart heuristic in order to establish at runtime which components may be subject to multiple thread access.

Since we do not want to rewrite completely the code of components, we will introduce a new component (called *ConcurrencyManager*) that embeds the heuristic and that will, at runtime: explore the application description structure, establish the list of components that should be monitored (candidate components), instantiate and initialize one proxy component (acting as a monitor and called *Monitor*) per candidate components, and alter the application structure in order to monitor candidate components.

In order to understand how it works, we will not consider a real AR application but rather a toy case organized around three main components: two components (named *loop1* and *loop2*), each in a separate thread, iterate from 0 to 9, and one component (named *sum*), in a third thread, adds all iteration values received from the two other components. The components are, by default, developed

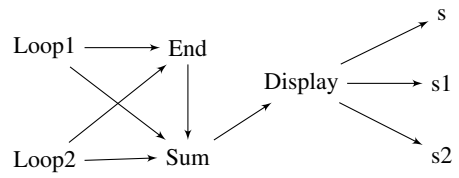
```
For i from 0 to 9
  RandomSleep()
  Emit i
EndFor
Emit "end"
```

(a) *Loop* pseudo-code

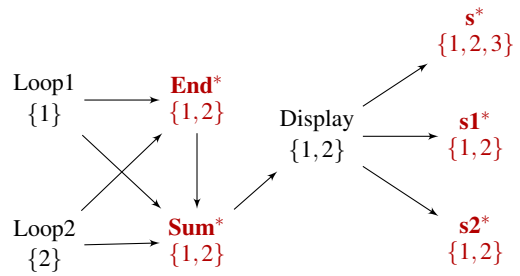
```
tmp <- i
RandomSleep()
sum <- sum + tmp
```

(b) *Sum* pseudo-code

Figure 2: *loop* and *sum* components pseudo-code.



(a) Simplified connection graph



(b) Connection graph with final process annotations. Components with stars are to be mutexed

Figure 3: Simplified connection graph of components in the toy case. Components with stars are to be mutexed.

without any code that synchronize them (Fig. 2 is showing their pseudo-code). Therefore, our *sum* component, which should reach a total of 90 ($2 \times \sum_{i=0}^{9} i$), do not necessarily reach the right total in a multi-threading context (the counting slot is called from two different threads and therefore *sum* is in a race condition situation).

Figure 3(a) represents the simplified connection graph of the toy case where we can find the main components as well as thread controllers (statemachines *s*, *s1* and *s2*) and some logical glue components (*finish* which triggers the end of the application when *loop1* and *loop2* have finished iterating, and *display* which displays the final result). *s*, *s1* and *s2* are controllers that may be accessed by different threads. A first thread triggers *loop1* and a second thread triggers *loop2*. Therefore, components *finish*, *sum*, *s*, *s1* and *s2* should be monitored.

In our heuristic, this set of component is automatically computed by analyzing the connection graph as follows:

1. Create the simplified connection graph ;
2. Initialize the heuristic by tagging each components with the set of process known to be attached to them (for example processes that are invocating components) ;
3. Propagate the sets of process through the connections until the sets are stable. Given a source and a destination component sharing the same edge, the propagation consists in modifying the destination process set by making the union of the source and destination process sets ;
4. Determine which components should be monitored. It is the case if components (nodes of the simplified graph) match one of the following rules.

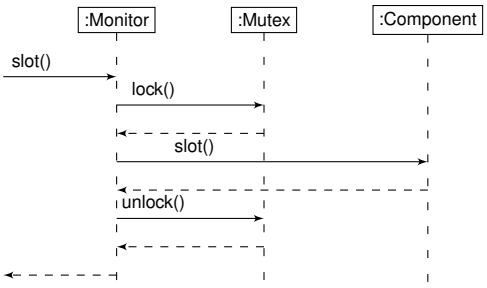


Figure 4: Behaviour of wrapper component implementing the monitor aspect as a UML sequence diagram

- *currentNode* is a controller ;
- $\exists n \in currentNode.parents$ such as $n.processSet \neq currentNode.processSet$.

For each of these components, a *Monitor* will be generated, associated to an object managing a mutual exclusion (*Mutex*). The *ConcurrencyManager* will substitute *Monitor* to actual *Components* and locks and unlocks *Mutex* when a slot is called on the actual component as it can be seen on figure 4.

The *Monitor component* and the *ConcurrencyManager component* has been implemented and the application has been modified to import the *ConcurrencyManager* component (it results in modifying one line in application description). It has been verified that *Monitors* have been put around the right components and that the race condition was suppressed by such modifications. None of the components of the application has specific code for handling mutual exclusion, therefore the *ConcurrencyManager* seems viable in order to manage concurrency on components that were not specifically designed to handle it. However, it will not prevent deadlocks which can be detected by applying formal methods before running the applications.

3.3.2 Networking

Given the fact that ARCS aims at supporting development of state of the art AR applications, offering transparent networking functionality is mandatory. Some previously quoted frameworks use custom protocols while others use generic middleware to manage the application distribution. In particular, CORBA which is used in DWARF and MORGAN. For ARCS, we chose a specific custom architecture for two main reasons. The first one is the preservation of the synchronous signal/slot mechanism and the second one is the ambivalent role of the components that can be, in our case, clients and services at the same time. This makes our architecture easier to setup, to maintain, to scale-up and more flexible since it can also work in a client/server manner.

In addition to these two important constraints, the extension of the architecture was designed keeping in mind a set of qualitative objectives (transparency, context awareness, interoperability, semantic naming, ease of programming...) and quantitative performance goals (required throughput, fault tolerance, latency and other criteria based on AR applications requirements).

The resulting lightweight middleware allows to transparently link remote components. The main idea is that an intermediary component is generated if a component is supposed to have some communication with another component on another machine. These intermediary components, built on the proxy design pattern [7] include, among other data, all the connection data needed in order to communicate (remote host address, port).

In the description of a sheet, if a component A needs to connect to a component B over the network, the component A is in fact connected (in signal/slot meaning) to the intermediary component

on the machine where it is located. A proxy slot is created for each connection of a signal of this component with a slot of a distant component. A proxy signal is also created on the component B side to receive incoming remote signals and spread them locally.

These proxy signals/slots include network deserialization/serialization mechanisms and are instantiated within specific components called *network configurators* (the intermediate components). Each remote component (network component) is considered as a service and is attached to one *network configurator*.

A connection manager (central component) lists all connections between components. It is used to set up the new data flow and activate different connections depending on the active sheet. Since all machines can be clients or servers, we decided to call the machine hosting the connection manager a *Master*. Other terminals are called *Slaves*.

Once all connections have been set up, components on slave machines communicate without going through the master as shown (the connections manager is only a repository of existing connections). The components can communicate in both directions and regardless of the location of the distant component since this is handled by the middleware (transparent communication). To have a detailed study of the middleware and different connection types, one may be interested in [22].

To make development of ARCS distributed applications easier, a graphical global distributed architecture viewer has been developed. Also, the architecture has been assessed on the basis of the previously identified qualitative and quantitative criteria and following an SPE (Software Performance Engineering) [16] inspired methodology. Large scale simulations have been made using OMNet++ network simulator.

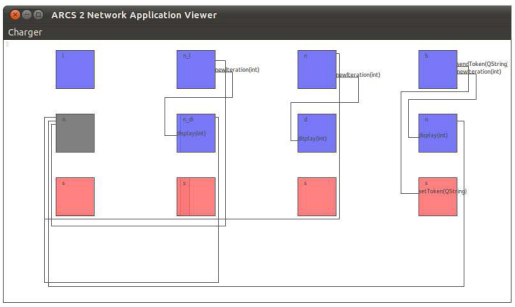


Figure 5: ARCS Network application viewer

3.4 On the cost of extensibility

Since the ARCS framework allows to integrate exogenous component systems through wrappers, a performance issue arises because of the well-known trade-off between performances and genericity.

We carefully implemented some caching mechanism in order to speed the access to components and invocation data. In order to determine if the whole mechanism is costly in terms of computational time, we put aside the fact the engine worked on real applications and ran some tests to examine the engine behavior under heavy load.

The hardware platform test was a Pentium IV at 3.2GHz with 1GB of memory. Linux was the operating system that ran the engine application. One of the objective was to estimate the time needed by a sheet activation. Therefore, we needed to estimate the cost (in terms of computational time) of invocations and establishing communications channels. We were also interested in estimating the additional costs of communicating data through the signal/slot mechanism.

We ran several series of tests. They mainly consisted in building sheets with a large amount of invocations and connections (up to 10 000) and measure the amount of time needed to perform them. About 14 microseconds are needed to perform an invocation, about 12 microseconds are needed to establish a component connection and less than 1 microsecond is needed to communicate data through a signal/slot connection.

```
<component id="cube" type="Cube"/>
<component id="mat"  type="Material">Material {
    diffuseColor 1.0 0.0 0.0}</component>
<component id="scene" type="Separator">
    Separator {
        RotationXYZ { axis X  angle 0.707 }
        RotationXYZ {
            axis Y
            angle 0 = ElapsedTime { speed 0.6 } .
                timeOut
        }
        Material { diffuseColor 1.0 0.025 0 }
        Cube { width 2 height 1 }
    }
</component>
```

Listing 1: Component declaration of inventor nodes

Related to an actual application like RAXENV (see section 4.3) the estimated average cost to switch from one sheet to another is therefore around 650 microseconds, which is negligible compared to the time needed to process images for example.

4 ARCS INTO USE

4.1 Integrating Inventor as a component system

In order to assess our design, we built a wrapper library to access to Inventor [20] (which is a scene graph library) nodes as if they were ARCS components. If we look at the current Coin3D [17] implementation of Inventor, it is composed of 350 000 lines of code and gives access to about 300 different kind of nodes and engines.

Nodes and engines objects in Inventor can be seen as components: they derive from a same class named *SoFieldContainer* that contain fields (which can be seen as properties) that could be initialized in order to modify their behavior. Moreover, the Inventor API possesses some type introspection capabilities concerning field containers: the actual type of a field container can be retrieved as well as the types and names of fields. Inventor is a scene graph library which means that field containers are organized hierarchically in order to describe a 3D scene according to parent–child relationships. At the same time, a field from one field container can be connected to a field from another field container in order to propagate data and information through the scene graph. Fields can be categorized into three types according to their capabilities: a *Normal field* can update its value from another field value and send updates to other fields, an *Event In* can only be updated and an *Event Out* can only update other fields.

The adaptation of an Inventor field container to an ARCS component is then straightforward: field containers become components and fields become signals and/or slots according to their update capabilities. Field to field connections will be mapped to signal/slot connections and are therefore subject to connection composition. Some specific slots are added to grouping Inventor nodes (nodes that can manage children) in order to compose scene graphs through the invocation composition. Inventor also has some parsing capabilities and can describe field containers using a textual representation (the Inventor file format is the ancestor of the VRML format). It allows us to quickly implement serialization/deserialization capabilities concerning Inventor field containers.

As a result and as we can see in listing 1, the engine can handle an inventor node encountered in an application description. This

description can be without any specification as the first component of the listing, or contain field values (second component), or even a complete subgraph (third component).

Figure 7: Screenshot of the marker-based AR example

```
<component id="camera" type="CameraOpenCV"/>
<component id="viewer" type="\arcsViewer"/>
<component id="converter" type="ImageCv2So"/>
<component id="monitor" type="\arcsMonitor"/>
<component id="al" type="Obj2So"/>
<component id="scene" type="Separator">
  Separator {
    Translation { translation 0 0 0.03}
    TrackballManip {}
  }
</component>
<component id="inverter" type="PoseInverter"/>
<component id="et" type="EdgeTracker"/>
<component id="ts" type="ThresholdSampler"/>
<component id="ppe" type="PatternPoseEstimator"/>
<component id="pattern" type="Composite"
    file="../legacy/blocks/patternblock.arcs2.xml"
        />
<component id="statemachine" type="StateMachine">
  <statemachine>
    <first name="a"/><last name="end"/>
    <transitions>
      <transition source="a" token="end"
          destination="end"/>
    </transitions>
  </statemachine>
</component>
```

Listing 2: Extract from component pool of the marker based AR application

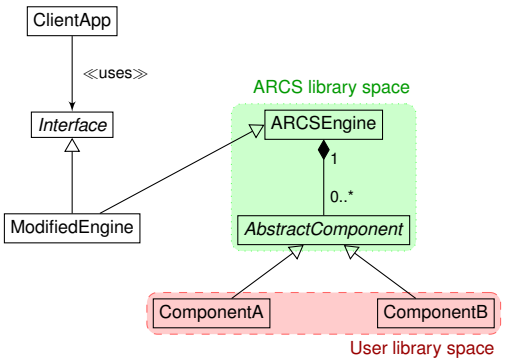
The integration Inventor is made in only 2000 lines of code and fulfill its objectives. That is to say, the ARCS engine gains full access to all 300 field containers and can handle the different field types. The main work was to design and implement two main classes: *InventorFamily* and *InventorComponent* and some satellite classes that should act as proxies between native components and inventor components.

4.2 A marker-based AR example

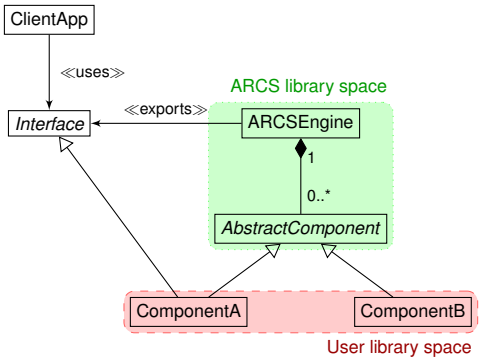
In order to show a set of inventor and native components integrated together, we will present a marker-based AR application implemented using our framework. The listing 2 is an extract of the component pool of the application showing 12 components. Amongst them, the *scene* component is an Inventor component, *pattern* is a composite component (a component obtained by aggregating other components as well as composite components), *statemachine* is the application controller which is also described as a component and the 9 other components are of the native component type.

Instead of showing XML code of connections between components, we chose to represent them graphically in figure 6. For readability purposes, name of signals and slots have not been reported, whereas the type of the conveyed data are indicated. As we can see, the *camera* component grabs frames from a webcam and dispatches the images to several components. Some of them perform image processing in order to detect coded markers and compute the position and orientation of the camera viewpoint (components *ts*, *pattern*, *ppe* and *inverter*). Other components are dedicated to scene graph visualization (*viewer*) and management: *al* imports wavefront models; *converter* transform the image taken from the camera into a texture that is exported by *viewer*; *scene* completes the scene graph for the *viewer*. As one can notice, *scene* is here composed differently from other components since it is composed by invocation composition in order to make it viewable. A screenshot of the resulting application (Figure 7) has been made in order to display the resulting scene graph registered over images taken from the camera. The obtained result illustrates as a proof of concept that it is possible to integrate exogenous component systems inside the ARCS framework with limited development effort.

4.3 Integrating the ARCS engine as a third party



(a) Subclassing the engine



(b) Exporting a reference to a component implementing a specific interface

Figure 8: Integrating ARCS as a third party

Since the ARCS engine is designed to be lightweight, it can also be integrated into other software as a third party. This second type of software integration problem can be stated as follows: How can

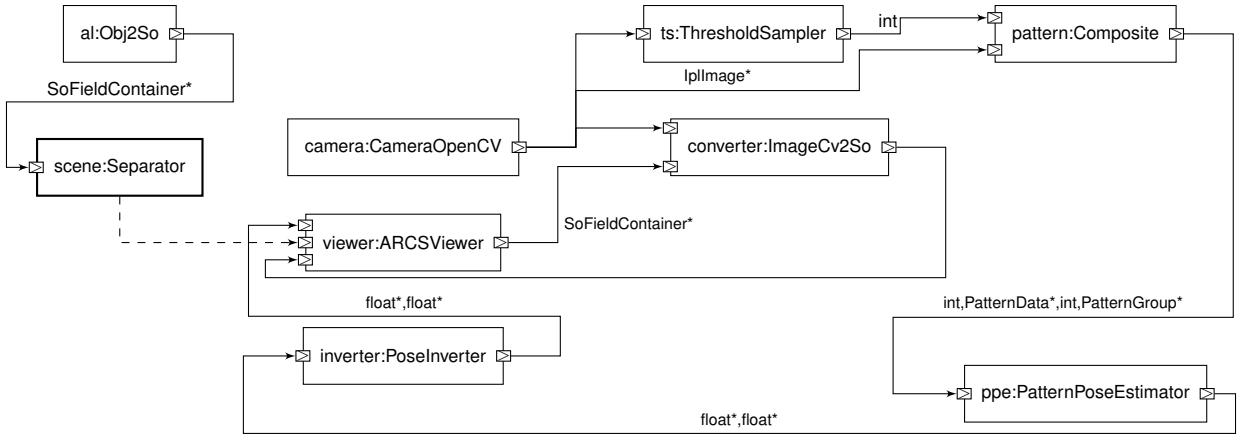


Figure 6: Graphical view of components composition of the marker-based AR application

a software integrating the ARCS engine access to data and computations performed by the components it stores ?

As we can see in figure 8, two solutions are proposed to solve this problem. The first one (figure 8(a)) consists in subclassing both the ARCS engine and a common *Interface* between the engine and a client application (*ClientApp*) and produce a *ModifiedEngine* that allows the *ClientApp* to access to components through a specific *Interface*. The problem in this solution is that the integrator should have a complete knowledge of the engine he is extending. Moreover, the modification of the engine should be made for each application that needs to integrate the ARCS engine.

The second solution to this integration problem relies more on the development of a specific component (in figure 8(b), it is *ComponentA*). The ARCS engine then exports a reference to the component which is implementing the required *Interface* and is then usable by the *ClientApp*. Therefore the implementation of the common interface is shifted from the ARCS library space to the user library space, which is the space of library components dynamically loaded by the engine. This solution does not require a full knowledge of ARCS engine and thus simplifies the integration problem because the engine remains untouched.

ARCS has been integrated in the RAXENV[2] project which aims at demonstrating the practical use of an outdoor AR system for environmental sciences and techniques, both in terms of technology development as well as end-user adoption.

RAXENV targets two different applications: the management of a complex geotechnical site and the diffusion of geological information for education and tourism. One task of the project was to develop and finalize a tracking and registration module for the application. This module also had to acquire data from a hybrid sensor composed of a GPS, an inertial sensor and a camera. As a result, the ARCS engine was integrated as a module in the final application and was able to communicate with other parts of the system (the resulting architecture is represented in figure 9). The performances of the localization part of RAXENV was then presented and evaluated in [23]. The ARCS application description embedded inside RAXENV, counts 6 sheets, a total of 105 component connections and 71 component invocations in order to perform its task.

4.4 Other applications

Beside the RAXENV project, the Open Inventor integration example and previously cited applications, there are numerous other applications that we can't describe all in detail in this paper. Among most noticeable ones, the integration of OGRE (Object-oriented Graphics Rendering Engine) as a rendering engine to ARCS applications. This rendering engine has been used in a distributed

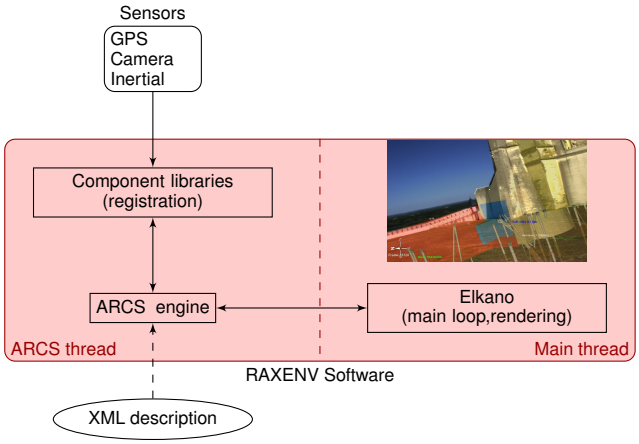


Figure 9: Global interaction between ARCS and RAXENV software modules

underwater Augmented Reality application following different scenarios (contextual augmentation, marker-based tracking and natural features tracking). We can also quote another low latency application that consists of a collaborative SLAM (Simultaneous Localization And Mapping) Structure From Motion application allowing to generate 3D models from video sources in real time.

All these applications provided very significant feedback that have led not only to identification of best practises when developing ARCS application but also to improve the engine itself.

5 Conclusion

We provided some hints on how to design a component based framework in order to ease the integration of other component systems and ease its integration by third party software. The proposed solutions have been implemented inside our own component-based framework ARCS and we provided examples of use as well as a study of the additional computational time needed by such a framework.The first results seem promising.

Future works on the framework will be to investigate deeper the multi-threading modeling of an application and the programming issues that such an environment raises. Another point we would also like to cover is the aspect of AR system concerning real-time constraints that could arise. In one of our previous work, we considered some real-time issues and the use of formal methods in order

Figure 10: Screenshot from an ARCS-based underwater application using OGRE as a rendering engine

to check timing constraints in Mixed Reality applications with the MIRELA framework [21]. One of our objectives is to unify these frameworks.

Finally, some AR applications must cope with several challenges that are common with the ones that were at the origin of the emerging of the system engineering discipline. Indeed, AR systems tend to grow in complexity and are integrating more and more heterogeneous parts. Furthermore, designing an AR system is not only covering software but also hardware. Therefore, designing AR framework is not only a software engineering issue but must also cover system engineering aspects.

ACKNOWLEDGEMENT

This work has been partly funded by the ANR (French National Research Agency) during the RAXENV Project.

REFERENCES

[1] M. Bauer, B. Bruegge, G. Klinker, A. MacWilliams, T. Reicher, S. Riss, C. Sandor, and M. Wagner. Design of a component-based augmented reality framework. In *Proceedings of the International Symposium on Augmented Reality (ISAR)*, pages 45–54, octobre 2001.

[2] BRGM. The raxenv project. http://raxenv.brgm.fr/?lang=en.

[3] J.-Y. Didier, S. Otmane, and M. Mallem. A component model for augmented/mixed reality applications with reconfigurable data-flow. In *8th International Conference on Virtual Reality (VRIC 2006)*, pages 243–252, Laval (France), 26-28 avril 2006.

[4] R. Dörner, C. Geiger, M. Haller, and V. Paelke. Authoring mixed reality. a component and framework-based approach. In *First International Workshop on Entertainment Computing (IWEC 2002)*, pages 405–413, Makuhari, Chiba, Japon, 14-17 mai 2002.

[5] C. Endres, A. Butz, and A. MacWilliams. A survey of software infrastructures and frameworks for ubiquitous computing. *Mobile Information Systems Journal*, 1(1):41–80, janvier–mars 2005.

[6] A. Fuhrmann, D. Schmalstieg, and W. Purgathofer. Fast calibration for augmented reality. In *Proceeding of ACM VRST'99*, pages 166–167, Londres, December 1999. ACM.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns. Elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[8] C. E. Hughes, C. B. Stapleton, D. E. Hughes, and E. M. Smith. Mixed reality in education, entertainment, and training. *IEEE Comput. Graph. Appl.*, 25:24–30, November 2005.

[9] R. Kuck, J. Wind, K. Riege, and M. Bogen. Improving the avango vr/ar framework - lessons learned. In M. Schumann, editor, *Virtuelle und Erweiterte Realitat : 5. Workshop der GI-Fachgruppe VR/AR*, pages 209–220, Aachen, 2008.

[10] ObjectManagementGroup. Omg's corba website. http://www.omg.org/corba/.

[11] J. Ohlenburg, I. Herbst, I. Lindt, T. Frhlich, and W. Broll. The morgan framework: enabling dynamic multi-user ar and vr projects. In *Proceedings of the ACM Symposium on Virtual Reality Software and Technology, VRST 2004*, pages 166–169, Honk Kong, China, 10-12 novembre 2004.

[12] C. Owen, A. Tang, and F. Xiao. Imagetclar: A blended script and compiled code development system for augmented reality. In *Proceedings of the International Workshop on Software Technology for Augmented Reality Systems*, pages 23–28, Tokyo, Japon, 7 octobre 2003.

[13] W. Piekarski and B. Thomas. An object-oriented software architecture for 3d mixed reality applications. In *The Second IEEE and ACM International Symposium on Mixed and Augmented Reality (ISMAR'03)*, pages 247–256, Tokyo, Japan, octobre 2003.

[14] M. Ponder, G. Papagiannakis, T. Molet, N. Magnenat-Thalmann, and D. Thalmann. Vhd++ development framework: Towards extendible, component based vr/ar simulation engine featuring advanced virtual character technologies. *Computer Graphics International Conference*, 0:96–104, 2003.

[15] G. Reitmayr and D. Schmalstieg. An open software architecture for virtual reality interaction. In *Proceedings of the ACM symposium on Virtual reality software and technology*, pages 47–54. ACM Press, 15-17 novembre 2001.

[16] C. U. Smith, L. G. Williams, Lloyd, and G. Williams. Performance and scalability of distributed software architectures: An spe approach. *An SPE Approach, Parallel and Distributed Systems*, 13, 2002.

[17] SystemsInMotion. Sim – coin3d 3d graphics development kit. http://www.coin3d.org.

[18] C. Szyperski. *Component Software - Beyond Object-Oriented Programming (Second edition)*. Addison-Wesley, Harlow, England, 2002.

[19] TotalImmersion. D'fusion studio. http://www.t-immersion.com/en,on-stage-presentation,33.html, 2008.

[20] J. Wernecke. *The Inventor Mentor: Programming Object-Oriented 3d Graphics with Open Inventor, Release 2*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1993.

[21] xxx. Blank to preserve anonymity. *Journal of Virtual Reality and Broadcasting*, 2009.

[22] xxx. Blank to preserve anonymity. In *Proceedings of the 5th International Conference on COMmunication System softWAre and MiddlewaRE (COMSWARE 2011)*, July 1-3 2011.

[23] I. Zendjebil, F. Ababsa, J.-Y. Didier, and M. Mallem. Large scale localization for mobile outdoor augmented reality applications. In Springer, editor, *International Conference On Computer Vision Theory and Applications*, 2011.