A Component Model for Augmented/Mixed Reality Applications with Reconfigurable Data-flow

Jean-Yves Didier Samir Otmane Malik Mallem
Laboratoire IBISC - Université d'Evry
40, rue du Pelvoux - 91000 Evry - FRANCE
{didier, otmane, mallem}@iup.univ-evry.fr

Abstract

In this paper, we will introduce a new component-based framework for mixed and augmented reality applications. We will see the developed framework intends to meet several requirements such as portability, variable component granularity, scalability, and a high level of abstraction for end-users.
The provided framework, at runtime, is composed of a set of several libraries where components are stored, an XML file in which configuration and communications between components are described and a runtime program that deals with the two previous parts of the system. Our component model can also deal with reconfigurable data-flows by the use of a finite state-machine, using several component configurations within an application.

Keywords: system architecture, components, mixed/augmented reality

1 Introduction

A component-based software architecture is a challenging area to develop frameworks for MR (Mixed Reality) applications. During the last years, MR community has proven the need of an infrastructure. Almost 30 different projects of framework have been developed for AR (Augmented Reality) [5].

Why does such a field of research tends to use Component-Based Software Engineering (CBSE) more and more? Augmented reality (AR) or Mixed Reality (MR) is the way to mix virtual entities with real world, both being semantically linked. An AR system can reach this goal by knowing the real world. This is achieved using several classes of sensorsfor example to locate the MR system in the real world. Indeed, AR is depending a lot on sensor technology and, since it is a quite new field of research, and is relying on data-flow processing algorithms that are evolving quite fast.

This is where CBSE is useful: it can provide components for sensors data acquisition, data processing and rendering of virtual entities. If components have compatible inputs/outputs, they would be easily replaced when the embedded algorithm is deprecated or when the sensor's type has changed. This makes CBSE very attractive for AR.

In this paper, we will expose a new component based framework, issued from our need in AR, An application build using this framework will be composed of three different parts:

- Dynamic libraries where custom components are stored,
- A file describing how components are connected to each other,
- A lightweight runtime program that reads the previous configuration file and manage components.

As we will see, we added to our system a way to reconfigure dynamically connections between components by adding a finite state machine. The aim is to produce a component architecture meeting the requirements of portability (i.e. running on several operating systems), variable component granularity (i.e. components can either be services or be reduced to specific algorithms) and high levels of abstraction for end-users.

First, we will have a look on existing component-based frameworks for augmented reality.

2 Customized component-based frameworks for Mixed reality

Among the numerous projects in this research field working specifically on modular architectures, we selected some of the more representative projects using CBSE.

2.1 StudierStube

Studierstube is a project from the Technical Universities of Vienna and Graz (Austria) designed in 1997. This framework aims to explore new 3D interaction media and to find the representation of the desktop 2D paradigm in 3D. It is based on the OpenInventor API and is using a distributed scene graph. This project is embedded two tools based on XML solutions: Opentracker [11] for sensors configuration and APRIL

[7]for scenarios prototyping. For our needs, this architecture is relying too much on scene-graphs structure. At the same time, this framework needs some programmer skills to design an application.

2.2 Tinmith

Tinmith is an AR architecture for outdoor experience. It provides a serialized data-flow going from the trackers input to the rendering part (see figure 1). It introduces a Hardware Abstraction Layer for the acquiring data [9]. The sensors data are indeed converted into C++ objects. Tinmith is based on the implementation, it is in fact a set of low level primitives enabling the programmer to quickly access trackers data and displaying them into a 3D rendering display. Designing an application with this framework will require some programming skills hence furnishing no abstraction level at all.

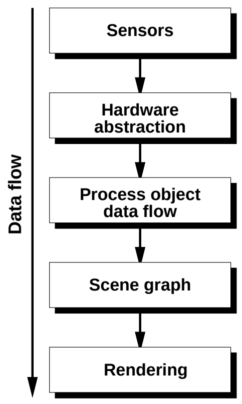


Figure 1. Tinmith data-flow.

2.3 DWARF

Started in year 2000, the DWARF (Distributed Wearable Augmented Reality Framework) project is based on the concept of collaborating distributed services [2]designed for a generic use in Augmented Reality (AR) applications [8][10]. Service are developed on an extension of a CORBA implementation.

They are described using XML (eXtensible Markup Language) files and are independent from each other. They are activated dynamically during application execution, forming a distributed data-flow graph. Each application will use a service manager that will automatically discover where are located over the network the needed services.

However, this system presents several drawbacks:

- meta-data concerning each service are broadcasted over the network,
- the granularity of the system is quite large (components are already independent systems and not, for example, specific algorithms),
- this system is dependent of a distributed environment and might not indicated for the use of small augmented reality applications.

2.4 AMIRE

AMIRE [1][4] (for Authoring MIxed REality) is a European joint project between several partners including SIEMENS and Fraunhofer AGC. It took place during 27 months, from 2002 to 2004. The aim was to develop a rapid prototyping framework for end users who do not necessarily know how the underlying AR technology works.

The partners of the project decided to develop from scratch their own component system to meet some specific requirements of stand-alone augmented reality applications. Components are of two different types: gems and components. Gems are the real components according to standard software component definition, whereas components (in this project) are actually an aggregation of components. In this system, each component (gem or component) has a configuration interface, input slots and output slots.

This framework is also including a graphical editor. It allows a user to connect components to the others through the interface according to a recorded design pattern that acts as a guide for people not familiar with augmented reality applications. The connections and configurations are recorded in a XML file. This one is parsed and interpreted by a runtime that will launch the application. This last system is the one that is the more similar, in many points, to the system we will present in this paper but it doesn't manage the life cycle of a component, that is to say, it's life cycle within an application will be the same as the one of the application.

In AR systems, some operations must be performed on real-time. We then wanted something where we could trigger the data processing when the data are acquired, and not with a delay we cannot determine without a complex algorithm, as it is the case with communications relying on events or network. In the same way as AMIRE, we relied on something derived from direct procedure calls according to Cox and Song taxonomy [3] : the signal/slot mechanism. The choice was then to develop or rely on a component system that meets several requirements such as portability, a variable granularity, some scalability, and a high level of abstraction for end-users. As we will see, we will introduce component life-cycle management mechanisms in ours application developed with our framework. We will also combined it with with a system allowing on-line reconfiguration of data-flow.

3 The framework core

Since we're proposing a component based architecture, we will define what is a component in our system and how it can interact with other entities to build an application.

3.1 The component

According to szyperski's definition [12], "*A component is 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*".

It means that, for our component architecture, we should describe the way components will communicate with each other, how they're implemented, how they're initialized and how we can use them to build applications. The main communication mechanism in our components is based the signal/slot paradigm we will introduce now.

3.1.1 Signal/Slot paradigm

Our solution is based on the signal/slot communication model between components. This paradigm, mainly used in user interfaces API, tends to reach other scopes of specific programming. Some libraries are implementing it, for example: QT , libsigc++ derived from GTK+, sigslot and boost. Signals represent callbacks with multiple targets. Signals are connected to some set of slots, which are callback receivers. They are triggered when the signal is "emitted". Signals and slots can be connected and disconnected dynamically at runtime and are managed. That is to say, connections are tracked by objects owning signals and slots. They're able to automatically disconnect signal/slot connections when one of the object involved in communication is destroyed. This allows the user to make signal/slot connections without expending a great effort to manage the lifetimes of those connections with regard to the lifetimes of all objects involved.

Once the communication mode is chosen for our components, we can easily graphically represent them.

3.1.2 Representation of a component

Basically, a component in our system will be an entity or object able to export by itself its interface which is made of signals and slots. As we will see a little bit later, we will often use both the terms of *components* and *objects* to refer to the same thing, since the components we will develop are objects. Such component could be seen as in figure 2.

Since a component is also a piece of compiled code, we should enter into the "box" and discuss about the choices of implementation.

3.1.3 Choices of implementation

We chose C++ as a programming language for our components. The signal/slot mechanism being already implemented into several libraries, we chose one of these libraries that were fitting our needs. We took Qt as a library implementing signal/slots for several reasons:

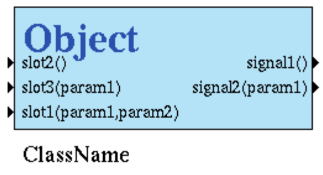


Figure 2. Component view with its signals and slots

- Qt is a cross-platform library. It gives an API (Application Programming Interface) which covers a lot of low level functionalities from operating system.
- The signal/slot mechanism developed allows connections/disconnections on demand.
- The meta-object model implemented in some Qt classes allows developing objects with self-introspection capabilities, especially signals and slots.

Our components will be objects deriving from classes inheriting from `QObject`, which is the class managing meta-informations about objects.

3.1.4 Dynamic libraries

Components are compiled pieces of code stored in dynamic libraries. For our libraries we will need only three specific entry points:

- one for describing objects stored in the library,
- one for instantiating an object contained in the library,
- one for destroying an object instantiated from this library.

Once components are stored in dynamic libraries, we can load and use them on demand. Therefore, a component may require some configuration so we need a system to set component properties.

3.2 Components initialization

Components initializations are performed through slots. However, these slots must meet a few requirements. They must have only one parameter and this parameter belongs to a list of simple types such as strings, numerical values and other components.

Further, we will establish a distinction between two kinds of initializations depending on when they're triggered. Indeed, a slot call can also trigger a signal that belongs to the same component; this last one may call another slot if a signal/slot connection is established between them. So, we have *Pre-connection initializations* that are performed before the component is connected with others and we have a second category we call *post-connection initialization* that are triggered after the component is connected to other ones. This second one allows propagating initializations through connections.

Since a component can be connected to other components, we will see the different solutions we have to connect components.

3.3 Laws of composition

In our system, there are two ways to make components communicate with each other. One is what we call an explicit composition and the other is an implicit composition.

3.3.1 Explicit composition

An explicit composition is the easier way to make components communicate to each other. It is explicit because we know exactly how components are connected. An explicit composition is performed by connecting signals to slots.

Using Qt signal/slot paradigm, connections are allowed as soon as signals an slots share the same signature which is composed of the list of parameters types.

The mechanism described for instance is explaining how to create logical glue between components. We will further see how to switch to a higher level of abstraction that doesn't require programming skills.

3.3.2 Implicit composition

There is another kind of composition between components. This kind is called implicit composition because without component documentation it would be difficult to exactly know how components are interacting with each other. This composition is using slots with only one parameter which is of type `QObject*`. Qt can track down the inheritance tree from class `QObject`.

It means we can initialize some components using others. It allows them to have specific ways to communicate without using the signal/slot mechanism. It's then a kind of obfuscated communication : an implicit composition.

3.3.3 The sheet concept

To sum up all we know about our components and their composition in our system, we introduce the concept of sheet. In a component based framework, an application could be seen as a set of components working (or more properly communicating) together. We choose to name this set a "sheet". Its graphical representation may look like figure 3. A sheet is storing how components are initialized and how components are communicating to each other.

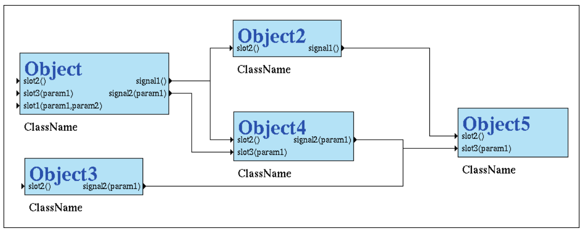


Figure 3. A Sheet is a set of objects communicating together

3.3.4 Macro-blocks

A sheet can also be used to represent a subset of components working together. They will be called macro-blocks. A macro-block is defined by global slots and signals to which classical components of our system can connect and interact. These global inputs/outputs are mapped to internal components of the macro-block. It allows to make the macro-block like a black-box, i.e., we can only give its inputs/outputs without telling exactly what's inside it. Another strong point of macro-blocks is that we can limit the post-initialization propagation to the components within the macro-block and then, not affect the other components of the application.

The recursivity issue: In our component model, macro-blocks can contain macro-blocks. This lead to the recursivity issue. Since macro-blocks can be designed independently from applications, it means component names inside a macro-block can possess the same name as an object from the main application. To solve this problem, a namespace is added to components registered as parts of a macro-block. This namespace is formed of the names of the macro-blocks containing the component. For example, if a component '`A`' is nested in a macro-block '`B`', its name will be, when loaded by the system, '`B::A`'.

However, sheets and macro-blocks are looking like a quite static sight of an application. Within an application, a component can have its own life cycle. This is not something only a sheet can reflect. Indeed, an application may have several behaviors or several states. For example, an application may have a configuration state different from its normal state or a calibration step that is needed before reaching the configuration state. We will then discuss this matter.

3.4 Extensions of composition mechanism

We will introduce two more mechanisms, the first is consisting in using a state machine and the second is dedicated to components life-cycle management.

3.4.1 Adding a state machine

Indeed, a sheet reflects a state of an application. An application could be seen as a stack of sheets. We then need a mechanism to switch between sheets. We achieve this by adding a state machine. An application could then be represented as in figure 4.

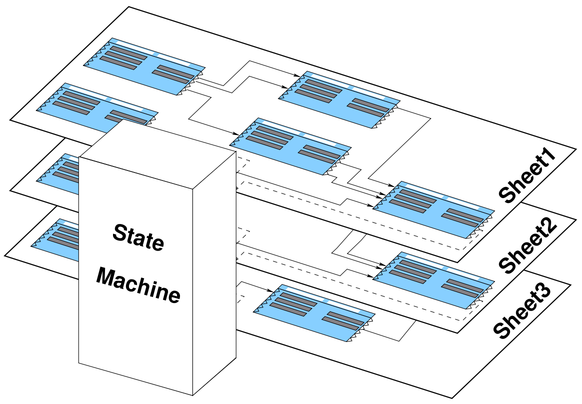


Figure 4. Sheets can be activated one by one by a state machine

Apart from strong automaton formalism, a state machine is, in our case, composed of states and transitions. The transitions are described by an initial state, a token and a final state. When the state machine receives a token, it may trigger a state change. The new state can be a terminal state . If it is the case, the runtime is forced to shutdown properly.

The sheets must be slightly modified to communicate with the state-machine. It is achieved by using a single object per sheet that is allowed to send a token to the state machine. Each token sent to the state machine can make it switch to another application state that will be represented by another sheet.

If we suppose a sheet (current sheet) is active and that the state machine has received an identified token triggering a transition, the sheet switch mechanism will follow these steps:

1. *Disconnections*: each signal/slot connection described in the current sheet is destroyed. All components specifically instantiated for the current sheet are destroyed. At the end of this step, components cannot communicate between each other,
2. *Change of current sheet*: the final state of the triggered transition becomes the current sheet. Each object that needs to be specifically set up for the sheet is instantiated,
3. *Pre-connection initializations*: before connecting signal/slot couples of the current sheet, components can be initialized separately,
4. *Connections*: each signal/slot connection described in the current sheet is activated. At the end of this step, components can communicate between each other according to the new scheme designed in the current sheet,
5. *Post-connection initializations*: reserved to initializations that propagate through different components when they're connected to each other.

This described mechanism allows us to change the data path between components. We then have a reconfigurable data-flow. As we have seen, some part of life-cycle component management are introduced in steps 1 and 2. We will detail it now.

3.4.2 Component life-cycle management

In our case, component life-cycle management means we can instantiate and destroy components on demand. We will adopt two types of strategy:

- There are components whose life-cycle is identical to application life-cycle. Those will be instantiated at the beginning and destroyed at the end of the application.
- There are components whose life span is smaller than the application life span. They will be managed through special initialization reserved to sheets that allow to create or destroy components.

A flag will be added to objects to know if their life span is the same of the application life span or not.

Once we described the basis of our system basis, we can now build over it a few layers of abstraction so that an end-user without any programming skills can build applications using components.

4 Layers of abstraction

To reach our goal, we added two layers of abstraction on top of our architecture. The first layer is a medium abstraction level since it is consisting in writing an XML (Extensible Markup Language) file that will be parsed and interpreted by a runtime managing components. This layer will require some knowledge of the XML syntax and, more specifically, knowledge of the markups set we developed for our framework. This step will be needed to reach a higher level of abstraction we will detail later. The second layer is a graphical builder helping to design applications.

4.1 XML abstraction level

We choose to use XML, the eXtensible Markup Language which is now a widely spread standard for many applications. It helps us to formally describe an application with all the concepts introduced before.

The XML formalization has several advantages when it comes to rapid prototyping, compared to more classical scripting languages:

- a convenient XML editor with a graphical user interface allows the end user to design an application without to master the syntax. Since the XML parser is checking syntax, it's reducing the number of syntax or semantic errors,

- building tools that create code or configuration files from specifications is simplified by the use of XML.

We will describe two sets of XML markups, one is for the applications and the second one is for macro-blocks. After this, we will explain how we will be able to parse and interpret it.

4.1.1 Markups sets for applications description

In our case, the XML document describing an application is composed of five blocks named:

- *defines*: listing pre-defined values for initializations, works like `#define` directives of C preprocessor,
- *libraries*: naming the libraries the runtime will need to load,
- *objects*: grouping objects (or components) instantiated by runtime. Each object is referenced by an *object* markup,
- *sheets*: describing the sheets stack, each sheet being embedded in a *sheet* markup,
- *statemachine*: describing the state machine that is switching sheets when needed.

Each sheet is described by a set of four markups:

- *tokensender*: indicating the component allowed to communicate with the state machine,
- *preconnection*: containing all components initialization before connection,
- *connection*: telling which couple object/signal will communicate with a couple object/slot,
- *postconnection*: containing initializations that could propagate from one component to another.

For the application description, we do not rely on any textual content but we are rather using the content model given by the Document Type Description (DTD). It is describing how the elements within the document are nested within each one. The organization of our markups is summed up by the figure 5. When a markup has specific attributes, those are written in the same box under the name of the markup.

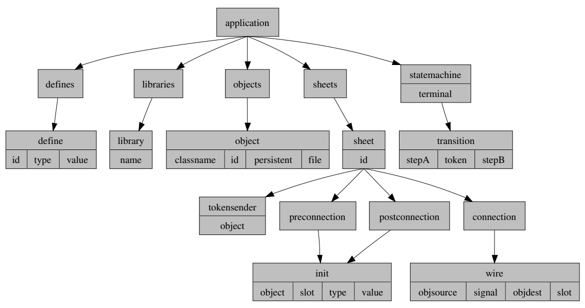


Figure 5. Markup organization of an application description

4.1.2 Markups set for macro-block description

As we already wrote, macro-block internals look like a sheet. On the set of markups used to describe a macro-block, there will be only a few changes:

- the section *sheets* is transformed into one section named *sheet* where pre-connection initializations, connections and post-connections are stored,
- a section *slots* where inputs and their link to internal components are described,
- a section *signals* where outputs and their link to external components are described.

The resulting markup hierarchy for macro-blocks is exposed in figure 6.

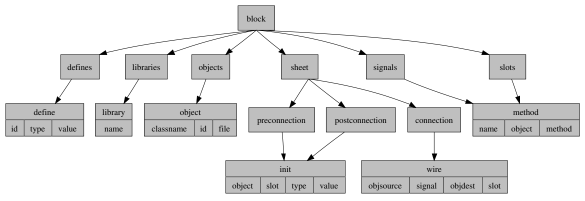


Figure 6. Markup organization of a macro-block description

4.1.3 Inside the application runtime core

Once we have some libraries implementing components and an XML description file of an application, we can write a runtime aimed to specifically load objects from libraries and make them communicate according to the description of the application contained in the configuration file. The architecture of the runtime core is only composed of four elements as seen in figure 7:

- an *XML parser*, to read description of the application,
- a *State Machine manager*: Its role is to switch the sheets when it receives a token or shutdown properly the application if it reaches a final state,

- a *Communication manager*: This one is connecting and disconnecting components according to the current state of the application,
- a *Component manager*: This element is loading components from the libraries and is instantiating them on demand.

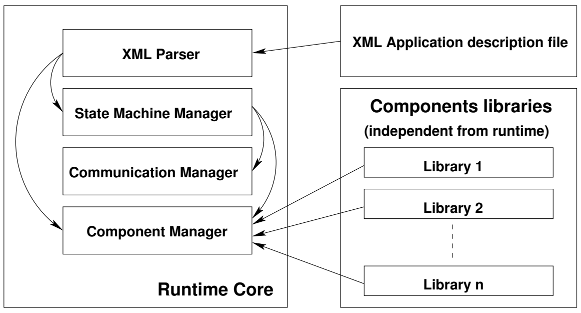


Figure 7. Runtime core with peripheral component libraries and XML description file

At this point, to develop an application, one must only be able to edit an XML description file to initialize and link the objects together. By editing such file, we're actually scripting the behavior of components and, by extension, of our application. However, we can reach one more level of abstraction to allow end-users to build applications using this architecture.

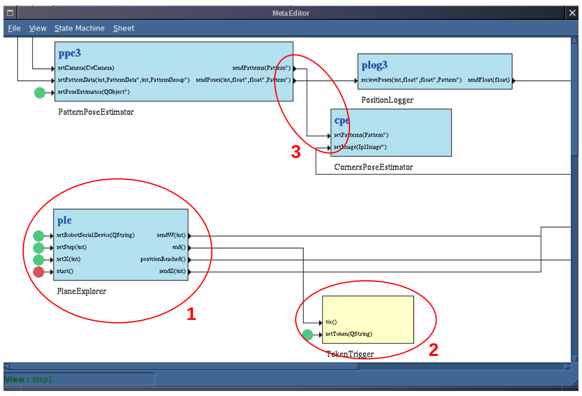
4.2 Graphical interface

We then created a graphical user interface to handle and manipulate components. The only thing required apart the editor are the dynamic libraries containing compiled code. Its allows a user with no programming skills to develop an application by manipulating graphical metaphors instead of scripting the components behavior as we can see on figures 8, showing a sheet being edited, and 9 showing a statemachine being edited.

5 A brief example of application

As a working proof of our framework concepts, we developed a small application using our architecture. This small tool is intended to provide an of the shelf demonstration of a mixed reality application, embedding a calibration camera step and an ARToolkit-like fiducial tracking step. The application is starting with a menu inviting the user to choose if he has to directly start with the MR demonstration or if he should calibrate camera first.

Camera is calibrated using Zhang's method [13] developed in the OpenCVAPI. Fiducial tracking in an ARToolkit-like manner [6]. This application is running on a Toshiba Portege Tablet-PC using a Logitech QuickCam USB camera as we can see on figure 10(a). The whole is orchestrated by a Linux operating system.



1. Standard component (blue), disks are initializations,
2. Component linked with statemachine (yellow),
3. Connection between 2 objects

Figure 8. Graphical user interface in sheet editing mode

5.1 Composition of the application in states term

Our application will be composed of four states which are:

- *Menu*, a state where only the main menu of the application is enabled,
- *Calibration*, the camera calibration step,
- *MRDemo*, the fiducial tracking demo step,
- *End*, the terminal state of the application.

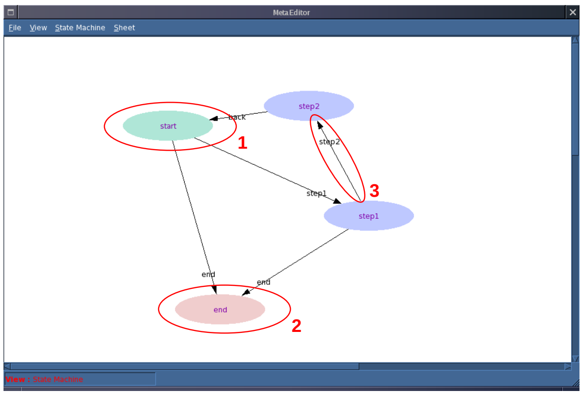
This means we have a state machine like in figure 11, showing the tokens exchanged by sheets and state-machine to switch from one sheet to another.

5.2 Sheets decomposition in components

The application needed various atomic blocks in order to satisfy its purpose: from acquiring images with the webcam, to calibration and visualization tools. Each component organization within sheets can be seen in figure 11. One can notice that some components are used in several sheets, where some only appear within one sheet. This shows the utility of managing component life-cycle.

We can sort all the employed components class names by categories:

- Graphical User Interface: *VLayout, Button, ImagePointSelector, VisualBell, GraphicsMixer,*
- Data acquisition: *CameraPWC,*
- Image processing filters: *RawCVImageConvertor, CheckerDetector, PatternDetector,*



1. Initial state (pale green),
2. Final state (rose),
3. Transition with its token.

Figure 9. Graphical user interface in statemachine editing mode

- Pose parameters estimation: *Checker2DTo3D, ZhangCalibrator, PatternPoseEstimator,*
- Parameters loaders/savers: *CameraLogger, Pattern3DLoader, OBJLoader,*
- Token manipulation for state machine: *TokenConcentrator, TokenTrigger,*
- Various useful components: *CVImageTrigger, TicCounter, GraphicNode.*

Once the components are developed using our framework, we can design the sheets and the statemachine using our graphical interface. This last one will generate the XML file describing the application. Then, using the runtime, we can launch and execute our MR demonstration.

5.3 Results

Figure 10(b) and 10(c) are showing to us the screens configurations for the *Calibration* sheet (where the application is using a classical calibration grid providing camera intrinsic parameters) of our application and the *MRDemo* sheet (based on fiducial - here square patterns - tracking allowing the pose estimation of a virtual character).

The system, using 6 libraries of components, can run to process up to 17 frames per second on a standard Pentium III 1.1GHz using a Linux operating system. As we can see, we have several granularity for components. Some are written with less than 10 lines of code (for example the *GraphicsNode*) whereas some of them are using hundreds of line code to process images like *PatternDetector, PatternPoseEstimator*. It is showing the variability of the component's granularity. Since we're using cross-platform libraries, the portability of our system is ensured. Indeed, we managed to port our runtime on a Microsoft Windows operating system to perform some tests.

6 Conclusion

We exposed a new component based framework. Such component system is relying on signal/slot paradigm for communications between entities. We introduced the sheet and macro-block concept. They are describing a part of an application. To circumvent the static point of view offered by sheets and to manage component life-cycle in an application, we introduced a state machine managing sheets. Once a state is activated, the corresponding sheet is connected. This also allows us to change the configuration of the dataflow within an application.

Then, we introduced two more levels of abstraction to help users with no programming skills to handle application design. One is a medium level relying on several XML markup sets and a runtime parsing and interpreting it. The other one is a graphical builder helping to design the applications. Such editor relies only on the compiled libraries of components.

This project has produced very encouraging results. We are now investigating some more general aspects about the need of specific documentation for this kind of components, the automatic checking of right components life-cycle management and the introduction of design patterns to help the end-user building applications from developed components.

References

[1] D. Abawi, R. Dörner, M. Haller, and J. Zauner. Efficient mixed reality application development. In *1st European Conference on Visual Media Production (CVMP)*, pages 289–294, Londres, 15 Mars 2004. IEEE.

[2] 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, Oct. 2001.

[3] P. Cox and B. Song. A formal model for component-based software. In *HCC '01: Proceedings of the IEEE 2001 Symposia on Human Centric Computing Languages and Environments (HCC'01)*, page 304, Washington, DC, USA, 2001. IEEE Computer Society.

[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)*, Makuhari, Chiba, Japon, 14-17 Mai 2002.

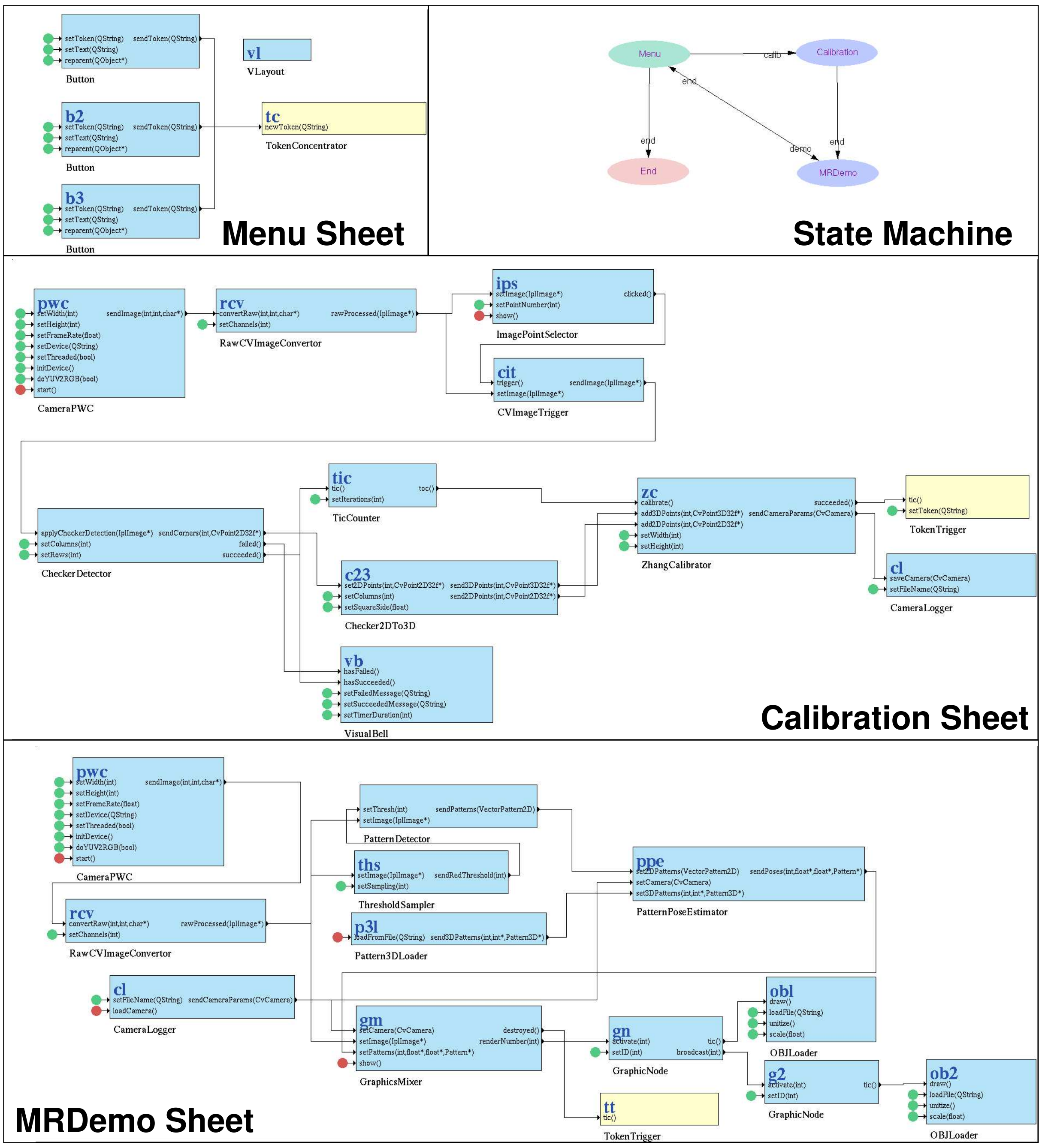
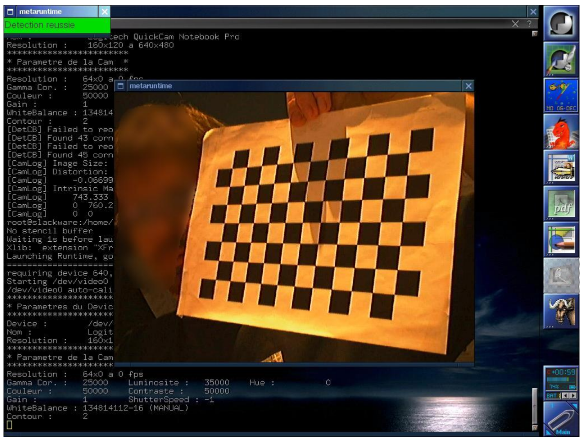


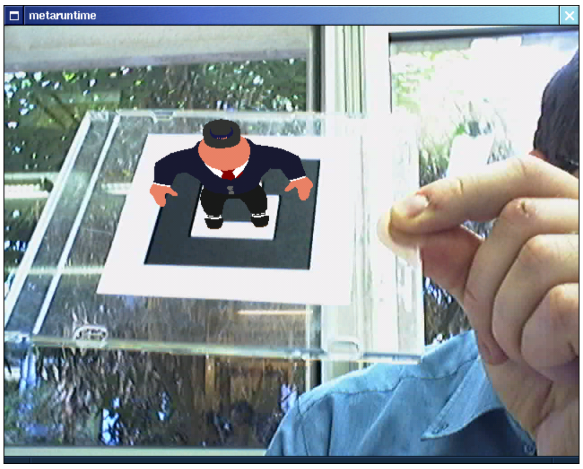
Figure 11. MR Demo application: state machine and sheets (since the *End* sheet is empty, it doesn't appear here). Components in yellow color are allowed to send tokens to statemachine. Initializations are represented by the small discs linked to slots of components.

(a) Tablet-PC with its webcam.



```
        Logitech QuickCam Notebook Pro
Resolution :    160x120 a 640x480
************************
* Parametre de la Cam  *
************************
Resolution :   64x0 a 0 fps
Gamma Cor. :   25000
Couleur :      50000
Gain :         1
WhiteBalance : 134814
Contour :      2
[DetCB] Failed to reo
[DetCB] Found 43 corn
[DetCB] Failed to reo
[DetCB] Found 45 corn
[CamLog] Image Size:
[CamLog] Distortion:
[CamLog]     -0.06699
[CamLog] Intrinsic Ma
[CamLog]     743.333
[CamLog]     0  760.2
[CamLog]     0  0
root@slackware:/home/
No stencil buffer
Waiting 1s before lau
Xlib:  extension "XFr
Launching Runtime, go
=====================
requiring device 640,
Starting /dev/video0
/dev/video0 auto-cali
*********************
* Parametres du Devic
*********************
Device :        /dev/
Nom :           Logit
Resolution :    160x1
*********************
* Parametre de la Cam
*********************
Resolution :   64x0 a 0 fps
Gamma Cor. :   25000    Luminosite :   35000    Hue :         0
Couleur :      50000    Contraste :    50000
Gain :         1        ShutterSpeed : -1
WhiteBalance : 134814112-16 (MANUAL)
Contour :      2
```

(b) A successful calibration step.



(c) MR Demonstration step.

Figure 10. MR application example: hardware and screen-shots.

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

[6] H. Kato, M. Billinghurst, I. Poupyrev, K. Imamoto, and K. Tachibana. Virtual object manipulation on a table-top ar environment. In *Proceedings of the International Symposium on Augmented Reality (ISAR 2000)*, pages 111–119, Munich, Germany, Oct. 2000.

[7] F. Ledermann. An authoring framework for augmented reality presentations. Master's thesis, Vienna University of Technology, 2004.

[8] A. MacWilliams, T. Reicher, G. Klinker, and B. Brüegge. Design patterns for augmented reality systems. In *Proceedings of the International Workshop exploring the Design and Engineering of Mixed Reality Systems (MIXER), Funchal, Madeira, CEUR Workshop Proceedings*, Jan. 2004.

[9] W. Piekarski and B. H. 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)*, Tokyo, Japan, Oct. 2003.

[10] T. Reicher, A. MacWilliams, B. Brügge, and G. Klinker. Results of a study on software architectures for augmented reality systems. In *Proceedings of the International Symposium on Mixed and Augmented Reality (STARS)*, Tokyo, Japan, Oct. 2003.

[11] 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, 2001.

[12] C. Szyperski. *Component Software - Beyond Object-Oriented Programming*. Addison-Wesley, Harlow, England, second edition, 2002.

[13] Z. Zhang. Flexible camera calibration by viewing a plane from unknown orientations. In *International Conference on Computer Vision*, volume 1, page 666, Corfu, Greece, Spetember, 20-25 1999.