A generic framework for game development

Michael Haller
FH Hagenberg (MTD)
AUSTRIA
haller@hagenberg.at

Werner Hartmann
FAW, University of Linz
AUSTRIA
werner.hartmann@faw.uni-linz.ac.at

Jürgen Zauner
FH Hagenberg (MTD)
AUSTRIA
jzauner@hagenberg.at

1 INTRODUCTION

This article describes a generic framework used in the MR based AMIRE (Authoring Mixed Reality) project. The concepts of the presented framework can be used in every 3D based application (e.g. game, VR-/AR- applications etc.) and our approach allows an easy communication between the objects (game objects).

If you are working on several games or VR/AR based applications you will see a common pattern in all these different engines: all the objects have to communicate with each other. Rabin shows in [1] a very interesting AI engine, which in fact is more a communication concept than an AI engine. His approach is interesting for several reasons: Firstly, his system is based on the so called State Machine (FSM) concept. Secondly, his AI engine is implemented in C and not in C++ - an important thing for console based applications. Finally, his approach uses an Event-Driven mechanism – his game objects are not actively watching the world (polling) but sitting back and waiting for news. Consequently it becomes difficult to extend the AI engine (both, the engine itself and the FSM of the game objects).

2 RELATED INTER-OBJECT COMMUNICATION

Inter-object communication is often done by directly calling methods of objects. This is indeed a very fast and easy to understand way of communication between two objects. In the C++ standard there is available only a minimum of meta information and interfaces [2], which does not include a concept like the Java reflection API (application programmer interface) [3]. For the development of authoring frameworks or applications a subset of such a method call abstraction by meta interfaces is mandatory. Due to this lack of such a reflection API in C++ it is unrealistic to develop such authoring frameworks or applications in C++ by using direct method calls for inter-object communication.

2.1 Signals and slots

In the GUI (graphical user interface) library QT Trolltech Inc. uses a design pattern called signals and slots for inter-object communication [4]. This is a concept for connecting objects, which is easy to understand and to use. Trolltech Inc. also presents a generic authoring tool based on the signals and slots. The implementation of this concept is achieved by a MOC (meta object compiler). This MOC provides all the necessary meta information needed by the signals and slots. But the implementation of this concept only allows the authoring tool to create source code for the application. For modifications on the application it is inevitable to recompile the sources.

2.2 Beans and properties

Another concept for authoring tools is the beans concept [5]. It is a software component model for Java, which allows the generic manipulation of such components. This is done by defining a set of properties for each bean. The Java reflection API provides the interfaces to

manipulate these properties and to connect beans by using properties. Sun demonstrates the applicability of the beans concept for authoring tools by a small application called BeanBox [6].

2.3 State oriented listeners

For the communication between objects we also need a mechanism allowing the objects to react on a specific situation like a pressed button or a collision with a chair. Java affords the listener concept for this requirement [7]. This mechanism is a simplified version of the well known MVC (model-view-controller) design pattern [8]. It provides a mechanism to notify views about the change of a specific model. This model change can also be considered as a change of the object state. This interpretation is realized in the state machine of Rabin [1]. Rabin's state machine uses a centralized mechanism routing state change messages between objects. Such a centralized architecture is a restriction for the future design. Moreover, it becomes a bottleneck in the future. With a decentralized architecture the prevention of bottlenecks would be possible. The centralized architecture can be simulated by a central object used for delegation and rooting.

3 COMPONENT ARCHITECTURE

We have decided to develop a new architecture which combines the advantages of the above concepts into a fast, easy to use and generic concept of inter-object communication for C++. First of all we will look at the requirements for a component. Components have to be:

- connectable by a simple and easy to use mechanism
- configurable in a generic way
- state oriented
- instancable by a classifying name
- persistent

3.1 Components

Compared with the design of QT objects which uses signals and slots, components (as seen in figure 1) have two slot types (in- and out-slots). In fact this is a more symmetric naming than signals and slots. In contrast to the QT design, component slots are named dynamically and not statically by the compiler. So it becomes possible to access the in- and out-slots by their names and to show a list of available slots for each component.

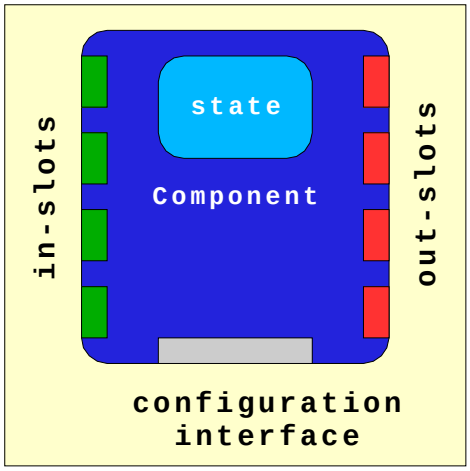

Figure 1: The component interfaces includes the in-slots, the out-slots and the state.

3.2 Connecting components

As already mentioned all slots have a name. So basically a connection between two components consists of the in- and out-slot names and the references of the emitting and receiving component (as seen in figure 2). A polling mechanism will costing unnecessarily performance and implies a polling thread, which will lead to a complex synchronization strategy. Therefore, a connection also needs a trigger state for the emitting component. This implies a well formed state interface, which has to be part of the component interfaces. It is recommended to map the slot name to a slot id. This will increase the performance due to the high costs of comparing strings and the fast comparison of numbers.

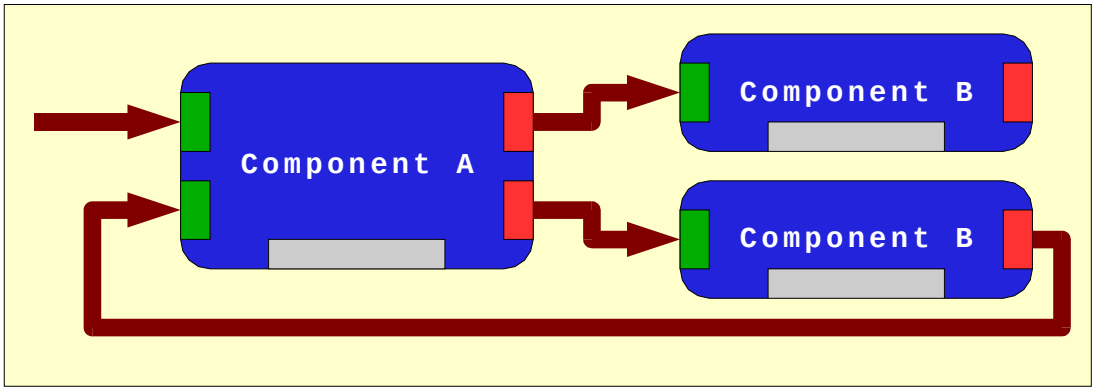

Figure 2: Connection of emitting and receiving component.

3.3 Composed components

A component can be considered as a black box. It has in- and out-slot, a state visible to other components and it has to be configurable. So, when this matches for each component, we can make a network of components and connections. The trigger states will be the states of the components containing the tagged out-slots. These states are visible for other components, which will be connected to the tagged out-slots. To get an abstraction of this network we put it into a composed component and we give these composed components a name.

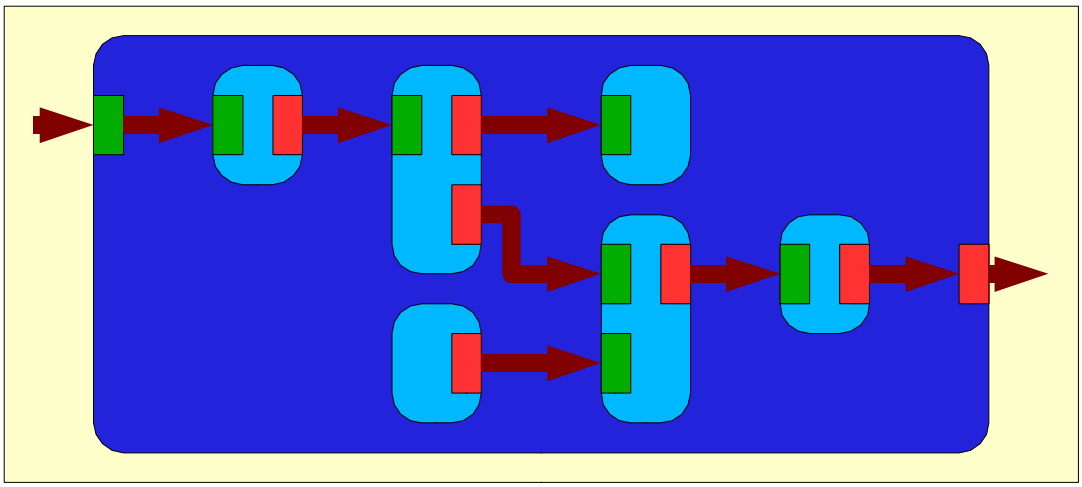

Figure 3: Composed component consisting of a set of components and connections.

3.4 Component management

In Java there is an easy mechanism to create an instance of a specific class. You can lookup a class by its name and call a constructor to create a new instance of this class. In C++ this is not possible. So we need an object, which manages the creation and the lifetime of the components. We call the object a component manager (as seen in figure 4). You can register a component prototype at the component manager by its name. To create a new instance of a component, each component must have an interface to create a clone of itself. The original prototype must not be modified after the registration to ensure the same default behavior for each instance. The component manager is also the central storage of all components used in the application. This will bypass the missing garbage collection in C++. Deleting a component will also delete the connections it is assigned to.

To make fast state comparison possible the component manager is also a storage of all system or application wide states. Each state used in application components must have a name and must be registered at the component manager. The component manager must not be a singleton to ensure that more than one applications can be loaded without conflicts.

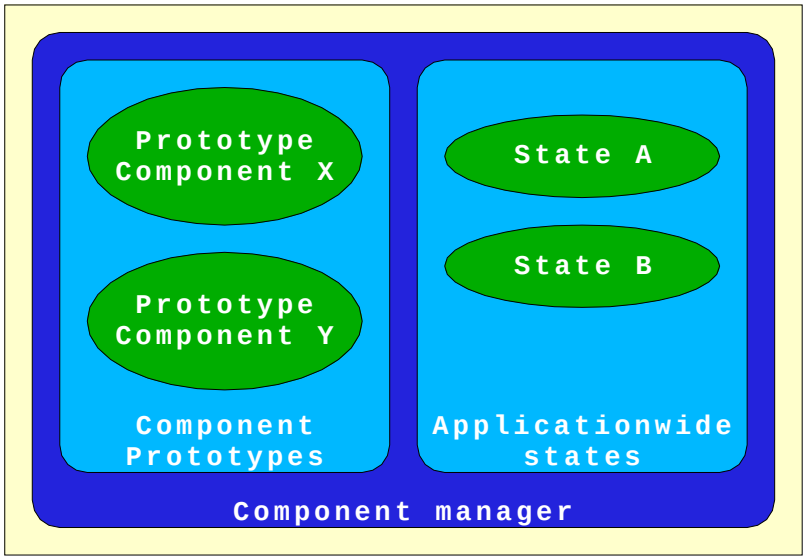

Figure 4: Components and all the states are managed by the component manager.

4 PROPERTIES

A generic communication between components must have a generic and well defined data format. The beans concept uses the so called properties for configuration of object and communication between objects. The properties, presented in the beans concept, have the possibility to use all the meta and reflection interfaces of Java.

Therefore, we have designed a property concept for our framework, which supports the following data type:

- Base types (boolean, integer, float, double, character, string, structures)
- Vectors of base type

A overview of all types is shown in figure 5.

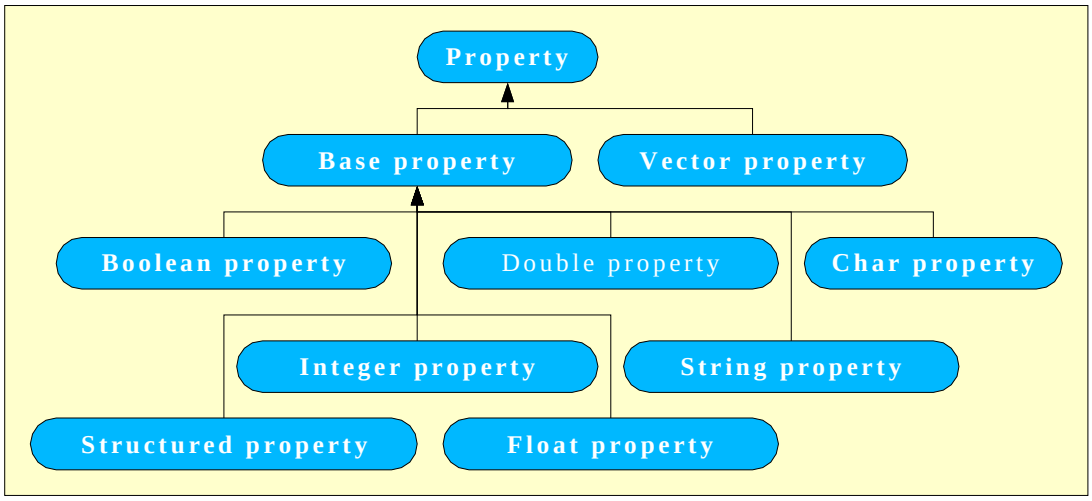

Figure 5: The property design.

There are two important properties to model the communication and configuration parts of components. The first is the vector property, which holds more than one values like the vectors in Java. The second is the structured property which works like an abstract data structure known from languages like C. It maps a name to a property value of a type shown in figure 6. The names and types are described in the structured property type of the structured property. This type is a named list containing all field names of the structured property and the its type.

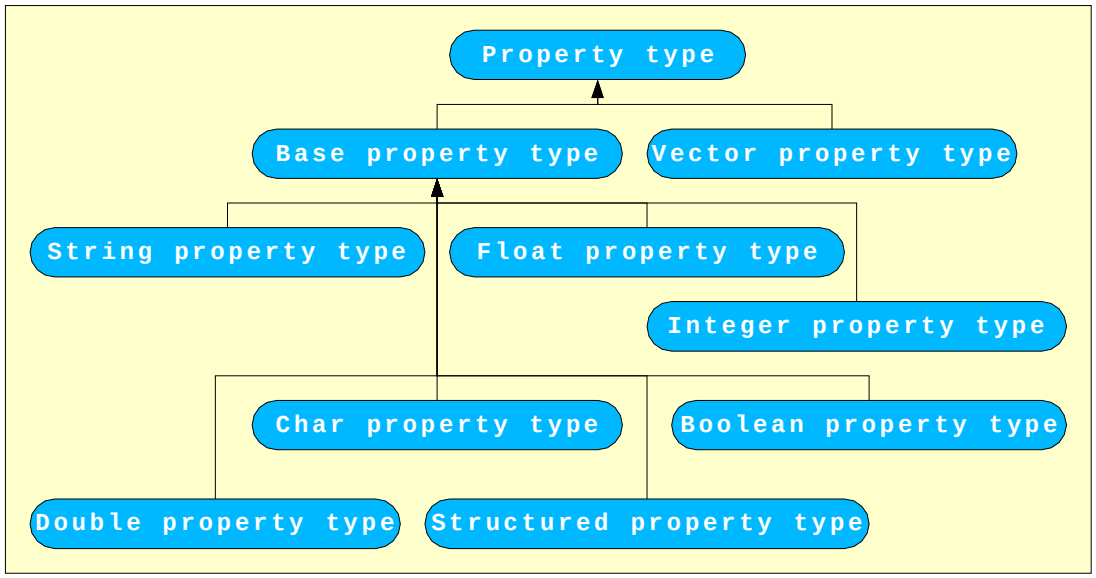

Figure 6: Property types.

To compare the type as fast as possible we designed a property type manager. It is responsible for registration and the lookup of property types. So, a simple comparison of the references returns true if two property types are equal.

4.1 Typing in components

Each out-slot of a component delivers data in a format which is specific for the slot. We already presented the property concept. A property will contain the data emitted by an out-

slot. Further, it contains the type of the property describing the format of the out-slot. An in-slot also expects the data in a defined format. So, it is recommended, that a component must provide typing interfaces for each slot. Only in- and out-slots of the same type are connectable.

Also for configuration of components with an generic authoring tool typing is very important. Components must be configurable by a property, which describes structured data formats. Using the type information it becomes easy to create a generic configuration tool based on the properties.

4.2 Persistence of components and properties

An abstraction of the reader and writer is recommended to easily provide support for different storage categories like XML based formats, stream formats or databases.

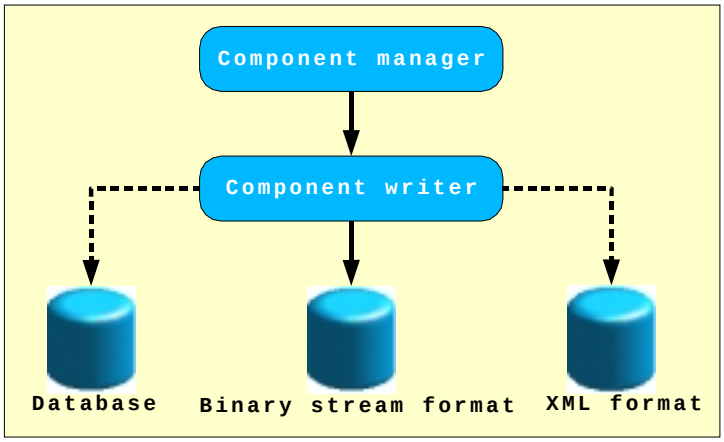

Figure 7: Exporting components.

As seen in figure 8 the component writer should be used by the component manager to export all component contained in the component manager. Each component must provide an interface to export itself to a component writer. The export must not store any connection data or data which is not necessary for the component. It is recommended that the component should use the configuration property to store itself into the component writer. So the component manager must write the name of the component category to the given output format and tell the component to export itself. After the component manager has stored all components it must write the connection data by using numerical identifications of the components. Which should easily be mapped to a database table or any other format.

The import of components (as seen in figure 9) is the reverse mechanism of the component writer. An empty component manager must be provided to the component writer to ensure correct results. The component writer reads the category name of the next component from the given input format and creates a new component of this type. This component must read itself by using a read interface of the component. As we have mentioned it before, it is recommended that this interface should use the configuration mechanism for both, the import and the export.

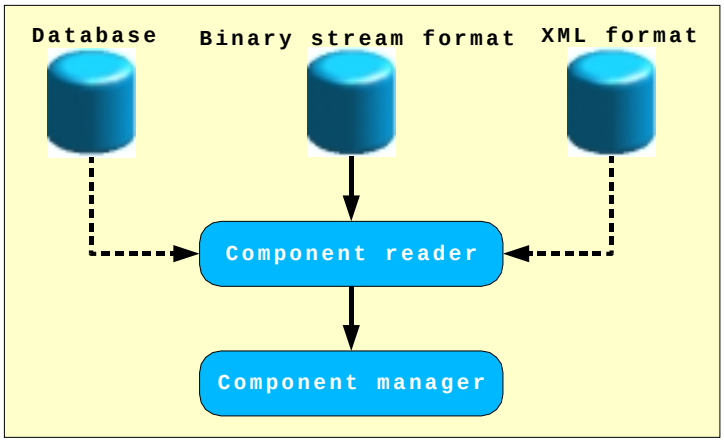

Figure 8: Importing components.

5 BRINGING IT ALL TOGETHER

The principles used for collaboration of components by using states, slots and messages are best explained by an example. Assume for instance a component that represents a gate valve. The gate valve is flange mounted between an incoming tube and an outgoing tube.

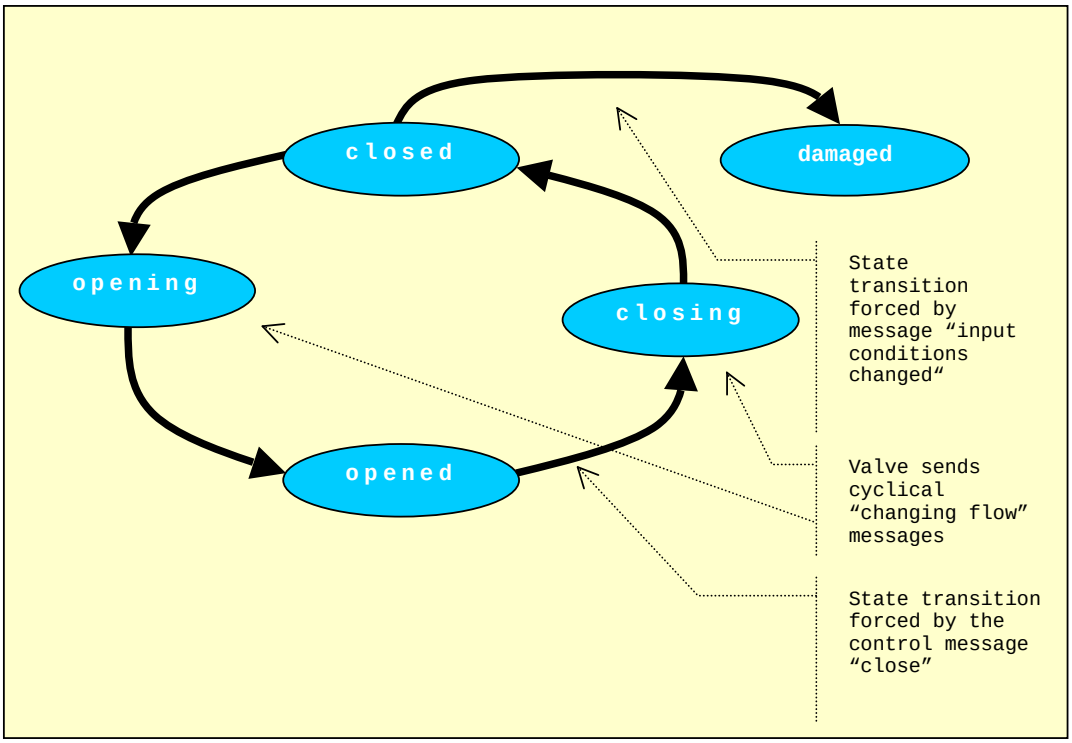

Figure 9 State-graph of a valve.

The valve component has two in-slots, which represent the conditions at the input tube of the valve and a control input to open and close the valve. Both input slots affect the component’s state. The control input for instance changes the state of the valve from “closed” through “opening” to “open” and vice versa. The in-slot that represents the condition at the input tube might change the valve’s state to “damaged” or “destroyed” if the pressure at the input tube exceeds a limit. The state graph of the valve and the tube can be found in figure 9 and figure 10. The type of the in-slot that controls the valve a string or an integer, while the type of the

in-slot that describes the input condition would be a structured property containing information about the type of liquid transported.

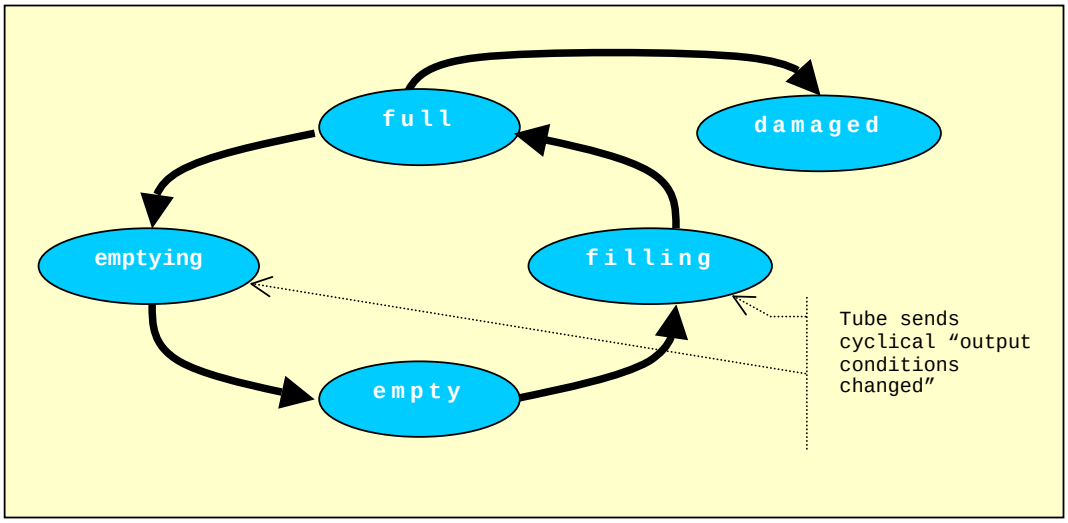

Figure 10 State-graph of Tube.

The components that represents the tubes on both sides of the valve have an in-slot which represents the input conditions of the tube and a out-slot which represents the output conditions of the tube. Input and output conditions can be described by the type of liquid inside the tube, by the pressure of the liquid and by the flow velocity. The input conditions will be propagated to the output condition after a defined time delay which depends on length and diameter of the tube. After the delay, a message “output conditions changed” will be emitted. The tube component state consists of “empty”, “full”, “filling”, “emptying” and “damaged”.

If the input tube changes its state from “empty” to “filling” it sends the message “output conditions changed” to the connected valve. The valve can propagate this message to the outgoing tube, if it is in the state “opened”, “opening” or “closing”, or it can ignore the message if it is closed.

Because the valve’s state influences also the behavior of the input tube (for instance the filling of the input tube is quicker if the valve is closed) the valve may return the “output conditions changed” message of the input tube by a “conditions not propagated” message to the input tube which tells the input tube component that it should react on changes of its input conditions like a closed tube.

If the valve changes its state from “closed” to “opening” it has to inform the connected tubes by sending a “valve flow changed” message to both of them. After the valve is completely opened, it acts like a normal piece of tube and propagates the input conditions its output slot after a certain delay. The collaboration between the valve and the connected tubes is depicted in figure 11.

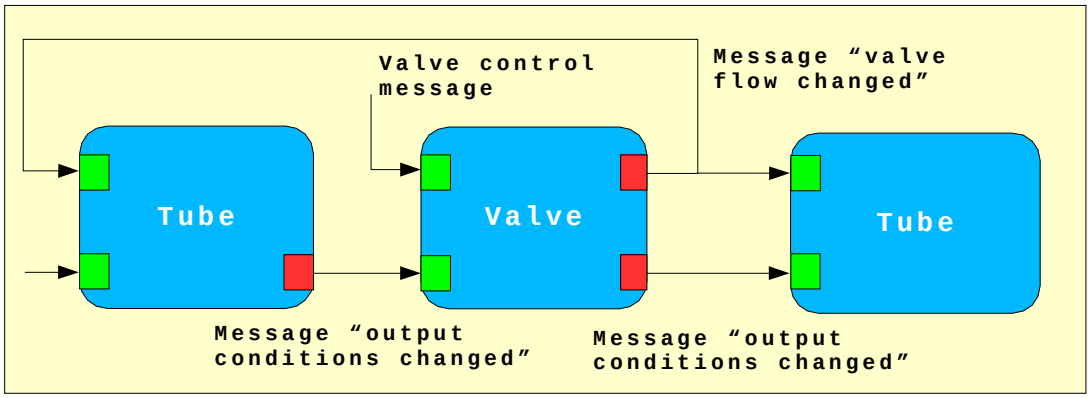

Figure 11 Collaboration of valve and tubes.

6 CONCLUSION AND FUTURE WORK

This paper presented a generic framework for 3D based applications (e.g. games, VR/MR applications). Our framework is very flexible and easy to extend – in fact, every 3D application can be realized with small effort. Our gems (software solutions, libraries) build the base of the framework. Well designed components use these gems and allow an easy integration in the framework. The concepts of the presented work will be realized in the AMIRE project, a Mixed Reality based application supported by the European Commission under the IST programme (project IST2001-34024). Actually, we are working on the first prototype based on the above concepts.

REFERENCES

[1] Steve Rabin, *Designing a General Robust AI Engine*, In Game Programming Gems, Charles River Media, pp. 221-236, 2000.

[2] Bjarne Stroustrup (The Creator of C++), *Run-Time Type Information*, In The C++ Programming Language (Third Edition), Addison Wesly Longman Inc, pp. 407-418, 2000

[3] Sun Microsystems Inc (Dale Green), *The Reflection API*, http://java.sun.com/docs/books/tutorial/reflect/index.html, 2002

[4] Trolltech Inc, *Qt 3.0 Whitepaper*, http://www.trolltech.com/products/qt/whitepaper/whitepaper.html, 2002

[5] Sun Microsystems Inc (Graham Hamilton), *JavaBeans*, http://java.sun.com/products/javabeans/docs/spec.html, 2002

[6] Sun Microsystems Inc, BeanBox, http://java.sun.com/docs/books/tutorial/javabeans/beanbox/index.html, 2002

[7] Sun Microsystems Inc, *Writing Event Listeners*, http://java.sun.com/docs/books/tutorial/uiswing/events/index.html, 2002

[8] Design Patterns in Smalltalk MVC, Design Patterns, Addison Wesly Longman Inc, pp. 4-6, 1995