Joint Virtual Reality Conference of EGVE - ICAT - EuroVR (2009)
M. Hirose, D. Schmalstieg, C. A. Wingrave, and K. Nishimura (Editors)

Short Paper: A Modular Framework for Distributed VR Interaction Processing

Ingo Assenmacher and Bruno Raffin

INRIA, LIG Laboratory, France

Abstract

Interactions are a key part of Virtual Reality systems and can lead to complex software assembly for multi-modal and multi-site collaborative environments. This is even harder, when each participant is interacting in the same virtual world by very different hardware and software capabilities. This paper outlines a software architecture and interaction processing framework developed to couple different sites in a collaborative set-up using a data-flow oriented approach. We show how we transform the site-specific capabilities to a common interface. This is used for application state processing based on a distributed actor and property model.

Categories and Subject Descriptors (according to ACM CCS): I.3.1 [Computer Graphics]: Hardware Architecture—Parallel processing I.3.4 [Computer Graphics]: Utilities—Software support I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Virtual Reality

1. Motivation

We are currently researching on a collaborative virtual environment where scientists from three distant sites can work on an interactive flooding simulation in a virtual terrain at the same time. Although collaborative environments are subject of research for more than a decade now, this project environment imposes interesting requirements: the shared application state is modified and reflected by very different input- and output-systems on each site. For example one site features a full body 3-D real-time reconstruction of the human user in the environment. At the same time, another site uses a standard desktop environment for interaction. There is no globally used Virtual Reality (VR) system on each site to implement the application. Instead existing interfaces and simulation code developed by each member of the project are used. The different codes a coupled using a distributed data-flow network middleware. We use the scenario to research on interface requirements for distributed applications in the context of cloud-computing for highly interactive applications.

The coupling of the different codes is implemented by a message passing architecture. This is a common method to design distributed applications. However, existing applications have to be adapted to this paradigm. Especially in the context of collaborative interaction, this is not a straightforward thing to do. We define a software architecture that gives design guides to the adaptation process. The motivation is to simplify the incorporation of new user modules and functionality in the collaborative environment. Existing VR systems typically offer limited support for distributed interaction processing, with the exception of remote device dispatching.

This paper presents a work-in-progress report with focus on the interaction concept of our framework. Our main goal is to have a consistent, but adaptable architecture that allows to exploit parallelism. The contributions of this paper are summarized as follows.

- We present a design that structures the physical data *aggregation and transformation* stage on top of a data-flow paradigm. This way, we are able to adapt to varying site requirements and capabilities.
- An *actor model* helps to abstract site specific capabilities in a common concept. Furthermore it allows to model the overall application requirements. Actors are defined by a varying number of properties which can communicate and change the system state.

We needed to address the interaction design, as the straightforward “distribute-and-collect” approach does not fit well to the problem of distributed interaction process-

ing. For example, motion trajectories have to be inspected by size varying partitions over time, while a button press indicates a single event in a wider time-span. The typically small amount of data to process does not justify additional overhead introduced by distributed processing and synchronization. However, an isolated component working on interaction data can be instantiated multiple times in a distributed setup. Additionally, the component can be parametrized independently and compute a different function in parallel on similar input. These are beneficial properties, so we researched on a generalization of the structuring for the distributed case.

The remainder of this paper is structured as follows. We briefly reflect influential related work in section 2. After that, we outline our method to structure the interaction processing from *physical* to *application* level. We conceive this as a clear interface to map local VR system constraints of every site to the requirements of the overall application. A two step process ensures replicability, namely the definition of a *transformation* step followed by the *specification* of application *registers*. We see the definition of these registers a crucial step in the process of assembling the application. The information in the registers can be distributed, implementing a publish-subscribe mechanism. The architecture on the physical level is described in section 3. On top of that, the main communication interface between the site applications and the data-flow network in between them has to be addressed. Here, we suggest to model *all* system entities as *actors* that have a varying number of *properties*. The latter make up the capabilities of each object in the system. For example actors with varying representations on the individual sites use different *visual properties*, whereas an object that can not be seen, does not have a visual property at all. The actor model is outlined in section 4. We close the paper by a brief explanation of the mapping of our architecture to the FlowVR middleware for distributed VR applications and give a short conclusion in section 5.

2. Related Work

A number of distribution strategies for visual reproduction exist, for example [HHN*02, AR05]. These allow to decouple rendering from application processing. However, there is no common ground for doing the same with interaction code. Usually the only type of distribution pattern is to run the data acquisition on different nodes in a distributed setup. This is considered to be a "best practice" approach and several toolkits support this traditionally, for example [THS*01]. Even when no remote data acquisition is available, most VR toolkits allow to decouple the data retrieval on thread level from the main application [RS01, BJH*01]. Common to these approaches is that their support for parallel processing ends at the point where the acquired data is passed to the interaction processing loop.

The technique of data-flow programming for interaction modelling is frequently used, for example in [RS01, FBB*08]. [JDM99] distinguish explicitly between *continuous* and *discreet* interaction paths of the data-flow network. Few described data-flow approaches discuss the need for explicit resource mapping or distribution support, for example [DJ05]. We chose to implement our approach on top of FlowVR [AR06]. This enables the transparent mapping of resources on a per-component level by declarative configuration. The FlowVR run-time environment takes control of the message passing, and any component can be executed on any known resource in a distributed set-up. The combination of different interaction streams may need an adapted mapping of resources depending on the current system layout. Clearly, a slow network connection should not be used for sending device data produced with high frequency. Approaches like [FR00] use application knowledge in order to provide sampling heuristics to ensure a low latency data acquisition. This is typically implemented by either discarding data or getting data just-in-time. The interaction modeling based on explicit registers as we present it here can be found in some VR tool-kits, for example [KBB*06]. Usually it is contained in a simpler form and more interleaved with other system interfaces than we see it.

3. Physical Processing

We suggest to distinguish between taking decisions about *physical* and *logical* interaction in order to structure the design process. The physical interaction deals with the *acquisition* of data that is used for interaction processing and *data source abstraction*. We identify three abstract stages that have to be adapted to the application needs on the physical level.

- *Data acquisition*: independent sources of data can be run asynchronously. A data source should be close to the hardware resource needed for the acquisition to ensure a first low latency access.
- *Data transformation*: interaction data coming from the sources has to be adapted to local requirements or constraints of the application.
- *Data application*: data that is available as output of the transformation stage has to be applied.

The choice of the distribution on the three stages has a direct impact on the latency of the system. The acquisition stage consists of modules which output data messages of varying types. Naturally, each module on this stage can be calculated in parallel and be bound to different resources. Information from different acquisition modules can be related using a globally comparable clock. The existence of the latter has to be ensured by the run-time environment.

A collection of acyclic graphs models the transformation stage. All data from acquisition stage serves as input to this stage. The graphs are fed to the system in a declarative style as a set of nodes and edges. Edges thereby represent data-flow from one node to the other. Nodes contain a variable

number of named *in-* and *outports*. Any inport can only be connected with a single edge. The structure supports fan-out of data from one node to a number of other nodes on every outport. The data that is passed along edges consists of an abstract data type (ADT) and timing information. Any port has a default value after creation, defined by the implementation of the ADT or given by the user. The above depicted structure defines a nested function using a graph language. Functions represented by nodes can be freely combined as long as the ADT between out- and inport matches. We assume that every node can compute when at least *one* of its inports is marked dirty. This behavior is guaranteed by the default value property of each node. The advantage of the graph approach is its flexibility to model important strategies typically needed for VR processing.

- *Exchange of transformation paths*: this is a crucial feature in the context of the collaborative application. Different hardware set-ups have to be mapped to the same virtual environment. For example when sensors are mounted differently or calibration data is acquired differently, the transformation has to account for that.
- *Aggregation of virtual devices*: different sources can be combined and 'normalized' by setting up different subgraphs and a combination of them. This property allows "device substitution". The level of interaction on each site can this way be adapted to the given system constraints.
- *Modularization and re-usability*: nodes can themselves contain graphs. Inner ports of the contained graph are exported as in- or outports of the composite node.
- *Parallel processing of independent trees*. Different trees can be processed in parallel, as long as the nodes work on sharable resources.

The resulting output of a graph is pushed to a *register*. The concept of a register can be seen as an array of ADTs combined with a notification system. Figure 1 depicts the principle of the processing on the physical stage. Each element of the register keeps a value of an ADT. Upon each update, it sends a message to N receivers waiting for a change of the element. For example the user decides on having a 3-D pointer metaphor in his application. The origin and the direction of a pointing ray in world space is needed to implement this function. In a WIMP environment, this requires the knowledge about additional system *state*, for example the viewport size and mouse position. The task of the transformation layer is to compute this function. Hardware devices, such as the mouse state, or abstract system information, for example the viewport size, act as input to this stage. The resulting pointing ray is then stored in the register. A number of subsystems may need the same pointing information for further processing. For example in an immersive environment, a visualization of a pointing ray needs to be updated as well as the input to a selection algorithm for scene manipulation. In our model, a "pointing actor" embodies the desired application stage. It has a ray as visible output and triggers selection of scene objects.

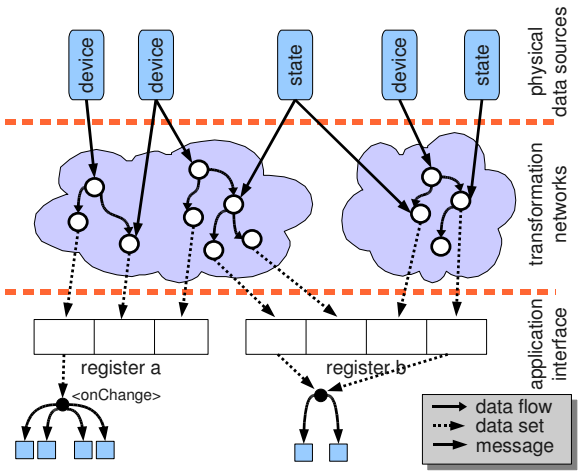

Figure 1: *The basic architecture of the physical processing layer. The transformation stage is user-definable. For our collaborative scenario, we have multiple instances of this stage. Each adapts a site's input and output behavior to the overall application needs.*

In general, the register approach forces the user to specify the main data sources for the interaction wanted, in *name*, *structure* and *semantics*. This inverts the process of thinking about interaction. In our experience, the shift goes from "what types of events does the framework provide" to "what type of information is needed to realize my interaction".

4. Logical Processing

On a logical level, we describe the virtual world as a collection of *actors*. Each actor represents an internal and external state. The latter consists of a collection of *actor properties* which can vary on a per-actor level. These properties build a hierarchy of representations that communicate with each other. For example, an actor that can be perceived visually owns a *visual* and a *spatial* property. Traditional approaches map this to the ability of modifying geometry and transformation for an object independently in a scene graph. However, we suggest to model all *necessary modalities* of a scene object in that way. Figure 2 shows a non exhaustive set of actor properties that we have identified for our applications. Each interdependency results in an exchange of messages between properties upon a change of their state. For example the change of the visual appearance of the actor needs to update information in the collision detection sub-system as well as in the graphical rendering. At the same time it may or may not have impact on the acoustic rendering of the actor. We see the advantage of the dynamic property mechanism in the adaptability of the application to the targeted types of interaction. It is thereby based on a consistent mechanism: in case a certain aspect has to be addressed by an application

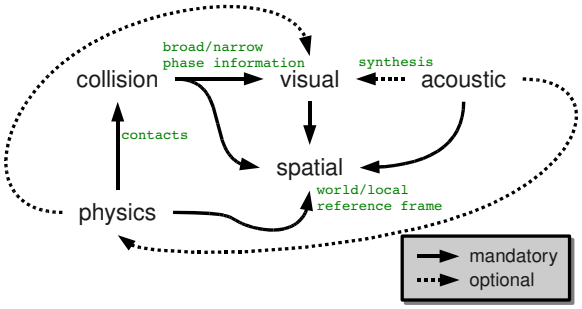

Figure 2: *A relation diagram of actor properties, as we identified them for our application. The edge labels hint at the purpose of the communication relation between the properties.*

object, it has to expose a specific property usable for that purpose. For example, an object that has to be recognized by a collision detection sub-system has to export a *collision* property. That way, the system is offered an interface to access all information it needs to process the object for the purpose of collision detection. Each result of a property change can be computed in parallel, as the interface and the state data is separated from the main object by design.

5. Implementation and Conclusion

We implemented our above discussed concepts using the FlowVR middleware. Especially the distributed rendering approach as described in [AR05] is a key concept to implement visual properties. Generally, FlowVR allows to model an application based on modules and hierarchies of modules, called meta-modules. An actor is implemented as a meta-module containing its inner state and a collection of property modules. The actors reside in an *actor home*, which is a meta-module as well. In the current implementation of FlowVR, connections between modules are not mutable after the network creation. The actor home is a mean to avoid a complete graph between N actors. For the collaborative application we are working on, three actor homes naturally map to each site. An actor home offers a single input port. A message coming in on that port contains routing information that defines the recipient actor. This is implemented on the *stamps* concept of FlowVR [AR06]. The message will be routed inside the actor home to the actor. It will then act accordingly on the actor's properties and subsequently change the system state.

This paper presents an outline of our distributed approach to create interactive, distributed VR applications. Our distributable approach allows data source abstraction and virtual device composition, as well as parallel device processing. Two of our current strategies in implementing a collaborative application scenario were described. One is the software architecture for adapting to site-specific hardware and software environments. A data-flow oriented transformation network is used for that purpose. The interface to the application stage is modeled by a register approach. The information contained in the registers is propagated to a set of actors and their properties. These actors define the overall application semantics. With the tools described, we hope to build an adaptable, low-latency and efficient prototype that will be of interest to the VR community.

Acknowledgment

This work was partly funded by Agence Nationale de la Recherche, contract ANR-06-MDCA-003.

References

[AR05] ALLARD J., RAFFIN B.: A Shader-Based Parallel Rendering Framework. In *Proceedings of IEEE Visualization '05* (Minneapolis, USA, October 2005).

[AR06] ALLARD J., RAFFIN B.: Distributed Physical Based Simulations for Large VR Applications. In *Proceedings of IEEE Virtual Reality '06* (Alexandria, USA, March 2006).

[BJH*01] BIERBAUM A., JUST C., HARTLING P., MEINERT K., BAKER A., CRUZ-NEIRA C.: VR Juggler: A Virtual Platform for Virtual Reality Application Development. In *Proceedings of IEEE Virtual Reality '01* (March 2001), Takemura H., Kiokawa K., (Eds.), Virtual Reality Society Japan, pp. 89–96.

[DJ05] DELINGIANNIDIS L., JACOB R. J.: Improving Performance of Virtual Reality Applications Through Parallel Processing. *The Journal of Supercomputing 33* (2005), 155–173.

[FBB*08] FIGUEROA P., BISCHOF W. F., BOULANGER P., HOOVER J. H., TAYLOR R.: InTml: A Dataflow Oriented Development System for Virtual Reality Applications. *Presence Teleoperations and Virtual Environments 17*, 5 (September 2008), 492–511.

[FR00] FRÖHLICH T., ROTH M.: Integration of multidimensional interaction devices in real-time computer graphics applications. In *Proceedings of Eurographics* (2000), Blackwell Publishers, U.K.

[HHN*02] HUMPHREYS G., HOUSTON M., NG R., FRANK R., AHERN S., KIRCHNER P. D., KLOSOWSKI J. T.: Chromium: a stream-processing framework for interactive rendering on clusters. *ACM Trans. Graph. 21*, 3 (2002), 693–702.

[JDM99] JACOB R. J., DELINGIANNIDIS L., MORRISON S.: A Software Model and Specification Language for Non-WIMP User Interfaces. *ACM Transactions on Computer-Human Interaction 6*, 1 (March 1999), 1–46.

[KBB*06] KREYLOS O., BERNARDIN T., BILLEN M. I., COWGILL E. S., GOLD R. D., HAMANN B., JADAMEC M., KELLOGG L., STAADT O. G., SUMNER D. Y.: Enabling Scientific Workflows in Virtual Reality. In *Proceedings of the ACM SIGGRAPH International Conference on Virtual Reality Continuum and Its Applications (VRCIA'06)* (2006).

[RS01] REITMAYR G., SCHMALSTIEG D.: An Open Software Architecture for Virtual Reality Interaction. In *Proceedings of the ACM Symposium on Virtual Reality Software and Technology (VRST'01)* (New York, NY, USA, 2001), ACM, pp. 47–54.

[THS*01] TAYLOR R. M., HUDSON T. C., SEEGER A., WEBER H., JULIANO J., HELSER A. T.: VRPN: A Device-independent, Network-transparent VR Peripheral System. In *Proceedings of the ACM Symposium on Virtual Reality Software and Technology (VRST'01)* (New York, NY, USA, 2001), ACM, pp. 55–61.