Flexible Architecture for the Development of Realtime Interaction Behavior

Gerwin de Haan* Frits H. Post

Delft University of Technology, The Netherlands

ABSTRACT

In this paper, we describe our work in progress on the architecture of StateStream, a software model for the development of interaction behavior in realtime interactive systems. The StateStream design aims to support the design and development cycle of complex interaction techniques, object behavior and interaction scenarios and is currently targeted at concurrent and fluent direct manipulation in Virtual Environments. StateStream provides a dual modeling approach to model both discrete behavior and continuous interactive behavior. The architecture integrates these modeling features with the underlying graphics and tracking libraries. By design, a dynamic programming language is used both throughout the architecture and in the behavioral model descriptions. We highlight some of the features of the StateStream model and illustrate how its tight integration through the Python language offers a flexible framework for studying and adapting the complex interactions between description models and architecture components. We believe this approach provides flexible run-time experimentation environments and tool chains, and can lead to improved insights and requirements for future interaction description models and architectures for realtime interactive systems.

Keywords: Virtual Reality, 3D Interaction, Scripting languages.

Index Terms: I.3.7 [Computing Methodologies]: Computer Graphics—Virtual Reality; D.2.6 [Software]: Software Engineering—Programming Environments

1 INTRODUCTION

Well-behaved *interactivity* is a core aspect of Virtual Reality and other *realtime interactive systems (RIS)*. Recent advances in computer technology triggered a new generation of post-WIMP interfaces and enable new forms of interaction [6]. Although several interface and interaction descriptions methodologies and taxonomies exist, more complex interaction techniques and scenarios become harder to design, to model, to program, to debug and to evaluate. The introduction of new input modalities (e.g. multi-touch displays) and a trend towards multi-user and distributed systems further complicate the design and development of interfaces, interaction and the underlying software models and architectures.

With the next generation of devices, interfaces and interaction scenarios well within reach, we should strive for a good understanding of the properties and issues of interaction modeling. As interaction and the related software are strongly interconnected, it is an important element in the requirements of any RIS architecture. Given this importance, we believe that interaction design and development issues should be considered an essential part of the RIS architecture development, as well as a part of application design cycles. We consider that powerful design patterns are not designed, but rather discovered through trial and error during iterations in *reflective practice*. However, support for the iterative (process of) design, development and evaluation for interactivity, often lacks in current RIS frameworks. It is also hard to design either an interaction specification model or a RIS framework separately with the other in mind.

Therefore, during our ongoing design and development efforts of our *StateStream* interaction model, this is *gradually* integrated with our VR software architecture. While the model is in development, we use the Python language as the integrating layer between the modeling components and the underlying architecture and experiment with its possibilities. This allows us to quickly transfer and integrate functional elements between model and software architecture.

In the remainder of this paper we will elaborate on this integrated approach and demonstrate the current state of our StateStream model and environment. First, we discuss related work on interactivity modeling for VR and underlying architectures of current systems in section 2. Then, in section 3, we describe our current StateStream model and how this integrates with the underlying architecture. Finally, we conclude in section 4 and indicate possible directions and areas for future research.

2 RELATED WORK

Generally, flexibility in the description of interactive behavior in VR systems is supported through one or more (formal) models, a special API, scripting or a combination of either one.

Modeling languages are used to describe interaction behavior while avoiding programming and validity issues for the end-user. For example, with the InTml [3] specification language one can describe 3D interaction techniques, input devices and their connections. It uses a data flow model to describe interaction techniques in terms of filters. Also UNIT [9] uses a similar data flow model here and focuses on flexible redefinition of continuous behavior of interaction techniques. The IFFI system [10] provides an even higher abstraction to allow for reuse of techniques across different VR toolkits. In contrast with the mainly continuous behavior descriptions of interaction technique components, other models such as statemachines better grasp discrete behavior. For example, CHASM [12] is a recent example of the use of *StateCharts* to describe VR interaction techniques. In the work by Jacob et al. [7][11], a StateChart model and a data flow model are used concurrently to describe and program non-WIMP user interfaces. Our StateStream system is build on a similar, dual-modeling approach where both state mechanisms and data flow are first-class entities. In this way we can model both event-based and behavioral-based interactivity in our approach.

We avoid the use of a specialized declarative model language in this early stage of development, and instead explicitly describe interaction techniques using Python scripted constructions. Zachmann [13] already proposed the use of scripting to define interactive behavior and provided special-purpose scripting language on a fixed interaction API. Early versions of Alice toolkit [1] also provided Python-based scripting environment for interaction and low-level extensions, to lower the language learning barrier. Al-

*e-mail:g.dehaan@tudelft.nl

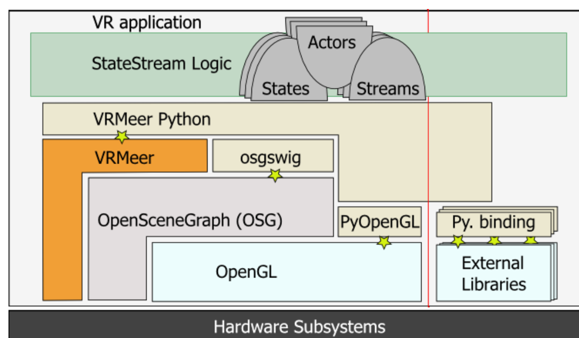

Figure 1: Overview of the StateStream model and VRMeer software layers (see Section 3.1). A pure VRMeer application can have native access to various components and external software or, preferably, through Python. A StateStream application uses actors with state-based and streaming behavior model descriptions. Stars indicate Python bindings on the underlying libraries.

though the use of a scripting language provides some flexibility, it does not directly provide higher level tools which might be helpful in the interaction modeling process. Hendricks [5] highlighted the need for VR interaction to allow a smooth migration of novice users to becoming more experienced and proposed the use of *meta-authoring tools* which assist in the creation of interactions by specifying higher-level, event-action pairs. By connecting our base VR layers with a wealth of available Python libraries, we are able to gradually build up the architecture and development front-end tools over various iterations. As a result, the Python dynamic language serves as a unified syntax and semantics for describing interactivity between objects, connect application components and produce development and analysis tools.

3 System Description

In our group we created the *VRMeer* VR toolkit to develop Virtual Reality applications in C++ for our various VR hardware systems. The toolkit provides common, hard-coded interaction techniques which cannot readily be extended for complex interaction scenarios. The StateStream model builds upon the toolkit and can replace the existing interaction mechanisms.

3.1 Base Architecture

The VRMeer toolkit mainly builds upon the OpenSceneGraph library, and is a redesign of our previous *iVR* toolkit which was OpenGL Performer based. In a similar fashion to our early VR toolkit, *iVR*, we use *SWIG* to create Python bindings for both OpenSceneGraph (*osgswig*[1]) and the VRMeer toolkit. Through sets of flexible Python abstraction layers, one can quickly integrate various Python and C++ toolkit functionality in a VR application, also see [2]. A schematic overview of the software layers in the *VRMeer* system is shown in Figure 1. On top of the base abstraction layers, our proposed StateStream interaction model runs to facilitate interaction modeling integrated with system functionality.

A VR application is written in (sets of) Python file(s), which contain the loading of 3D models, setting up devices, connecting interaction tools, defining specific callbacks, etcetera. For ease of use, a standard set of procedures and callback functions to interact with existing scene-graph traversals is provided. Through execution of the main file by a Python interpreter, necessary libraries are loaded, and the main loop of the VR run-time environment is invoked. This Python-based environment further controls the execution of the VR application. Currently, as we extend Python by importing the VR libraries in a running Python interpreter instance, this main Python instance is the controlling process over the entire application.

[1] http://code.google.com/p/osgswig/

3.2 StateStream Model

In this section, a short overview is given of the current StateStream interaction model. StateStream currently uses a dual modeling approach of interaction, similar to PMIW [7]. The first modeling primitive is the *statemachine*, a state-based mechanism intended for describing discrete behavior. The second modeling primitive is the *streammachine*, intended for modeling conceptually continuous streams of information. The description of the complete model with in-depth properties and design considerations is currently in progress and outside the scope of this paper.

We first split behavioral functionality into conceptual *actors*, each of which contain the behavioral primitives as described above. In our case, an actor often consists of a visible VR object and its behavior specification. For interaction techniques, actors include the graphical elements such as cursors, icons and rays.

For describing discrete behavior, we use a custom *StateChart* variant, which consists of hierarchical, concurrent statemachines. A graphical representation, automatically generated from an instance of such a statemachine, is shown in Figure 3B. For each StateStream application, we instantiate a main, top-level statemachine. All other statemachines existing in the applications are attached as (concurrent) child statemachine of this statemachine, or descendants thereof. Typically, state variables of actors and its graphical objects are directly related to, or even modeled as statemachines. State transitions can occur if certain *events* match the conditions of available transition. Custom functionality is defined in callback functions such as state entry and exit functions, or at transitions. A main functional element here is the sending of events to other objects, and the creation, enabling and disabling of connections between other streammachines.

The *streammachine* model is intended for modeling conceptually continuous streams of information. For this, we use a data flow graph structure, which consists of a set of connected nodes or filters with various input and output ports, see Figure 3C. The ports of the nodes can be connected through *connections*, over which information of various data types can be transported. The custom functionality of the nodes is defined in callback functions on the incoming and outgoing ports. Typically, these functions are executed every frame. Therefore, continuous variables of an actor and its graphical objects, e.g. position, rotation, color or size are easily modeled and connected through filters. A second element is the use of a specialized streammachine, which can trigger the generation of an event based on incoming streams.

A StateStream-based application consists of the description of the actors, their statemachines and streammachines. After initialization, the main loop proceeds with the distribution of aggregated events to the top-level statemachine. Then, the acyclic data flow graph of streammachines is ordered, and values are pushed through the graph. Functionality in both models can influence underlying content, for example nodes in the scene graph. To obtain events and streams in the StateStream model, a set of converters is used to translate events and variables for the underlying libraries, e.g. the scene graph or tracking libraries. An example of this is the conversion of button presses and coordinates of a 3D tracker device, where button presses are converted to StateStream events, while the coordinates are handled through streammachine nodes. By already performing this conversion at a low-level, the fine-grained details of interactivity can be flexibly modeled, composed and inspected through their explicit models.

3.3 StateStream Example

To give some insight on the model's use in practice, we take a simplified ray-based selection and manipulation scenario as an exam-

ple. Two simple VR objects can be interacted with at the same time by two interactors connected to 6DOF styli in both hands (Figure 3A). The application's state contains four concurrent sub-states: two for the interactors and two for the objects. Continuous device streams, such as stylus position and orientation, are connected to ray-intersection streammachines which can generate "hit" events. An interactor's statemachine (Figure 3B) can respond to these events, notify the VR objects it hits and connect to some of its stream ports (Figure 3C). A separate streammachines can be used to calculate the length of the ray. When a "button" event is received, the interactor can reconnect streams to impose a new position and rotation onto the hit VR object. As a result, the object's position is defined as the stylus position "filtered" through a chain of connected streammachines.

The combined behavior of both interactor and object is defined in the actors' statemachines and streammachines. We can define special interactors to work on regular objects or define special object behavior for various interactors, all on a fine-grained level. Through concurrent state-based behavior, each interactor or object can also maintain a separate child state- and streammachines for each related actor. In this example, we can extend one interactor's behavior to support manipulation of multiple objects at the same time. Two handed object manipulation of a single object can also be described by connecting interactor streams to different functional ports of the object, such as position and rotation. The complexity in sets of states and stream connections quickly explodes in such a scenario, and will require extra tools for visualization and development.

3.4 Integration with Python

The complexity and unclarity in the model requirements and the supporting architecture withheld us from directly specifying which functional features should be placed in which part of the system. The dynamic properties Python language gives us the advantage in the development of the modeling and architecture. A main feature is that both the two modeling primitives are described through Python class structures, and also their executing engines are Python based. First, it avoided the up-front specification of protocol structures and static data types. We can rapidly transport various types of structures through the event and stream channels. Second, we introduce a bridge between descriptive and imperative programming, as flexible syntax constructs mimic declarative programming. The class inheritance mechanism provides an expressive way for reusing constructions. Third, the introspective properties allows us to inspect types and to act accordingly at run-time or display them in the interface. In conclusion, it brings us a tight integration between description model and its execution engine, as well as underlying VR architecture. We observe that the descriptive modeling of interaction integrates functionality at a single level of abstraction, which is orthogonal to the software layers.

3.5 Iterated Development

Since its early conception, the model and underlying architecture have gone through several iterations of advancement. More and more interaction parts of the original VR toolkit were converted to use StateStream primitives. Early versions consisted only of state-based primitives, after which it quickly became clear that this was unpractical for modeling continuous relations. During a transition of both the StateStream model from our old iVR toolkit to the new VRMeer toolkit, the logic of the interaction techniques was preserved. Only a limited set of relations to the underlying scene graph and toolkit had to be changed in order to restore functionality. More recently, we introduced the derivation and instantiation of StateStream-based component templates. This allows more elegant re-use of code and provides more concurrent state-based functions.

3.6 Run-time Environment

We provide an interactive (graphical) debugging or development environment, available concurrently with the VR application. This is achieved through the integrated use of the *GTK* GUI toolkit and an *iPython* interactive Python shell. As all components of the application can be accessed through the same single Python instance, one can rapidly create GUI components and adjust relations between them. Through small pieces of code or a layout designer, a development workspace can be created, see Figure 2. This workspace can be displayed on a separate screen or machine, with the benefit that a user can perform an action in the Virtual Environment while the interaction designer or developer inspects the flow of events and the correct handling of the states, or even interrupts or assists in the sessions by performing extra actions. This possibility is especially useful when used for inspecting StateStream modeled behavior. For example, we display a list of active statemachines, each of which when clicked, generates a graphical representation of its current state.

4 Conclusions and Future Directions

The design and evaluation of the interactive components of RIS remains a complex task. Simple testing and evaluation of system properties as a whole are not adequate, as Olsen stated in [8]. Therefore, our proposed StateStream model and architecture focuses on providing the suggested *visibility*, *flexibility* and the *expressiveness* necessary to rapidly iterate to a better user interface system, and resulting user interfaces in general. At its current state of development, it is difficult to objectively assess the added value of the proposed StateStream model and the supporting architecture. Nevertheless, the increasing rate of the addition of useful features and growing insight in dynamic relations between our interaction model and architecture already indicate we are making good progress. We feel that, with the proposed combination of evolving descriptive, semi-formal models and tool-chains, we increase our understanding of the complex logic of interactivity, the software involved, as well as its development cycle. The use of a single unified language accelerates these iteration cycles.

We hope that through this line of work, we demonstrate designers of next-generation RIS systems the importance of flexibility in interaction design, and encourage the adoption of the base technologies that allow similar development strategies. In this way, our research community might be better prepared to offer robust system designs for useful next-generation interfaces and devices. In the line of the current approach, we also see two important directions of future developments in interaction modeling and RIS system development in general.

The first direction is to work towards a extensible development, rapid-prototyping and evaluation environment for fine-grained interaction behavior development for VR applications, similar to what *d.tools* does on a higher level for external devices [4]. Our StateStream work increased our understanding that in interaction models are difficult to predict, design and realize. The complexity of technical and aesthetic relations between interaction models, implementation and understandable end-user interfaces make this hard, and we do not expect that a single model or development tool can alleviate this. As also stated in [5] and [2], we therefore continue the use of many flexible layers of abstraction and prototyping cycles in the development cycle for interaction. In these cycles, the model, (visual) languages and front-end GUIs and evaluation strategies gradually become available and continuously evolve. Depending on the application, developers, interface design and end-users, one should quickly be able to adapt available language constructs, models and visual development tools to fit that specific case. The main work needed in this area is on rapid prototyping, deriving interaction models, construction of new front-end interfaces and the creation of analysis tools for interaction models.

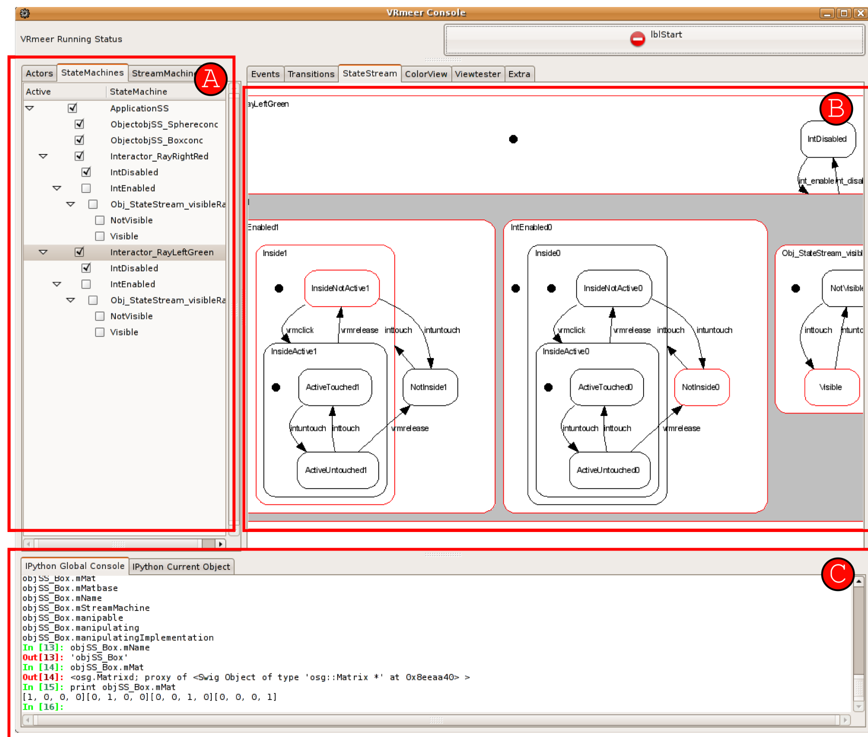

Figure 2: Typical layout of a StateStream application within the interactive VRMeer run-time environment: List overviews of active state- and streammachines (A), generated graphical StateChart representations (B) and interactive IPython shells to inspect and control objects (C).

A second direction is to obtain more useful, (semi-)formal properties in the interaction model. This is necessary to enable reasoning about concurrent and connected components of interaction behavior. One goal is to offer robust serialization for recording, analyzing and replaying interactive sessions. A second goal is that of optimization and distribution. Through analysis of interdependencies and requirements of the streammachine components optimization and guarantees of responsiveness and latency of interaction techniques can be made. Of special interest are the separation and distribution of parts of the statemachines or streammachines over multiple, connected StateStream instances on independent processes or machines. To provide such model features, it is necessary to make sure the related implementation part does not invalidate newly introduced model assumptions, e.g. no side effects occur with model operations.

Acknowledgements

Part of this research has been funded by the Dutch BSIK/BRICKS project.

References

[1] M. J. Conway and R. Pausch. Alice: easy to learn interactive 3D graphics. *SIGGRAPH Comput. Graph.*, 31(3):58–59, 1997.

[2] G. de Haan, M. Koutek, and F. H. Post. Flexible Abstraction Layers for VR application development. In *Proc. IEEE VR 2007*, pages 239–242.

[3] P. Figueroa, M. Green, and H. J. Hoover. InTml: a description language for VR applications. In *Proc. Web3D 2002*, pages 53–58.

[4] B. Hartmann, S. R. Klemmer, M. Bernstein, L. Abdulla, B. Burr, A. Robinson-Mosher, and J. Gee. Reflective physical prototyping through integrated design, test, and analysis. In *Proc. UIST 2006*, pages 299–308.

[5] Z. Hendricks, G. Marsden, and E. Blake. A meta-authoring tool for specifying interactions in virtual reality environments. In *Proc. AFRIGRAPH 2003*, pages 171–180.

[6] R. J. Jacob, A. Girouard, L. M. Hirshfield, M. S. Horn, O. Shaer, E. T. Solovey, and J. Zigelbaum. Reality-Based Interaction: A Framework for Post-WIMP Interfaces. In *Proc. CHI 2008, to appear*.

[7] R. J. K. Jacob, L. Deligiannidis, and S. Morrison. A software model and specification language for non-WIMP user interfaces. *ACM TOCHI*, 6(1):1–46, 1999.

[8] D. Olsen Jr. Evaluating user interface systems research. In *Proc. UIST 2007*, pages 251–258.

[9] A. Olwal and S. Feiner. Unit: modular development of distributed interaction techniques for highly interactive user interfaces. In *Proc. GRAPHITE 2004*, pages 131–138.

[10] A. Ray and D. A. Bowman. Towards a system for reusable 3D interaction techniques. In *Proc. VRST 2007*, pages 187–190.

[11] O. Shaer and R. J. K. Jacob. Toward a Software Model and a Specification Language for Next-Generation User Interfaces. *ACM CHI Workshop on The Future of User Interface Software Tools*, 2005.

[12] C. Wingrave and D. Bowman. "CHASM": Bridging Description and Implementation of 3D Interfaces. In *Proc. IEEE VR Workshop on New Directions in 3D User Interfaces*, pages 85–88, 2005.

[13] G. Zachmann. A language for describing behavior of and interaction with virtual worlds. In *Proc. VRST 1996*.

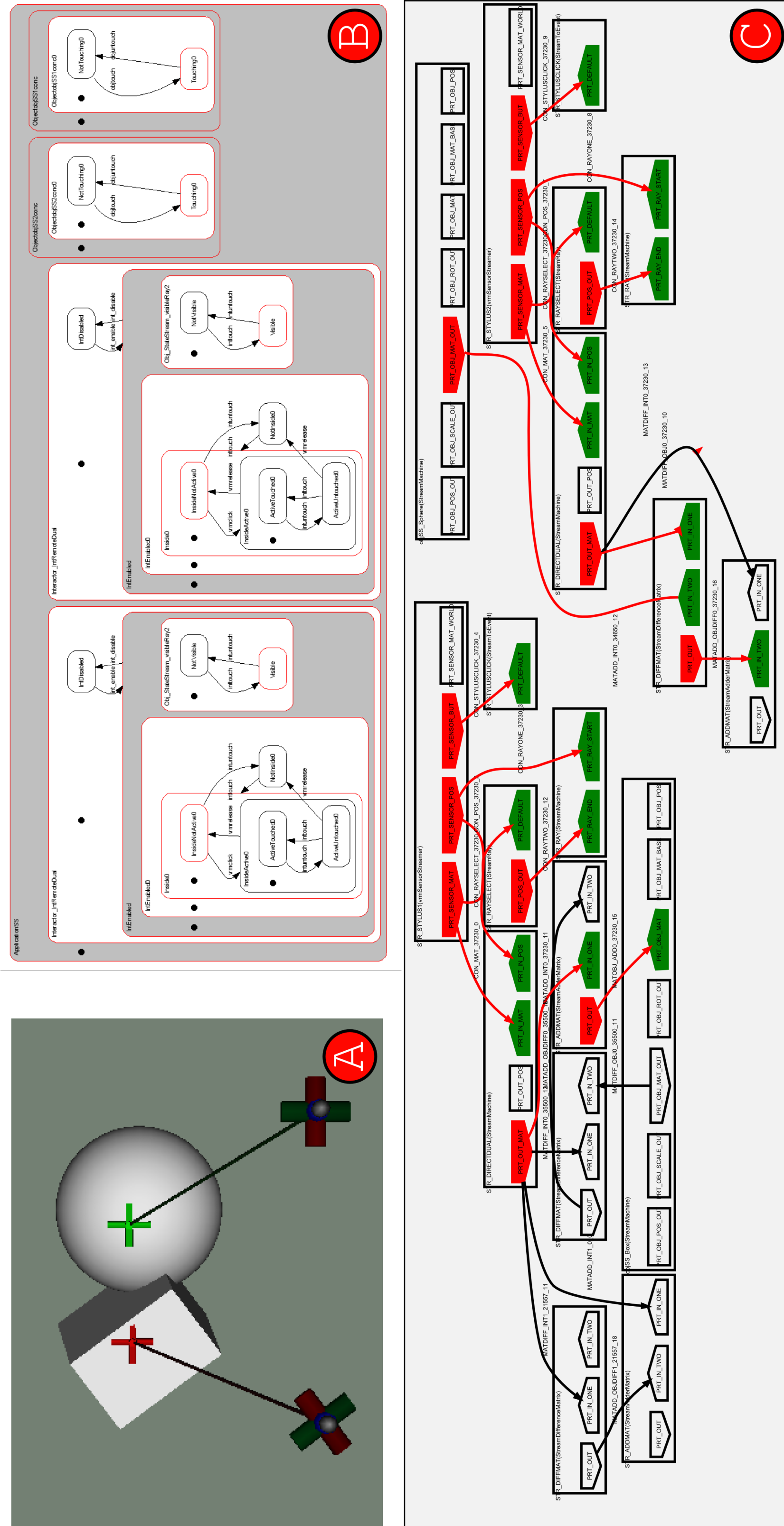

Figure 3: Overview of a basic VR interaction scenario modeled with StateStream. Two VR objects(A) are selected and manipulated in various ways over time by two different interactors. The diagrams for states (B) and streams (C) graphically represent the changing relations and transitions between various interaction components.