StateStream: a Developer-Centric Approach Towards Unifying Interaction Models and Architecture

Gerwin de Haan and Frits H. Post
Data Visualization and VR Group
Delft University of Technology
P.O. Box 5031, 2600 GA, Delft, The Netherlands
{g.dehaan, f.h.post}@tudelft.nl

ABSTRACT

Complex and dynamic interaction behaviors in applications such as Virtual Reality (VR) systems are difficult to design and develop. Reasons for this include the complexity and limitations in specification models and their integration with the underlying architecture, and lack of supporting development tools. In this paper we present our *StateStream* approach, which uses a dynamic programming language to bridge the gap between the behavioral model descriptions, the underlying VR architecture and customized development tools. Whereas the dynamic language allows full flexibility, the interaction model adds explicit structures for interactive behavior. A dual modeling mechanism is used to capture both discrete and continuous interaction behavior. The models are described and executed in the dynamic language itself, unifying the description of interaction, its execution and the connection with external software components.

We will highlight the main features of StateStream, and illustrate how the tight integration of interaction model and architecture enables a flexible and open-ended development environment. We will demonstrate the use of StateStream in a prototype system for studying and adapting complex 3D interaction techniques for VR.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques— *State diagrams; user interfaces*; H.5.2 [**Information interfaces and presentation**]: User Interfaces— *Graphical user interfaces; prototyping*

General Terms

Design, Algorithms

Keywords

Model-Driven Engineering, User Interface Description Language, 3D interaction, Python

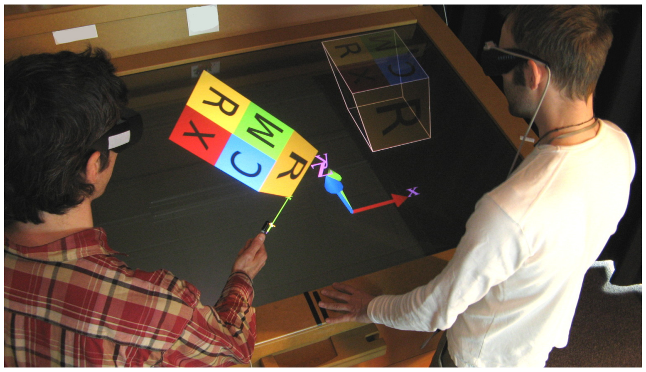


Figure 1: Example of a dynamic, 3D manipulation technique in a multi-user VR setup. When multiple interaction tools can operate on multiple objects, interaction behavior can become difficult to design and program.

1. INTRODUCTION

The look and feel of a well designed interaction technique appears simple, logical and intuitive to its users. Little thought goes to the often painstaking and time-consuming development process of making interaction techniques work as designed, error-free and well-tuned. Even with use of existing interface and interaction modeling methodologies, taxonomies and software tools, it remains hard to design, to model, to integrate, to debug and to evaluate complex interaction techniques and scenarios. Trends in new input modalities such as multi-user displays and distributed systems further complicate the design and development of interfaces, the underlying software models, architectures and tools.

In our recent efforts in developing multi-user and multi-handed input 3D interaction techniques [12], we encounter many situations where the complexity of the interaction description explodes. Simple interaction techniques are relatively straightforward to design and implement, but their re-use through variations and combinations can easily lead to unexpected results, often only to be discovered while already applied in a VR application.

Consider the example in Figure 1, where a 3D box can be selected and freely manipulated with a ray controlled by a tracked stylus. This works intuitively for a single user operating the VR system, but when a second user joins in and

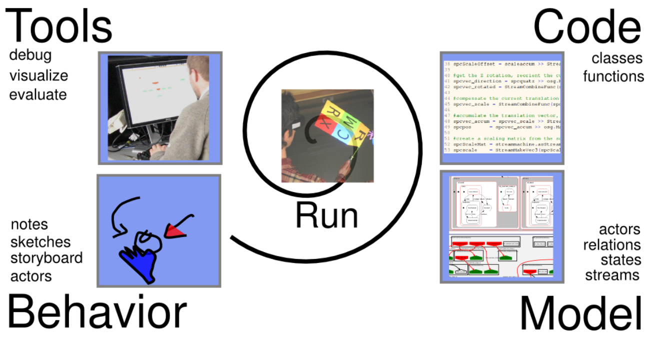


Figure 2: Cyclic development. From an informal behavior description, one generates a model abstraction which closely maps to running code. Errors or unexpected situations may occur during interaction, and need to be detected, analyzed and fixed. StateStream integrates model, code and tools.

grabs the same object, the interaction behavior can become more complex. As a side-effect of our original event-based implementation, the object's control is just taken over by the last selecting user. Naturally, one would want to decide which alternative behavior is used, such as averaging interacting forces, bending one user's ray or scaling the object. The relations between all components depend on which combination of interaction tools operate on which object types. To allow for these alternatives, one would need to rethink and re-implement the interaction behavior in detail.

Although much effort can be put into capturing these new situations in hard coded, imperative commands of callbacks and event-handlers, problems concerning the number of object relations and *exceptions to the rule* either get overlooked or quickly overwhelm development tasks. In practice, classic interaction modeling concepts and development tools often lack integration and rarely provide the right level of abstraction for effective design and problem solving. These restrictions often prevent developers and designers from adopting and customizing more sophisticated 3D interaction techniques in real VR applications.

Our motivation for this work follows from these issues. We feel that (combinations of) interactive behavior are inherently complex and require model-based design and supporting analysis tools. At the same time however, one wants to avoid restrictions a model imposes on the flexibility of existing software tools and existing design skills. A flexible integration of model, architecture and the supporting tools is important to support developers with varying skills and backgrounds, ranging from graphics programmers to interaction designers. The main problem is that many existing interaction models are far separated from other, external software components. This separation limits their descriptive power, thereby restricting the visibility of features and issues when integrated in run-time environments. This makes it difficult to appreciate a model's value, especially in agile scenarios with many software components and a cyclic process of design and development, see Figure 2. In this paper, we address this issue of separation and discuss our developer-centric approach. We present *StateStream*, a pragmatic software approach to unify interaction models and architecture.

The contribution of StateStream is the developer-centric approach of using a dynamic language to unify an interaction model with underlying architecture and tools of interactive applications. The interaction model provides semantic structure, but is described using in the same language with familiar syntax as the other system components. It allows integration and transition of existing code and control structures, but also benefit from dynamic language features such as dynamic execution, introspection and extension at run-time. With this approach we have implemented and integrated a dual interaction model, consisting of separated StateChart and Data Flow primitives. We describe how this model provides powerful composition patterns for code-reuse, how it eases integration with underlying system components. To demonstrate the functionality of our model and approach, we describe the creation, adaptation and reuse of several interaction tools within our VR framework.

The remainder of this paper is organized as follows: We first discuss related work on interaction models, tools and architectures of current interactive graphics systems in section 2. Then, in section 3, we describe the StateStream model and its components in technical detail. After a description of the implementation in section 4, the process of creating several 3D interaction techniques within our VR demonstrator is described in more detail in section 5. Finally, we discuss the results of this work in section 6 and conclude and give our view on future work in section 7.

2. RELATED WORK

In this section we discuss the position of the StateStream approach with respect to related work. We consider three main themes of interest: model-based design, its practical integration in the underlying architecture, and the supporting software tools in the development cycle.

2.1 Model-based Design

Modeling languages are considered an essential asset in describing and implementing interaction behavior while avoiding detailed execution and validity issues for users. Among the various models, the Data Flow paradigm and state based models are the most well-known for user interaction. The Data Flow model is widely applied to describe system flow and interaction techniques in terms of filters, often with a focus on reconfiguration. For example, with the InTml [9] specification language one can describe 3D interaction techniques, input devices and their connections. UNIT [22] uses a similar model and focuses on flexible redefinition of continuous behavior of interaction techniques. The IFFI system [23] provides an even higher abstraction to allow for reuse of techniques across different VR toolkits. FlowVR [1] extends the Data Flow approach over its entire VR architecture to provide distributed capabilities. Data Flow models excel in their description of continuous behavior components, but often require ill-formed constructions to support simple tasks such as message passing and event handling.

State-based models are better suited to model discrete behavior and integrate with event-based systems. For specifying reactive user interface behavior, special interest goes to StateCharts [13], which provides a visual formalism for hierarchical, concurrent statemachines. The hierarchy and concurrency can avoid state explosion, see [28]. Many StateChart variants exist with different model properties and operational semantics [3, 18]. Recent examples of StateChart-inspired user interface modeling approaches include *HsmTk* [4] for 2D direct manipulation interfaces, *CHASM* [27] for

3D user interfaces and *d.tools* [14] for physical prototyping. The *SwingStates* system extends the Java Swing user interface toolkit with concurrent statemachines [2].

At discussions during the IEEE VR 2008 SEARIS workshop (Software Engineering for Realtime Interactive Systems, [17]), participants acknowledged that the use of key aspects of multiple modeling techniques is a promising approach. We observe that this integration does exist in many approaches to some extent, but often lack an explicit separation between these two models. This quickly introduces complex *feature interaction*, which complicates application behavior analysis. Similar design issues were already addressed earlier in the specification of interactive systems. For example, a separation of *status* and *events* phenomena to model state and continuous relationships between interface components is used, for a recent implementation see Dix et al. [8]. An alternative approach to specify user interaction in the context of Virtual Environments is proposed by Smith et al. [19, 25]. They extend high-level Petri Nets with Data Flow, so-called *FlowNets*, and use a semi-formal notation to model discrete and continuous components. This work is requirements-centric as it aims to provide only a *sketch* of interaction to enable analysis of usability requirements.

In contrast, in our developer-centric approach we emphasize unification of model and architecture for many development iterations. The familiarity of developers with models and their mapping to underlying language and system structures is important. Therefore, we chose to mix state-based models and Data Flow to model event-based and continuous interaction. An early example of this model separation is shown in [5], where StateChart diagrams are expanded with Data Flow and constraint specifications to design custom user interface widgets. The use of this type of model separation in real-time interactive graphics systems is demonstrated in the HCSM driving simulator [7]. Jacob et al. transfer this approach to the description and programming of non-WIMP user interfaces [16, 24]. Our State-Stream system builds on a similar, dual-modeling approach where StateChart mechanisms and Data Flow are separate, first-class primitives. The main difference of our work with the systems above is the way the models are described and implemented.

2.2 Model Integration

Model-based techniques are of practical relevance if results can be effectively integrated with the underlying software architecture, such as a scene graph system. From a historical perspective, Myers [20] reports that both formal language based tools and model-based techniques for UIMS's suffered from a high *threshold* of acceptance. In many model descriptions, a specialized description language is compiled or interpreted to running code. For example, XML-based descriptions, sometimes augmented with native code snippets, are first converted to C-style code, then compiled and run, see e.g. [9, 26, 8, 28]. However, Carr already stated the potential of editing specifications and directly executing them [5].

We consider the semantics and syntax of the modeling language to be determining in this high threshold. First, declarative description model semantics may restrict expressiveness, especially with respect to the pure imperative coding practice on underlying architecture. Second, language syntax is often different. Third, once code is compiled and run, the relation between the running code and the mixed description and code is difficult to grasp. As a result, conceptually related elements may exist in different language semantics, syntax, files and scope, which make development and debugging difficult.

To alleviate some of these issues, we avoid the use of a specialized declarative model language. Instead, we explicitly describe interaction techniques using the constructs in Python, a dynamic, easy-to-learn programming language which unifies our framework. Zachmann proposed the use of a special-purpose scripting language on a fixed VR interaction API [29]. Early versions of *Alice toolkit* provided a Python-based scripting environment for interaction and low-level extensions to lower the learning barrier [6]. Although the use of a dynamic language by itself does not provide a ready solution, its flexible syntax and interpreter allow a clean integration of *pseudo-declarative* models. The usefulness of a similar integration approach is demonstrated in the *SwingStates* toolkit, where statemachines are described in native Java inner classes [2].

We chose Python because the dynamic language offers more flexibility, such as introspection, for building development tools. Also, the integration with a scene graph system builds upon earlier work on flexible abstraction layers of our base VR architecture [11]. The StateStream model is self-executing and operates at run-time in Python, allowing interaction techniques and scenarios to be studied and modified, often without restarting the application.

2.3 Development Environment

Finally, we consider the development environment and run-time system to be an essential element of interaction design software. Wingrave et al. [28] report on the complexity of interface development and argue the need for better, developer-centric methodology and tool support through the entire design and development cycle. Hendricks [15] highlights 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. Some of these ideas appear in the *d.tools* environment [14], which provides an integrated design, test and analysis environment. However, like *NiM-Mit* [26], d.tools is a strongly visually-oriented design tool from which code is generated and run. This visual approach generally does not scale well to larger systems and restricts low-level inspection or modification of application states and flow of control by more experienced designers and developers.

Our approach focuses on the low-level, fine-grained aspects of 3D interaction and detailed model behavior, and less on an Integrated Development Environment (IDE) for end users. Because of this low-level approach, the heart of our approach is an accessible, programming language-based interaction description. This description can be run directly, while components for analysis and visualization such as graphs, traces and lists for the front-end GUI communicate with the running model. As a result, the dynamic language serves as a unifying link for describing interactivity between objects, connect application components, and to produce front-end development and analysis tools. We feel this integration helps to lower the barriers between different stages of development from design, programming and analysis. With this approach, we attempt to reduce system

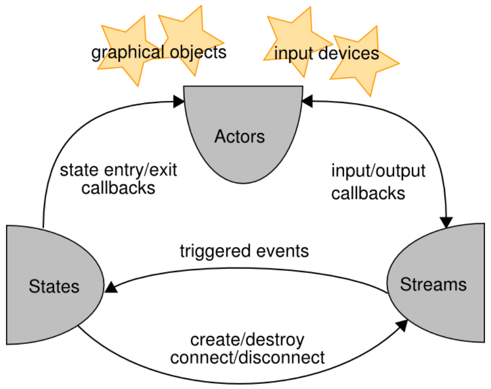


Figure 3: Three main StateStream components. Arrows indicate the main mechanisms through which components influence each other.

viscosity by providing higher *flexibility* and *expressiveness*, necessary to rapidly iterate to a better user interface system, as suggested by Olsen [21].

3. MODEL DESCRIPTION

In this section, an overview is given of the StateStream *model primitives* and how they integrate *domains* of behavior description. As described in the previous section, a dual modeling approach is used, similar to PMIW [16]. The first modeling primitive is the *statemachine*, a StateChart-like mechanism intended for describing behavior in the *discrete domain*. The second modeling primitive is the *streammachine*, intended for modeling conceptually continuous streams of information, thus in the *continuous domain*. A third domain is the *actor domain*, which essentially forms an interface to the underlying architecture and contains the statemachines and streammachines. Figure 3 gives an overview of the various components and the control mechanisms through which they influence each other. Before we describe these primitives in more detail, we first motivate the choice of description language for model primitives and application description, which is essential in our developer-centric approach.

3.1 Description Language

One key aspect of our approach is that the StateStream model primitives are described directly as class structures in a dynamic, interpreted language, i.e. Python in our case. This avoids the use of a specialized model description language, parser and compiler or interpreter. Object-oriented techniques such as class hierarchy and method overloading are well-known and extremely useful for composition and inheritance of StateStream model primitives. It allows a transparent communication between domains, as well as integration with functionality of the underlying Python application. Regular Python syntax is easy-to-read, and StateStream's syntax slightly extends this to have a descriptive rather than imperative appearance, although its semantics are not formally declarative, see Figure 4 and 5. In contrast to many other description languages, StateStream primitives are self-executing within a simple execution engine in a standard Python interpreter. This means they can be run and tested stand-alone, or be integrated within external Python programs, classes and libraries. The internal StateStream models and execution flow can be inspected and adapted at run-time to allow for dynamic model behavior. The main aspect of this is that new model primitives can be created, loaded and inserted during an application, which is essential when live prototyping applications or when the scene is not known in advance.

3.2 Actor Domain

Behavioral functionality is split into conceptual *actors*, each of which contain the two behavioral primitives as described above. In the context of a VR application, an actor often "is" or consists of a visible VR object and its behavior specification. For interaction techniques, actors include the graphical elements such as cursors, rays and information labels. Some actors do not have a graphical representation, but instead represent a proxy for example an external algorithm or interaction device. Although it is not technically restricted, actors preferably use streams and events instead of direct references to other StateStream modeled objects, nor do they contain behavior-specific logic. This is to prevent for unexpected feature interaction outside the model logic.

3.3 Discrete Domain

For describing discrete actor behavior, we use a simplistic *StateChart* variant, which consists of hierarchical, concurrent statemachines. A graphical representation, automatically generated from an instance of such a statemachine, is shown in Figure 6. In this Figure, each rounded box represents a single state. An hierarchical state can have children, which in turn can be a statemachine. Simple states allow only one child state to be active at the same time (red outlines), while concurrent states (grey, filled) have all their states active. Template states (yellow, filled) represent concurrent states that are dynamically generated and replicated. Each state can have transitions to another state. State transitions can occur if certain *events* match the conditions and filters of available transition. When a transition among hierarchies occurs, this can cause a cascade of state entries and exits. Custom functionality is defined in callback functions such as state entry and exit functions, or at transitions. These functions can be part of the actor, see Figure 4. For consistency, it is advisable to set properties or perform actions in a state *entry* function which can be reset or undone in the respective state *exit* function. A main functional element here is the broadcasting of events to other objects and to set state-sensitive properties of the related actor. In this domain, the flow of control of the actors and their states can be clearly modeled and visualized.

3.4 Continuous Domain

The *streammachine* primitive is intended for modeling conceptually continuous streams of information. For this we use a simple Data Flow graph structure, which consists of a set of connected nodes or filters with various input and output ports. The ports of the nodes can be connected through *connection* objects, over which information of various data types can be transported. The custom functionality of the nodes is again defined in callback functions on the incoming and outgoing ports. Simple Python syntax and constructions are used for creating a new streammachine class and connections, see Figure 5. A part of a Data Flow net-

```
#Actor reacts to "Touch" and "unTouch" events
class TouchyActor(ssActor):
    def initActor(self):
        #initialize actor, states, streams
        ssActor.initActor (self)
        self.st_NotTouching.addTransition(self.st_Touching,
                                           "Touch")
        self.st_Touching.addTransition(self.st_NotTouching,
                                        "unTouch")
        self.conn = StreamConnection()
    def st_NotTouching(self):
        #NotTouching State will be created,this is enterfunc
        self.conn.disconnect()
    def st_Touching(self):
        #connect stream to guiPrinter, broadcast event
        self.conn = self.portout_position >> guiPrinter
        addSSEvent("Hurts",self)
    def portout_position(self):
        #stream callback
        return self.getPosition()
```

Figure 4: Sample Python code for an actor containing states, transitions and connections. With simple syntax conventions and introspection we can create objects implicitly for states, ports, connections etc. at run-time.

```
#overloaded streammachine for widget connection
class StreamGTKValue(StreamMachine):
    def __init__(self,pwidget):
        StreamMachine.__init__(self)
        self.mwidget = pwidget
        #create an explicit port
        self.mPort_in = self.createInputPort \
                    ("IN",self.updateIn)
    def updateIn(self,pValue):
        self.mwidget.set_value(pValue)
    def portout_myvalue(self):
        #implicit output port "myvalue" from function name
        return self.mwidget.get_value()

#connect mySlider widget to a scaling function
StreamGTKValue(mySlider) >> sceneObject.setScale
```

Figure 5: Sample code for making a streammachine class for a GTK widgets. We use the $>>$ operator to connect streams to ports or directly to variables or functions.

work is shown in Figure 6, this is generated by introspecting run-time objects. Continuous variables of an actor and its graphical objects, e.g. position, rotation, color or size are easily modeled and connected through filters. In a standard VR scenario, updates are typically executed every render frame and values are pushed through the network, but more advanced and optimized update strategies on the graph are possible.

One can clearly see how actor properties are related through explicit connections, when in a certain state of an application. This separation of concerns allows one to better reason on intended behavior and analyze its implementation. Although the streams are conceptually continuous, in implementations they are sampled and separated in function calls or events. The separation of the discrete domain also has an important practical implication. We do not *pollute* the discrete event system with often-called "update" events, which do not contain any state information. Especially when inspecting event streams in a VR application with a rendering frame rate of 60Hz and more, the amount of event information would be simply overwhelming.

3.5 Integration

A StateStream-based application consists of the description of the actors, their statemachines and streammachines. For a StateStream application, a main, top-level statemachine is maintained which reflects the application state. On application start-up, main actors are instantiated, which in turn can activate their own actors. During actor initialization, both its statemachine and streammachine structures are created. All statemachines existing in the applications are attached as (concurrent) child statemachines of the application statemachine, or descendants thereof. The framework maintains a global *event broker* for event queuing and broadcasting. Event objects can contain various cargo, and are injected in the active statemachines by the event broker component. A *stream broker* component is in charge of the maintenance of the stream graph. It sorts the acyclic Data Flow graph of streammachines, and pushes values through the active connections of the graph.

The discrete domain can influence the continuous by requesting the creation, connection, disabling etc. of streammachines and connections in the stream graph. The continuous domain can influence the discrete domain by broadcasting events to the event broker. This is done through specialized streammachines, e.g. triggers, which generate an event based on the value of incoming streams.

Naturally, functionality in both domains can influence the actor domain through their respective callbacks. This is often necessary, for example to obtain events and streams from the underlying architecture into the StateStream domains, or to reflect states and values through actor properties, e.g. position or visibility status. In section 5, we will use several interaction examples to demonstrate these relations.

4. STATESTREAM PROTOTYPE

In this section, we demonstrate how the StateStream integrates interaction modeling with the components of our VR architecture.

4.1 Base Architecture

StateStream integrates on a Python level with our in-house VR toolkit *VRMeer*, which mainly builds upon the C++ *OpenSceneGraph* library. We use *SWIG* to create Python bindings for both OpenSceneGraph 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. A run-time development front-end interface is provided through *iPython* interactive Python shell, optionally integrated with the *PyGTK* GUI toolkit. A more detailed overview of abstraction layers is given in [11], and for the VRMeer architecture with front-end GUI see [10]. Running on top of the base abstraction layers, our proposed StateStream interaction model integrates interaction modeling with other system functionality.

4.2 StateStream integration

In our current implementation, a Python interpreter controls the execution of a StateStream enabled VR application. The VR system's main loop interleaves the distribution of the event to statemachines through the event broker, the execution of the streammachine graph through the stream broker, the scene-graph update and the update of an optional GUI. Events can be identified by their name and often accompanied by a cargo. This cargo can denote the destination object(s) and possibly extra parameters. Statemachines and their transitions can use a variety of filters to determine if an event should be processed. For each rendering frame the event broker's queue is processed until it is empty. Cycles

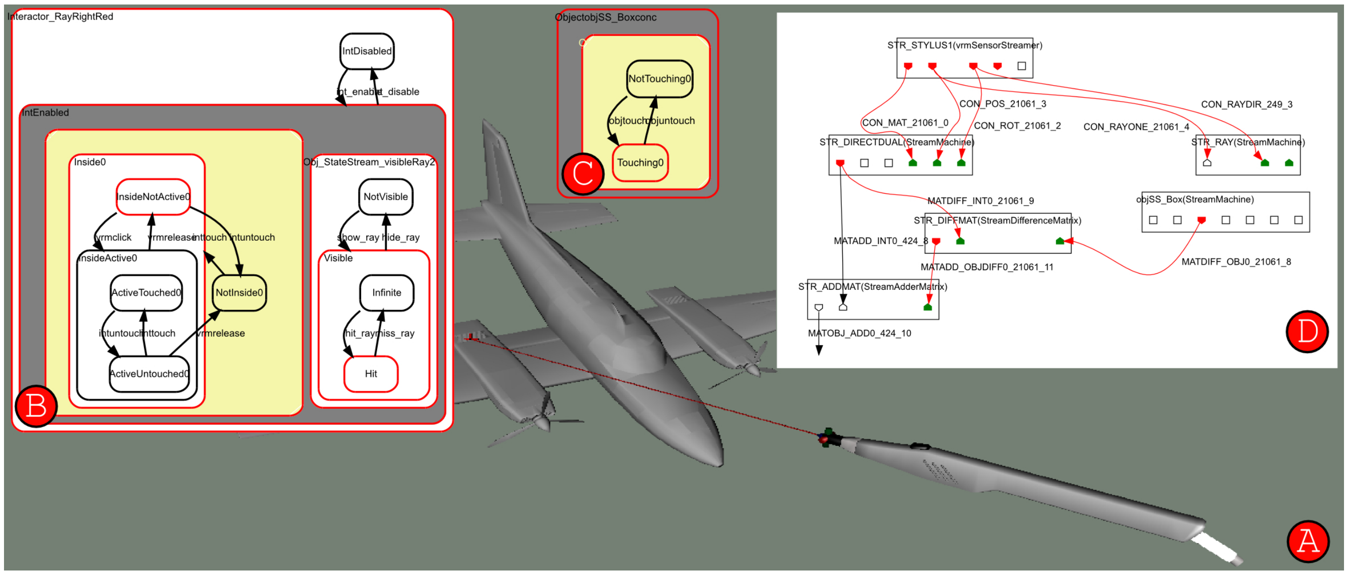


Figure 6: Composition of a StateStream modelled 3D ray casting selection technique, see also section 5.1. A stylus device can be used to point a ray at objects in the VR scene (a). Details of the modeling primitives can be found in section 3. Generated graphical StateChart representations for the statemachines are shown for both the selection technique (b) actor and the plane (c) actor. The statemachine of the ray actor is contained in (b, right sub-state). The related Data Flow graph (d) displays the current streammachines, their ports and connections that are in use for this state.

in event execution can be detected and avoided.

A set of basic system actors can convert events and variables of underlying scene graph or tracking libraries to abstract, StateStream compatible events and streams. By performing this conversion at a low level, the fine-grained details of interactivity can already be flexibly modeled, composed and inspected through their explicit models. As a result, behavior is modeled *orthogonal* to system abstraction layers, so components in virtually all layers can be used from within control of StateStream primitives. An example of this is the conversion of button presses and pose of a 3D input device, where button presses are converted to events, while positions and rotations are made available through streammachine nodes. Other system actors *monitor* streams to perform, for example, an INSIDE test on a set of VR objects and trigger a TOUCH event as a result.

4.3 Front-End Interface

Concurrently with the running VR application, a graphical interface for interactive debugging and development environment is available. It provides an interactive Python shell and visual elements that reflect the internals of the VR application, see Figure 7. The availability of the front-end on a separate display or remote computer enables live debugging sessions, which is especially useful for immersive, tracked VR applications. We currently include visual widgets such as lists, trees and graphs that reflect current application and StateStream state and relations. New application-specific widget elements can be created through code and existing layout designers, and can be loaded and activated at runtime. A powerful approach is to extend widgets to be StateStream actors as well, in order to integrate some of their logic and behavior in the application. For example, a group of text widgets can show the values of an incoming stream connection. We envision that interaction techniques can be made available with GUI panels and a programming API to configure and tune their use.

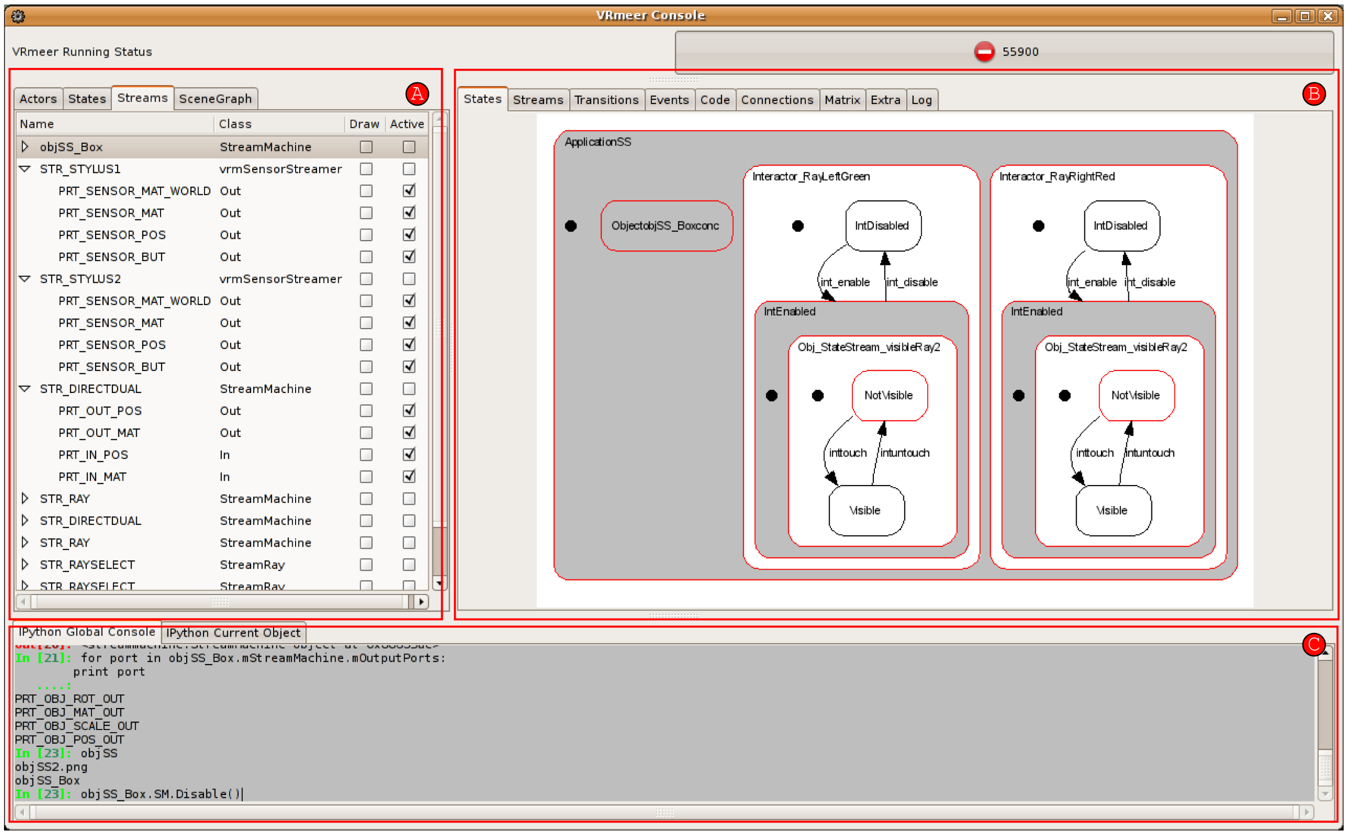
5. RESULTS

In this section, we will give an overview of some of the resulting interaction techniques we have obtained with our system and discuss the process of designing them. We will first illustrate a straightforward VR selection technique to explain the basic working of the system. This description is deliberately kept at a low level, so it will be clear how the flow of execution is. At the same time, this shows that even for a toy interaction technique a textual description can sometimes become quite elaborate, see also [28]. We will then continue and gradually extend them to more elaborate techniques, where combinations of states and streams soon become overwhelming. Also keep in mind, that some simple constructions in StateStream appear more intrusive than direct imperative coding would be for the sake of extension, replacement and reuse in other techniques.

5.1 Selection and Manipulation

In direct object selection, a 6DOF stylus is used as a straightforward selection tool of a single VR object. If the tip of the stylus is inside, or touches, the bounding box of a VR object, it should be selected and highlighted. We will gradually work towards a ray casting selection and manipulation technique, so some of the elements described here can be found in the more complex Figure 6.

The direct selection actor class is sub-classed from a base *interactor* actor, which represent the display of a cursor icon. Its base statemachine consists of two states ENABLED and DISABLED, and two transitions to switch between them if a INT_ENABLED or INT_DISABLED event is received. In addi-



Name	Class	Draw	Active
objSS_Box	StreamMachine	☐	☐
STR_STYLUS1	vrmSensorStreamer	☐	☐
PRT_SENSOR_MAT_WORLD	Out	☐	☑
PRT_SENSOR_MAT	Out	☐	☑
PRT_SENSOR_POS	Out	☐	☑
PRT_SENSOR_BUT	Out	☐	☑
STR_STYLUS2	vrmSensorStreamer	☐	☐
PRT_SENSOR_MAT_WORLD	Out	☐	☑
PRT_SENSOR_MAT	Out	☐	☑
PRT_SENSOR_POS	Out	☐	☑
PRT_SENSOR_BUT	Out	☐	☑
STR_DIRECTDUAL	StreamMachine	☐	☐
PRT_OUT_POS	Out	☐	☑
PRT_OUT_MAT	Out	☐	☑
PRT_IN_POS	In	☐	☑
PRT_IN_MAT	In	☐	☑
STR_RAY	StreamMachine	☐	☐
STR_DIRECTDUAL	StreamMachine	☐	☐
STR_RAY	StreamMachine	☐	☐
STR_RAYSELECT	StreamRay	☐	☐
STR_RAYSELECT	StreamRay	☐	☐

```
In [21]: for port in objSS_Box.mStreamMachine.mOutputPorts:
       print port
   ....:
PRT_OBJ_ROT_OUT
PRT_OBJ_MAT_OUT
PRT_OBJ_SCALE_OUT
PRT_OBJ_POS_OUT
In [23]: objSS
objSS2.png
objSS_Box
In [23]: objSS_Box.SM.Disable()
```

Figure 7: Screenshot of a typical layout of the StateStream front-end GUI. Visual widgets such as lists, trees (a) and graphs (b) are grouped in two main panels, and dynamically reflect StateStream state and relations of the current running VR application. New interaction tools and widget elements can be loaded and activated at run-time from the interactive Python shell (c).

tion, the streammachine provides the connections to position and rotation streams of an interaction device to update the cursor. The ENABLED state of the base statemachine is extended with an INSIDE and NOTINSIDE child state for selection. Two transitions are added: TOUCH to go from NOT_INSIDE to INSIDE, and UNTOUCH to go from INSIDE to NOT_INSIDE.

A simple VR object is also modeled as a StateStream actor. Its statemachine consists of a TOUCHED and NOT_TOUCHED state, with transitions OBJTOUCH and OBJUNTOUCH, see Figure 6. The streammachine contains several ports for both incoming and outgoing streams for position, rotation etc.

If in the direct selection actor an incoming TOUCH event is received from a system actor, the state is changed to INSIDE. In turn, in the state entering function of the INSIDE state, the selected object is sent a OBJTOUCH event. In the NOT_TOUCHED state entering function of the VR object, a highlighting function can be called to change its appearance. Note that in this toy example, we see mainly state-based communication between object and interaction actor.

5.1.1 Ray Casting Extension

To extend the previous example by allowing remote selection with a ray, we have to make a number of changes. First, a ray actor is created that serves to visually represent the indicated direction of the stylus. We create the statemachine of the ray actor in the ENABLED state of the interaction actor, and it contains states to indicate visibility and RAY_HIT or RAY_MISS, see Figure 6(b, right sub-state). In the state exit and entry functions of the interactor INSIDE and NOTINSIDE child states, visibility events for the ray are generated.

Second, the ray's streammachine is connected to the interactor's streammachine to communicate starting position and direction. In the value update function of the ray's streammachine, these values are used to update the visual ray object. Third, a system actor is activated that performs the actual ray casting hit algorithm with scene objects. This actor generates TOUCH and NOT_TOUCHED events, but additionally activates a continuous stream for the position of intersection points. The interactor's INSIDE state is extended to create the connection of the intersection point streammachine to an end point port in the ray's streammachine, see Figure 6(d). The ray actor itself is responsible for its continuous visual updating. In the RAY_HIT state, we choose to use the end point instead of the direction for drawing. In this example, one observes a clear interplay between discrete, state-based communication but also continuous updating of values through streams.

5.1.2 Object Manipulation

If the stylus button is pressed during selection, one should be able to manipulate it. Once one is manipulating an ob-

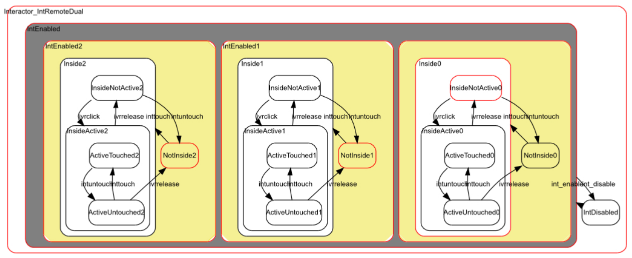


Figure 8: A StateChart diagram of a multiple object selection actor. For two recently selected objects and a currently selected object, an instance of a template statemachine was generated. These keep the object-interaction relation consistent.

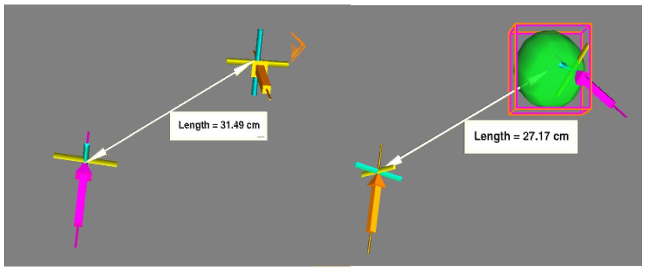


Figure 9: Snap Measurement technique. A measurement actor can be controlled by two 3D cursors (left) or the connection of one endpoint snaps to the object by starting a measurement inside the object (right). One cursor concurrently manipulates the object, while the measurement actor remains consistent.

ject, the new object position and rotation are calculated based on the original stylus pose just before the start of the manipulation, and the movement of the stylus. Apart from state changes, object manipulation therefore also requires several continuous communication streams between the object and interaction actors. This is achieved by creating streammachine that, in the TOUCHED state, connects to the stylus pose and to the object pose, and calculate the difference. As soon as the stylus button is clicked to start manipulation, this difference stream is connected to another streammachine that adds it to the new stylus pose stream. The output of this adding streammachine is the new resulting pose of the object. This output is connected to the object's streammachine, and it can use this to update its position.

In this example, one sees a carefully orchestrated combination of discrete and continuous mechanisms. Also we show the use of streammachines as functional filters to combine several streams.

5.2 Multiple Object Selection

The previous examples work on a single "object to interactor" relation. If the scene contains multiple objects, for example, an interaction technique might be required to manipulate more than one of them simultaneously.

To ensure correct handling, this requires that the interaction actors maintain a separate set of states and streams for each object that it is involved with. To avoid modeling in advance of all possible relations, we can dynamically instantiate a *templated* actor. Each templated child instance reflects a different "object to interactor" relation. As a result, we can extend an existing selection interactor to be used on multiple objects.

To achieve this, events reaching the interactor are intercepted and, if necessary, a new template is instantiated. The template actor creates both a new child statemachine and new streammachines, including all transitions and connections. A templated state of the actor is indicated in Figure 8 in yellow. The originally modeled state changes and stream connections still work as advertised without changes. This property is important for extending techniques to work with multiple actors while maintaining overall consistency. In this example, the templated actors have no interaction with other instances. The dynamic instantiation of extra model compontents is made possible by the dynamic features of Python. We are not aware of other systems where similar extension of existing techniques can be done without requiring much manual recoding and bookkeeping.

5.3 Snap Measurements

This third example demonstrates how an interaction technique can transfer its continuous operation to objects. The interactive element here is a visual measurement tape that can measure distances between arbitrary positions in space or between objects, see Figure 9. With one or two styli, a measurement tape can be drawn. In a regular, empty scene, the measurement actor is drawn from stylus tip to the other stylus tip, or from the last clicked point to stylus tip. A panel is positioned and its text updated. When the stylus enters an object however, it interprets the user action as wanting to measure from this object and not in empty space. The stream connection to the stylus interactor is not directly used for drawing anymore, but the object position is used instead. In such a way, a measurement tape actor can be created that measures position between two objects. The advantage here is that the objects can still be moved around while the measurement tape automatically updates. This demonstrates the transparency of the actors, e.g. that the original relation of interactor-to-object can be replaced by object-to-object. This relation is often fixed in other interface implementations. It also gives insight in how streaming can be used to generate constraints dynamically, while the Data Flow graph is kept consistent.

5.4 Two-Handed Scaling

The final example combines the earlier examples into a two-handed object scaling technique, similar to the popular two-finger scale technique on multi-touch displays. It extends interaction demonstrated earlier, but uses a tri-fold relation between an object and two interactors. The core modeling technique used here is state ordering. Two interactors are individually used as regular selection and manipulation tools. As described, manipulation updates are communicated through stream connections to the object. When the two manipulate the same object, a "fight" for preference might occur, as they would normally have parallel state machines and connections as generated from the template. In the object, we want to redirect and combine incoming connections to different properties of the object, e.g. depending on the order of incoming styli. For scaling, the changing distance between the two points of interaction determines the scaling factor of the object, where the first stylus determines the pivot point. Individually, the interaction technique is hand-symmetrical, but it can change to asymmetrical depending on the scaling mode. Through the use of template ordering, the functionality of each object-interactor relation may change. Although this example illustrate the complex interaction between concurrent sub states, for clarity a separate actor to control state ordering might be more comprehensive.

5.5 Development Use

We observe that the separation of concerns from the actor, discrete and continuous domain serves as a guide already in an early design phase. Developer discussions tend to focus more on how to get into a certain state, and what connections to ensure in that state. Through containment of difficult situations, overall understanding is enhanced. Also, the development of actor appearance and behavior gets implemented and tested as self-containing components, before they are integrated with other actors. The interactive front-end assists in quick prototyping on top of existing actors, by loading, activating and stepping through situations while observing and adding connections.

A wide array of streammachines is being constructed to provide flexible combinations of streams. In that sense, a bottom-up approach is used to provide a wide platform of tools. Because of the hierarchy in states, also a top-down approach is used. By working from a basic state and gradually specifying new sub states, one can postpone new detailed behavior specification. From use by different developers, several design patterns emerge with a focus on re-use. To improve composition and reuse, it is preferred for actors to maintain as much state as possible, and to consume and replicate streams instead of just connecting external objects. Implementation difficulties mainly arise in conversion of external components. For example, concepts of event systems may differ, such that the absence of an event can be considered an important "idle" event in a different system.

6. DISCUSSION

In this section, we will shortly discuss results and limitations of the current model and its integration in the architecture. We are aware that, for a good understanding of the strengths and weaknesses of current approach, we require a long-term evaluation of the development of complex interaction techniques in real-life applications, on different types of interactive systems. Similar to SwingStates [2], an evaluation where students implement existing techniques is part of future work. It must be stated that the current model and architecture have already evolved over the last year as a result of iterative application development experience.

First, we currently use basic StateCharts and Data Flow mechanisms to serve as a proof of concept. More elaborate variants of these might be used in the future, such as different StateChart approaches [3]. Second, our practical focus is on prototyping: model integration, the avoidance of a pure declarative description and dynamic model changes. Although simple consistency checks can be made on the models, real formal verification and operational semantics are more difficult, especially for dynamically changing models. The model is executed from within a Python interpreter, so off-line verification for full applications will involve complex parsing of Python code trees. Third, some advanced model and introspection properties are currently restricted to dynamic features of the Python interpreter. The use of Groovy[1], a dynamic programming extension for Java, is an interesting alternative. Fourth, good system and model integration results from running a single Python interpreter with a single event broker and stream broker. Performance-wise this is sufficient for most applications and we are investigating optimization and distribution. For systems or stream components that are critical in latency and performance one would have to resort to advanced multi-threading schemes. Finally, because of our developer-centric, language-based approach we currently do not have a unified graphical model representation which can be edited graphically from a full-fledged IDE. Although we consider this essential for a transfer to a larger audience, we see this as a substantial amount of system engineering to be included in a later stage.

7. CONCLUSIONS AND FUTURE WORK

In this paper we described StateStream approach towards unifying the integration of model, architecture and tools for the development of 3D interaction techniques through a dynamic language. The dynamic, interpretative nature of the language provides familiar syntax, and allows us to employ flexible communication between model domains and to introduce novel modeling features, for example the generation and activation of templated behavior descriptions at run-time. We demonstrate the language-based approach on a dualistic model, in an effort to provide two primitives that fit specific discrete and continuous domains of interactive behavior. In this way, we provide structure and separate concerns when designing or studying complex interaction between actors. As demonstrated in several examples, this interplay between the two main model domains and the actor domain is suitable to define complex behavior, but can be extended with the dynamic language. The observation we want to make here is that in practice, aside from toy examples, no single, purely model-based approach can be expected to be complete and elegant. That is, it is inevitable to encounter situations that cannot be modeled within the current model specification, or require a more complex construction pattern. The use of a dynamic programming language and dynamic adaptation of models at run-time can create complex interaction constructions. However, to maintain an overview, custom tools can be integrated to selectively analyzed and visualized their hierarchical structure and communication. With this in mind, we feel our iterative integration approach of interaction models, its underlying architecture and development tool sets is a useful asset towards bridging the gap between models and practical use.

The StateStream system, model, syntax and tools are continuously refined. Naturally, we strive to enlarge the set of available actors and front-end tool widgets to enhance developer productivity. From a research perspective, we aim at more understanding of the interaction model to allow reasoning on actor relations. We believe that this understanding is needed for extending the current state-of-the-art with optimization and distribution over multiple processes or machines. Finally, we plan to investigate alternative visualization techniques to enable better insight and evaluation

[1]http://groovy.codehaus.org/

of interaction techniques over time.

8. ACKNOWLEDGEMENTS

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

9. REFERENCES

[1] J. Allard, C. Ménier, E. Boyer, and B. Raffin. Running Large VR Applications on a PC Cluster: the FlowVR Experience. In *Proc. of EGVE/IPT '05*, pages 59–68, 2005.

[2] Caroline Appert and Michel Beaudouin-Lafon. SwingStates: adding state machines to the swing toolkit. In *Proc. of UIST '06*, pages 319–322, 2006.

[3] Michael von der Beeck. A Comparison of Statecharts Variants. In *Proc. of ProCoS '94*, pages 128–148, 1994.

[4] Renaud Blanch and Michel Beaudouin-Lafon. Programming rich interactions using the hierarchical state machine toolkit. In *Proc. AVI '06*, pages 51–58, 2006.

[5] David A. Carr. Specification of interface interaction objects. In *Proc. of CHI '94*, pages 372–378, 1994.

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

[7] James Cremer, Joseph Kearney, and Yiannis Papelis. HCSM: a framework for behavior and scenario control in virtual environments. *ACM Trans. Model. Comput. Simul.*, 5(3):242–267, 1995.

[8] Alan Dix, Jair Leite, and Adrian Friday. XSED — XML-Based Description of Status—Event Components and Systems. In *Proc. of EIS 2007*, pages 210–226, 2008.

[9] Pablo Figueroa, Mark Green, and H. James Hoover. InTml: a description language for VR applications. In *Proc. of Web3D '02*, pages 53–58, 2002.

[10] G. de Haan and F. H. Post. Flexible Architecture for the Development of Realtime Interaction Behavior. In *Proc. of SEARIS '08*, pages 71–75, 2008.

[11] Gerwin de Haan, Michal Koutek, and Frits H. Post. Flexible Abstraction Layers for VR application development. In *Proceedings of IEEE Virtual Reality 2007*, pages 239–242, 2007.

[12] Gerwin de Haan, René Molenaar, Michal Koutek, and Frits H. Post. Consistent Viewing and Interaction for Multiple Users in Projection-Based VR Systems. *Computer Graphics Forum*, 26(3):695–704, 2007.

[13] David Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, 1987.

[14] 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. *Proc. of UIST '06*, pages 299–308, 2006.

[15] Zayd Hendricks, Gary Marsden, and Edwin Blake. A meta-authoring tool for specifying interactions in virtual reality environments. In *Proc. of AFRIGRAPH '03*, pages 171–180, 2003.

[16] R. J. K. Jacob, L. Deligiannidis, and S. Morrison. A software model and specification language for non-WIMP user interfaces. *Transactions on Computer-Human Interaction*, 6(1):1–46, 1999.

[17] Marc Eric Latoschik, Dirk Reiners, Roland Blach, Pablo Figueroa, and Raimund Dachselt. *IEEE VR 2008 Workshop on Software Engineering and Architectures for Realtime Interactive Systems (SEARIS)*. Shaker Verlag, Aachen, Germany, 2008.

[18] Gerald Lüttgen, Michael von der Beeck, and Rance Cleaveland. A compositional approach to statecharts semantics. *SIGSOFT Softw. Eng. Notes*, 25(6):120–129, 2000.

[19] Mieke Massink, David J. Duke, and Shamus P. Smith. Towards Hybrid Interface Specifications for Virtual Environments. In *Proc. of DSV-IS*, pages 30–51, 1999.

[20] Brad Myers, Scott E. Hudson, and Randy Pausch. Past, present, and future of user interface software tools. *ACM Transactions Computer-Human Interaction*, 7(1):3–28, 2000.

[21] Dan R. Olsen. Evaluating user interface systems research. In *Proc. of UIST '07*, pages 251–258, 2007.

[22] Alex Olwal and Steven Feiner. Unit: modular development of distributed interaction techniques for highly interactive user interfaces. In *Proc. of GRAPHITE '04*, pages 131–138, 2004.

[23] Andrew Ray and Doug A. Bowman. Towards a system for reusable 3D interaction techniques. In *Proc. of VRST '07*, pages 187–190, 2007.

[24] O. Shaer and R. J. K. Jacob. Toward a Software Model and a Specification Language for Next-Generation User Interfaces. *CHI '05 Workshop on The Future of User Interface Software Tools*, 2005.

[25] Shamus P. Smith. Exploring the Specification of Haptic Interaction. In *Interactive Systems. Design, Specification, and Verification*, volume 4323 of *Lecture Notes in Computer Science*, pages 171–184, 2007.

[26] Davy Vanacken, Joan De Boeck, Chris Raymaekers, and Karin Coninx. NIMMIT: A notation for modeling multimodal interaction techniques. In *Proc. of GRAPP*, pages 224–231, 2006.

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

[28] C. Wingrave and D. Bowman. Tiered Developer-Centric Representations for 3D Interfaces: Concept-Oriented Design in Chasm. In *Proc. of IEEE VR '08*, pages 193–200, 2008.

[29] G. Zachmann. A language for describing behavior of and interaction with virtual worlds. In *Proc. of VRST '96*, pages 143–150, 1996.