An Implemented Approach for a Visual Programming Environment in VR

Peter Biermann, Ipke Wachsmuth
Laboratory for Artificial Intelligence and Virtual Reality
University of Bielefeld
http://www.techfak.uni-bielefeld.de/ags/wbski/

Abstract: In this paper we describe a visual programming environment which consists of computing nodes that are described in an XML notation and can be interactively configured and wired. The dataflow is realized via field connections, which are implemented in the AVANGO toolkit. The connections and parameters of the computing nodes can be modified during execution via the Scheme scripting language.
The computation of the networks is embedded in the render-loop of the VR application, but frame-rate independent computation can also be done using attribute sequences as field values.
The nodes can act as scene-graph nodes, which makes it very easy to visualize the nodes itself or the output of a node in the VR environment.
Keywords: *Visual Programming, VR Applications, Scene-Graph manipulation, XML description, Scripting*

1. Introduction

Visual environments for programming languages have become popular in many applications and especially for computing on dataflows [Hil92]. In VR applications visual programming environments could be a great help when used while processing for example the input of user devices. Most input devices, like 6 DOF-trackers and data-gloves in VR applications produce continuous data-streams and therefore can be ideally processed by a dataflow oriented programming language.
The approach presented in this paper combines a scene-graph oriented and render-loop dependent program style with a visual dataflow programming paradigm. This is achieved by using field-containers of the SGI-Performer based AVANGO toolkit [Tra99]. Since these containers can be instantiated as nodes of the Performer scene-graph, they are ideal for manipulating scene-graph structures and can easily generate geometry for visualization in the virtual environment. More complex nodes can also replace the normal Performer rendering with their own rendering technique.
The field-containers were extended to gain general purpose computing nodes that can process data-streams independent of the frame-rate of the VR application. For optimal performance, which is inevitable for real-time VR applications, the nodes are written in C++. Small computing nodes provide a very fail-safe and stable way of coding, because small code parts of the nodes can be easily tested and verified.
To keep the node library flexible and easily extendable, the nodes are described in an XML style and are then translated to C++ code by using an XSLT (Extensible Stylesheet Transformations [Dou2001]) translation.
Most parameters of the compute nodes are also accessible via field values and can be altered while the program is running by using a Scheme (see [Dyb96]) scripting interface. This allows a direct feedback of the effects on the application. For more complex control of the application the user can define self-chosen methods for the compute nodes that can also be called via the Scheme interface.
New nodes and field connections can be added in the scheme interface, so that whole programs can be dynamically altered and extended – even while running.
The compute networks can be visualized in the virtual environment to give the user a direct access to the program structure while testing the program, for example in a Cave.

2. Attribute Sequences

The program code of the single nodes is executed once per render-frame, but most input devices in VR applications produce data in a higher time resolution (60-120 Hz) and detectors e.g. for natural gesture need these high data-rates for precise results. To realize computation on a data-stream independent of the frame rate a data-structure had to be developed that contains all data produced by the input devices. These data-structures are called Attribute Sequences (AS) and are based on a framework for multimodal interaction: PrOSA (Patterns on Sequences of Attributes) as presented in [Lat2001]. This framework describes the PrOSA concepts as fundamental building blocks for multimodal interaction in VR. It is focused on gesture and speech recognition and multimodal integration and interpretation. Within this framework attribute sequences are mainly used for the gesture processing, consisting of gesture-detectors and manipulators that affect the scene-graph structures.

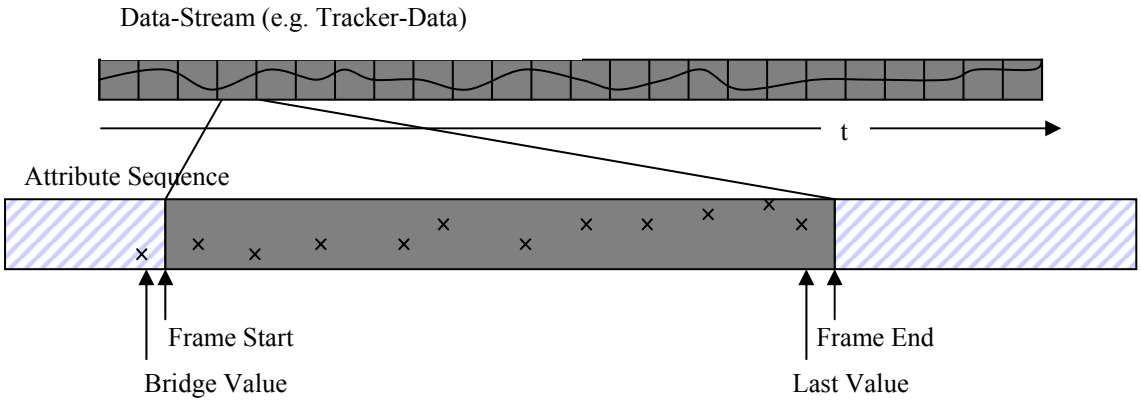

Fig. 1: An attribute sequence with multiple time-stamped values for one render frame.

An AS contains all data, which arose during the last render frame. The data is stored as time-value pairs to maintain the timing information, especially for sources that do not produce time-equidistant data. Each AS also holds the last value of the previous frame to bridge the gap between two sequences.
Attribute sequences provide methods to compute a linear interpolation for queries of arbitrary times in the time span they cover. Valid query times for an AS lie between the time of the bridge value and their last value (see Fig.1).
Two or more ASs can be combined in channels, which provide an interface for the resulting valid time for querying values. Since in general the times of the values of two ASs differ, it is necessary for two or more ASs in one channel to have a master clock, which can be a fixed rate or be given by a master AS.

3. Compute Nodes

The compute nodes are realized within the AVANGO toolkit, developed by the GMD. The AVANGO library is an object oriented framework for distributed, interactive VE applications, which extends the SGI-Performer library among other things with a field concept and a scripting interface. Fields act as an interface for various types of data in AVANGO nodes. Field values can be queried and set by the Scheme interface and field connections can also be established.
This allows the creation of a dataflow graph orthogonal to the scene-graph. Field connections automatically transfer the value from the sender to the receiver field once per render frame.

3.1. Field container in AVANGO

All compute nodes are derived from AVANGO field containers, which provide input and output fields that can be connected for exchanging data. The evaluation-method in these containers, which does the computation of the field values, is executed once for each render-frame if one of the input field has changed its value. This allows a sort of lazy evaluation, so that the computation is halted as long as no new values are provided. The resulting changes of field values are propagated to connected containers, so that the complete path of the data-stream is computed in one render frame.

3.2. XML Description

Often programs can not be created by the connection of a set of predefined nodes. Many visual programming languages offer a fixed set of building blocks, and are difficult to extend. In our approach we tried to generate a very flexible tool, where new computing nodes can be generated and added very fast@.
Since the Field Container in AVANGO have a much wider purpose than building compute nodes, an XML description of the functionality of compute nodes was developed to simplify the creation of new nodes.
With this XML description it is possible to:

- define input and output fields of the container
- provide the C++ code for computing the resulting field-values
- define Scheme callbacks for complex and interactive queries, independent of the render-loop

```
< Container Name=„prBoolAnd“ ParentClass=„fpDCS“ >
  < InFields >
    < Field Name=„In1“ Type=„bool“ Init=„true“
      Mode=„F,AS“ />
    < Field Name=„In2“ Type=„bool“ Init=„true“
      Mode=„F,AS“ />
  </ InFields >
  < OutFields >
    < Field Name=„Out“ Type=„bool“ Init=„true“
      Mode=„F,AS“ />
  </ OutFields>
  < compute > <![CDATA[
    bool result = In1_Value && In2_Value;
    set_Out (result);
  ]]> </ compute >
</ Container >
```

Fig. 2: An XML description of a simple compute node for a Boolean operation.

```
< Container Name=„prBoolHistory“ .....

  .....
  < SchemeCallbacks >
    < Callback Name=„add-entry“ >
      < arg Name=„Entry“ Type=„bool“ />
      < arg Name=„Time“ Type=„double“
        Init=„pfGetTime()“/>
      < code > <![CDATA[
        return fp_scheme_bundle
          (self->Hist.addEntry(Entry,Time,ToTime);
      ]]> </ code >
    </ Callback >

    .....
  </ SchemeCallbacks >
</ Container >
```

Fig. 3: A sample fragment of a XML description defining a Scheme callback

The example in Fig. 2 shows a simple container with two input and one output field, which are of type bool and can all be accessed as (one-value) fields or as attribute sequences. The fields, which support ASs are named with the Suffix '_AS' to distinguish them from fields that only hold single values. In this example all field entries in the XML description result in two fields: one supporting Boolean values and another one supporting ASs of Booleans.

In the compute-callback an AND operation of the two input fields is evaluated and the output field is set to the resulting value.

The 'compute'-tag contains C++ code that is inserted in the compute-callback of the AVANGO field containers when the description is translated.

The variables In1_Value and In2_Value are defined by the programming environment and are automatically set to the actual input values of the corresponding field. The output function set_Out() is also provided by the environment and can be used to set the value of the Out field directly in case of simple fields with one-value per frame.

The Scheme callbacks are also defined in the XML file with the description of their arguments, the code, and the return value.

The example in Fig. 3 shows the definition of a callback to add a new entry in a container that holds a history of time-stamped values. The first argument of the callback is the value of the entry and the second argument defines the time of the value. The second argument is optional, because a default value – the current time as given by an internal timer – is provided.

To build the resulting C++ library with the nodes resulting from the XML descriptions, the XML code is translated to C++ headers and code files with an XSLT translator. The resulting code contains the definitions for the fields and Scheme callbacks, and the environment for the computation and field operations. The compiled library can be dynamically linked to an existing AVANGO application.

3.3. Computing with Attribute Sequences

If one of the input fields is filled with an attribute sequence, the code in the compute callback is not used to fill the output field(s) directly. Instead an AS is filled with values for the output fields which support ASs as values. Therefore the compute callback is not only called once for each frame, but is evaluated several times according to the time span of the input ASs. This functionality is provided by the programming environment, so that the user does not have to care about that.

In case of the AS-computing the set_... function for the output field is used to add a new, time-stamped value to the AS of the according output field. When computing the last value for the AS the AS is assigned to the output field and the single value field is also set to this most current value.

Internally all input ASs are added to an input channel, which provides an interface for the time points, which are accessible for all input ASs. Two modes are selectable for calculating the times: In the *fixed-rate mode* all ASs are queried in equidistant time periods, ignoring the times in the single ASs; in *AS-master* mode one AS serves as master-beat and the values are queried at the time spots of the values of this master-AS.

All values which are not queried at the concrete times of the values in the AS – generally for all AS which are not master – are computed using linear interpolation between the two nearest values.

Fore each time point the ..._Value variables are set to the (interpolated) value of the according AS at that time and the compute callback is called. If some of the input fields are only filled with single values the variables are set to that value for all computations within the current frame.

3.4. Visualization

Almost all field types can be visualized by special visualization nodes. Since all AVANGO field containers can be instantiated as dynamic coordinate systems and therefore be added to the existing scene-graph of the VR application, it is easy to generate nodes which generate their own geometry to visualize field-values. This visualization is shown directly in the virtual environment and is ideal for testing and debugging new compute networks, which work on user-input from tracking-devices or data-gloves, directly while interacting in the VR scene.

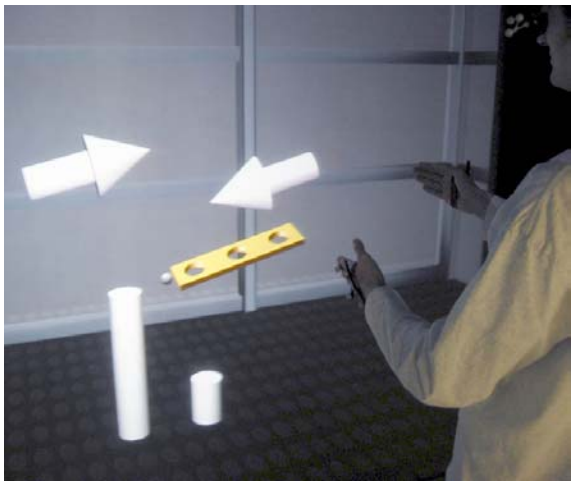

Fig. 4: A VR scene with different visualizations of field values.

Figure 4 shows a VR scene with different visualizations of field values. The cylinders visualize floating point numbers and the arrows show directions of vectors. In this example the two arrows visualize the directions of the palm-normals of a user while performing a scaling gesture. The cylinder on the right side shows the angle between these normals and the right one shows the distance of the hands.

Coordinate systems, represented by a sphere with 3 small arrows, can show the resulting transformation of a 4x4-Matrix, and simple spheres represent a position in the VR-space.

There are also visualization nodes for segments, trajectories and strings, which are represented by 3D-text that can be placed in the VR-scene.

Finally, it is possible to build a geometry for every computing node and arrange them in the scene-graph according to their field connections to get a complete visualization of the compute networks currently used.

Fig.5: A simple visualization of a compute network. in a CAVE-like display system

Figure 5 shows a visualization, automatically generated from a compute network. The single nodes are represented by squares with inner boxes, which show their fields. Field connections are visualized as arrows, which represent the dataflow through this network. The hole network is a container with its own input and output fields that are shown as dark boxes and are associated with normal fields in the network (see Section 4.4).

4. Scheme Scripting

The Scheme interface is a feature of the AVANGO toolpack. It makes it very easy to build and configure scene-graph structures, with a simple and user-friendly interface. Since the AVANGO shell comes with a Scheme interpreter it is possible to execute Scheme code during a running application. This allows the user to instantiate new compute nodes, set field values and field connections and call the user defined Scheme functions to modify and tune the application while it is running.

4.1. Instantiation of new nodes

New computing nodes can be instantiated within the Scheme interpreter. The corresponding Scheme function allocates a new instance of the C++ class and initializes the fields with the values, defined in the XML description. The user-defined Scheme callbacks are registered to the interpreter, so that they can be accessed by the commandline (see Section 4.3).

```
(define average-node (make-instance-by-name "prVecAverage"))
```

Fig. 7: Scheme code for creating a new node

Figure 7 shows an example of the Scheme code for creating a new compute node, which can smooth a stream of vector values by using the average within a defined time-window.

The following groups of compute nodes were implemented so far:

- Simple arithmetic calculation on Boolean, integer and floating point values
- 4x4-matrix and vector calculations
- Scene-graph manipulating nodes, like node-switches and size adjuster
- Complex render nodes which implement e.g. CSG-rendering in OpenGL
- Visualization-nodes for almost all types of field values
- 'History'-nodes, which store their input values for a certain amount of time and can be queried for values within this time range

4.2. Setting field values and connections.

Most of the settings of the compute nodes are done via field values, so that the parameters can be changed interactively. Parameters which have to be adapted to optimize the results can be changed in the running system to directly see the changes to the program.

```
(set-value  average-node 'AverageTime 0.2)
(connect-fields tracker-input-node 'OutPos_AS  average-node 'In_AS)
(connect-fields average-node 'Out_AS  subvec-node 'In1_AS)
```

Fig.8: Scheme code for field manipulations

In this example (Fig. 8) the time in which the average of the values is computed in the average-node of the example in Fig. 7 is set to 0.2 seconds. After that the input field for ASs of the average-node is connected from a node which provides the position of a tracking device and the filtered output is then connected to an other compute node. Since the computation on tracking data should be independent from the frame-rate of the rendering system, the fields are connected as AS values.

4.3. User defined Scheme Callbacks

For input and output, where the setting of field values is awkward – e.g. if multiple inputs have to be set at the same time or a discrete query is desired – the user can define Scheme callbacks as another interface to the functionality of the compute nodes. As seen in Section 3.2 the user can define methods with various arguments and can return complex Scheme structures.

Taken the example of Fig. 3 the add-entry function of a previously defined bool-history-1 can be called using the Scheme code shown in Fig. 9:

```
(->   bool-history-1    'add-entry     #t     10.0)
(->   bool-history-1    'get-value-at  6.7)
```

Fig. 9: Scheme code for calling user-defined callbacks

The first line adds the Boolean value TRUE to the history node at time 10.0 (seconds).

The second line retrieves a previously stored entry from the history node.

4.4. Node Containers

To build reusable groups of compute nodes, it is possible to instantiate node containers which define their own input and output fields and contain wired compute nodes.

Within these containers it is possible to instantiate nodes, make their field connections, define input and output fields and assign them to fields of the contained compute nodes.

```
(define test-container-1 (make-instance cContainer))
(send  test-container-1
       'add-component "prVecAverage" 'average-node)
(send  test-container-1
       'add-field "fpVec3" 'InPosition)
(send  test-container-1
       'assign-field 'InPostion 'average-node 'In_AS)
(send  test-container-1
       'connect-fields 'average-node 'Out_AS  'next-node 'In_AS)
```

Fig. 10: Scheme code for instantiating and manipulating containers

5. Example: A simple detector for a scaling-gesture

The following code is an example for a rather simple detector for scaling gestures, i.e. gestures which exert a scaling operation on a virtual object in the dimension indicated by hand movement. The two input fields are 4x4-matrices from tracking devices representing the position and orientation of the hands of the user. Then two normal-vectors, which represent the orientation of the palms and the angle between them, are computed. These values are also visualized in Fig. 4 together with the distance of the hands that defines the scaling-factor for the object. The visualization of this compute network is shown in detail in Fig. 11.

```
(define ScaleDetector (make-instance cContainer))
(send ScaleDetector 'add-field "fpMatrix" 'InLeftMat)
(send ScaleDetector 'add-field "fpMatrix" 'InRightMat)
(send ScaleDetector 'add-field "double" 'OutDist)
(send ScaleDetector 'add-field "fpVec3" 'OutDir)
(send ScaleDetector 'add-field "bool" 'isGesture)

(send ScaleDetector 'add-component "prMatField" 'LeftHandTracker )
(send ScaleDetector 'add-component "prMatField" 'RightHandTracker )
(send ScaleDetector
      'add-component "prMatRotVec" 'LeftPalmNormal )
 (send ScaleDetector
      'add-component "prMatRotVec" 'RightPalmNormal )
(send ScaleDetector 'add-component "prMatXVec" 'RightPos )
(send ScaleDetector 'add-component "prMatXVec" 'LeftPos )
(send ScaleDetector 'add-component "prVecAngle" 'Angle )
(send ScaleDetector 'add-component "prVecSub" 'LtoR )
(send ScaleDetector 'add-component "prDblThreshold" 'isAntiParallel )
(send ScaleDetector 'add-component "prVecNorm" 'Direction )
(send ScaleDetector 'add-component "prVecLength" 'Distance )

(send ScaleDetector
      'assign-field 'InLeftMat 'LeftHandTracker 'matrix_AS )
(send ScaleDetector
      'assign-field 'InRightMat 'RightHandTracker 'matrix_AS )
(send ScaleDetector  'assign-field 'OutDist 'Distance 'Out)
(send ScaleDetector  'assign-field 'OutDir 'Direction 'Out)
(send ScaleDetector  'assign-field 'isGesture 'isAntiParallel 'Out)
```

```
(send ScaleDetector 'connect-fields
      'LeftHandTracker 'matrix_AS 'LeftPalmNormal 'InMat_AS)
(send ScaleDetector 'connect-fields
      'RightHandTracker 'matrix_AS 'RightPalmNormal 'InMat_AS)
(send ScaleDetector 'connect-fields
      'LeftHandTracker 'matrix_AS 'LeftPos 'InMat_AS)
(send ScaleDetector 'connect-fields
      'RightHandTracker 'matrix_AS 'RightPos 'InMat_AS)

(send ScaleDetector 'connect-fields 'RightPalmNormal 'Out 'Angle 'In1 )
(send ScaleDetector 'connect-fields 'LeftPalmNormal 'Out 'Angle 'In2 )
(send ScaleDetector 'connect-fields 'Angle 'Out 'isAntiParallel 'In )

(send ScaleDetector 'connect-fields 'RightPos 'Out 'LtoR 'In1 )
(send ScaleDetector 'connect-fields 'LeftPos 'Out 'LtoR 'In2 )
(send ScaleDetector 'connect-fields 'LtoR 'Out 'Distance 'In )
(send ScaleDetector 'connect-fields 'LtoR 'Out 'Direction 'In )

(send ScaleDetector
      'set-value 'LeftPalmNormal 'InVec (make-vec3 0 0 -1))
(send ScaleDetector
      'set-value 'RightPalmNormal 'InVec (make-vec3 0 0 -1))
(send ScaleDetector 'set-value 'isAntiParallel 'ThresVal 2)
```

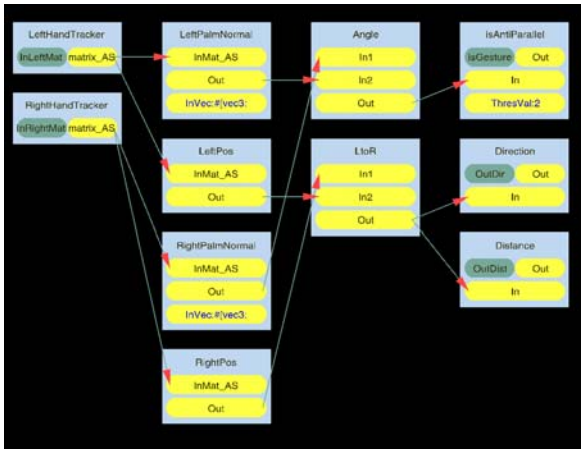

Fig. 11: The visualization of the container for a scaling-gesture

6. Results & Outlook

In this paper we have presented an approach for visual programming in a VR-environment. The computation of the single nodes is embedded in the render loop, but it is also possible to compute sequences of data, independent of the frame rate. The XML description of the nodes allows fast development of new units to gain a very flexible tool that is facile to be administered. With the possibility to build containers of compute networks with their own field definitions the programmer can produce highly reusable code, with a well-defined interface and which can be easily adapted to new applications.

The close relationship to the scene-graph nodes allows the visualization of results and intermediate data of the compute nodes. A simple visualization in the virtual environment of the compute nodes and their connections was presented.

Additionally to an improved visualization, it is planned to build a tool for altering field values and connections in the virtual environment using a pointing device, like a stylus. The stylus can be used to select the field or connection which should be altered, and to manipulate a user interface to change the values. With the direct feedback of the effects in the VR application of the changes in the compute network, the tuning and debugging of programs is readily achieved.

Since only the adjustment of the compute networks is done via the comparably slow Scheme interface and all computing and data flow is handled in C++ code, even large networks can be computed very fast, which is extremely important for real-time VR applications.

With the possibility of distributing field connections over network – which is supported by the AVANGO toolkit – very large computing networks can be divided on computer clusters to gain optimal performance.

Acknowledgement

This work is partially supported by the Deutsche Forschungsgemeinschaft (DFG) in the *Virtuelle Werkstatt* project.

Literature

[Lat2001] M.E. Latoschik: A General Framework for Multimodal Interaction in Virtual Reality Systems: PrOSA. In: Broll, W & Schäfer, L. (Editors): *The Future of VR and AR Interfaces - Multimodal, Humanoid, Adaptive and Intelligent.* Proceedings of the workshop at IEEE Virtual Reality 2001, Yokohama, Japan. GMD Report No. 138, March 2001, pp. 21-25.

[Tra99] Henrik Tramberend. A distributed virtual reality framework. In *Virtual Reality*, 1999.

[Dyb96] R. K. Dybvig. *The Scheme Programming Language: ANSI Scheme*. P T R Prentice-Hall, Englewood Cliffs, NJ 07632, USA, second edition, 1996.

[Dou2001] Doug Tidwell. *XSLT*. O'Reilly, August 2001.

[Hil92] Hils, D.D. *Visual Languages and Computing Survey: Data Flow Visual Programming Languages*. JVLC, March 1992, pp. 69-101