Chapter 1

Data Flow Languages for Immersive Virtual Environments

Anthony Steed
Department of Computer Science
University College London

Abstract

Current immersive virtual environment (IVE) systems use a wide variety of programming methodologies and scene description languages, and although many support importation of VRML 1.0 scenes, there is no agreement on how behavior and interactions should be described. Prototyping IVE behaviors can be a tedious process. To solve this, an immersive programming language, the virtual environment dialogue architecture (VEDA), was built to allow rapid and immediate changes to be made. Underlying VEDA is a data flow model similar to that of the recent VRML 2.0 standard. This paper contrasts the two models and highlights the strengths of both. Various approaches to creating an immersive VRML 2.0 programming environment are proposed from this comparison.

Introduction

A data flow model describes a system in terms of the data being passed between functions that transform its state. The tracing out of all such data flows through the system forms a directed graph, with nodes corresponding to functions and arcs indicating the possible routes for data to take. Such a model has a direct visual representation, and so-called *data flow visual programming languages* (Hils 1992) have proven useful and instructive in a number of application areas, such as scientific visualization.

The current virtual reality modelling language specification, version 2.0 (1996) uses a data flow model of operation to describe the virtual environment (VE). The nodes represent functions for describing behaviors, interactions, geometry, and appearance, and the arcs represent the manner in which node properties will be altered. A visual representation of the data flow of VRML 2.0 would certainly be possible, but as of yet, no tools exist that can directly use such a metaphor for immediate and interactive editing of scene graphs.

The utility of such a system has been shown through the design of implementation of a similar system for immersive virtual environments (IVEs). The virtual environment dialogue architecture (VEDA) system (Steed and Slater 1996; Steed 1996) was designed to allow immersed participants to rapidly prototype new behaviors and interactions without leaving the environment. Essentially, VEDA is a hybrid object hierarchy and data flow model with an immersive 3-D representation. VEDA's data flow model is similar to that of VRML 2.0.

This paper begins by describing current work on visual languages for virtual environments. Then it describes the motivation and design of the VEDA system. This is contrasted with the VRML 2.0 design later in the chapter. The final section proposes a number of approaches to creating a visual editor for VRML 2.0.

Current Technology

Data Flow Model

A data flow model is composed of nodes, that transform data, and arcs, which represent the passing of data between those nodes. Figure 1.1 shows an abstract representation of a data flow graph. Data flow starts from particular nodes, known as triggers or sensors, that are the external interface of the system. They might sense devices or react to a certain event in the VE, such as collision detection. Once a trigger is activated, data flows along all its connections to connected processing nodes. These nodes transform the data and then pass it on to further nodes that process or have an effect on the external world. Data flow stops at a node when the reception of data causes no further outputs. When constructing the entire graph of data flow, both *fan-in* and *fan-out* are usually allowed. Respectively, these are data from multiple sources arriving at the same receiver, and data from a source being sent to multiple receivers.

VRML 2.0 uses such a model to describe transformations of the properties of objects in the scene database. VRML 2.0 contains similar elements to those in VRML 1.0 (Bell, Parisi, and Pesce 1995) to describe the visible elements of a scene: geometry, appearance, lights, and transformations. The addition of a data flow model, through the use of sensors, interpolators, scripts, and routes, allows dynamic and interactive worlds to be built. Each VRML 2.0 node may have one or more inputs, or *eventIn* fields, where data can be received. The reception of data on certain fields triggers the calculation of an internal function, which may subsequently generate values on one or more *eventOut* fields, which are propagated to all connected nodes. The types of data that can be passed around include boolean values, colors, positions, orientations, and other nodes. In addition, nodes might have plain *fields,* which are not exposed to the data flow and are used for static state that

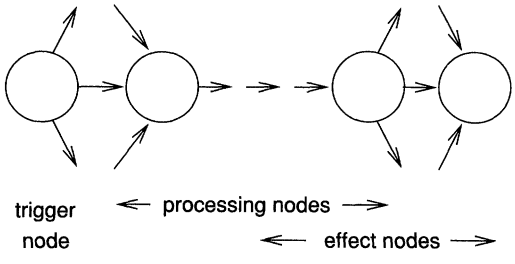

Figure 1.1 Components of a data flow model.

is initialized at start-up, and *exposedFields,* which serve as both eventIn and eventOuts. That is, data received has an effect on the node and is routed through to further nodes.

As an example, the fields of the *OrientationInterpolator* are given in Figure 1.2. The OrientationInterpolator's purpose is to generate values on the eventOut *value_changed,* whenever the eventIn *set_fraction* is changed. These values are chosen by interpolating between the values defined by the exposedField *keyValue* where the exposedField *key* gives the reference fraction values.

Figure 1.3 shows a sample VRML 2.0 file that uses the OrientationInterpolator, and Figure 1.4 the corresponding scene graph and data flow graph. The behavior is very simple; when the user clicks upon the red cube, it starts spinning. The TouchSensor senses the user clicking on the geometry and generates a time value on a data stream. This then flows to the TimeSensor node, which in turn generates a fraction value. This is then used by an OrientationInterpolator to start the generation of orientation values, which are routed to the position of the cube to provide an animation. The four elements of the data flow graph map onto the different types of data flow nodes in the abstract graph of Figure 1.1. In this case,

```
OrientationInterpolator {
eventIn    set_fraction
exposedField MFFloat key []
exposedField MFVec4f key value []
eventOut    MFVec3f value_changed
}
```

Figure 1.2 Fields of the OrientationInterpolator.

```
#VRML V2.0 utf8
Viewpoint {
 position 0 0 5
}
DEF BOX_POS Transform {
 children [
  Shape {
    geometry Box {}
  },
  DEF BOX_TOUCH TouchSensor {},
 ]
}
DEF BOX_RIMER TimeSensor {
 cycleInterval 5
 startTime - 1
}
DEF BOX_ENGINE OrientationInterpolator {
 key [ 0, .5, 1]
 keyValue [ 0 1 0 0, 0 1 0 3.14, 0 1 0 6.28]
}

ROUTE BOX_TOUCH.touchTime TO
BOX_TIMER.set_startTime
ROUTE BOX_TIMER.fraction TO BOX_ENGINE.set_fraction
ROUTE BOX_ENGINE.value_changed TO
BOX_POS.set_rotation
```

Figure 1.3 Sample VRML 2.0 file.

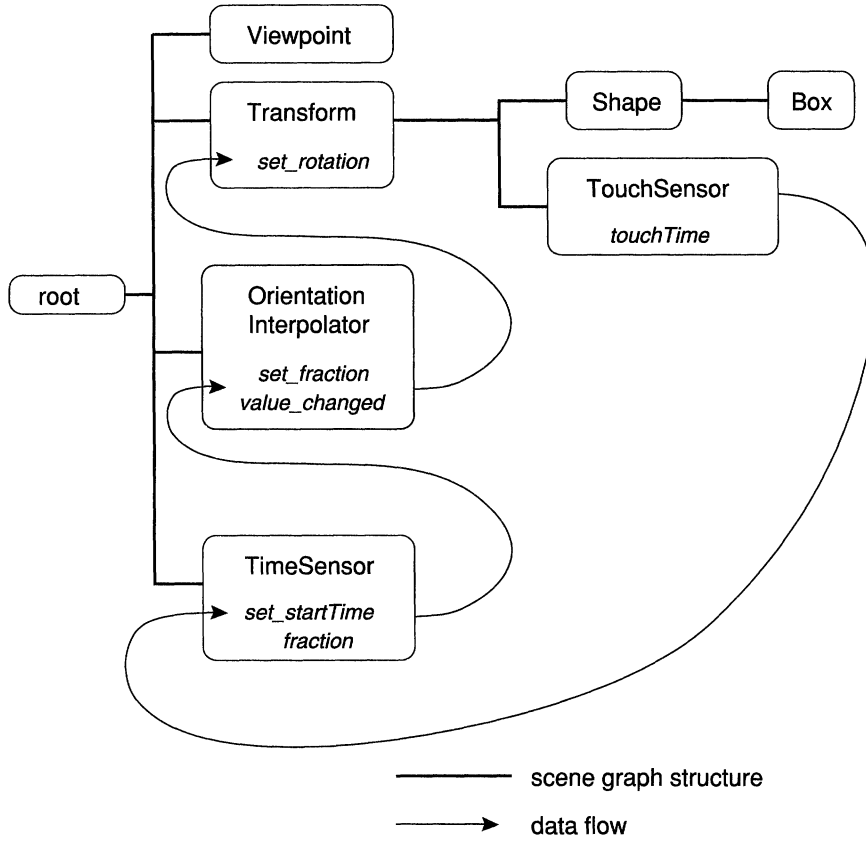

Figure 1.4 Scene graph of sample file.

the trigger node is the TouchSensor since it reacts to an event external to the scene graph (the user). The TimeSensor and OrientationInterpolator are the processing nodes, and the cube is the effect node, since the modification is visible to the user through the use of a browser.

VRML 2.0 is only one example of the data flow model, and many variations in the types of nodes and arcs allowed are possible (Hils 1992). Given a data flow model, there is a direct visualization of the data flow (see Figures 1.3 and 1.5), and a data flow visual language makes the obvious visualization into a programming environment. Many examples exist; IRIX Explorer (Foulser 1995) and AVS (Upson et al. 1989) are two well-known examples.

Visual Languages for Virtual Environments

In describing current visual programming systems for virtual reality, we have to distinguish between two classes:

- Visual languages to *describe* virtual environments
- Visual languages *within* virtual environments

Body Electric from VPL is an example of the first class of language (Kalawsky 1993, 212–219). Body Electric ran on a Macintosh adjacent to the main virtual environment generator. It used a node and arc-type diagram with the data flowing from the input devices

through data "massage" units to drive events in the virtual environment. Since Body Electric ran on a separate system to the display generator, it was not accessible to the immersed participant.

Other more widely used 2-D data flow visual languages have been extended to support VEs. These include the AVS scientific visualization (Sherman 1993) and IBM's Data Explorer (Gillilan and Wood 1995). However, in both cases the VE presentation is an effect of a node within the data flow diagram rather than being the specification environment.

The second class of system involves a visual language, which, rather than being manipulated on a 2-D display, is manipulated within a 3-D environment. Such an approach was advocated by Glinert (1987) who proposed extending his BLOX method, which converts flat jigsawlike pieces to 3-D, using cubes that can be snapped together much like building blocks.

Several visual programming languages with 3-D representations have been built. They vary greatly in the style of interface and level of liveness at which they operate.

VPLA (Lytle 1995) is an interpreted language specified by a 3-D network editor that describes components of an animation in terms of hierarchical objects and actions on them, such as transformations, modelling operations, deformations, particle systems, and recursive procedures. However, VPLA is not "live" since it is used to produce RIB scripts that are subsequently rendered off-line.

Two virtual environment languages for programming that have been implemented are CUBE (Najork 1994) and Lingua Graphica (Stiles and Pontecorvo 1992). CUBE allows visualization of expressions in a functional language and some limited editing through a desktop-based interface. Lingua Graphica is a 3-D editor for a C++ based language. The Lingua Graphica workspace looks like a tool board with the tools being the various library functions, primitives, and predefined types. These can be juxtaposed with the spatial relationships between the objects corresponding to the syntax rules of the language. Once a procedure has been constructed, it can be written out directly in a text form that is compilable.

A visual language that belonged to both of these classes would both describe a virtual environment and exist within it. It would also ideally operate in a *live* mode. That is, the changes to the data flow could be made without leaving the environment and with immediate effect.

It is worth noting that it is not unusual for a mature IVE system to contain immersive tools with which to modify the environment. A tool such as dVISE 3.1 from Division Ltd. (1996) includes immersive and standard desktop menu systems that allow certain behaviors and properties of the objects to be modified. However, not every function of the underlying system is available to the immersed participant, and many programming tasks have to be conducted outside the environment.

Virtual Environment Dialogue Architecture

Model

The virtual environment dialogue architecture (VEDA) is a model for the programming of VEs and also an environment and set of techniques within which the behavior of the environment components can be manipulated. It uses a data flow model whose three types of functions map directly on to the three classes in Figure 1.1:

- **Devices** that return information about the participant and generate data streams
- **Filters** that process data samples on one or more incoming data streams and then generate new data streams
- **Tools** that take input streams and act upon the objects of the VE

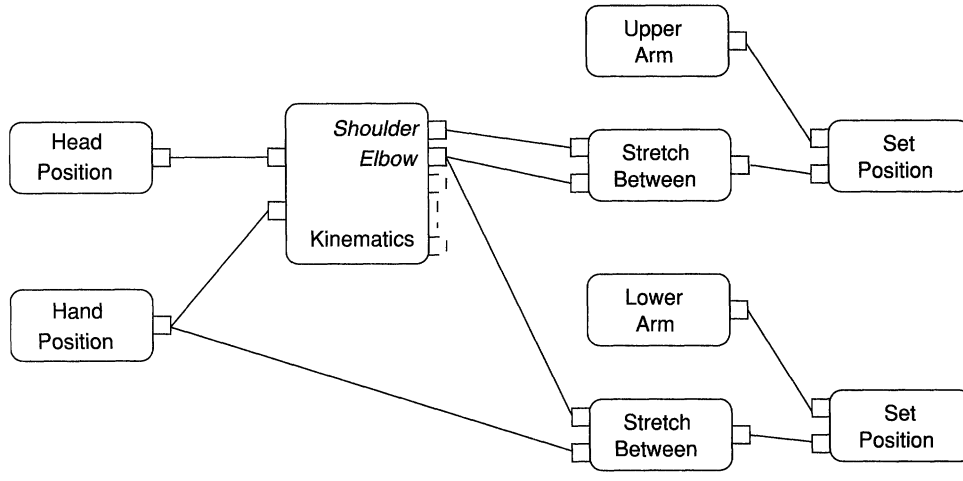

Figure 1.5 The abstract data flow model of the arm.

Figure 1.5 gives an example of how the arm of a participant is modelled using inverse kinematics. The data flow starts with two devices, one that returns the head position and one that returns the hand position. The kinematics function takes both of these as inputs and then generates further streams that correspond to other joint positions based upon a standard set of parameters for the body. The stretch-between function takes two positions and generates a data stream containing the transformation that would stretch a unit object between them. Finally, the setposition tool takes a position stream and applies this to an object, the id of which is passed across a second data stream.

Representation

The data flow model of the VE is presented immersively to the participant. This means that they can make changes to the live application without exiting the environment. Since the descriptions can be highly detailed, there are tools to hide and reveal parts of the data flow and to encapsulate several functions into a higher-level function.

An example of the immersive representation is shown in Figure 1.6. This representation implements a navigation metaphor called the *virtual treadmill* (Slater, Usoh, and Steed 1995). The functions represented are the head sensor on the left, the move function on the right, a gesture recognizer at the bottom, and the object referencing the participant at the top. The effect of this data flow is to move the participant in the direction they are looking when the head is making the gesture of walking in place. Data flow is represented by the pipes between input and output gates of functions. The shapes below the functions represent the types of the data and are colored, depending on whether they are input or output.

Basic Functions

The set of functions provided by VEDA was designed with the aim of providing interesting interactions and behaviors. These include the following:

- Gesture recognition—path and neural net based
- Object properties—color, scale, and position

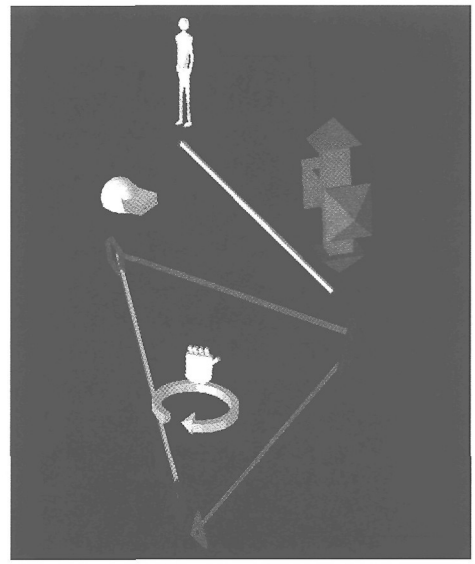

Figure 1.6 Immersive representation of the virtual treadmill navigation metaphor. (*See color plate on page 305.*)

- Object behaviors—collisions, animation, and constraints
- Logical and timing—toggle, double click, and so forth

These serve as the static parts of a scene graph, but VEDA was designed with the aim of allowing editing of the graph, so any part of the data flow representation can be picked and moved, functions can be copied and deleted, and the tube connectors can be dragged and snapped between function inputs and outputs.

Meta Objects

The immersive representation of the functions form the specification layer of the virtual environment, but the presentation layer certainly should not contain these objects if they clutter the space or detract from the purpose of the environment. The first way in which this is tackled is to provide level of detail tools within the environment. When applied to a function node, they can either hide or reveal the connecting tubes between two nodes.

VEDA also uses the concept of *meta objects* to provide complete hiding of function nodes and higher level abstractions of complicated behavior. A meta object can have any representation, so at a basic level it serves to allow arbitrary geometry to be included in the presentation of a world. A meta object also serves as a container, in that it can have a collection of other objects as children and a selection of their input and output streams as an interface. When combined with the level of detail tools, this allows whole sections of the data flow to be hidden or exposed, since the lowest level of detail for a meta object is for its children to be made invisible. Thus, an application environment will consist of a number of top-level meta objects that represent the form of the environment, and each one of these will be hiding within it the functionality of the data flow associated with that object.

Using these mechanisms, many higher-level objects have been built within VEDA that allow easy manipulation of new environments. These include virtual sliders, path tools, color tools, scale tools, and a virtual multimeter.

Motivation

The foremost reason for using an immersive representation of the scene behavior was so that editing could take place within the VE and not require the participant to repeatedly enter and leave the environment when making modifications. Apart from the obvious time penalty this involves, exiting the environment leads to a loss of *presence* of the participant in the environment, which might in turn lead to their forgetting exactly what the state of the environment was. Also, in many systems, editing the environment involves rerunning or, at worst, recompiling the environment, which leads to an application loss of state that could be time-consuming to reconstruct.

Another motivation is that the immersive 3-D nature of the environment should provide a good environment to display the network of data flow between objects. There is direct evidence to show that a 3-D node and arc diagram may offer substantial benefits to comprehension of the dialog structure over the 2-D diagram approach (Ware and Franck 1994).

From a design point of view, there might be another, more subtle benefit to immersively editing behavior. That is, since the participant has a sense of presence within the environment they are designing for, it may be the case that any techniques or design decisions that are effected would be more appropriate to that environment. Essentially, one would hope that since the participant is present within the VE, anything they would conceive of designing or altering would be consistent with the VE's overall style.

Comparison of VRML 2.0 and VEDA

Although VRML 2.0 and VEDA were both designed to describe virtual environments, their scopes are not identical, and thus the models and nodes that each supports differ on a number of points. VRML 2.0 is primarily a description of a file format for the transmission of interactive 3-D worlds across the Internet. VEDA was designed as an interactive VR tool kit for describing interfaces for both desktop and immersive systems. Thus, although there is a large overlap in nodes types, VEDA does not deal with URLs, scripting, or external interfaces, and VRML 2.0 does not support some forms of interface, a broad editing API, or a visual representation. The common ground is the similar descriptions of object properties such as color, position, and animation, and the basic data flow model of object behavior.

Models

Scene Graph

The data flow model is common to both systems, though the nodes of the data flow are different in each case. In VRML 2.0, the primary database structure comes from the scene graph, which is a property hierarchy, and the scripting and routing mechanisms are largely independent of this in that their behavior does not depend upon position. In VEDA, the primary database structure defines the relationship between meta objects and the components that make up their behavior. In particular, VEDA has no property hierarchy. Every VEDA function or meta object has a collection of properties: geometry, appearance, and position. Figure 1.7 shows that to change the position and scale of a scene element involves operating on a transform property in VRML 2.0, or on an object in VEDA.

Function Style

Figure 1.7 also illustrates the fact that VEDA effect nodes were designed to be function orientated rather than node orientated. In VEDA the data flow graph to change both the position and scale contains two functions, SetScale and SetPosition, which operate on a

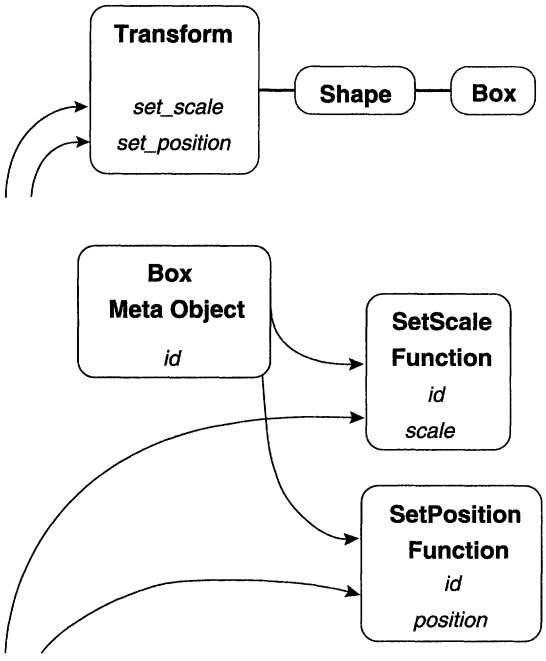

Figure 1.7 Contrasting data flow hierarchies for a simple task in VRML 2.0 and VEDA.

reference to the object, whereas in VRML 2.0, this is achieved with two data flow inputs to the Transform node. VEDA's approach was chosen to allow easier visual representation of the data flow, since the irrelevant part of the data flow could be hidden or revealed. It is possible in both systems to use prototypes or encapsulation to make data flow constructs that look like the corresponding ones from the other language.

Flow Model

VEDA was designed around a *data driven* model of data flow. This has the data flow starting at trigger nodes, which are updated at a continuous frequency. The alternative is the *demand driven* model that does a reverse, request-based polling of functions. In this case, data flow starts at nodes that have an effect in the displayed environment and propagates backwards until enough data has been accumulated in order to compute the correct state of the world. The advantage of the second approach is that it can be done in a lazy fashion, with only visible effects being computed. This means that some functions need not be evaluated, and if the time to propagate all the required data can be estimated, requests can be timed to be as close to the display refresh as possible. VRML 2.0 does not impose a model on the browser author, but in effect, VEDA had to, since external sensor updating and rendering were controlled by separate processes (see later section on Extensions). This meant it was impossible to do lazy propagation since there was no way of testing whether an object was being rendered, and it was also impossible to time the data flow to the rendering cycle.

Interaction Model

Since one of the motivations for building VEDA was to allow customization of the interaction metaphors, VEDA allows direct interaction with the devices that the participant is using. In VRML 2.0 the user can interact with an object only through Sensors, which

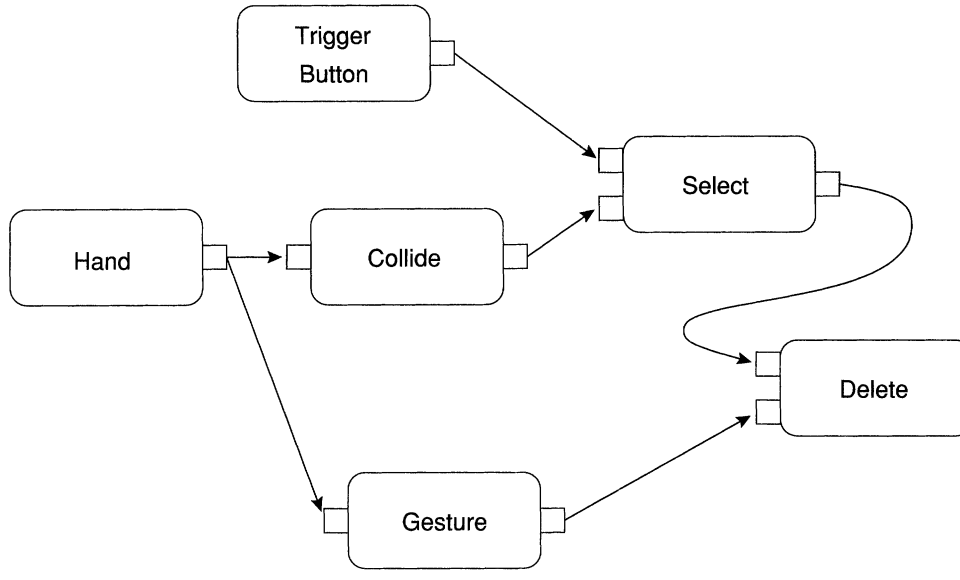

Figure 1.8 Abstract data flow for a delete mechanism.

impose an interaction style on the world since they also constrain the interaction in some way. In particular, there is no sense of the participant having a body in VRML, whereas VEDA considers the participant to be presented by a set of objects that are being updated by sensing devices, both position sensors and buttons. Thus, VEDA trigger nodes give the position and state of parts of the participant's body continuously rather than only when they interact.[1] The use of a body model makes the interaction style largely independent of the interaction devices, since as the example in Figure 1.5 showed, the body state can be inferred from few sensors. This allows fundamental changes to be made to the interaction structure within the data flow graph. For example, Figure 1.6 showed the definition of an interaction metaphor within VEDA based on where the participant was looking.

Editing Mechanisms

Another motivation for the design of VEDA was to allow live editing of the environment, so VEDA incorporates effect functions that copy and delete other nodes. This, in combination with collision detection, allows a simple tool-based approach to environment construction. The mechanisms by which the editing functions are used is in fact described within VEDA. Figure 1.8 illustrates the standard way in which the delete function is used. The objects to delete are a set selected by touching them with the hand and pulling on the trigger button. Once selected, they can be deleted by making the delete gesture with the hand. This means that within VEDA, it is possible to delete the delete function, but there are situations where you might want this, such as the last stage of reconfiguring an environment for use by a novice.

[1]Continuous extraction of the position of the viewpoint is actually possible in VRML 2.0 using a large ProximitySensor, but it is not possible to get the position of the hand.

Table 1.1 **VEDA and VRML 2.0 Node Comparison**

Node Class	*VEDA Support*	*VRML 2.0 Support*
Gestures	Training and recognition nodes	Possible with scripts
Logic	And, or, not, then,	Possible with scripts
Collision Detection	Supported	Viewer collision, but no object collision
Constraints	Supported	Possible with scripts and prototyping
Animation	Path following and description	Interpolators give a flexible approach
Sensors	Device and collision sensors, no visibility sensor	Constrained interaction sensors
Appearance and geometry	Properties excluding texture and geometry	Full support
Sound	Play recorded sounds	Spatialized playing of recorded sounds
Lighting	Is an object property	Supported
Node editing	Delete, copy, selection	Indirect through scripting

Nodes

The previous section described the differences between the models and introduced some of the nodes in areas of nonoverlapping scope. Table 1.1 summarizes the node differences where there is common support.

Apart from the differences mentioned already, such as the role of sensors in both systems and the lack of collision detection in VRML 2.0, there is better support in VRML 2.0 for changing properties. This is partly due to the property graph structure of VRML 2.0 where each property, such as texture and geometry, is a candidate for inclusion in the data flow. Part of the reason VEDA does not support these data flow types is that the platform it was written on supported neither texture mapping nor dynamic geometry editing. However, given the function-oriented approach of VEDA, such functions are simple to add as long as the underlying database API supports them.

Extensions

One of the most interesting proposed extensions to VRML 2.0 is the External Authoring Interface (Marrin 1996). This will allow external control of a VRML 2.0 scene through an API within the browser. This will allow other applications to alter the scene graph independently of the data flow model. In effect, such a layer is inappropriate for VEDA since it acts as an external process, operating on an object store through an API. VEDA was written as a dVS application and is only one of a number of actors sharing a database (Division Ltd. 1994). Figure 1.9 illustrates the separation of the two, with the clear interfaces between the database and event mechanisms.

The VEDA equivalent of a process using VRML 2.0's external API is simply another dVS client. As a consequence of this, VEDA automatically supports multiple users since in the dVS model, a second user is simply a second set of trackers and renderers. However, since the data flow is not represented in the dVS database, there is no capability for two clients to interact except through surface interactions; that is, interactions that can be detected from the current state of the scene rather than any application level semantics.

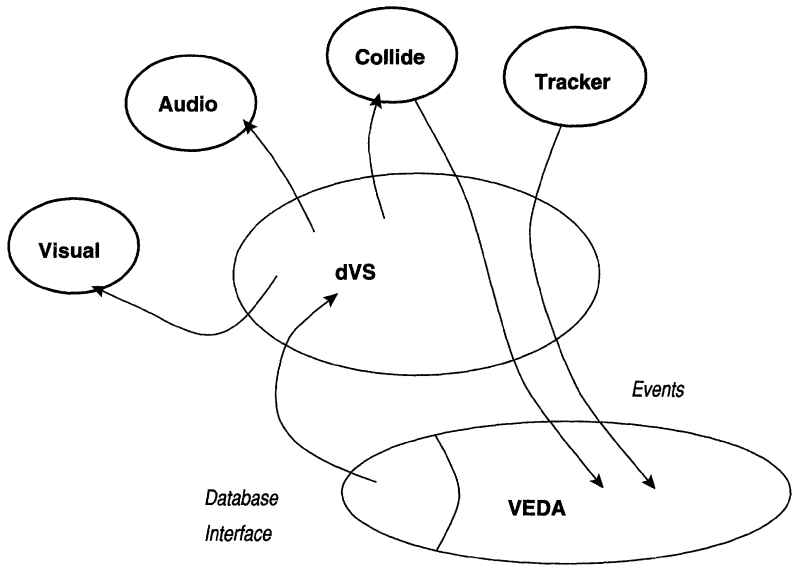

Figure 1.9 Relationship between VEDA and dVS.

Towards a Visual Representation of VRML 2.0

The experience with VEDA has shown that it is useful to have the capability to make changes to the world while the participant is immersed. This allows aspects of the behavior of the environment to be demonstrated in situ and take full advantage of the full body interaction style.

The two data flow systems have a number of parallels in the node classes they provide, but they differ in some important areas: sensor information and collision information, which were discussed previously. The second could be prototyped in VRML 2.0, but the first would either require access through some sort of browser API to devise information or through new nodes that modelled the participant's body. The body model approach has advantages for portability across browsers, but the exposed device approach allows the browser users greater scope for customizing their representation and behavior in the environment.

As it stands, VEDA would require a handful of new nodes in order to be able to support the import of VRML 2.0 worlds. Export is more problematic since the mapping of gestures and interaction techniques into VRML 2.0 could not be made without new browser-dependent nodes. However, since one of the aims of a VEDA style system would be to provide editing for the end-user rather than just scene authors, it would be necessary to provide the required functionality across different browsers, so the application itself must be written in VRML 2.0.

As noted earlier, VEDA is simply a dVS client and interacts with it through a database API. Since the separation is quite distinct, it would be simple to write a similar application that used the proposed external authoring interface VRML 2.0. This would avoid the problem of accessing tracking devices since the external applet could interface to them directly. Either the browser or the scene itself would have to have a hook that allowed the presentation level to be augmented with the VEDA tools, and these could be downloaded as required.

Conclusions

A number of systems have started looking at applying a visual approach to the description of virtual environments. The VEDA system is the first to combine a data flow model with a live immersive programming environment. It allows the participant to alter the behavior of a running application by examining and editing the underlying data flow model.

While the recent VRML 2.0 standard also supports a data flow model of execution, it does not support or define a visual representation. The interaction model lacks the flexibility to be used to describe novel manipulation or navigation techniques, but through the use of VRML's external API, this can be rectified to some degree. The external API also gives us the ability to introduce editing tools into the environment, and thus a VEDA-like approach to environment design could be supported as a VRML 2.0 scene.

References

G. Bell, A. Parisi, and M. Pesce, "The Virtual Reality Modeling Language, Version 1.0 Specification," Web document, available at http://vrml.wired.com/vrml.tech/vrml10-3.html (May 26 1995).

dVISE for Unix Workstation, User Guide, Division Ltd., Almondsbury, Bristol, U.K., 1996.

dVS Technical Overview Version 2.0.4, Division Ltd., Almondsbury, Bristol, U.K., 1994.

D. Foulser, "IRIS Explorer: A Framework for Investigation," *Computer Graphics,* May 1995, pp. 13–16.

R.E. Gillilan and F. Wood, "Visualization, Virtual Reality, and Animation within the Data Flow Model of Computing," *Computer Graphics,* May 1995, pp. 55–58.

E. Glinert, "Out of Flatland: Towards 3-D Visual Programming," *Proc. 2d Fall Joint Computer Conference,* IEEE Computer Soc. Press, 1987, pp. 292–299.

D. Hils, "Visual Languages and Computing Survey: Data Flow Visual Programming Languages," *J. Visual Languages and Computing,* Vol. 3 No. 1, 1992, pp. 69–101.

R.S. Kalawsky, *The Science of Virtual Reality and Virtual Environments,* Addison-Wesley, 1993.

W. Lytle, *Vpla: Visual Programming Language for Animation,* technical sketch, SIGGRAPH95, 1995.

C. Marrin, "Proposal for a VRML 2.0 Informative Annex, External Authoring Interface," http://vrml.sgi.com/moving-worlds/spec/ExternalInterface.html (Sept. 12, 1996).

M. Najork, *Programming in Three Dimensions,* PhD thesis, Univ. of Illinois, Urbana-Champaign, 1994.

W. Sherman, "Integrating Virtual Environments into the Dataflow Paradigm," *Fourth Eurographics Workshop on ViSC,* Abingdon, UK, April 1993.

M. Slater, M. Usoh, and A. Steed, "Taking Steps: The Influence of a Walking Metaphor on Presence in Virtual Reality," *ACM Transactions on Computer Human Interaction,* Vol. 2, No. 3, 1995, pp. 201–219.

A. Steed, *Defining Interaction within Virtual Environments,* PhD thesis, Queen Mary and Westfield College, Univ. of London, Dept. Computer Science, 1996.

A. Steed and M. Slater, "A Dataflow Representation for Defining Behaviours within Virtual Environments," *Proc. VRAIS'96,* IEEE Computer Soc. Press, 1996, pp. 163–167.

R. Stiles and M. Pontecorvo, "Lingua Graphica: A Visual Language for Virtual Environment," *Proc. 1992 IEEE Workshop Visual Languages,* IEEE Computer Soc. Press, 1992, pp. 225–227.

C. Upson et al., "The Application Visualization System: A Computational Environment for Scientific Visualization," *IEEE Computer Graphics and Applications,* Sept. 1989, pp. 30–42.

"The Virtual Reality Modeling Language Specification, Version 2.0," http://vag.vrml.org/VRML2.0/FINAL/ (Aug. 4, 1996).

C. Ware and G. Franck, "Viewing a Graph in a Virtual Reality Display Is Three Times as Good as a 2D Diagram," *Proc. 1994 IEEE Symposium Visual Languages,* 1994, pp. 182–183.