

Research Report

Dialogue Structures for Virtual Worlds

J. Bryan Lewis
Lawrence Koved
Daniel T. Ling

IBM Research Division
T. J. Watson Research Center
Yorktown Heights, NY 10598

LIMITED DISTRIBUTION NOTICE

This report has been submitted for publication outside of IBM and will probably be copyrighted if accepted for publication. It has been issued as a Research Report for early dissemination of its contents and will be distributed outside of IBM up to one year after the date indicated at the top of this page. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies of the article (e.g., payment of royalties).

Dialogue Structures for Virtual Worlds

J. Bryan Lewis
Lawrence Koved
Daniel T. Ling

Veridical User Environments
IBM Research
T. J. Watson Research
Yorktown Heights, New York 10598

Abstract:

We describe a software architecture for virtual worlds, built on a base of multiple processes communicating through a central event-driven user interface management system. The virtual world's behavior is specified by a dialogue composed of modular subdialogues or rule sets. In order to achieve high flexibility, device remappability and reusability, the rule sets should be written as independent modules, each encapsulating its own state. Each should be designed according to its purpose in a conceptual hierarchy: it can transform a specific device into a generic device, or transform a generic device into an interaction technique, or, at the top level, map interaction techniques to actions.

Keywords. User-interface design issues, user interface management systems, virtual worlds, virtual reality.

Background

The concept of a virtual world (VW) as an interaction metaphor is relatively new, enough so that there have been relatively few discussions of the preferred architecture, design concepts and software engineering techniques. Many investigators build their first virtual world with techniques similar to those that were useful in desktop applications, except for the additional work of interfacing to unusual devices such as the VPL DataGlove and Polhemus 3Space tracker. That is how we built our first such world, a handball game, which we structured as a single loop to poll the glove and tracker and to update the display. We began to learn that there are several essential characteristics of virtual worlds:

1. Real-world performance is crucial. When the user's hand moves, it's important that the virtual hand appears to move with an imperceptibly small delay. High display rate is important, but even more important are low latency and a smooth continuity. [Brooks, 1988].
2. The simultaneous use of multiple input and output devices is essential. The user's simple action of pointing a finger and saying "There!" causes three essentially simultaneous events from the tracker, glove, and speech recognizer. This characteristic, combined with the performance criterion, makes essential the concurrent processing of input and output devices.

We achieved a degree of concurrency early in our work by building dedicated servers for the glove and tracker [Wang et al., 1990]. We have continued to add new devices, including speech input, speech output and sound output, as servers. Communication takes the form of asynchronous message-passing over a private network. Coordination is handled by a central kernel-like executive process with which all the devices communicate. Thus the foundation of our architecture, shaped by the performance and multiple-device characteristics, is as shown in Figure 1.

Figure 1. Basic architecture for a virtual world

The central executive process fills the same role as a dialogue manager or user interface management system (UIMS) in conventional desktop applications. In fact we

adapted an existing event-driven rule-based UIMS [Rhyne, 1988] by adding a front end to receive asynchronous network messages and package them as events for the rule matcher. This allows us to handle events from newly defined devices like the glove, tracker, and speech recognizer, just as easily as the desktop version handled mouse and keyboard events. With this kind of UIMS the *dialogue*, which specifies and synchronizes the mapping of inputs to actions and results to output, is expressed as a set of production rules. Each rule is fired by an event or pattern of events, and produces a new event or accomplishes an immediate effect by executing semantics embedded in the rule.

3. VWs are relatively large and complicated collections of software, requiring the combined skills and efforts of several developers. A good illustration of this is our latest virtual world which enables visualization and manipulation of computational fluid dynamics data. This world, to be described in more detail later, is a relatively small one, comprising only one graphic object, four input techniques and three output techniques. Yet its development required close attention to the software engineering principle of decomposition into small, logically self-contained modules communicating via standardized interfaces.
4. VWs need to be constructed in a highly flexible way, to make it easy to change devices and techniques. VW devices and techniques are as yet not standardized or well understood, and we want to be able to introduce new ones easily. Ideally the introduction of a new device or interaction technique should cause only well-localized perturbations to the code.

Another way of looking at flexibility is the avoidance of arbitrary restrictions on interaction techniques. Restrictions should be controllable at a sufficiently high level that they can be dynamically changed. We shall see in the structure of our fluid dynamics world that we use the tracker for three different purposes depending on the user's current task; each task uses a subset of the available degrees of freedom.

How to achieve the last two requirements of VWs, modular decomposition and dynamic flexibility, is the focus of this paper. It is true that the UIMS gives benefits of simplification and economy, as does a UIMS in desktop applications. The dialogue can be written at a conceptually higher level, independent of device details; each rule describes the limited action to be taken upon receipt of a specific event. Yet it is still possible for the dialogue itself to grow large and complex; our first dialogues for simple worlds reached hundreds of lines. Thus we must still pay close attention to modularity. A certain rule-writing discipline and structure needs to be followed for the best results.

Previous Work

Jacob [1986] viewed the modular decomposition of a dialogue as a way to subdivide and encapsulate *state*. He defined an interaction object as “the smallest unit in the user interface that has a state that is remembered when the dialogue associated with it is interrupted and resumed.” The advantage of this decomposition was simplicity; each such object could be clearly described with a single-threaded state diagram. This concept of encapsulated state was helpful to us, although the particular style of UIMS in his illustrations was based on transition diagrams rather than an event model.

Foley et al. [1990] discussed the distinction between the meaning of a language and its form. They further distinguished three conceptual levels of design: functional or semantic design

which embodies the meaning, sequencing or syntactic design which embodies the form or style, binding or lexical design which establishes the interface to hardware primitives.

Similarly, Wiecha et al. [1989] emphasized the separation of *content* from *style*. We will see that this separation is a natural result of our structure. The *content* of our fluid dynamics world will be seen to be quite limited, embodied in only three rules in our dialogue; the remaining rules manage the *style* of using multiple simultaneous devices. In fact the style component of a virtual world is complex enough that an even finer structure is needed.

Description of the Vortex World

To provide illustrative examples for our subsequent discussion, we will use the dialogue we developed for our fluid dynamics world. It has been a good test-bed for the structuring of complex dialogues, since it combines four input devices — speech, tracker, glove and virtual sliders — and three output devices — speech, sound, and graphics.

We originally built a tool for visualization of fluid dynamics data obtained from a simulation developed at Rutgers University [Gu et al., 1990]. The subject of the simulation is a pair of fluid vortices at right angles; as the simulation steps forward in time the vortices interact and eventually cross-connect. We visualize the data as surfaces consisting of all points of a given constant value (“isovalue”) of vorticity.¹ The initial state of the data, when visualized in this way, has the appearance of two orthogonal hollow tubes.

To this visualization tool we added real-world-like interactivity. Forming a fist gesture grabs the vortices and rotate them. A flat-hand “flying” gesture navigates around the vortices. The flying metaphor can be thought of as a camera-bearing satellite flying around a globe; the angle of the hand determines the angular velocity and the closeness of the hand to the screen determines the satellite's radius.

The user can issue spoken commands to turn on a spotlight mounted on the camera, to reset the camera to its home position, and to pause or resume the use of the hand as an input device.

A set of five sliders allows the user to adjust the experimental conditions — simulation timestep and surface isovalue — and the red, blue and green color components of the graphic objects. The user selects a slider by pointing at it, and grabs the slider's adjustment knob with a fist gesture.

We found frequent sound feedback to be very useful. The application's response to an adjustment in timestep or isovalue can take as long as ten seconds, so we play “thinking music” in the interim.² Other uses of sound feedback are a click on every change of hand gesture, a helicopter sound during flying, and a vocal “okay” in response to a spoken command. These seem to help minimize the perceived latency.

¹ The vorticity is the magnitude of the curl of the velocity. Loosely speaking, it measures the turbulence, or rate of change in flow velocity.

² Interactivity is not sacrificed during this thinking interval; the user can continue to fly, rotate and issue commands. This is a natural benefit of the concurrent architecture.

Dialogue Structure

Our first dialogue for the vortex world was written as one large rule set. In the normal course of iteratively improving the dialogue and bringing in new interaction techniques, we learned that *remappability* was a very useful goal. For example, the techniques for manipulating the color controllers was first developed with very simple prototype devices that directly filtered the glove data stream and returned a color value. The dialogue required significant revision when we made replaced the prototypes with graphic slider devices. Dialogue changes were needed for minor style changes such as the use of relative motion instead of absolute location pointing, or even the arrangement of the sliders horizontally rather than vertically.

We therefore changed the slider dialogue to work with more abstract non-glove-specific events such as SELECT, PICK, MOVE, COMMIT, and DESELECT. Another advantage of the new set of interaction primitives was a wider applicability. The interaction techniques for many other devices such as pulldown menus, selection lists, and pushbuttons could be described by the same primitives. The dialogue's mapping could be changed from one device to another as long as the interaction techniques were similar, and the devices produced the same type of results.

Thus remappability seemed to be the right paradigm to guide the structuring of the dialogue. We wanted to be able to reuse the same subdialogues for devices with similar interaction techniques. Going further, we wanted to be able to remap devices and techniques dynamically, letting the user switch without interrupting the world. This would provide a good basis for the dynamic flexibility needed in virtual worlds. Doing this required the rule set to be divided into several levels such that the interfaces between levels provided convenient places for identifying similarities in techniques and outputs.

Figure 2. Structure of rule sets

In the lowest level, a rule set forms the interface to a *specific* hardware or software device. In particular the rule set's function is to encapsulate low-level details, such as packing more than one datum into a single message, or waiting for acknowledgment before sending new data, or smoothing of jittery input streams.

The *generic* level accepts events coming up from the lower level and further transforms them into more general interaction techniques such as rotation, flying or color events. It also

transforms events coming down from the executive as needed to match the requirements of the lower level.

The *executive* level maps interaction techniques to actions. It handles higher-level synchronization and dialogue mode, e.g., a fist gesture means grabbing a slider bar if it occurs after selecting a slider, but means rotation otherwise.

The three levels are conceptual. There can be many separate rule sets belonging to a given level, and typically this is true in the lower two levels. A level might need further subdivision in some cases, depending on how closely the device or technique matches the desired interface to the next level.

Clearly an additional benefit of this structure is reusability. If a device or technique has been given a generic enough interface to allow easy substitution, then it will be just as easy to reuse. Remappability and reusability are equivalent as guidelines for dialogue decomposition.

Glove Subdialogue. A sample rule set at the *specific* level is the following one for the glove. It encapsulates low-level details such as: the server reports the data for up to two gloves at once; even with a single glove, it might be either the right or left hand; the server performs gesture recognition and needs to be told which profile to use.

```
SET_GLOVE    { remember specified handedness;
               tell server specified profile };
QUERY_GLOVE { query server for data };
GLOVE        { split data into one-glove events }
               GLOVE1, GLOVE2;
GLOVE1       { determine whether gesture changed }
               iftrue gesture_changed
               GESTURE_CHANGED1;
```

The rules are represented schematically here. Pseudo-code in braces represents embedded semantics. Rules can produce new events, as represented by “ ”. We're omitting the rules for initialization and quitting which are commonly found in all the low-level rule sets.

The first two rules' left-hand sides, SET_GLOVE and QUERY_GLOVE, are events that typically “come down from” higher-level rule sets.

The GLOVE event typically “comes up from” the glove server; that's how the server replies to the query for data.

The last rule produces a new event GESTURE_CHANGED1 only if GLOVE1's gesture changed. The rule filters out gestures of brief duration which the user sometimes forms unintentionally on the way from one gesture to another.

The corresponding rule set in the *generic* level is the following, which transforms the separated and smoothed GESTURE_CHANGED1 event into a generic one-of-many device. The gesture fans out into one of seven mutually exclusive alternatives.

```
GESTURE_CHANGED1 { compare to gesture table }
                  oneof
                  POINT, TWOPOINT, FLAT, FIST,
                  THIRD, TWOTHIRDS, or THUMBOUT;
```

Finally, at the highest level, our dialogue maps these events to actions.

```
FLAT          START_FLYING;
START_FLYING { set mode = flying };
```

The highest-level dialogue has internal state represented by the *mode* variable; this state is encapsulated and not shared with any other rule set. A flat gesture starts flying mode; this causes the dialogue to select a different interpretation for the tracker device, as will be described next.

Tracker Subdialogue. A good illustration of dynamic flexibility is our tracker dialogue. The tracker's data stream is interpreted in one of three ways, depending on the current mode. The specific device interface remains constant, but there are three generic rule sets operating in parallel.

The lowest level rule set is specific to our server which can report data from up to four trackers at once; for the sake of illustration we assume only a single device here.

```
QUERY_TRACK { query track };
TRACK       { server data... no action required };
```

The TRACK event contains six floating point numbers: three positions (still expressed in the tracker's source coordinates) and three orientation angles. In our simple one-device example, TRACK requires no action at this level.

The next higher level contains three parallel rule sets. The first transforms the tracker into a generic device reporting two angles and a position, as needed for *theta*, *phi* and *rho* in our satellite flying model.

```
Tracker to Flying:
TRACK   { compute hand angles with respect to
         x and y directions; combine the two
         angles and z position into FLY event }
        FLY;
```

```
Tracker to Rotating:
TRACK   { transform the three angles to world
         space and put into ROTATION event }
        ROTATION;
```

```
Tracker to Slider Manipulation:
TRACK   { compute relative motions in x,y plane }
        DELTA_PLANE;
```

Note that the last rule reduces the tracker's six degrees of freedom to two; the tracker is essentially being transformed into a mouse.

All three generic devices are always active. It is only at the highest level that we dynamically choose one of their inputs.

```
DELTA_PLANE iftrue mode == manipulating_sliders
            MOVE_SLIDERS;
ROTATION    iftrue mode == rotating
            RENDER_ROTATE;
FLY        iftrue mode == flying
            { map angles and position to theta,
              phi, rho; convert to viewpoint }
            VIEWPOINT;
```

Application Subdialogue. A final illustration is the rule set that expresses the content of the original physics application. The application at the lowest level consists of a simulation process which supplies a set of data depending on the specified timestep. As was shown in Figure 1, The data are sent to a second process (actually a pipeline of three processes) which maps the data to a triangle mesh suitable for rendering. The data are passed directly from simulation to mapper to renderer; there is no point to routing such a low-level data stream through the UIMS when its destination is fixed and we have no need to derive abstractions from it for higher-level decisions. The information we do need is sent as events, e.g. MAPPER_GOT_DATA from the mapper to signal that it has the latest data and is ready to accept an isovalue. This synchronization decision represents the state in this subdialogue. The low-level rule set is:

```
SET_TIMESTEP      { send timestep to simulator };
SET_ISOVALUE      { if mapper has received the
                    latest simulation data
                    then send isovalue, else save it };
MAPPER_GOT_DATA   { send isovalue to mapper };
```

The application *content*, when fully separated in this way, comprises a very small part of the dialogue. Also notice that the application is treated like any other concurrent “device”; its behavior is coordinated by the executive level.

Conclusions

We have built a software framework for virtual worlds, based on a hierarchy of a distributed message-passing architecture, with multiple processes communicating and synchronized through a central event-driven, rule-based UIMS. The behavior of the virtual world is then specified by a dialogue composed of a number of modular subdialogues or rule sets. We've found that the way in which the dialogue is subdivided, the structure of the rule sets, is important for the sake of ease of development and dynamic flexibility.

The rule sets should be written as independent modules, in the sense that each one should encapsulate its own state and should not share the state of any other rule set.

Each rule set should be designed with a limited objective as appropriate to its level in the dialogue hierarchy: a rule set can transform a specific device into a generic device, or transform a generic device into an interaction technique, or, at the top level, map interaction techniques to actions.

The guiding principles and acid tests for this discipline are dynamic remappability and reusability. Any specific or generic device should be easily substitutable for any other that supplies the needed inputs or accepts the given outputs.

Acknowledgments

We would like to thank the members of our department, Veridical User Environments, for their invaluable contributions and support.

References

- Brooks, F.P. Grasping Reality through Illusion. *Proceedings ACM SIGCHI*, 1-11, 1988.
- Wang, C.P., Koved, L. and Dukach, S. Design for Interactive Performance in a Virtual Laboratory. *Computer Graphics*, Snowbird, Utah, March 1990. (Proceedings 1990 Symposium on Interactive 3D Graphics)
- Rhyné, James R. Extensions to C for Interface Programming. *ACM SIGGRAPH Symposium on User Interface Software*, Banff, Canada, October 1988.

- Jacob, R.J.K. A Specification Language for Direct-Manipulation User Interfaces. *ACM Transactions on Graphics*, 5(4):283-317, October 1986.
- Foley, J.D., van Dam, A., Feiner, S.K. and Hughes, J.F. *Computer Graphics, Principles and Practice*. Addison-Wesley, 1990.
- Wiecha, C., Bennett, W., Boies, S. and Gould, J. Generating Highly Interactive User Interfaces. *Proceedings ACM SIGCHI '89*, 277-282, 1989.
- Gu, Z., Silver, D. and Zabusky, N.J. 3D Visualization and Quantification of Evolving Amorphous Objects. *IEEE Proceedings of Visualization 1990*, 1990.

Copies may be requested from:

IBM Thomas J. Watson Research Center
Distribution Services F-11 Stormytown
Post Office Box 218
Yorktown Heights, New York 10598