October 29, 1998
RT0284
Engineering Technology 6 pages

Research Report

Event Processing For Complicated Routes In VRML 2.0

Masaaki Taniguchi

IBM Research, Tokyo Research Laboratory
IBM Japan, Ltd.
1623-14 Shimotsuruma, Yamato
Kanagawa 242-8502, Japan

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

Limited Distribution Notice
This report has been submitted for publication outside of IBM and will be probably copyrighted if accepted. It has been issued as a Research Report for early dissemination of its contents. In view of the expected transfer of copyright to an outside publisher, its distribution outside IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or copies of the article legally obtained (for example, by payment of royalities).

Event Processing For Complicated Routes In VRML 2.0

Masaaki Taniguchi*

Tokyo Research Laboratory, IBM Japan

Abstract

VRML 2.0 allows a content creator to dynamically control state changes in a virtual reality world by defining routes of events over VRML 2.0 nodes.

In the conceptual execution model defined in the VRML 2.0 specification, an event should be delivered to its destinations instantaneously. However it makes browser implementation difficult in routing connections that involve complicated requirements such as simultaneous events in multiple fields of a node, or cyclic dependencies between nodes.

This paper describes an event processing method we have implemented in our VRML browser, which is designed to handle complicated route connections.

CR Categories and Subject Descriptors: I.3.6 [Methodology and Techniques] Graphics data structures and data types; I.3.7 [Three-Dimensional Graphics and Realism] Virtual Reality.

Additional Keywords: VRML, ROUTEs, event notification.

1 INTRODUCTION

One of the most important features of VRML 2.0 is the capability it provides for dynamically changing the state of the scene graph. This capability is made possible by events and routes, which are among the major advances over VRML 1.0.

An initial event, which can be generated when the user interacts with an object, or when a specified time has elapsed, is sent to subsequent nodes through route connections. Nodes receiving events may respond by generating other events. This flow of events is expressed by a directed graph, which we refer to as a route graph in this paper.[1]

In the ideal event model, events are propagated instantaneously. However, in a real implementations there are the following considerations:

1. unpredictability of event generation by a node,
2. multiple eventIns,
3. loop prevention,
4. cyclic dependency,
5. direct outputs.

This paper describes the event-processing method used by our VRML 2.0 browser which addresses these considerations.

There are two basic approaches that a browser may take to process events over the route graph.

The first approach is to propagate an event upon its generation following the direction of the event flow, which is defined by the ROUTE statement.

The second approach is not to propagate events until they are required by receivers. This can be achieved by traversing the route graph in reverse until the source of an event is reached.

Daniel J. Woods et al. described a pull-and-push model, which is a combination of these two approaches [4]. A message is processed in two phases, the first of which is the notification phase. When a field value is updated, the browser merely invalidates fields in its route graph. The second phase is the transmission phase. The recipient requires valid data, which is obtained from the source of an event. The design of this model is intended to allow very efficient processing of events, but their paper did not describe how the browser handles more complicated cases.

Unlike the method presented by Woods et al., this method propagates events according to the first of the two approaches mentioned above. In order to propagate events, first the route graph is examined. Then a feasible order of node evaluation is generated, and events are propagated in that order. This approach exploits some fundamental graph algorithms that can be effectively used with complicated route connections.

The next section briefly describes the basics of event-processing. In section 3, the design considerations of the event processing implementation is discussed. Section 4 explains the event-processing method, and section 5 details our implementation. Section 6 gives a discussion on the VRML Script Clarification Working Group's resolutions relating to this implementation.

All discussions made by the present paper is based on the VRML 97 specification [1].

*1623-14 Shimotsuruma Yamato, Kanagawa, 242, Japan taniguti@trl.ibm.co.jp

[1] It is also called the event cascade in the VRML 2.0 specification.

2 EVENT-PROCESSING BASICS

Nodes are the fundamental components of the VRML 97 specification. Each node has a certain number of fields, which express its properties or attributes.

Fields that can receive events are called eventIns, while fields that can send events are called eventOuts. Fields that have properties of both eventIns and eventOuts are called exposedFields. ExposedFields can be treated as either eventIns or eventOuts, and are not discussed in this paper.

A route is a connection established from an eventOut field to an eventIn field, where both the field types match exactly.

The value of a field is sent through a route as an event, which is then evaluated by the receiving node and may result in the generation of other events. Event propagation proceeds in this manner. In the conceptual definition of the event execution model given in the VRML 97 specification, events generated during their propagation are considered to occur instantaneously, and a browser implementation needs to simulate this. Without serious consideration, however, it might lead to unexpected results.

3 DESIGN CONSIDERATIONS

Realization of event processing by a browser involves several considerations, which will be discussed in this section.

3.1 Unpredictability Of Event Generation By A Node

Events received by a node will be evaluated in order to change the status of that node and/or to generate additional events.

How a node responds to a received event depends on the type and status of the node, as well as the message and timestamp of the event, and it is generally unpredictable until the event is evaluated.

A Script node is a good example of such unpredictability, since it is difficult for a browser to predict how the scripting language will evaluate events.

A TimeSensor node is another example. It generates a fraction_changed eventOut only when the condition occurs such that startTime is less than or equal to the current time, the current time is less than or equal to stopTime, and enabled is TRUE.

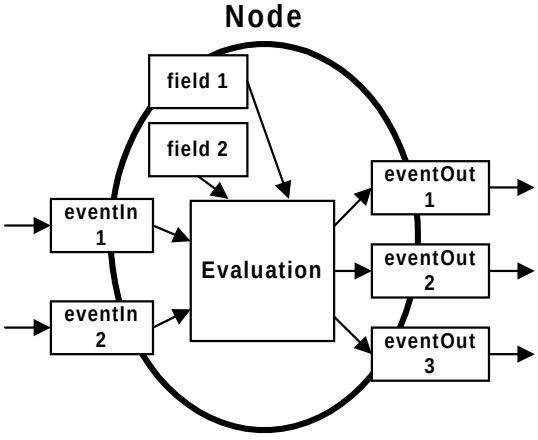

Figure 1: Schematic of event generation by a node.

3.2 Multiple EventIns

In the ideal event process definition, all events are propagated instantaneously. This allows a node to receive different kinds of events at the same time within the same event cascade.

This is not trivial if the paths of the event flow are different. For example, suppose there are three nodes, A, B, and C. Route connections are established as follows:

```
ROUTE A.out1 TO C.in1
ROUTE A.out2 TO B.in
ROUTE B.out TO C.in2.
```

If node A generates events from eventOut out1 and eventOut out2 at the same time, then node C will receive multiple events at eventIn in1 and eventIn in2. In that case, node C should be evaluated based on values from multiple eventIns, thus node B should be evaluated prior to node C. Therefore the browser implementation must take into account the route graph topology, so that the evaluation produces a unique result.

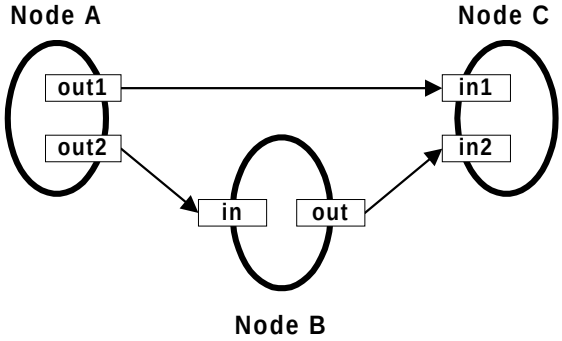

Figure 2: Example of multiple events being sent to a node simultaneously.

3.3 Loop Prevention

The VRML 97 specification defines the condition for the occurrence of loops, and specifies how a browser can break loops. A browser can prevent loops by checking fields to ensure that fields are not sending events with identical timestamps. Because of the way in which a node responds to the events it receives, whether an event loop is present or not can not be confirmed from the route graph topology, and a browser is required to detect loops at run time.

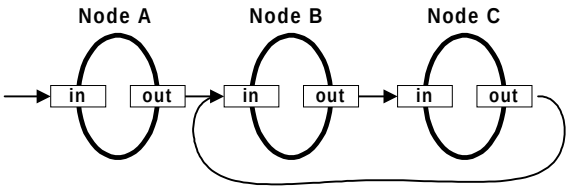

Figure 3: Example of route connections in which a loop exists at node B.

3.4 Cyclic Dependency

Routes can be connected in such a way that a cyclic dependency is generated between nodes.

Loop-preventing rule does not prevent a node from being evaluated again, since a node can respond through a different eventOut.

For example, three nodes A, B, and C have the following route connections:

```
ROUTE A.out1 TO C.in
ROUTE A.out2 TO B.in
ROUTE B.out TO A.in2
```

If an event is processed in this order:

```
A.in1 -> A.out2 -> B.in -> B.out
-> A.in2 -> A.out1 -> C.in -> C.out
```

then it is not a loop, since it does not visit the same field twice. However, in this case, there is a cyclic dependency between nodes A and B, since the evaluation of node A depends on the evaluation of node B, and vise versa.

Cyclic dependency is inconsistent with multiple eventIns. In the previous example, node A needs to send an event through eventOut out2 before it receives an event at eventIn in2, therefore multiple eventIns is not possible.

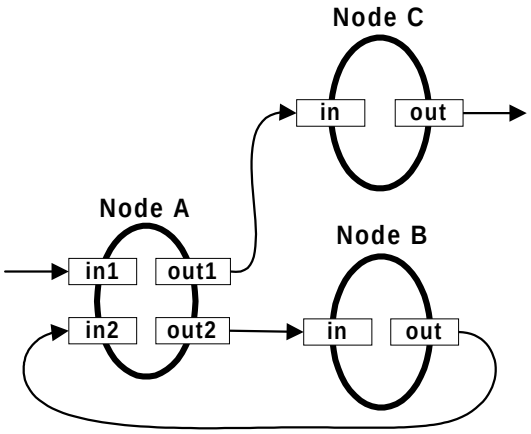

Figure 4: Example of route connections in which a cyclic dependency exists.

3.5 Direct Outputs

A Script node has direct outputs when its directOutput field is TRUE, at which time it can access other nodes and establish or destroy route connections or send events directly to their eventIns without establishing routes. Processing events which involves direct outputs is difficult, since it results in dynamic event cascade modifications at the event propagation stage. Such modifications are unpredictable and it is generally not possible to determine an event evaluation order in advance. Therefore special treatments are needed to deal with direct outputs.

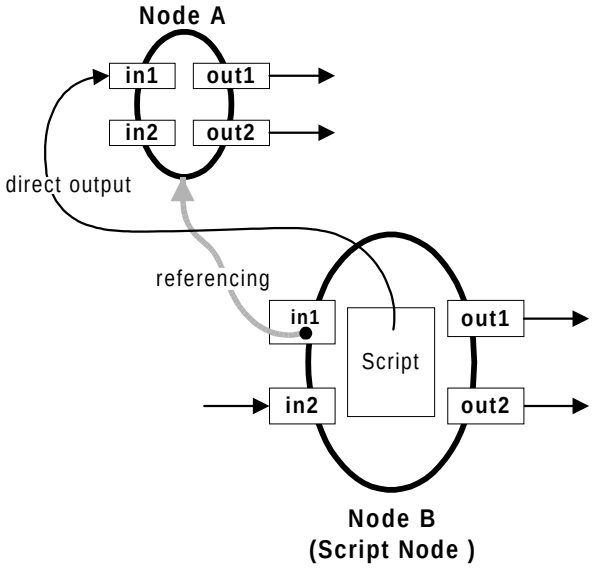

Figure 5: Schematic of direct output, where the Script node B sends an event to node A without an established route connection.

4 EVENT-PROCESSING METHOD

The event-processing method that we implemented in our VRML 2.0 browser propagates events as they flow through the event cascade. This is not only consistent with the event execution design of the VRML 97 specification, but is also suitable for propagating all events generated over the route graph.

However, processing all events is thought to be inefficient. The main purpose of event processing is to update the values of fields. With our browser implementation, time-consuming executions of invisible objects such as MovieTexture and AudioClip will be avoided by viewing frustum culling. Moreover, we can expect that not all of the event network will be active. There are ways to avoid generating unnecessary events. For example, a ProximitySensor node can be used to avoid generating events when objects are viewed from a distance, and a TouchSensor node can be used to avoid generating events until they are invoked by a viewer. As a result, even without any optimization of event processing, we anticipate that the processing cost will not be high.

For events to be processed, nodes in the route network need to be evaluated. Since we can evaluate one node at a time, we need to serialize the evaluation of nodes in such a way that it is not inconsistent with the route connection semantics; for instance, certain nodes should be evaluated before other nodes. In the example described in section 3.2, node B should not be evaluated after node C, since an event generated from node B will not be handled unless node C is evaluated again. However, evaluating node C twice might give different results.

To avoid such situations, the event processing is done by carefully determining the evaluation order in advance, then evaluating every node in that evaluation order. This simulates instantaneous event propagation.

4.1 Determine The Order Of Evaluation

If a graph does not contain loops or cyclic dependencies, it is called a *directed acyclic graph* (dag) .

Assuming that the route graph is a dag, we can obtain the order of the evaluation by topological sorting. The result is ordered in such a way that all nodes are evaluated after they receive all eventIns from the route graph.

Unfortunately, route graphs are generally not dags, since they contain loops and cyclic dependencies. We therefore need a technique that allows us to treat them as dags.

Strongly connected components are sets of nodes in a directed graph with the property that all the nodes in a set are mutually accessible. In other words, nodes inside a loop or nodes that create a cyclic dependency are strongly connected components.

Figure 6 illustrates how the cyclic dependency in the route graph is removed by dividing the graph into components. Since nodes A and B have a cyclic dependency, they are in the same component: component 1. Note that a node itself is a component. Therefore, node C is the only element of component 2. It is easy to see that component 1 should be evaluated before component 2. The order of evaluation will be:

```
Node A -> Node B -> Node A -> Node C.
```

However, if we apply topological sorting without separating nodes into components, the evaluation order

```
Node A -> Node C -> Node B -> Node A
```

will also be a valid topological ordering. In this case, an event will not propagate properly.

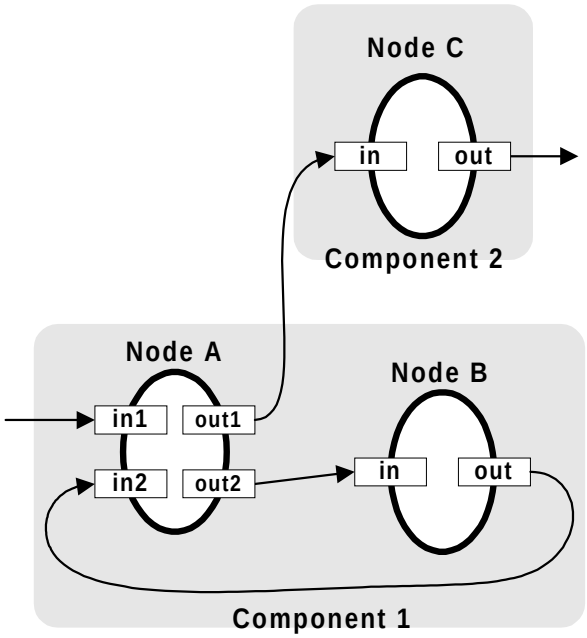

Figure 6: Strongly connected components generate a dag from a route graph.

To obtain the order of evaluation, the route graph is first separated by detecting the strongly connected components. The derived set of route connections among the components is a dag. Next, the order in which the components are processed is obtained by topological sorting. Finally, the evaluation order inside the components is decided.

4.1.1 Generating A Dag

The first step in deciding the order of evaluation is to divide a route graph by the strongly connected components. The route graph formed by the components is a dag.

The strongly connected components in the route graph are obtained by the algorithm proposed by R. E. Tarjan in 1972 [3]. Since the algorithm is a variant of depth-first search, the computation cost is linearly proportional to the number of nodes and routes in the route graph.

4.1.2 Sorting Components

Once a dag has been created, the next step is to obtain the evaluation order of the components, which is done by applying topological sorting to the components.

The topological sorting algorithm is also a variant of depth-first search. The computation cost is linearly proportional to the number of components and the number of routes connecting the components.

4.1.3 Sorting Inside A Component

The final step in deciding the order of evaluation is to sort the nodes inside a component, so that the order will yield a correct result when the component is evaluated.

As explained in Section 3.1, a node is unpredictable with regards to event generation. In general, there are many possible orders of evaluation.

Figure 7 shows a more complicated case of the example in Figure 6. There are two node evaluation orders. In most cases, node A generates events from eventOut out2 and eventOut out3 when it receives an event at eventIn in1. Then, node B generates events from eventOut out1 and eventOut out2. Finally, node A generates an event from eventOut out1. The node evaluation order inside the component 1 is

```
A -> B -> A.
```

However, in some cases, node A will only generate an event from eventOut out2. In this case, one expected event flow is:

```
A.out2 -> B.in1 -> B.out1 -> A.in2 -> A.out3
-> B.in2 -> B.out2 -> A.in3 -> A.out1 -> C.in.
```

The node evaluation order is

```
A -> B -> A -> B -> A.
```

In general, it is impossible to decide a single ordering before evaluating nodes, since the order may vary from case to case.

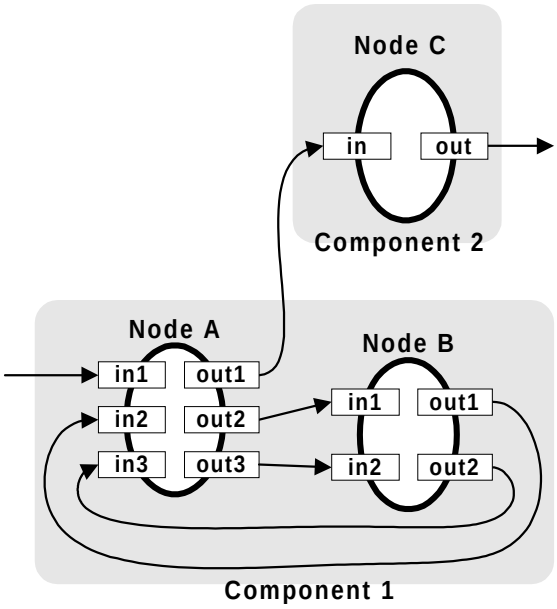

Figure 7: Example of complicated cyclic dependency.

One solution to the problem is to decide the evaluation order by topological sorting, which gives a fair result. This process is repeated until no node remains to be evaluated.

In the previous case, the topological sorting of the nodes inside component 1 is

```
A -> B.
```

The process will repeated two or three times, depending on the events generated by nodes A and B.

4.2 Propagating Events

If the route graph is separated into strongly connected components, events are propagated in an organized way by evaluating its components. A component is evaluated according to its internal nodes.

The order in which components and nodes are evaluated has already been decided. As explained in section 4.1.3, evaluation of nodes inside a component is repeated until there are no more nodes to be evaluated. Infinite loops are avoided by two event-processing strategies. One is to apply the same rule as for avoiding loops in event propagation, which was described in section 3.3. A node cannot send more than one event through the same eventOut. The second strategy is to skip node evaluation if the node does not receive any eventIns.

5 IMPLEMENTATION

To improve the performance, evaluation orders are created the first time evaluation of a route graph is requested, and are then repeatedly used until a change is made to the graph.

The evaluation order is generated before the browser actually evaluates events. The order is created on the assumption that a node generates all eventOuts when it receives an event. This may result in unnecessary nodes being included in the order. To minimize unnecessary evaluations, every node has an modification flag to indicate whether it has actually received events.

Given that direct outputs modify the route network dynamically, and since this method does not handle dynamic modifications, they are handled in another way. Requests to modify route connections made during the route graph evaluation are stored in a queue, and then processed after the evaluation is finished.

Similarly, events directly sent to other nodes by scripts are handled as different event cascades.

In addition to a timestamp, an event ID, which is a unique number to the initial event, is propagated over the route graph. Instead of using a timestamp, that ID is used for loop detection, since the time information obtained from the operating system may not be precise enough to distinguish between different event propagation.

A prototype is a declaration of a new node type composed of VRML 97 built-in nodes. Fields of a prototype are used as interfaces for access to internal node fields, which is enabled by the IS declaration. In our implementation, an IS is enabled by a pointer reference, and fields of a prototype are used to store values. The evaluation order when the route network includes prototypes is determined in such a way that it includes nodes inside prototypes.

6 DISCUSSION AND CONCLUSION

The method proposed in this paper provides a feasible event-processing implementation that allows users to create complicated route connections. This method properly handles route connections which include multiple eventIns and cyclic dependencies, by first determining the evaluation order and then propagating the events accordingly.

After the VRML 97 specification was released, the VRML Script Clarification Working Group was formed [2]. They have been discussing about script and event related issues. Among the issues that the group has reached conclusion on, “Timestamp Event Evaluation Ordering,” and “Fan In Clarification” are related to event processing implementations and are discussed here.

The problem regarding the “Timestamp Event Evaluation Ordering” issue was how events should be propagated when the propagation results in the modification of the route graph, such as addition and removal of a route connection, a node creation and deletion. The resolution they have reached defines the order of handling tasks which can not be included in a event cascade evaluation. As a result, a route graph will not be modified during the graph evaluation. Addition and removal of route connections are handled after the graph is evaluated. This is consistent with our event processing implementation. As mentioned in the previous section, our method decides the order of nodes evaluation prior to the start of event propagation. Therefore changes to the route graph will not be handled during the graph evaluation. The method can handle those requests after the graph evaluation is finished. This corresponds to the way the resolution handles the modification of a route graph.

The problem of regarding the “Fan In Clarification” issue was that handing of more than one event received by a field within the same timestamp was not clearly specified. Whether all the events received are used by the node evaluation or not varies from one implementation to another. The Scripting WG resolution is that a browser needs to process all fan-in events. Our method resolves dependencies among nodes in the route graph by topological sorting. It assures that all events will be sent to a field before the node will be evaluated. Therefore, our implementation complies to the fan-in handling described in the resolution .

7 ACKNOWLEDGEMENTS

This work was supported by grants from the Information-Technology Promotion Agency, Japan (IPA).

I would like to thank Akio Koide, Tatsuo Miyazawa, Ryo Yoshida, Takaaki Murao, and Michael McDonald for their advice and for reviewing of this paper.

References

[1] “The Virtual Reality Modeling Language Specification,” ISO/IEC DIS 14772-1 April 4, 1997. Available as “http://www.vrml.org/Specifications/VRML97/DIS/”.

[2] “VRML Script Working Group Clarifications,” Available as “http://www.vlc.com.au/~justin/vrml/script-wg/”.

[3] Robert Tarjan, “Depth-First Search And Linear Graph Algorithms,” SIAM Journal on Computing, Vol. 1, No. 2, June 1972, pages 146-160.

[4] Daniel J. Woods, Alan Norton, and Gavin Bell, “Wired For Speed: Efficient Routes In VRML 2.0,” in proceedings of VRML 97, pages 133-138.