Editing VRML Worlds in Multi-User Environment

Jaroslav Dorňák¸ Jiří Žára
cim@artax.karlin.mff.cuni.cz
zara@fel.cvut.cz

Faculty of Mathematics and Physics
Charles University
Prague / Czech Republic

Abstract

Existing multi user VR environments (MUE) are primarily aimed to replicate events generated by user interaction within formerly single virtual environment (VE) to other users in the same environment thus making a 'shared VR' feeling. These VE must be written in a specific way to allow such behavior. This paper presents a set of methods allowing users in MUE to cooperatively change/edit their environment without the need to have these actions already pre-defined in file describing VE. Our approach is targeted to VRML worlds.

KEYWORDS: VRML, editing, MUE

1 Introduction

Being a user in MUE means that we can perceive other users' presence and actions. We can interact with the VE in a predefined way that is given by VRML file defining this world. Creators of MUE often implement number of interactive features to make their VE satisfying needs of most users. Unfortunately it's not possible to presume *every* possible user actions and of course writing support for these actions would result in VE with size far beyond reasonable. Sometimes users want to go one step further than MUE allows them to go and they want to edit their VE in the ways that are not implicitly supported. In this paper we focus on methods allowing users to change their environment directly without the need to have special constructs prepared by the creator of the VE.

We present experimental system allowing users to edit arbitrary VRML worlds with predefined set of tools that are not part of the predefined environment. In section 2 we state requirements and features of such system. Section 3 describes system architecture while section 4 describes current implementation of the system.

2 System requirements and features

Objects in VE that our system allows to edit directly must have unique identification and a user must be provided with facility to choose such objects and with a set of tools that allows him/her to make desired changes to these objects.

User interaction in VE is performed via standard way of using predefined VRML [1] *sensors*. When we want to allow direct changes to the environment, we have to provide user with our own set of tools/sensors that are created dynamically and that are suitable for performing specific tasks. While standard interaction in VE results in actions that were already considered by its creators, direct editing can result in unpredicted results so we have to make sure that we won't break consistency of such environment by user's actions.

Facility that broadcasts changes made by users to other users in VE and that performs multiple user cooperation in VE must also be provided. We have to bear in mind that some users may connect into session later and we have to provide them with the VE that contains changes made by other users before.

Last we want to keep changes to the VE stored in some way to allow user to make a break in editing session and to continue in this session later.

3 Architecture

As we stated in this paper's title our system is intended to be used with VE defined using VRML. This allows us to use existing technologies for the development - WWW browser with VRML plugin is used for the VE representation and Java cross platform language communicating with the VE through the EAI [2] is used. Resulting system is thus virtually platform independent.

VRML plugin performs visualization of the VE and also captures user's interaction with the VE. It hosts VRML parts of tools that allow user to edit his/her environment. Java applet takes care of communication issues. It also contains Java parts of tools and a part responsible for scene updates through the EAI.

Implementation of system for cooperative editing of the VE has to address these issues:

1. User interaction with the VE
2. Identification and access to VE objects (VRML nodes)
3. Editing of node properties
4. Network layer: broadcasting/receiving informations about changes in the VE and concurrency control
5. Connecting/reconnecting user into session

3.1 User interaction in the VE

System is designed to work with VE created from VRML files. VE defined by VRML file consists of a multi-tree of nodes. Each node is of predefined type but it's also possible to create new types of nodes using either a *Script* node or a *PROTO* construct.

Each node defines a set of properties that can be divided into two categories based on their structure:

- single field value (SFField) - a property represented by a single value of some type
- multiple field value (MFField) - a property represented by an array of SFFields

Such properties can be set/read through VRML mechanism called *eventIn*(s) and *eventOut*(s). Unfortunately not all of node properties can be read and set. Some properties can be only set while others cannot be changed at all. This makes our task a bit more complicated because sometimes it's not possible to determine enough information about a node just by simply peeking at it's *eventOut*(s).

We change the VE by operating VRML nodes that define it. It also means that our actions in the VE are limited by limitations of those nodes.

User interaction in such environment is done via *sensor* nodes. These are special kind of nodes designed to react either when some property of the VE changes (*environment sensors*) or when a user operates such sensor in a predefined way (*pointing-device sensors*). When a user operates a pointing-device sensor an eventOut describing the change is created. It is propagated through the VE by *ROUTE*(s) that connect operated sensor with nodes that are listening for its changes.

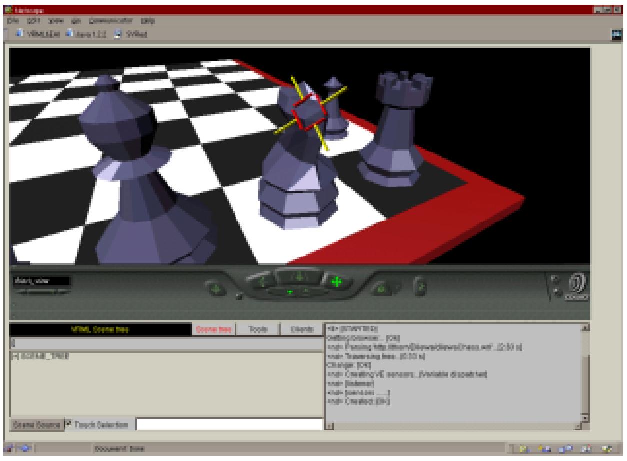
Direct editing of VE node means that we have to create and ROUTE events such that *would be created* if there was already support for this kind of action. These events have also to be propagated through network to the other users in the same VE. This will allow a user in a system to see changes made by other users. Using Living Worlds [3] vocabulary we call the edited node *pilot* and nodes replicating its behavior across network its *drones*. Because there are more users working in the same VE some kind of concurrency control has to be implemented to exclude conflicting user actions. This facility is described in sections 3.3 and 3.4.

3.2 Identification and access to VE objects (VRML nodes)

We will focus on what we call *simple nodes* in terms of plain VRML nodes that are not implemented by a *Script* node or by a *PROTO* construct. This allows us to determine exact type of such node and to provide user with set of tools that would allow him/her to make appropriate changes to such node.

Only named nodes (using *DEF* construct) can be directly obtained from VRML world through the EAI. To be able to access whole scene tree we have to load the scene as a child of some node (called ROOT) that we already have obtained through EAI. This way through ROOT's properties we have got access to every node in scene. Scene tree is then traversed and every node that interests the system is given an ID that is unique among other nodes in VE. This ID is later used for identification of this node.

Knowing world hierarchy we can insert into the VE some kind of virtual handles that allow a user to select objects in the VE. This is accomplished by using a *TouchSensor* for the selection of a node and a special visual symbol informing that the system supports specific operations on such node.



```
<b> [STARTED]
Getting browser... [OK]
<nd> Parsing http://[illegible]Chess.wrl [2.53 s]
<nd> Traversing tree [0.33 s]
Change: [OK]
<nd> Creating VE sensors...[Variable dispatcher]
<nd> [listener]
<nd> [sensors .....]
<nd> Created [OK]
```

Figure 1: Selection of node in VE

3.3 Editing VRML nodes

Nodes selected by a user can be edited by a set of predefined tools that are designed for specific tasks. A tool basically consists of VRML construct (PROTO) that combines sensors and other nodes into a virtual device that allows user to operate some properties of the node in the VE. Java part of the tool implements complex logic and ensures propagation of events through network. It also solves concurrency conflicts. As we stated before not all properties of nodes can be determined through the EAI. To be able to perform more complex tasks we had to include in the system a VRML parser that gives us more detailed information about edited nodes.

3.3.1 User cooperation conflicts

Although we want to allow a rich cooperation in editing of the VRML worlds, in some cases we have to exclude concurrent access to node properties to prevent inconsistency in the VE. This task is accomplished by a system of locks on node properties. Some tools have to obtain lock on a specific node property before they can change it.

Example 1: Concurrent work

SFField example: When two users change a *diffuseColor* field of a *Material* node they can both change it without risk of inconsistency in the VE and without the need for a lock on this property.

MFField example: Two users changing a *point* field of *Coordinate* node can also work simultaneously with assumption that they don't change number of elements in this field. Why they must not change the size of this field is described in the next example. If they would work on the same values in this field they would be simply destroying others work but this would not cause inconsistencies in the VE.

Example 2: Work with mutual exclusion A user changing geometry representation(number of vertices/faces) of an *IndexedFaceSet* node has to acquire locks on all properties that are dependent on changed property - that includes locking of color changes and so on.

Let's describe a situation that will explain the need for a lock in this example:

The VE contains an IndexedFaceSet that consist of one face with five vertices.

User (named *Client1*) is deleting one vertex from its face thus reducing number of vertices to four.

Another user (named *Client2*) is meanwhile changing color of the fifth vertex. When a Client2 is notified about the vertices number change the fifth vertex will simply disappear leaving his/her VE in consistent state.

Notification about the color changes of the fifth vertex made by Client2 will be inconsistent with Client1*'s* VE because in his/her VE the face has now only four vertices.

Solution:

Client1 has to obtain lock on vertex field and on all IndexedFaceSet properties that are dependent on it before he/she can change the number of face vertices. Client2 has to obtain lock on IndexedFaceSet color properties before he/she can change the color of the vertices. This assumption will assure that either Client1 will made his/her changes first and Client2 will then be able to change color only on four vertices that will remain in face or Client2 will change color of the fifth vertex and Client1 will then delete it.

3.4 Network layer

The network part of the system must accomplish several tasks:

- delivery of data through network to the other users
- solving concurrent user interaction conflicts

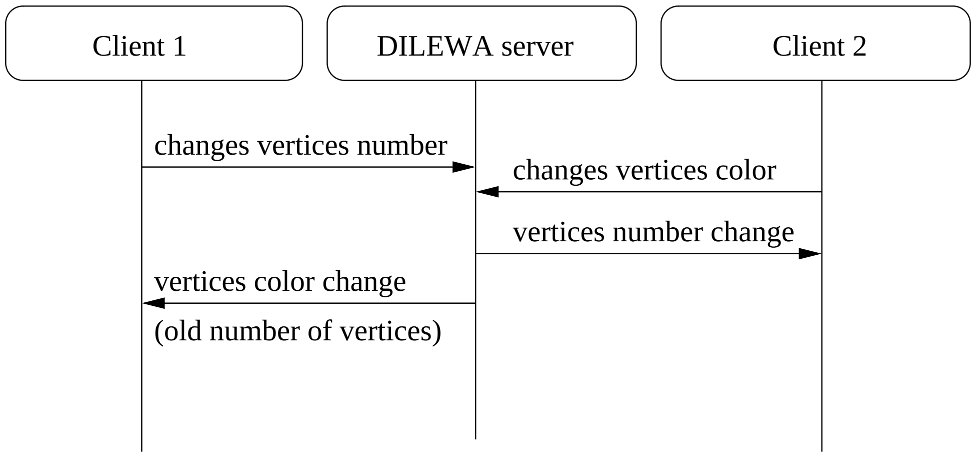


Figure 2: Concurrent work conflict example situation

The distributed editor has *client/server* network architecture. Server is an authority that is asked in questions of concurrent access. It also has all the informations about the VE up-to-date and provides newly connected clients with fresh data. Client allows VE editing and sends changes to server that propagates them to other clients.

As a *client* we consider WWW browser representing a user connected into MUE. The part of a *server* in the system takes DILEWA [4] system developed by M. Máša.

3.4.1 Delivery of data

In VE that would allow only interaction that is predefined in VRML file we could accomplish this task by inserting the *network routes* [3] in the middle of ROUTE(s) that route the events triggering scene changes. These network routes would deliver such events to corresponding destinations in other users' VE(s) thus making feel that these events were created there. Unfortunately this approach cannot be used in our system because there are no predefined ROUTE(s) in the VE delivering events triggered by user interaction simply because these routes are created dynamically and only on the side of the *pilot*. We solved this by naming every such event after its destination. When this event is delivered to the another user in the VE he/her can determine its destination simply by looking at its name.

Example Client1 is changing diffuseColor of a node with ID 41 (for node ID assignment see section 3.2) to value 1 0 0 (red). Event named *41.diffuseColor* with the value of 1 0 0 is created and propagated through network. When Client2 receives it he/her sends its value to destination determined from its name.

3.4.2 Solving concurrent user interaction conflicts

We have to provide facility to create and manage locks on node properties in VE.

We solve conflicting user interaction by a rule that a user (his/her tools) has to obtain locks on properties that may lead to the VE inconsistencies. Thus only one user is allowed to change such VE properties and the VE will be kept in consistent state.

3.5 Connecting/reconnecting user into session

A server holds all information about changes in VE. When some user looses a connection he/she is treated in the same way as a new coming user. Server sends him/her all the changes(in order of their time of arrival on server) that have been made in VE from the beginning of the session thus putting his/her VE in consistent state.

4 Implementation

This section describes main implementation issues of the system.

4.1 VE editing

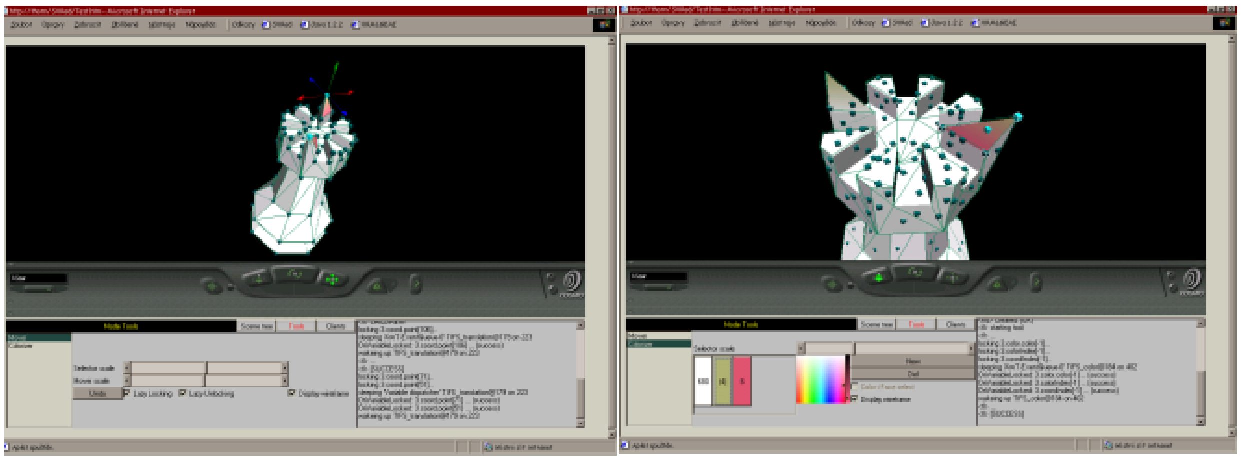


Figure 3: Example of concurrent VE editing session

We have created a set of tools that are dynamically inserted into user's VE and that connect themselves on desired nodes. When a user selects a node in VE a list of tools that are capable of performing specific tasks is offered in Java applet. These tools have to take care of proper node property locking to avoid conflicts.

A tool initializes itself from two sources of data. First part of needed information it gets from its VRML environment. But as we said in section 3.1 sometimes it is not possible to determine all information about nodes from VRML. Because of this we have included VRML parser in the system. This parser parses VE VRML file and provides tool with additional information (such as *colorPerVertex* property value of IFS).

When a user uses tool in the VE it generates events that are routed through EAI to its Java part. Java part sends these events to network to notify other users about the changes. It also sends them in a loopback back to user's VE thus visualizing changes produced by the tool.

4.2 Named net-variables

In cooperation with M. Máša, the author of DILEWA system, we introduce facility that can be called *named network variables.* This facility provides methods for creating of a named variable on the network for locking of such variable and for broadcasting of the changes of this variable. These variables are identified by a *name* and a number called *index.* A variable with an index equal to -1 represents a single value or the whole array. Index is used to allow representation of values in an array. A variable with name *name* and positive *index* represents a single value *name* with index *index* in an array.

Locking rules:

1. A single value can be locked if it's not locked by somebody else.
2. A single value in an array can be locked if it's not locked by somebody else or if somebody has not obtained a lock on the whole array (index equal to -1).
3. An array can be locked when there is no lock on any single value in the array or on the array itself.

This facility is used for distributing VE changes among users and for solving concurrency conflicts. When a user wants to change some property of the VE a lock on this property has to be acquired. Property is identified by node unique ID and property name.

SFField net-variables usage A SFField property uses a net-variable that has appropriate name for identification of this property. Variable index is always -1. Variable value is overwritten every time new value is acquired so this net-variable contains up-to-date state of this property.

MFField net-variables usage MFField properties use net-variables a bit differently that SFFields. Variable index is used to determine single values in array that this property represents. Variable value is appended by a new value every time the value is changed so this net-variable contains some kind of a *journal* of changes on this variable.

-1 index is used when we want to access the whole MFField. Given net-variable locking rules prevent acquiring of such lock when some user changes part of this MFField thus preventing possible inconsistency and collisions in the VE.

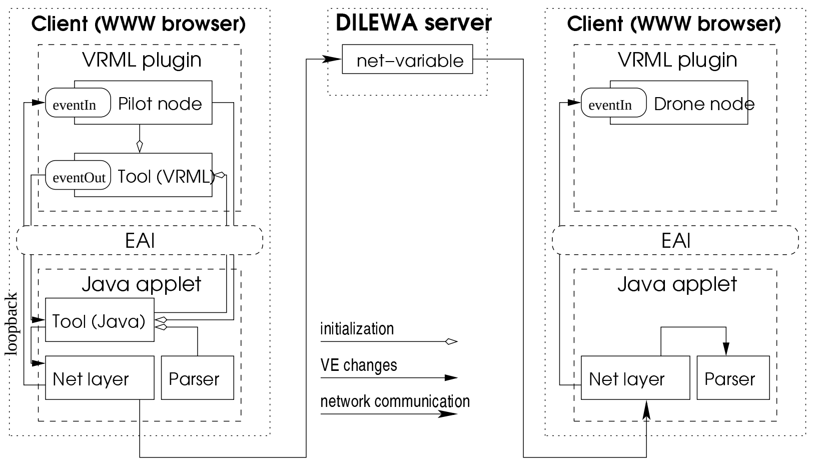


Figure 4: VE changes propagation scheme

4.3 World authoring

We have to address following issues: capturing user selection of nodes and locating of nodes that had to be changed depending on incoming data from network.

When a users connects into MUE session the system walks through scene hierarchy and creates internal tree of a nodes editable by system. Some of these nodes (i.e. *Shapes*) are extended by a *TouchSensor* thus making possible selection of these nodes directly in the VE. Nodes without visual representation (i.e. *Transform*) are not selectable through the VE but they are presented for selection by a Java applet part of the client in a form of a hierarchy tree.

Nodes that are part of system internal tree have unique ID(s) that are used to identify these nodes in VE. These ID(s) are used for sending information about changes on nodes and for dispatching of changes received from network.

4.4 Late coming users and work saving

When a new user connects into system or when some user loses a connection server sends him/her a fresh copy of the VE along with all the changes (net-variable values or respectively parts of net-variable values) made to the VE from the beginning of the editing session. These changes are sorted by the time of their arrival to server. Using this data a client reconstructs all changes to the VE as if he was a part of the MUE session from the beginning.

If we want to end editing session we have to save our changed VE. It's not reasonable to store net-variable values to save our work because that would limit usage of this system. Instead it would be better to generate VRML file that would represent VE in its current changed state. We already had to introduce VRML parser into a system. Every change to VE is propagated also into VE representation in parser so that it is able to generate VRML file depicting current VE state. This file can be then used as a base for the next editing session.

5 Conclusions and future work

We have presented possible implementation of a system that allows users to edit their VRML environment in MUE with predefined set of tools without the need to have a support for such actions written in VE VRML file. We addressed issues of selecting and manipulating objects in scene by more users simultaneously.

Future work concentrates on coexistence of this approach with DILEWA built-in cooperation [4] and on methods allowing to change the VE scene graph.

References

[1] "The Virtual Reality Modeling Language (VRML)", *International Standard* ISO/IEC 14772-1:1997, http://www.vrml.org/Specifications/VRML97/

[2] "The Virtual Reality Modeling Language (VRML) - External Authoring Interface", *Committee Draft* ISO/IEC 14772-2, http://www.vrml.org/Specifications/VRML97/

[3] *"Living Worlds"*, http://www.vrml.org/WorkingGroups/living-worlds/

[4] M. Máša : "DILEWA: The DIstributed Learning Environment Without Avatars", http://www.cgg.cvut.cz/DILEWA