Remote VRML browser control using EAI

Milan Kubec, Jiri Zara
<xkubec,zara>@fel.cvut.cz

Departament of Computer Science and Engeneering
Czech Technical University
Prague / Czech Republic

Abstract

The paper describes an implementation of experimental system, which allows multiple users to cooperate within one VRML world. The program is built up on architecture client/server. A Java applet is used for communication between several users connected by a network. The applet uses the *External Authoring Interface* (EAI) to influence/control a VRML world in a browser embeded in a HTML page.

Keywords: VRML, EAI, Java, Applet, client/server

1 Introduction

Recently we can hear words "collaborative systems" very often. It is such a system, where multiple users are engaged in particular shared activity, usually from remote locations. This kind of cooperation is very useful, e.g., two or more people in distant locations are able to develop or consult their work without travelling and wasting time and money. Up to now this kind of systems are mainly textually or 2D (drawings) oriented. Another approach is to allow users to share interaction in 3D. VRML and platform independent programming language Java were chosen to achieve this goal. The internet provides the communication background.

The aim of this project was to find out the way, how to remotely control VRML browser and particular parameters of the VRML scene, because it plays the main role in such collaborative systems.

The structure of this paper is organized as follows. The *External Authoring Interface* is discussed in Chapter 2. The implementation of the experimental system is described in two following sections. Applet implementation in Chapter 3 and server implementation in Chapter 4. Implementation details are described in Chapter 5. The last two chapters contain discussion on the future work and conclusion.

2 External Authoring Interface for VRML

EAI [3] defines an interface between a VRML world and an external environment. It contains a set of functions of the VRML browser that the external program can call to affect (or get parameters from) the VRML world. This contribution deals only with interface between a Java applet on a HTML page and a VRML world opened in a viewer embeded in the same page. There are some other interfaces such as COM or IPC based.

2.1 Access methods

EAI uses the existing VRML event model to access nodes in the scene. This model is based on sending and receiving events (eventOuts, eventIns). We can use four methods of accessing VRML worlds:

1. accessing the functionality of the *Browser Script Interface* [5], which is used by built-in script nodes. These functions provide a mechanism to get or set browser state. Functions such as `getName()`, `getCurrentSpeed()`, `loadURL()`, `createVrmlFromURL()`, `addRoute()` are available.
2. sending events to eventIns of named nodes inside the scene.
3. reading the last value sent from eventOuts of named nodes inside the scene.
4. getting notified when events are sent from eventOuts of named nodes inside the scene.

The first three access methods are conceptually identical to *Script Authoring Interface*, as described in ISO Standard [5]. The only difference is the way of obtaining the reference to a node through which its eventIns and eventOuts can be accessed. A script node can get a pointer to a node defined by DEF and USE statements (called instancing). An applet has no access to instancing mechanism, it has just a method to get the pointer to a node defined by DEF statement only.

The last access overcomes the problem, that ROUTE statement cannot be used for sending events from VRML scene to an applet. The applet has to implement a registered method which is to be called when specified eventOut occurs (i.e, callback mechanism).

2.2 Access to nodes

Only nodes, which are named using VRML statement DEF can be accessed by an applet. Once the pointer to a node is obtained, it's easy to access all eventOuts, eventIns and exposedFields of the node.

An instance of class **Node** can be obtained by function `getNode()` of the class **Browser**. This instance is used to get access to desired fields of the node. Calling `getEventIn()` method of class Node returns **EventIn** object, when passed

appropriate event name. The method `getEventOut()` returns an **EventOut** object when passed event name. Also exposedFields can be accessed either by passing name of exposedField itself (e.g., *rotation*), or by passing name of corresponding event (e.g., *set_rotation* for eventIn or *rotation_changed* for eventOut).

2.3 Sending events

Once an instance of **EventIn** class is obtained, an event can be sent to it. EventIn is abstract class so the cast to appropriate EventIn subclass is needed. This subclass contains a method for sending events of given type. If a VRML scene contains the folowing node:

```
DEF SCALER Transform { ... }
```

the scale can be set to values (1.0, 2.5, 1.0) from applet like this:

```
Node scaler = browser.getNode("SCALER");
EventInSFVec3f scale =
        (EventInSFVec3f) scaler.getEventIn("set_scale");
float sc[3] = { 1.0, 2.5, 1.0 };
scale.setValue(sc);
```

2.4 Reading events

Once an instance of **EventOut** is obtained, a value of event or exposedField can be read. Since EventOut is abstract class it needs a cast to appropriate subclass which contains method for getting events of given type. Current value of *scale* field can be read like this:

```
EventOutSFVec3f scale =
        (EventOutSFVec3f) scaler.getEventOut("scale_changed");
float current_sc[3] = scale.getValue();
```

2.5 Being notified

Another feature of EAI is to be notified when an EventOut is generated from the scene. In this case the applet must subclass the **EventOutObserver** class, and implement **callback()** method. The **advice()** method of EventOut is then passed the EventOutObserver. Whenever an event is generated for that eventOut the `callback()` method is invoked and passed the value and timestamp of the event. The `advise()` method is also passed a user defined object. The value of this object is passed to the `callback` method and can be used by programmer to pass user defined data to callback. It allows to handle events from multiple sources, the source is distinguished just according to this value. The following example shows how applet is notified when *position* field of ProximitySensor is changed:

```
public class VRMLApplet implements EventOutObserver {
  public void callback(EventOut event,
                       double timestamp, Object userdata) {
    // process new values of event with respect to userdata
    // userdata helps to distinguish among events
  }
  sensor = browser.getNode("PROXIMITY");
  position = (EventOutSFVec3f)
        sensor.getEventOutSFVec3f("position_changed");
  position.advise(this, new Integer(1));
    // number (1) is passed to callback method as userdata parameter
}
```

3 Applet implementation

The applet has to be placed into the same HTML page where the VRML browser window is, it is the only way how external program can communicate with an embeded VRML world. The applet provides interaction with user, communication with VRML scene and network communication with server. HTML page is divided into two frames, there is identification of applet mode (described later) in upper frame. VRML world together with applet are located in lower frame.

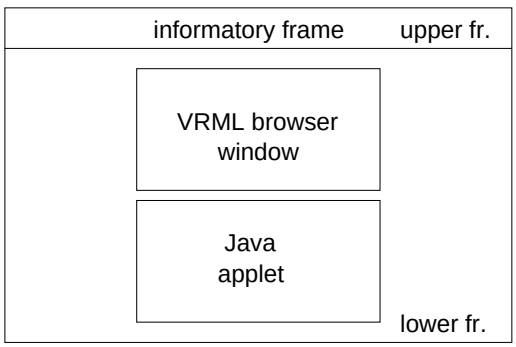

Figure 1: HTML page layout

3.1 Security restrictions

Applets by default cannot execute certain kinds of tasks from security reasons. Networking issues belongs to this kind of tasks too. Applet is not allowed to make a connection to computer other than it came from. To work around this restrictions we have used a client/server [2] model. The server runs on the same computer, which applet came from. All executed applets connect themselves to this server and then their mutual communication is possible. This approach has some drawbacks, two mains are: communication is slower than it would be among directly connected applets, and the server must be invoked before applet communication starts.

3.2 Applet modes

Due to different kinds of actions performed by applet, we divide operations of applet into modes in our implementation. The first mode is called **Master**, in this mode a user controls the world and all other participants can watch his/her activity, there is no limitation on user's actions. All actions (movements) are sent to server and then forwarded to all participants.

The other mode called **Slave** enables user either to watch actions of **Master** (submode called **Tracking**) or to work with VRML scene locally and independently (submode called **Self control**). No actions are sent to server. There is one limitation, when user watches actions of **Master** in submode **Tracking**, no interaction with VRML scene is allowed. The reason will be explained in section Problems & Future work. Upper frame shows current mode and submode. Titles can be: **Master**, **Slave/Self control** or **Slave/Tracking**.

3.2.1 Changing modes

Switching between applet modes is ensured by applet GUI. A user can change **Master** mode only by clicking *Release* button. Then mode is changed to **Slave/Self control** and the mode of next connected participant (chosen by cyclic shift) is changed to **Master**. Then server listens to this new Master and sends recieved messages to other participants. This is the only way how a participant can get control of VRML world.

Once the applet is in **Slave** mode a user can switch between **Self control** mode and **Tracking** mode (*Self control* button or *Tracking* button).

3.3 Applet GUI and control

Applet GUI consists of window for informing users about actions performed by applet. Then user has buttons to control applet. Buttons *Front View*, *Top View* and *Side View* sets defined Viewpoints in scene. Button *Set Viewpoint* adds current view as new Viewpoint to scene (then user can set this Viewpoint using VRML browser GUI). Clicking on button *Release* disconnect applet from server and causes changing applet mode to **Slave/Self control**. Button *Tracking* is active only in mode **Slave**, clicking this button causes changing mode to **Slave/Tracking** and disables all GUI elements for controling scene (even in VRML browser). Title of button is changed to *Self control*, then clicking it causes changing mode to **Slave/Self control** and enables back control GUI elements. Functions of *Help* and *Exit* buttons are obvious from their names. *Exit* button is enabled only in mode **Master**, because server listens only to applet in this mode.

Text fields under text area and buttons informs user about current position and orientation of view. This works in all modes and submodes.

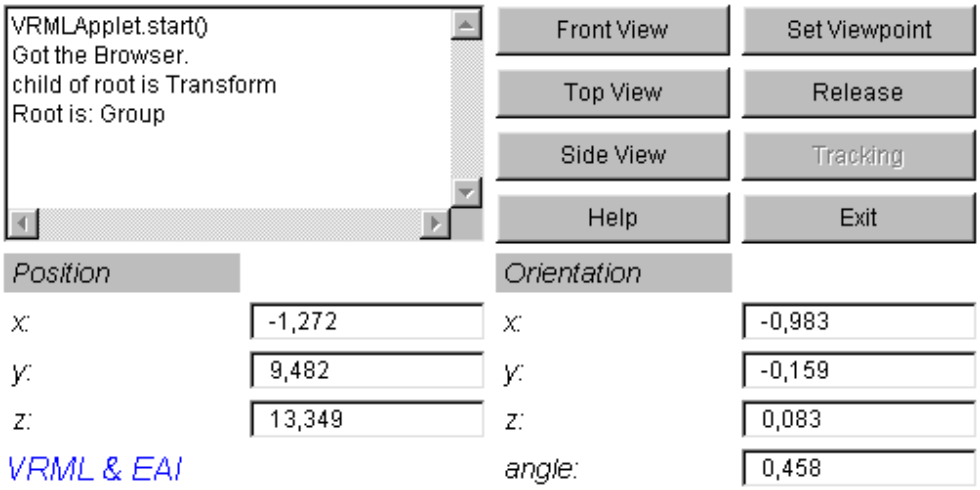

Figure 2: Applet GUI

3.4 Getting and sending parameters

Parameters of position and orientation are sent from **Master** applet to server to be forwarded to all participants. Applet gets these parameters from ProximitySensor (see section 5.1) by implementing EventOutObserver interface and its method `callback()`. Method `advice()` is called by two eventOuts: position and orientation. Applet doesn't send all new parameters immediately, but only when accumulative chages of their values exceed certain thresholds.

3.5 Users motion in the scene

When a user tracks actions of **Master** applet in **Slave/Tracking** mode, all recieved parameters of position and orientation from server are set to current Viewpoint. This Viewpoint has empty *description* field to be invisible in browser's list of Viewpoints and it is bound to VRML browser when switching to **Slave/Tracking** mode.

3.6 Applet in HTML page

The HTML file containing embeded `wrl` and `class` files has one required semantic. Field `MAYSCRIPT` means that the applet can get the VRML plugin browser object instatnce from the browser. This method should be functional in Netscape Navigator and Internet Explorer as well.

```
<EMBED SRC="VRMLScene.wrl">
...
<APPLET CODE="VRMLApplet.class" MAYSCRIPT></APPLET>
```

4 Server implementation

The server is intended to realize communication among applets. This is by principle a multithreaded task, the server consists of three threads. Server must run on the same computer which sends the applet and must be invoked before establishing first connection.

4.1 Establishing connection

Main thread listens on defined socket, which is known to all applets. When applet connects to this socket server stores its new socket number into internal structure. After second applet connects and its socket number is stored, two other threads are started. One thread which provides forwarding messages from one applet to the others (this thread has maximal priority). The other thread provides service of removing socket numbers of disconnected applets from internal structure of server.

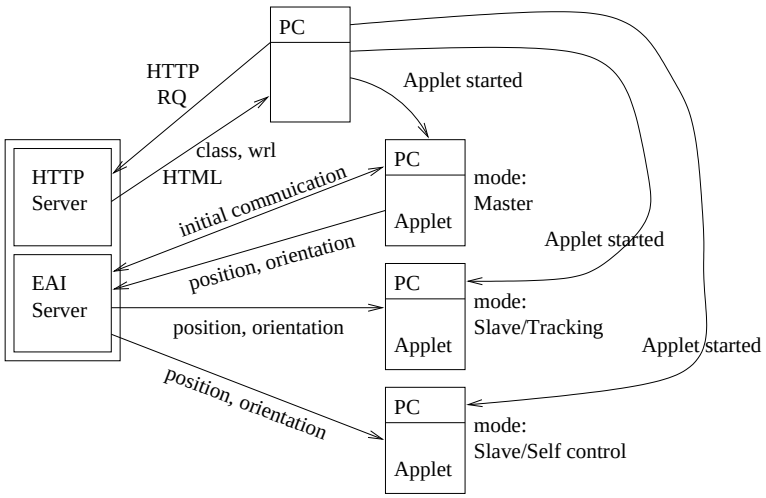

Figure 3: Establishing connection and forwarding

Server sends an order number to each new connected applet. From this number the applet can decide if it should act like **Master** or **Slave** applet.

4.2 During forwarding

Server forwards all recieved messages. When it recieves a message about releasing **Master** mode, it hands over the control to next applet (in cyclic order) and starts listening to this new **Master** applet socket.

4.3 Disconnecting

Only applet which is currently in mode **Master** can be disconnected and removed from server structures, because server listens only to this applet. In this case the server sends message to next applet (in cyclic order) that it become **Master**.

5 Details

5.1 Mandatory content of `*.wrl` file

Since the applet can access only nodes named by VRML statement DEF, there are some mandatory named nodes required in base shared `*.wrl` file.

- **NavigationInfo** named INFO,
  used for setting parameters of the scene, such as type of navigation or speed of viewer.
- **ViewPoint**s named FRONT_VIEW, TOP_VIEW, SIDE_VIEW,
  these nodes are used by GUI of the applet, to help user navigate in the scene. They should show the entire scene from front, top and side views.
- **ViewPoint** named SETTING_VIEWPOINT,
  position and orientation are sent to this node in mode **Tracking**.
- **ProximitySensor** named PROXIMITY,
  applet gets the position and orientation from this node, its bounding box should cover whole scene.
- **Group** named ROOT,
  main node for entire scene, all other nodes should be childrens of this node. The node is necessary if programmer wants to access all nodes in the scene.

5.2 Implementation and compilation

Both applet and server were implemented in language Java and compiled using JDK 1.1.5 (because of support in web browsers). Java uses an environment variable called `CLASSPATH` to look for classes. This variable should be set to contain classes: `vrml.external.Browser`, `vrml.external.Node`, `vrml.external.field.*`, `vrml.external.exception.*`. These classes are usually stored in zip file in VRML browser home directory (name of this file for VRML browser Cosmo Player is npcosmop211.zip or npcosmop21.zip, depends on version). This implementation was tested only on platform Windows using Netscape Navigator 4.07 and Cosmo Player 2.1.

6 Problems & Future work

We discover some problems during implementation of this system. The VRML scene during setting parameters *position* and *orientation* seemed to be jumping. It is caused by setting each of these parameters separately. When the position is set, new orientation should be set immediately too, but there is no function for setting more parameters at the time. Then there is a moment when these parameters

doesn't coresponds each other. This will be solved by couple of functions called `beginUpdate()` and `endUpdate()` from new proposal for EAI ISO Standart.

Another problem was with setting parameters for Viewpoint node in mode **Slave/Tracking**, because if user moved with scene during tracking, new position and orientation were added to current position of scene. This was manifested like uncontroled moving of scene, until a new Viewpoint was chosen from list. According to this all interaction with scene is disabled in mode **Slave/Tracking**.

Using *Back* button and then *Forward* button on web browser causes new start of applet and server adds to its structure a new socket, but the last socket is still there. It means that if server hands over to such unowned socked, communication stops.

There are many things to improve. Applets sends only information about current viewer position and orientation. The system might send whole VRML events to be more flexible. Then applet doesn't respond to closing of web browser so server doesn't find out about exiting some applet, this should be improved too. For real application of this system should be worthy to think of another model of choosing next controling applet. Another drawback is that there is fixed VRML file in HTML page, so users cannot choose own scene.

7 Conclusion

Described implementation shows how to take advantage of Java applet and VRML in different way than usual. Remote control of VRML world is very interesting issue. It can be used in many different tasks.

Currently the system doesn't work like real collaborative system. It allows users very few functions to perform. Only one user at time can actualy work with a VRML scene. The other participants just watch and wait to be chosen. On the other hand, the implementation provides good working basis for future extensions of Java platform and VRML cooperation.

References

[1] Flanagan, D.: *Java in Nutshell, A desktop quick reference – Java 1.1*, Second edition, O'Reilly, 1997.

[2] *Java Tutorial*, Sun Microsystems, 1998.
`http://java.sun.com/docs/books/tutorial/`

[3] Marin, Ch.: *Proposal for a VRML 2.0 informative annex, External Authoring Interface*, Silicon Graphics Inc., 1997.
`http://cosmosoftware.com/products/player/developer/eai/`

[4] Aktihanoglu, M.: *VRML 2.0 EAI FAQ*, 1998.
`http://members.xoom.com/muratak/eaifaq.html`

[5] VRML 97 Specification
`http://www.vrml.org/Specifications/VRML97/`

[6] Hartman, J., Wernecke, J.: *The VRML 2.0 Handbook – building moving worlds on the web.* Silicon Graphics Inc., Addison Wesley, 1996.