

# Using VRML EAI in Applets and Applications

Vladimir Stepan

15/12/2003

## Introduction

In the following text I would like to point out the problems that can be encountered when working with the External Authoring Interface (EAI), particularly with the implementation of EAI that is available with the installation of Parallel Graphics Cortona VRML Client version 4.1. I also present solutions to these problems. These solutions are not the only and most likely not even the best ones, but they work. Thus they meet the purpose of this text and that is to stop wasting time trying to figure out "why it doesn't work" and just make it work instead.

Using the EAI we can control the VRML scene from outside. That means that the application can insert/remove objects (VRML nodes) into/from scene and otherwise influence the parameters of the scene. In order to start any operation using the EAI we must establish the connection between the application and the VRML browser. This is the first potential source of troubles. I will dedicate it a section of this text. Another section will deal with the problems of inserting new nodes in the scene. Finally some miscellaneous troubleshooting will be presented.

## Getting the browser

### *Applet*

Let us have a VRML file *TestEAI.wrl*.

```
#VRML V2.0 utf8
DEF ROOT Group{
  children [

    DEF Sphere Shape {
      appearance Appearance
        { material Material { diffuseColor 1 0 0 }}
      geometry Sphere{ radius 1 }
    }

  ] }

```

This file describes the simple scene with red sphere. The *Shape* node that represents the sphere is a child of the *Group* node comprising the root of the whole scene. Later we will see that a VRML file with an empty *Group* node is the minimum requirement for any EAI application.

Let us embed our VRML file in an HTML code along with the applet that we are going to create (*TestEAI.class*).

```

<HTML>
<EMBED src ="TestEAI.wrl" width="600" height="400" border=0 >
<APPLET code="TestEAI.class" width="600" height="100" mayscript>
</applet>
</HTML>

```

Now we can describe how our applet gets the reference to the browser using the EAI methods. First we call:

```
browser = (Browser)BrowserFactory.getBrowser(this);
```

It is not that simple though. If we start calling browser methods right after this, we won't end up with the results we would expect, definitely not on the slower machines or over the Internet. Writing

```
Node sphere = browser.getNode("Sphere");
```

will not yet result in obtaining the reference to a node we required. The creation of the browser is asynchronous and this EAI doesn't provide a synchronization mechanism. What we should do is:

```

Node sphere = null;
while (sphere == null){
    try{
        sphere = browser.getNode("Sphere");
    } catch (vrm1.eai.InvalidBrowserException e){ ... }
}

```

Once we get the first node, we can safely read any other or call any other browser method, because the connection to the browser has been established.

## **Application**

When creating a stand-alone application the situation is a little different. But EAI implementation offers a method to create a browser and its window. Let us say we have the application implemented as a class

```
public class CApp extends Frame implements Runnable{...
```

The main() method will call the *CApp* constructor and set the frame visible:

```

public static void main(String[] args) {
    CApp app = new CApp();
    app.setVisible(true);
}

```

The most important part will be in the constructor. Using the BrowserFactory method we will create the UI component with the VRML browser window. This component should be an instance of the *VrmlComponent* class that implements *getBrowser()* method. The example code is:

```

java.awt.Component c=null;
BrowserFactoryImpl factory;
Class factory_class;
String[] st = ...
try{
    factory_class = Class.forName
    ("com.parallelgraphics.cortona.vrml.eai.BrowserFactory");
    factory = (BrowserFactoryImpl)factory_class.newInstance();
    c = factory.createComponent(st);

} catch ( Exception ex){...}

```

We have created the UI component, now we will get the browser object. Variable *st* used as a parameter at *factory.createComponent(st)* is an array of strings, each of them should be a valid URL of the scene that we want to display in the browser. If first URL fails for some reason, next one is checked. Many EAI methods working with URLs use this approach.

As we will see, this variable has to be used again, when the new browser object loads the scene. Don't ask me why...

After we are done with the browser, we need to register the component of its window just like any other UI component.

```

try{
    VrmlComponent vr;
    vr = (VrmlComponent)c;
    browser = vr.getBrowser();
    String [] par = null;
    browser.loadURL(st,par);
}catch (Exception ex){ ... }

c.setSize(790, 490);
this.add (c);
c.setBounds(5, 5, 790, 490);

```

Finally, since our application is runnable, we create its thread and start it:

```

thread = new Thread(this);
thread.start();

```

Thus the application needs the *run()* method to be implemented. The code of this method can look like this:

```

public void run(){
    boolean ready=false;
    while(!ready){
        try{
            Node v_node = browser.getNode("ROOT");
            ready = true;
            start();
        } catch (vrml.eai.InvalidBrowserException e){
            try{
                thread.sleep(1000);
            }catch(Exception ex){ ... }
            ...
        }
    }
}

```

We face the same problem as described in previous section. The code above describes basically the same solution – we try to read the VRML nodes until we succeed. In this case we use a local variable for test-reading and all real work is performed in the *start()* method that is skipped if the node reading throws an exception. The cycle is driven by a boolean value that is set after successful test-reading. If we catch the exception, we can use the thread's capability to sleep and wait before trying again.

## Inserting nodes

Once we have a browser object, we can control the VRML scene. We can access DEF named nodes and their *exposedFields*, send *eventIns* and listen to *eventOuts*. Often we need to add nodes into our VRML scene. For that purpose two methods of the Browser class exist. Unfortunately they differ in approach quite a lot. Let's describe them closer...

### From string

For simple VRML constructions that can be written down in Java code or in case we use a textual input we can use the method *vrml.eai.Browser.createVrmlFromString(String vrml)*. This method returns an array of nodes (Node[]) that can be sent to the scene as an EventInMFNode.

Example of the use can be inserting sphere and box in the root of the scene:

```
String vrmlString = "Transform { children Shape { "  
    + "geometry Sphere {radius 0.5} } }"  
    + "Transform { children Shape { "  
    + "geometry Box {size 1 0.2 0.5} } }"  
Node[] nodes = browser.createVrmlFromString(vrmlString);  
EventInMFNode inNodes = (EventInMFNode)rootNode.getEventIn("addChildren");  
inNodes.setValue(nodes);  
Node sphereNode = nodes[0];  
Node boxNode = nodes[1];
```

In this example we create VRML description of the two shapes and pass it as a parameter (*vrmlString* variable) to the *createVrmlFromString()* method. The result (variable *nodes*) is sent to the root (*rootNode*) via the *eventIn addChildren*. Both newly created nodes are also stored in special variables (*sphereNode* and *boxNode*) for possible later use.

### From URL

The method *vrml.eai.Browser.createVrmlFromURL(String[] url, Node node, String eventIn)* is another way to insert new nodes into the scene. This method allows us to work with the VRML files without the necessity having to implement reading them as a string. The bad news for a programmer is that this method returns void value.

It reads the .wrl file specified by the *url* parameter (array of possibly valid URLs) and sends it to the eventIn of the node (specified by parameters *node* and *eventIn*). This is OK if we just need to add something to the scene without further working with it. But we usually need to manipulate our scene, so we might want to have a reference to newly created nodes.

Of course we can help ourselves by reading the *children* field of the node we sent the new VRML to and taking the last ones (sending the *addChildren* event adds new nodes to the end of the *children* array). For this we need the a priori knowledge of the number of nodes (trees) in the file we read. Or we can keep track of the number of the target node's children

and subtract the old number from the new one, after we call the *createVrmlFromURL()* method.

The problem is that this method is asynchronous. We have to create a mechanism that would wait for the sent nodes to arrive to the VRML tree. We can do it by listening to the *children* field (*exposedField* acts as both *eventOut* and *eventIn*) of the target node. After the *children* field signals a change, our new nodes are ready.

The following code is the example of the mechanism that works in our programs:

```
EventOutMFNode ch = (EventOutMFNode)rootNode.getEventOut("children");
ch.addVrmlEventListener(new vrml.eai.event.VrmlEventListener(){
    public void eventOutChanged(VrmlEvent e){
        ready = true;
    }
});
```

We register the *eventOut children* of our node (*rootNode*) and add an event listener to it. This listener is implemented as an anonymous class and sets the boolean value (*ready*) to *true*. In order for the anonymous listener to see the variable *ready* it must be the global variable of the class whose method uses this piece of code.

Before we call the *createVrmlFromString()* method, we must set *ready* to *false* and than we use the waiting cycle.

```
ready = false;
browser.createVrmlFromURL(url, rootNode, "addChildren");
while(!ready){ boolean b = ready; };
```

For some reason an empty waiting cycle doesn't work everywhere that is why the simple assign instruction is there.

After the change of *ready* value caused by the event listener ends the waiting cycle, our new nodes have been delivered and we can start reading them.

```
Node[] nodes = ch.getValue();
Node newNode = nodes[nodes.length - 1];
```

In our example we assume that we send just one new node (in case that the file specified by *url* is just one VRML tree). At the end, we have our new node in the VRML tree (as a new child of the node kept at the variable *rootNode*) and referenced by new variable *newNode*.

## Other tips

### ***Path to file vs. URL***

When working with EAI, one often uses String variables to express the URL. This is frequently a source of mistakes. When writing an applet, we inquire for path to the current directory by calling *java.applet.Applet.getCodeBase()*. This method returns string that is a valid URL. On the other hand, we use *System.getProperty("user.dir")* to do the same in case of application. The string returned by this method IS NOT an URL it is the path to the file. The EAI methods will not find a file specified by its path. It is necessary to add "**file:///**" string to the beginning of the path. Keep that in mind to avoid wasting time in search for mistakes...