ProBrAl - 002/94
-
Development of a platform for communication and visual computing

“Extracting objects in a VRML 2.0 file”

This report is the result of the research activities of the visit at the **Computer Graphics Center** (**ZGDV**), **Department of Mobile Information Visualization**, Darmstadt, Germany.

This visit was sponsored by the cooperation between Brazil and Germany, under ProBrAl project 002/94 - CAPES/CNPq - (**UNICAMP / THD - GRIS**).

UNICAMP - Universidade Estadual de Campinas
THD-GRIS - Technical University of Darmstadt
- Interactive Graphics Systems Group.

Guest: Msc. Alberto Barbosa Raposo

Period of the visit: July to September 1996.

“Extracting objects in a VRML 2.0 file”

Msc. Alberto Barbosa Raposo *

* - Electrical Engineer (**UNICAMP**), 1994
- Master of Sciences (**UNICAMP**), 1996
- Ph.D. Student at **DCA** - **Department of Computer Eng. and Industrial Automation**, Faculty of Electrical and Computer Engineering, State University of Campinas - **UNICAMP**, Campinas, Brazil.

CONTENTS

1. Introduction **1**

2. Implementation details **3**
- **2.1 The application server (ParseVRMLServer.java)** **3**
- **2.2 Creating the script (ReadVRMLFile.java)** **4**
- **2.3 The client (Interface.java)** **9**
- **2.4 Creating the "sub-VRML" world (ParseVRMLServer.java)** **11**
- **2.5 Visualization of the results (SimpleViewer.java)** **15**
- **2.6 The data structure (Node.java)** **15**

3. Specification of the script **16**

4. References **19**

Appendix A - General observations **20**
- **A.1 The application in LCA - FEEC - UNICAMP** **20**
- **A.2 Deficiencies encountered** **20**

Appendix B - Source code **22**

1. Introduction

This work developed a strategy to accelerate the visualization of VRML 2.0 (Virtual Reality Modeling Language) worlds in a mobile terminal, by allowing the user to extract only a part of the complete VRML world. The extracted part of the world represents one or more sub-trees of the hierarchical structure of the application world´s objects[1] .

An application (written in Java) was developed in order to test this strategy. This application runs in a mobile terminal, which is connected to the application server. In this server there is a program capable of reading the VRML world (located in any WWW-server), parsing it and sending its hierarchical structure back to the client (the mobile terminal). The client, based upon the data received from the server, creates a simple graphical interface that gives the user the opportunity of selecting the part(s) of the world he wants to see. The selected sub-tree(s) is then sent back to the application server, which is capable of parsing the original VRML file and extracting from it only the desired parts, sending a valid "sub-VRML" file to the client, which can finally call a VRML 2.0 viewer.

The schema of this approach is shown in Figure 1 below:

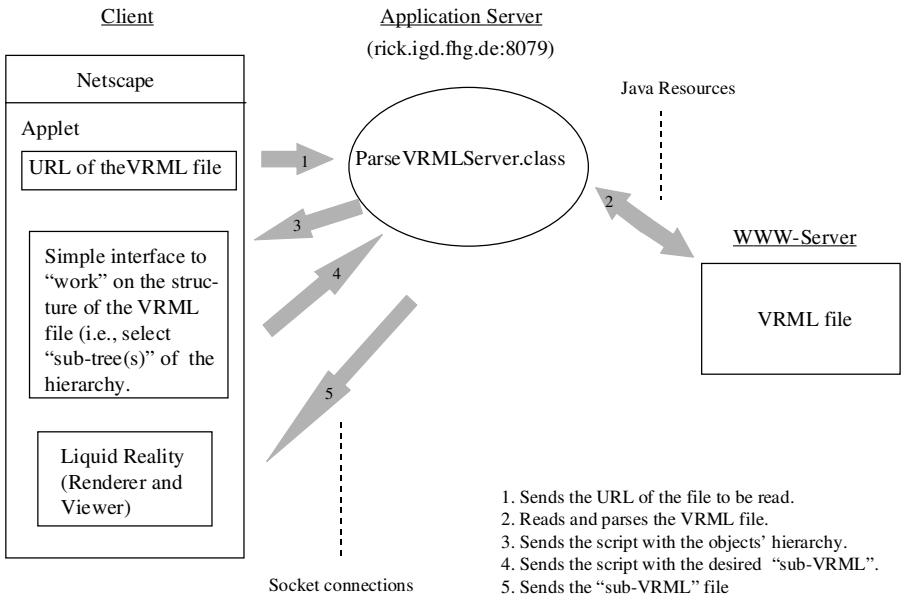

Figure 1: Schema of the strategy to extract sub-trees of a VRML 2.0 world.

The application runs in the client as a Java applet [Sun (1995)], [Cornell (1996)], [Hoff (1996)], [Lemay (1996)], downloaded with a HTML page (http://rick.igd.fhg.de:8080)[2] .

[1] The structure of the objects in a VRML 2.0 world is highly hierarchical. More details about the organization of the VRML 2.0 objects' structure can be found in [VRML2.0 (1996)].

[2] However, it is strongly suggested that the application be executed from UNICAMP. See Appendix A.

The process begins by calling the application home page using any Java compatible Web browser (like Netscape Navigator 2.x, for example). At this moment, a socket is created between the client and the application server (currently running in rick.igd.fhg.de:8079[3]). The next step is to write the URL of a VRML 2.0 world and send it to the application server (this is the arrow #1 in Figure 1). The application server will then use Java resources to connect to the server of the VRML file, read and parse it (arrow #2 in Figure 1). The first output of the server is a text script, representing the hierarchy of the objects of the VRML world, which will be sent to the client (arrow #3 in Figure 1). A brief specification of this script is in the section 3.

Based on the script received, the applet creates a simple interface which allows the user to select the desired sub-tree(s) from the file. After this selection, a new script[4] is sent to the server, describing the selected parts (arrow #4 in Figure 1). Based on this new script, the application server can create a new VRML 2.0 world from the original one, by extracting only the desired parts from it. This final "sub-VRML" world is then sent to the client, that will finally be able to see the desired world (arrow #6), using methods of the Liquid Reality library [Dimension X (1996)].

The division of the program into an applet (executed in the client) and an application server is needed due to a security restriction of the Netscape browser. As stated in [Cornell (1996)], "applets run on the computer browsing your Web page, but they can only access files and sockets on the computer serving the Web page". Because of this restriction, the client would only be able to read VRML worlds located in the WWW server of the applet's page (rick.igd.fhg.de:8080). To overcome this limitation, the application server was developed so as to act as a proxy server, a service that "grabs requested information from the Web and sends it to whoever requested it". The application server is able to grab that information anywhere in the Web because it is executed as an independent program, not subject to the applet restrictions.

[3] Also running in caolho.dca.fee.unicamp.br. See Appendix A.

[4] This script is similar to the previous one, simply adding *true* or *false* after the objects, indicating whether they are selected or not.

2. Implementation details

As stated, this approach can be used to accelerate the visualization of VRML 2.0 worlds in mobile terminals by reducing the data transmission involved in this process. The scripts are small text files and the “sub-VRML” file is also a text file that can be small, depending upon which sub-trees were selected.

The developed application can also be seen as a starting point for a larger application, which would parse not only the objects’ structure, but also the textures, geometry, animation, and other aspects of the VRML 2.0 files.

The application is composed of five source files:

- ParseVRMLServer.java - This is the application server main program. It has an infinite loop, that is always waiting for a connection on its port. When this connection occurs, it creates a new thread, which deals with the client communication protocol, reading the VRML file, translating it into a script and doing the final parsing of the VRML (i.e., removing the undesired nodes from the original VRML), as requested by the client.

- ReadVRMLFile.java - This is the program (also executed in the application server) that reads the VRML file and translates its structure into a script.

- Interface.java - This is the client program (applet). It deals with aspects of communication and interfacing, and is also responsible for the translation of the script into a tree data structure.

- Node.java - Simply the data structure of each "node" of the tree.

- SimpleViewer.java - The renderer, that uses methods of Liquid Reality library.

2.1. The application server (ParseVRMLServer.java)

The main method of this program simply waits for a connection to the server at the specified port (8079). When this connection occurs, a socket[5] is created between the client and the application server, and a new instance of the class `VRMLParser` is established, using this socket. The new

[5] The Java language has predefined classes for distributed programming using socket communication. Refer to [Cornell (1996)] for more information.

instance of `VRMLParser` class will be executed in a separated thread[6], allowing that more than one client be connected to the server simultaneously. The piece of the code created to do that is shown below:

```
// Creating a server, that waits for a connection
ServerSocket ss = new ServerSocket(8079);

// Continuously monitoring for connections
for(;;)
  {
  // Creating a socket when a client asks for a connection
  Socket incoming = ss.accept();
  // Starting a new thread, that will run class VRMLParser
  new VRMLParser(incoming).start();
  }
```

The `VRMLParser` class manages the communication protocol between the client and the application server (this protocol is represented by the arrows in Figure 1). In this section, and in sections 2.2 and 2.4, the communication protocol will be analyzed at the application server's side, while in section 2.3 it will be analyzed at the client's side.

The communication starts with a signal from the client, which wants to read a VRML 2.0 world. When the application server receives this signal, it becomes prepared to read the URL of the VRML world, which is the next information the client is going to send (arrow #1 in Figure 1). After reading this URL, the application server connects to the WWW server where the desired VRML file is located, and reads this file (arrow #2 in Figure 1).

After these initial steps, the application server has already downloaded the original VRML file, which is now stored in a string variable of the program. This variable will be used to the creation of the script that will be sent to the client (section 2.2) and in the final parsing of the file, when the undesired nodes will be eliminated (section 2.4).

2.2 Creating the script (ReadVRMLFile.java)

After reading the VRML 2.0 file, the application server parses it to write a text script, that will be sent to the client. The application server reads each word of the VRML 2.0 file (stored in a string variable) and calls the method `getKeyWords`, that will compose the script correctly, while the interested keywords are being found in the original file. The `getKeyWords` method is a method of the `ReadVRMLFile` class.

This method looks for the following keywords in the VRML 2.0 file:

[6] Java is a multithreaded language, i.e., it allows that many programs be executed simultaneously, each one carrying its own set of data. This is an important facility to multitask programming. More information about Java threads can be found in [Cornell (1996)], [Drake (1996)] and [Lemay (1996)].

- Shape: establishes the beginning of a Shape node, i.e., a geometric object of the scene. When this word is found, it is written in the script. Inside a Shape node, the method also looks for the auxiliary keyword geometry, that describes the geometric characteristics of the node. When this word is found, it is written in the script, and the program reads the next word in the file, that is the name of the geometric node representing the geometry of the object (Sphere, Cone, Text, and so on). This word is also written in the script.
- PointLight, DirectionalLight and SpotLight: are the keywords for illumination nodes. When one of these words is found, it is written in the script, and the program looks for the auxiliary keywords location and direction, describing some features of the light source.
- Viewpoint: establishes the beginning of a camera node. This word, when found in the original file, is written in the script, and the program starts the search for auxiliary keywords for the camera: position and orientation.
- children: every time this word is found in the file, we entered a deeper level of hierarchy (i.e., the following nodes are children of the current one). When this word is found, a global variable indicating the current hierarchical level is updated and the value of the new level is written in the script. The children nodes are enclosed by brackets. Because of that, [and] are also considered keywords (a] can determines the return to an upper hierarchical level).
- Switch: indicates the beginning of a Switch node, i.e., a node that allows a choice among the various nodes defined inside of it. This word is written in the script and the method looks for the auxiliary keyword whichChoice, that indicates which node of the Switch is going to be visualized.
- LOD: keyword for the Level Of Detail node.
- DEF: this word is used to give names to specific nodes, that can be later reused. This word is considered by the method if it is related to any of the previous key nodes (Shape, Viewpoint, PointLight, etc), otherwise it is ignored. In the case of relating to a key node, the word is written in the script, followed by the name given for the node and the node itself (followed by auxiliary keywords, as stated previously).
- USE: indicates the use of a predefined node. This word is considered only if the equivalent DEF wasn't ignored (i.e., USE will be written in the script only if it is related to a key node).

An important observation about the keywords Switch, LOD and DEF is that their definitions are enclosed by { and }. By reason of that, these symbols are also considered keywords (like the brackets, for the children keyword).

The source code of the discussed method is given below:

```
public void getKeyWords(String s)
// The variable s is the word read in the VRML file
      {
      boolean b = false;
      String str;
      int c = 0;

      // Found a Shape node
      if (s.equals("Shape"))
            // Write it in the script
            this.writeScript("Shape\n");

      // Form of the Shape node - auxiliary keyword
      else if (s.equals("geometry"))
            {
            // Write it in the script
            this.writeScript("   geometry ");
            // The geometric node
            String st = this.readString();
            // We must rerun getKeyWords if the type of the
            // geometry is a DEF or USE
            if (st.equals("DEF") || st.equals("USE"))
                  {
                  this.getKeyWords(st);
                  this.writeScript("Def");
                  }
            else this.writeScript(st);
            this.writeScript("\n");
            }

      // Found a point light
      else if (s.equals("PointLight"))
            // Write it in the script
            this.writeScript("PointLight\n");

      // Found a directional light
      else if (s.equals("DirectionalLight"))
            // Write it in the script
            this.writeScript("DirectionalLight\n");

      // Found a spot light
      else if (s.equals("SpotLight"))
            // Write it in the script
            this.writeScript("SpotLight\n");

      // Position of a light source - auxiliary keyword
      else if (s.equals("location"))
            {
            // Write it in the script
            this.writeScript("   location");
            // Write the point where the source is located
            for (int i=0; i<3; i++)
                  this.writeScript(" "+this.readString());
            this.writeScript("\n");
            }

      // Direction of a light source - auxiliary keyword
      else if (s.equals("direction"))
            {
            this.writeScript("   direction");
            // Write the vector representing the direction
            // of the light source
            for (int i=0; i<3; i++)
                  this.writeScript(" "+this.readString());
```

```
      this.writeScript("\n");
      }

// Found a camera
else if (s.equals("Viewpoint"))
      // Write it in the script
      this.writeScript("Viewpoint\n");

// Position of a camera - auxiliary keyword
else if (s.equals("position"))
      {
      this.writeScript("   position");
      // Write the point where the observer is located
      for (int i=0; i<3; i++)
            this.writeScript(" "+this.readString());
      this.writeScript("\n");
      }

// Orientation of a camera - auxiliary keyword
else if (s.equals("orientation"))
      {
      this.writeScript("   orientation");
      // Write the orientation of the camera
      for (int i=0; i<3; i++)
            this.writeScript(" "+this.readString());
      this.writeScript("\n");
      }

// A new level of the hierarchy
else if (s.equals("children"))
      {
      // Updating the hierarchical level
      hlevel++;
      // Write the level in the script
      this.writeScript(String.valueOf(hlevel)+"\n");
      // We use a vector and a counter to know when
      // the children will be closed
      opened.addElement(new Integer(count_colch));
      }

// Found a switch Node
else if (s.equals("Switch"))
      {
      // Write it in the script
      this.writeScript("Switch\n");
      // To "mark" that a Switch node is opened and
      // to know when it is closed, we use a vector
      // and a counter
      opened2.addElement(new Integer(count_bracket));
      }

// Definition of the choice of the Switch node
else if(s.equals("whichChoice"))
      // Write it and its choice in the script
      this.writeScript("   whichChoice " +
                        this.readString() +"\n");

// Found a LOD Node
else if (s.equals("LOD"))
      {
      // Write it in the script
      this.writeScript("LOD\n");
      // To "mark" that a LOD node is opened
      opened3.addElement(new Integer(count_bracket));
```

```
      }

// Found a definition
else if (s.equals("DEF"))
      {
      // The name of the DEF
      String defName = this.readString();
      // The kind of node defined
      String nodeType = this.readString();
      // Only some kinds of DEFs will be considered
      if (nodeType.equals("Shape") ||
          nodeType.equals("Viewpoint") ||
          nodeType.equals("Switch") ||
          nodeType.equals("PointLight") ||
          nodeType.equals("DirectionalLight") ||
          nodeType.equals("SpotLight"))
            {
            // Write the definition in the script
            this.writeScript("DEF " + defName + "\n");
            // To "mark" that a Definition is opened
            opened1.addElement(new Integer(count_bracket));
            // Put this definition in a vector of used
            // DEFs
            this.usedDefs.addElement(defName);
            }
      // Continue normally with the next word
      this.getKeyWords(nodeType);
      }

// Found a use of the name
else if (s.equals("USE"))
      {
      // The name of the USE
      String useName = this.readString();
      // Search if the name is equal to a name in the
      // vector of used DEFs. If it is a used DEF, we
      // write it in the script.
      usedefloop:
         for(int i=0; i<this.usedDefs.size(); i++)
            {
            // Found the name in the vector
            if(useName.equals((String)this.usedDefs.
                                        elementAt(i)))
                  {
                  // Write it in the script and
                  // break the loop
                  this.writeScript("USE " + useName + "\n");
                  break usedefloop;
                  }
            }
      }

// If the signals [, ], { or } are found, auxiliary
// methods are called, to treat each case
// Open a statement "["
else if (s.equals("["))
      this.openStatement(COLCH);
// Close a statement "]"
else if (s.equals("]"))
      this.closeStatement(COLCH);
// Open a statement "{"
else if (s.equals("{"))
      this.openStatement(BRACKET);
// Close a statement "}"
```

```
else if (s.equals("}"))
      this.closeStatement(BRACKET);

// End of the file
else if (s.equals("EndOfFile"))
      this.writeScript(s+"\n");
}
```

The other methods called inside this one can be seen in the Appendix B of this document, where the source code is completely shown.

After the creation of the script, it is transmitted to the client (arrow #3 in Figure 1), that creates an interface for the selection of the desired objects. This interface and the details of the client side will be discussed in the next section.

2.3 The client (Interface.java)

Interface.java is the program being executed at the client. It is an applet embedded in the application home page (http://rick.igd.fhg.de:8080). When this applet is initialized, it opens a socket connection to the application server, that should be waiting for a connection (as explained in section 2.1). At this moment, the applet also creates a simple user interface, with which the user will be able to select the VRML world and the objects of this world he wants to see (Figure 2).

On the top of the applet window there is a text field to write the URL of the VRML file the user wants to work with. When this name is correctly written in the text field, the user can click one of the two buttons “Read it” to send this URL to the application server (arrow #1 in Figure 1). After that, the application server will read that VRML file, parse it, and send the script back with the hierarchical structure of the VRML file (all this process was discussed in the previous sections).

When the client receives the script (arrow #3 in Figure 1), it is written in the large text area below the buttons “Read it”. In fact, the script is shown in the applet only for demonstration purposes, since the user does not have to work on it (actually, the user does not even have to know about the existence of this script).

The user can select the desired objects of the scene using the buttons below the text area. To select among the geometric objects of the scene, the user should click the button “Geometric Objects”, to select among the light sources, he should click “Light Sources”, and to select among the cameras, he should click “Cameras”[7] . When one of these buttons is clicked, a colored panel appears, with the name of all the objects of that kind in the first hierarchical level. If there are objects of the selected kind in a deeper hierarchical level, a button “Next Level” can be used to change the panel contents to the names of the objects in the next level.

[7] It’s important to note that the Viewpoint node (that defines the camera) of VRML 2.0 is a blindable node, i.e, there may be many instances of it in a scene, but only one instance can be active at any instant of time (if more than one instance is active, the renderer will use the last active viewpoint).

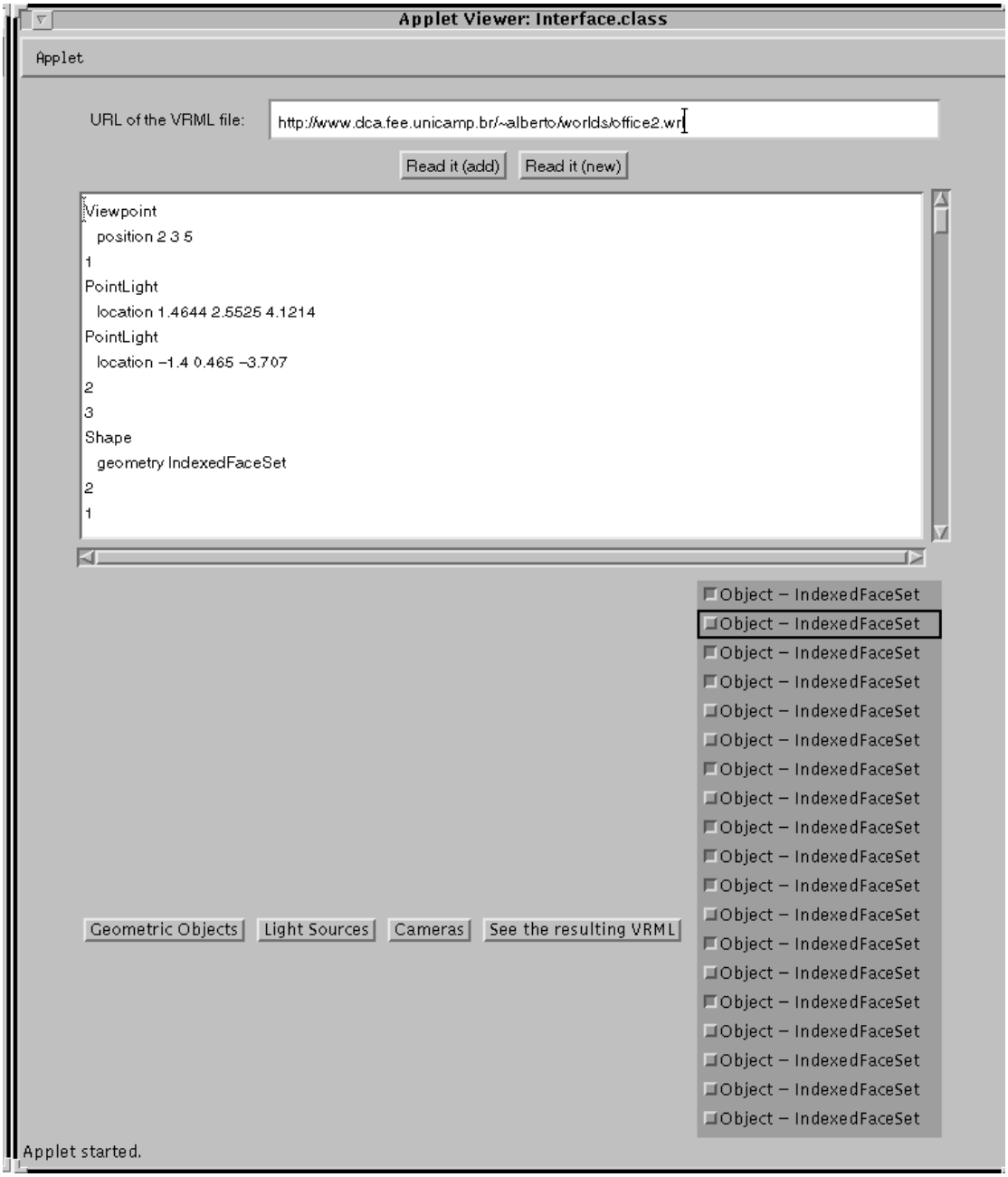

Figure 2: The client's interface

By default, all the objects are chosen; the user has to click only on the objects he doesn't want to visualize. If he deselects/selects an object, all its descendants will be deselected/selected automatically.

Once the user has deselected the objects he doesn't want to visualize, he can click the button "See the resulting VRML" to visualize the "sub-VRML" world he has created. At this moment, the work of the user is finished; he must only wait for the image.

After the button "See the resulting VRML" is clicked, the client rewrites the script, adding the words "true" or "false" after each object, indicating whether the object is selected or not. This new

script is sent to the application server (arrow #4 in Figure 1), which will create the "sub-VRML" world based on it. This is the topic of the next section.

Before going back to the application server, a final observation about the applet's interface should be made, regarding the difference between the two buttons "Read it". The button "Read it (new)" makes the application server read the VRML world as a new file, throwing away the information about the "sub-VRML" world previously created. The button "Read it (add)" makes the application server maintain that information, allowing the merging of two "sub-worlds". In this way, the application allows that objects of different VRML worlds be joined in another world (the tool can be used not only as an object-extraction tool, but also as a merging tool).

2.4 Creating the "sub-VRML" world (ParseVRMLServer.java)

When the application server receives from the client the script containing the information about which objects were selected (arrow #4 in Figure 1), it calls the method `searchInFile`, that functions similarly to the `getKeyWords` method, shown in section 2.2. The main difference is that the `searchInFile` method rewrites the VRML file (or, in other words, writes the "sub-VRML" file) based on what is read from the script, while the `getKeyWords` composed the script based on what was read from the original VRML file.

The `searchInFile` is a method of ParseVRMLServer class. Like `getKeyWords` method, `searchInFile` looks for the following keywords in the script received:

- Shape, PointLight, DirectionalLight, SpotLight and Viewpoint: when one of these keywords is found in the script, the program reads the next word to know if this object was selected or not. If the word is "true" (i.e., it was selected), the node is copied from the original VRML to the "sub-VRML". If the word is "false", the node wasn't selected and will not be copied to the "sub-VRML".
- DEF: when a definition is found in the script, an additional search is needed to know if there is at least one selected node in that definition (a definition itself is not a node, but comprises one or more nodes). If there is at least one selected node in that definition, it should be written. Otherwise, if no node of a definition was selected, the definition will not be copied at the moment, but should be stored in a vector of unused DEFs, because it can be reused later.
- USE: if this node wasn't selected, it will be simply skipped. However, if it was selected, the program must search the vector of unused DEFs. If it is the case of using a definition previously skipped (i.e., stored in the vector of unused DEFs), the definition should be written in the place of the USE, and removed from the vector.

- whichChoice: this word is considered as a keyword here because, based on the choice made in the client, the corresponding Switch node could have its value changed (the interface at the client allows that the user changes the choice of such node). The new choice is transmitted in the script. So, when this word is found, it cannot be copied from the original VRML file, but from the script received. This is a first experiment using the idea that the client could also be able to change characteristics of the original VRML.

All other nodes that are not treated in this application will simply be copied from the original VRML file to the "sub-VRML".

Following is the source code of this method:

```
public void searchInFile(String word)
      // The variable "word" is the word read in the script
      {
      // One of these keywords are found
      if(word.equals("Shape") ||
         word.equals("PointLight") ||
         word.equals("DirectionalLight") ||
         word.equals("SpotLight") ||
         word.equals("Viewpoint") )
          {
          int i;

          // Skip the spaces
          for(i=0; Character.isSpace(this.script.charAt(i)); i++)
                  ;  // Do nothing
          //Update the script
          this.script = this.script.substring(i);

          // Find the "state" of the node (true or false)
          for(i=0; !Character.isSpace(this.script.charAt(i)); i++)
                  ;  // Do nothing
          boolean b = Boolean.valueOf(
                     this.script.substring(0, i)).booleanValue();

          // Update the script
          this.script = this.script.substring(i+1);
          // Find the next node of this type in the VRML file
          this.lookFor(word, b);

          // Write or not the Shape Node
          if(!b)
            this.ignoreNode();
          }

      // Found a DEF
      else if(word.equals("DEF"))
          {
          int i;
          int i1 = 0;

          // Find the name of the node and update the script
          for(i=0; !Character.isSpace(this.script.charAt(i)); i++)
                  ;  // Do nothing
          // The name of the definition
          String defName = this.script.substring(0, i);
          this.script = this.script.substring(i+1);
```

```
      // Check if there is at least one active node inside
      String aux = "";
      boolean b = false;
      i = 0;
      while(!aux.equals("ENDDEF") && !b)
        {
        // Read each word
        for (i1 = i ; !Character.isSpace(
                         this.script.charAt(i1)); i1++)
              ; // Do nothing
        aux = this.script.substring(i, i1);
        i = I1+1;
        // Found an active node
        if(aux.equals("true")) b = true;
        }

      // If there's no active node in the DEF
      if (!b)
        {
        // Update the script
        this.script = this.script.substring(i1+1);
        // Find the next DEF <name> in the VRML file
        String strTmp = "DEF " + defName;
        this.lookFor(strTmp, false);
        // "body"of the definition
        StringBuffer defBody = new StringBuffer(5000);
        defBody.append(strTmp);
        defBody.append("  ");
        defBody.append(this.ignoreNode());
        // Store the name and the body of the definition
        // in the vector of unused DEFs
        def1.addElement(defName);
        def2.addElement(defBody.toString());
        }
      }

  // Found a USE
  else if(word.equals("USE"))
      {
      int i;

      // Find the name of the node and update the script
      for(i=0; !Character.isSpace(this.script.charAt(i)); i++)
              ;  // Do nothing
      String useName = this.script.substring(0, i);
      this.script = this.script.substring(i+1);

       // Find the "state" of the node (true or false)
      for(i=0; !Character.isSpace(this.script.charAt(i)); i++)
              ;  // Do nothing
      boolean b = Boolean.valueOf(
                  this.script.substring(0, i)).booleanValue();

  // The node is active (true)
      if (b)
        {
        // Check if the node used is in the
        // vector of "unused" DEFs
        for(i=0; i<this.def1.size(); i++)
           if(useName.equals((String)this.def1.elementAt(i)))
                    break;

        // If the use node is in the vector
```

```
            if(i<this.def1.size())
               {
               // Find the next node USE in the VRML File
               this.lookFor("USE " + useName, false);
               // Put the definition in the sub-VRML file
               this.theFile += (String)this.def2.elementAt(i);
               // Remove this definition from the vector
               this.def1.removeElementAt(i);
               this.def2.removeElementAt(i);
               }
            }
         else
            {
            // Find the next node USE in the VRML File
            this.lookFor("USE " + useName, false);
            }
          }

      // Found "whichChoice"
      else if(word.equals("whichChoice"))
        {
          int i;

          // Find the choice
          for(i=0; !Character.isSpace(this.script.charAt(i)); i++)
                  ;  // Do nothing
          String choice = this.script.substring(0, i);

          // Update the script
          this.script = this.script.substring(i+1);
          // Find the next word "whichChoice" in the VRML
          this.lookFor(word, true);

          // Substitute the original choice in the file for
          // the one given in the script
          // Read the original file and do nothing
          this.readString();
          // Put the new choice
          this.theFile += choice + " ";
          }

      else if(word.equals("FINISH"))
          {
          this.lookFor("EndOfFile", false);
          }
      }
}
```

The other methods called inside this one can be seen in the Appendix B of this document, where the source code is completely shown.

The "sub-VRML" world created is stored in the application server having the following name: *subVRMLddmmyyhhiiss.wrl*, where *ddmmyy* is the current date and *hhiiss* is the instant when it was created. This name will appear in the applet window at the client when the resulting image is visualized, and the user will be able to download the "sub-VRML" world he created.

Now that the "sub-VRML" world has been stored in the application server, it can send the name of this world to the client, in order to allow the visualization (arrow #5 in Figure 1).

2.5 Visualization of the results (SimpleViewer.java)

To visualize the resulting "sub-VRML" world, the program uses classes of the Liquid Reality library [Dimension X (1996)]. Liquid Reality is a set of Java class libraries that gives the user VRML 2.0 functionality (in other words, Liquid Reality implements the VRML 2.0 specification, providing everything needed to write out VRML 2.0, to render, and to manipulate the scene graph). Although it is a commercial product, there is currently a beta version freely available at the Web (http://www.dnx.com/products/lr).

When the client receives the URL of the "sub-world" created (arrow #5 in Figure 1), it executes a method of `SimpleViewer` class, that utilizes the `BrowserNexus` class of Liquid Reality. The `BrowserNexus` class is suitable for making browsers, since it implements the essential functionality of a browser, and parses the appropriate input. In fact, this class acts like a "black box", where the input is the URL of the VRML 2.0 file and the output is a frame showing the rendered world (this frame appears in the applet window at the client).

In the `SimpleViewer`'s frame, the user is able to visualize the "sub-world" he has created and to "walk" through it (i.e., with the mouse he can change the observer's position, simulating a "walk" in the world).

2.6 The data structure (Node.java)

The last class of the application (`Node` class) is merely a data-storing class. It has global variables that store the characteristics of the VRML nodes considered in this application (the type of the node - Shape, Viewpoint, and the like; the geometric type - in the case of a Shape node, and so on).

This class is also used to implement the tree structure representing the hierarchical structure of the original VRML file. For that, it has a Vector variable which can store other instance of `Node` objects. In this vector, all the children nodes of a current node are stored.

3. Specification of the script

The script proposed to send the information regarding the hierarchical structure of a VRML 2.0 world is simply a text file that represents each hierarchical level by a number (every time the word "children" is found on the VRML file, a deeper hierarchical level is defined and when the definition of the child is finished, we come back to the upper level).

Besides the information about the hierarchical level, the script gives information about the objects of the scene (here, "objects" means geometric objects, light sources and cameras). For example, the script contains information about the geometry of the objects, the position and direction of light sources, etc. This information, although not directly necessary to select the sub-trees, can be used to help the user in knowing with which object he is dealing and to facilitate future extensions of the application (for example, the client could be able to change the characteristics of an object). To transmit this information, the script uses the same keywords as the VRML 2.0, ensuring consistency.

Figure 3 represents a possible structure of a VRML 2.0 file.

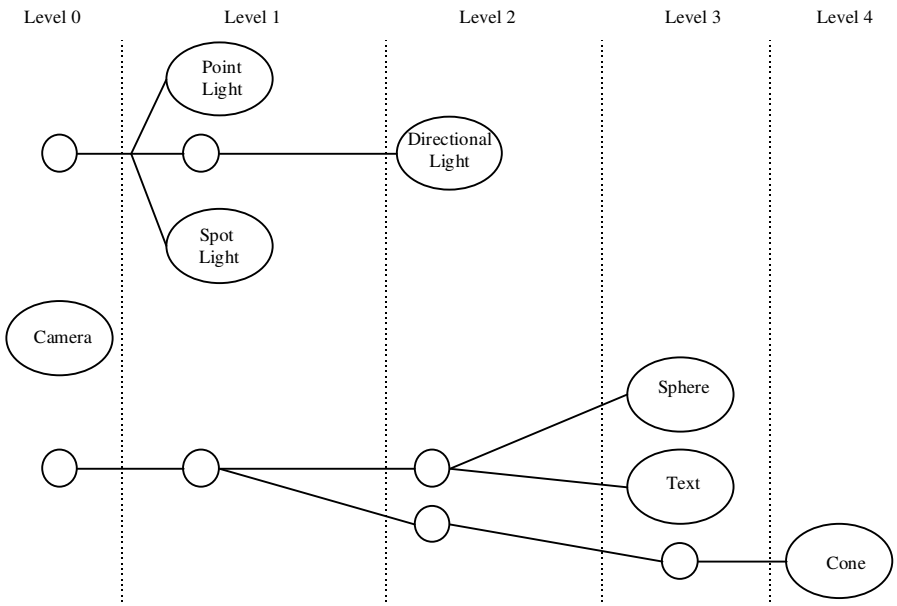

Figure 3: Example of a VRML 2.0 hierarchical tree.

The script for the file represented in Figure 3 would be:

```
1
PointLight
   location x y z
2
DirectionalLight
   direction x y z
1
SpotLight
   location x y z
   direction x y z
```

```
0
Viewpoint
   position x y z
   orientation x y z
1
2
3
Shape
   geometry Sphere
Shape
   geometry Text
2
3
4
Shape
   geometry Cone
3
2
1
0
```

The `x`, `y` and `z` in the previous script are only symbols, representing the numbers in the actual script.

Instancing (keywords “DEF” and “USE” in VRML 2.0) is also treated by the script (it delimits the definitions and transmits the reuses of them). The following example defines a sphere called „ball“ at level 1 and uses it further in another hierarchical level.

```
1
DEF ball
Shape
   geometry Sphere
ENDDEF
2
3
USE ball
2
1
0
```

The nodes “Switch” and “LOD of the VRML 2.0 are also considered in the script. The first allows to choose one among several children nodes (all the children and the choice are passed in the script). The “LOD” node allows to give different definitions for an object, according to the distance from it the observer is located. All these definitions are passed in the script. The next example shows a script with the “Switch” node used to select between two cameras (the first one is selected) and a “LOD” node, defining two shapes for an object.

```
1
Switch
   whichChoice 0
Viewpoint
   position 10. 10. 20.0
   orientation 0 0 1
Viewpoint
   position -10. 10. 15.0
   orientation 0 0 1
```

```
ENDSwitch
0
1
LOD
Shape
   geometry Sphere
Shape
   geometry Cone
ENDLOD
0
```

Some VRML 2.0 files can be found in the directory http://www.igd.fhg.de/~alberto/worlds, and their scripts can be seen by executing the current version of the client application (http://rick.igd.fhg.de:8080).

4. References

[Cornell (1996)] - G. Cornell and C. S. Horstmann, *Core Java*, SunSoft Press, 1996.

[Dimension X (1996)] - *Liquid Reality*, Dimension X, 1996. (*http://www.dnx.com/products/lr/*).

[Drake (1996)] - D. G. Drake, *Introduction to Java threads*, Java World (online magazine), v.1, Issue 2, April 1996 (*http://www.javaworld.com/javaworld/jw-04-1996/jw-04-threads.html*).

[Hoff (1996)] - A. van Hoff, S. Shaio and O. Starbuck, *Hooked on Java: Creating Hot Web Sites with Java Applets*, Addison-Wesley Publishing Co., 1996.

[Lemay (1996)] - L. Lemay and C. L. Perkins, *Teach Yourself JAVA in 21 Days*, Sams.net Publishing, 1996.

[Sun (1995)] - *The Java Language Specification*, Sun Microsystems Computer Corporation, 1995.

[VRML2.0 (1996)] - *The Virtual Reality Modeling Language Specification - Version 2.0*, ISO/IEC CD 14772 - August, 1996. (*http://vrml.sgi.com/moving-worlds/*).

Appendix A

General observations

A.1 The application in LCA - FEEC - UNICAMP

The application described in this document, as stated before, has been developed during the visit at the ZGDV, Darmstadt, Germany. For this reason, the application server and the VRML 2.0 examples are located in their WWW sites. However, the application have already been successfully installed in the LCA - Laboratory of DCA - FEEC - UNICAMP, and future improvements on it are going to be implemented in this laboratory. Because of that, and also because it is not guaranteed that the application will continue running properly in Germany, it is strongly suggested that the application be executed from UNICAMP.

In LCA, the application server is currently the machine caolho.dca.fee.unicamp.br, and the application's home page is now http://www.dca.fee.unicamp.br/~alberto/vrmlExtractor/index.html.

Some examples of VRML 2.0 worlds can be found at the directory http://www.dca.fee.unicamp.br/~alberto/worlds.

A.2 Deficiencies encountered

The tests made with many VRML 2.0 worlds have detected some problems in the application that have not been solved yet, but they should be corrected in a near future.

- The application doesn't consider nodes declared in other VRML files, i.e., "Inside" nodes are ignored in the model of the hierarchical structure of the VRML (Inside nodes are used to make a link with a node that is located in another VRML file).
- A Switch node must have all its nodes of the same type (or geometric objects, or cameras, or light sources). Although generally a Switch node is used to select among nodes of the same type, there is nothing in the VRML 2.0 specification that obligates all the Switch children to be of the same type. If this happens, the selection among the children could not work correctly.
- If a Switch node follows another Switch node with the same type of children, in the same hierarchical level, things may not work properly.
- If the number of nodes of the same type in a hierarchical level is too large, there could be not enough space for all of them in the client's interface window.

- The user interface is very simple, not showing the hierarchical structure of the world in a graphical way.

Other deficiencies of the application are related to problems of the current beta version of Liquid Reality. These problems are supposed to be solved in the final version of the tool. Refer to http://www.dnx.com/products/lr for more information about this tool.

- Liquid Reality doesn't run properly under Netscape 3, it must be executed using Netscape 2.x (actually, it is not guaranteed that it would work correctly under any version of Netscape; it works properly only under the appletviewer of the JDK - Java Development Kit [Cornell (1996)], [Hoff (1996)], [Lemay (1996)]).
- Liquid Reality doesn't implement the complete specification of VRML 2.0 yet. For instance, Text nodes and some kinds of textures are not implemented in the current beta version..

Appendix B

Source Code