VRML parser in Java

Filip Sixta
E-mail: xsixtaf@hwlab.felk.cvut.cz

Czech Technical University
Faculty of Electrical Engineering
Deptartment of Computer Science and Engineering
Prague / Czech Republic

Abstract

A VRML parser in Java language, as a part of a specialized VRML browser for blind people, is presented in this paper. The parser reads an input URL with a VRML 2.0 scene description and creates an inner representation of the scene tree structure and the list of event routes, used by the other modules of the browser.

A part of the input file analysing block contains also the management of the event route list and the event queue. Therefore, this work includes not only the parser, but represents a complete base-layer, called VRML API. It consists of classes representing the entities of VRML, especially of the class Browser, whose methods serve other modules for obtaining the inner structures and some basic properties of the system.

There will be shown, how the inner scene structure is built from instances of class Node, how the route list and event queue are managed, how these and other VRML objects are specified, how the input file analyser works etc.

KEYWORDS: VRML, virtual reality, Java, parser.

1 Introduction

The goal of this work was to create **an analyser of the dynamic virtual scene describing language VRML 2.0**. It was a part of so called **project "Blind's dog"**, which takes place in the Dept. of Computer Science and Engineering at Czech Technical University, and whose goal is to create a complete navigating system for blind people. The idea that led to the birth of this project is to allow people handicapped in this way to browse virtual worlds. The project is an attempt to provide them the advances of virtual reality, like conventional VR-navigators provide to other people.

Obviously, the navigating system for blind people is noted for some differences from that conventional. The weight is not laid on the output graphics processing - instead of this, the system must have special module, that forms an audio output information about the scene topology, the user position in the scene (user's environment), his interactions with the scene and dynamic behaviour of the scene etc.

Similarly to a general virtual reality browser, this system can be demonstrated as a connection of specialized blocks for input scene analysing, scene-dynamic

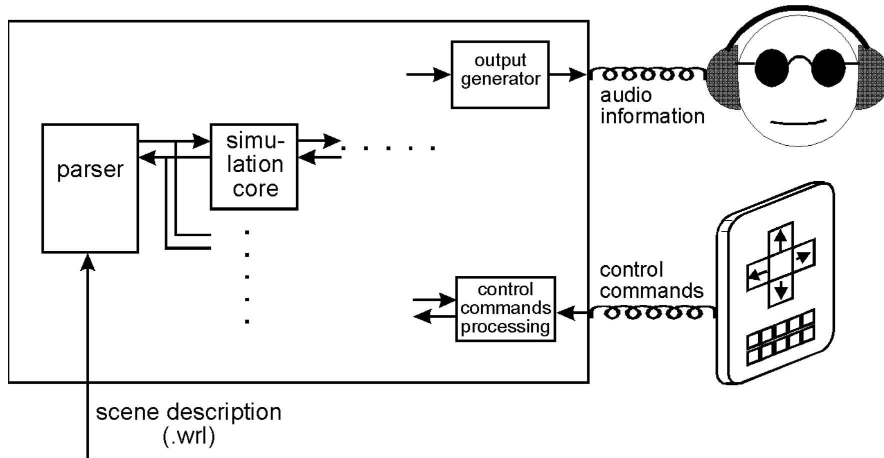

Figure 1: The structure of navigating system for blind people

processing, topology analysis, user commands execution, output generation etc. The rough structure of the browsing system is depicted in fig. 1.

It is obvious, that the input scene analyser represents the first module in the scene processing pipeline. Its more detailed context is shown in fig. 2. The role of the parser is to read and analyse the input file (URL) describing a virtual scene and to create the **inner tree structure**, used by other modules in the system. To this structure also belongs the **list of event routes** used for the inter-node communication and defined also in the input file (as **ROUTEs**). However, the parser treats the scene as static, i.e. with the inner structure creation it's particular role ends, and the following changes and dynamic behaviour of the scene objects is a matter of other modules (especially the simulation core).

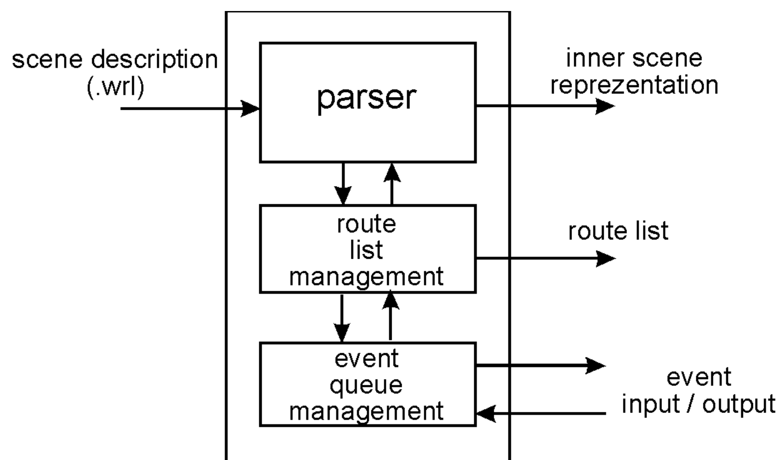

Figure 2: The structure of the input file analysing block

Due to the primary role of the parser implementation in the project, which implies the need of designing a base-layer of the system, this work includes also the implementation of route list and event queue managing functions. Via these functions, these data structures are provided to the simulation core after the input file analysis.

2 VRML API

2.1 Motivation

Before the implementation of parser, an important step had to be made - to design a collection of classes and their methods, which provide functions of parser and other input file analysing module parts to various other modules of the system. This collection is called **VRML application programming interface** (**VRML API**), and represents the above mentioned fundamental layer. It includes a class for the browser itself, VRML-scene nodes, their attributes, routes, events, fields of all types etc. Because the system is implemented in Java language, the VRML API can be covered by members of so called **package** *vrml*.

The VRML API presented here was inspired by the API described in VRML 2.0 specification, appendix C - *Java Scripting Reference*. Although this interface was designed for interacting of *Script* nodes with their associated scenes, a noticeable part of it seemed to be suitable for any powerful VRML scene browsing system. This holds e.g. for most of methods of the class *Browser*, class *Event*, and whole package *vrml.field*, representing the VRML 2.0. **data types**.

Java Scripting Reference was not the only one source of ideas for designing the VRML API. Very interesting, from this point of view, is JavaSoft's product **Java 3D API**, which provides suitable VRML support. Unfortunately, this API wasn't implemented yet, when this project and its parts started to be worked out. Moreover, we tried to simplify the VRML API in the sense that all VRML 2.0 node types were represented by instances of the same class (named *Node*), and differed by their attributes, while Java 3D API provides particular class for each of the standard node types. Therefore our philosophy would be probably broken by using this API. Another fault was, that Java 3D API has not very suitable classes for primitive VRML 2.0 data types (field types).

For all these reasons, the use of Java 3D API was dismissed. However, some parts of this API can be pretty exploited for other entities (not so exclusively specific for VRML), such as geometry and mathematical objects, transformations etc., but it concerns rather with some specialized modules (e.g. topology builder).

2.2 Package *vrml* and its function

Appendix A shows the hierarchy of classes of package *vrml*, in the context of **Java Platform 1.1.4 Core API**. The implementation of classes with the names typed in **bold** was the object of this work. There can be seen, among others, the classes mentioned above: the class representing the interface of whole navigating system (*Browser*), classes for resources and entities of VRML - nodes, prototypes, routes, events, node attributes and data types, classes for the parser itself (*LexicalAnalyser* and *SyntaxAnalyser*) and the system of exceptions, specific for manipulation with all these classes.

2.2.1 Browser

First of all, let's look at the class *Browser*, which represents the core of the API. It holds and provides information about the properties and the state of the navigating system; especially it provides functions of the input scene analyser and route-list and event-queue management. Browser's declaration is very similar to that one described in VRML 2.0 specification, section 4.7.10 - *Browser Script Interface*, and both declarations considerably penetrate. The declaration of the

class Browser, with the simple explanation of the function of it's methods is in the Appendix B.

The class is not static - the constructor *Browser* creates a particular browser instance with it's own, independent properties, which implies the theoretical possibility to use more browser instances in the framework of the navigating system, and to switch between them.

In the scope of this work, the most important method is *createVRMLFromURL*. This method gets a URL of scene description in VRML 2.0 and returns the inner representation of the scene - as a list of pointers to root nodes of the scene tree structure (or better forest structure). As a side effect of the creation of the inner scene structure, there is also created the list of event routes. Inside this method, the syntax analyser is initialized and the input scene description is passed to it. One of the following chapters discusses it in detail.

2.2.2 Classes for VRML entities

Node, *Attribute*, *Prototype*, *Route*, *Event*, *Field* and their descendants are representing the VRML entities. This part of the article briefly explains their sense.

The special attention, from the viewpoint of the semantics, should be given to the classes *Node*, *Attribute* and *Prototype*. It could be seen, that there are no specific classes for different node types - all VRML scene nodes are represented by instances of the same class named *Node*, including both the nodes of standard type (*Transform*, *Shape* etc.) and the user defined. The only exception is the *Script* node, which has it's particular class *Script* - the descendant of class *Node*, because of it's specific properties.

The type of the node represented by the instance of class *Node* is indicated by the type name (held by the object) and by associated list of attributes. In this context the word 'attribute' means the node parameter, which can be field, exposed field, input or output event. For attributes, there is class *Attribute* established. Objects of this type are specified by the kind (*field*, *exposedField*, *eventIn* or *eventOut*), the data type (*SFBool*, *MFNode* etc.), the name (*size*, *children*, *fraction_changed* etc.) and in the case of field or exposed field the pointer to an instance of some descendant of the class *Field* (as explained later).

When creating a new node, the node type and the attributes associated with this node type must be known. In other words, there must exist the appropriate **prototype**. Hence, there exists the class *Prototype*, which instances define for the respective node types the list of attributes and their default values. These instances are created by the syntax analyser on the basis of the PROTO definitions in the input file. (As will be shown later, this holds also for the standard node types.)

The routes - instances of the class *Route* - are determined by the **sending node** and it's **output event** (a specified *eventOut*) and the **receiving node** and it's **input event** (a specified *eventIn*). Routes are elements of the route list, which is held by the browser, and which is a part of the inner scene representation. Analogously, instances of the class *Event* form the event queue. They are determined by a **sending** or **receiving node**, respectively, a **name** and a **time stamp**. Objects of both these types (routes and events) can be created, added to and deleted from their respective sets, and their parameters can be obtained, but they cannot be explicitly modified.

The set of descendants of the abstract class *Field* represents all single (SF-) and multiple (MF-) VRML **data types**. Instances of these classes contain values of the appropriate type. Every class provides several methods for setting and

getting these values; types of the values passed into them and returned by them are in most cases the primitive data types of the Java language. For example, if the *SFInt32* object inner value is set or get, the value of Java type *int* is passed, in case of *SFRotation*, the four-tupple of *float* values is passed and so on.

3 The input file analyser

3.1 Principles of the analysis

The most important part of this work is the **syntax analyser** of the VRML scene description. The parser is based on the principle of recursive descending. The process of analysis is derived from the grammar defined in VRML 2.0 specification, appendix A - *Grammar*, which is transformed to LL(1) grammar. This transformation is made due to the easier implementation of the parser (the analysis is then deterministic) and brings some little changes, as shown in the following example:

Rules of the original grammar:

```
<mfcolorValue>  ->  <sfcolorValue> |
                    [ ] |
                    [ <sfcolorValues> ]  ! two right sides start
                                           with the same symbol
<sfcolorValues> ->  <sfcolorValue> |
                    <sfcolorValues>      ! the left recursivity
```

Corresponding rules of LL(1) grammar:

```
<mfcolorValue>  ->  <sfcolorValue> |
                    [ <mfcolorValue'>
<mfcolorValue'> ->  ] |
                    <sfcolorValue> <mfcolorValue'>
```

The syntax analysis are performed by two classes of the package *vrml* - *SyntaxAnalyser* and *LexicalAnalyser*. The class *SyntaxAnalyser* provides a method named *vrmlScene*, which is called from the *Browser*'s method *createVRMLFromURL*, and which responses to the starting symbol of the VRML grammar. From this method, the methods for specific statements (such as node, proto or route statement) are iteratively called, according to the content of the input file.

Every non-terminal symbol in the modified LL(1) grammar has it's appropriate method in *SyntaxAnalyser*. The terminal symbols are considered to be the **lexical elements** and they are read (parsed) by the **lexical analyser**. They include keywords, braces, brackets, the point, identifiers, numbers and strings.

Parsing of each particular statement results in creation of the appropriate VRML object - especially the node-statement parser returns a representation of a VRML node (namely the object of type *Node*) and the route-statement parser returns a representation of a route (the object of type *Route*). The whole parsing then results in the inner structure, consisting of the list of root nodes of VRML scene and the list of routes.

The process of analysis is roughly illustrated in fig. 3. In this figure, there is denoted another important thing, touching the semantic knowledge of the standard node types. Because of the uniform approach to both standard and user-defined node types, both of them must have their prototypes. Hence, the knowledge of the standard prototypes is achieved using the file with the standard PROTO-definitions, which is an inseparable part of the parser.

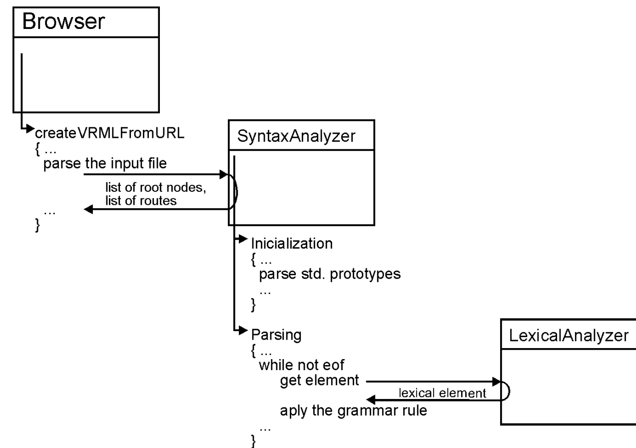

Figure 3: The process of the input file analysis

3.2 The inner structure construction

The next example shows, how the inner tree structure of the VRML scene is built up by the parser. Let's have the following VRML world file fragment:

```
...
  Transform {
      translation 3 0 1
      children [
          DirectionalLight {
              direction 0 0 -1
          }
          Shape {
              geometry Sphere {
                  radius 2.3
              }
          }
      ]
  }
...
```

The fragment is a node statement. Therefore, the appropriate method for node parsing is called, when the identifier 'Transform' is read. This method calls sequentially the method for parsing node attributes and it consequently calls parsing of the particular fields. Each called method returns the appropriate object (e.g. an attribute), while the calling method composes objects of higher level (e.g. a node). Then the result of parsing the previous code fragment can be depicted by fig. 4.

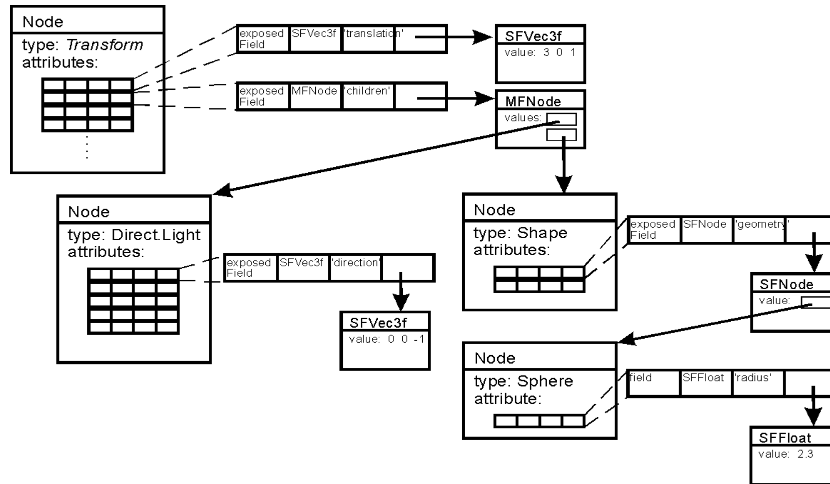

Figure 4: The inner VRML scene representation

3.3 Route list and event queue management

Important parts of the inner representation of the world are the **list of routes** and the **queue of events**. None of them is returned directly by the method *createVRMLFromURL* to the calling block, but they are members of *Browser* instance and they are accessible by appropriate methods.

The route list is built up by the syntax analyser, which adds the routes to it, as a result of parsing the ROUTE-statements. When the simulation core of the browsing system has to simulate the event cascades, it searches through the route list, to determine, how to pass the events between nodes.

The event queue is not so tightly bound to the parser - it is only a preparation for the simulation core, which uses it for buffering events during processing the event cascades. The class *Browser* provides these methods for the event queue manipulation: *addEvent*, *getEvent* and *getEventQueue*. In the contradistinction to routes, which form an unordered list, events form a queue, ordered ascendantly by the **time stamp** - the absolute time value of the event rising moment. It is guaranteed, that the *getEvent* method returns an event with the lowest time stamp, every time it is called, and this event is immediately removed from the queue.

4 Testing

For testing the capability of the parser, a set of scenes was created, using various features of VRML. The tester was a simple program, which only let the parser to read the scene and create it's inner representation and then it took this representation and transcribed it to the text output. As better testers would serve the other modules of the navigating system, but unfortunately, developing of them did not proceed simultaneously with the parser. Therefore, only the module for the topology processing was utilizable, in the time of testing the parser. It processes, however, only the static information of scene. The best part for testing would be probably the simulation core, which would test also the event management perfectly, but it is currently being implemented.

The results of testing can be viewed in three levels:

1) Some scenes are not parsed at all, because they use features, which the parser cannot process. This concerns in fact only the EXTERNPROTO state-

ments. The parser does not read other URL's during parsing of the scene, and hence the external declaration is not known. Therefore, the parser cannot analyse the node of user-defined type, because of missing it's prototype.

2) Some scenes can be parsed, but the inner representation is not proper - the parser then serves only for the syntax control. This happens, for example, when the scene uses the *Script* node. It is well read, but not interpreted, because there is not implemented any scripting language interpreter. It also happens, when there is a user PROTO-definition in the scene, i.e. the PROTO statement with a non-empty body. The parser uses only the declaration of attributes in the PROTO statement (the part in the brackets), which is sufficient for standard node types; the body (the part in the braces) is read but ignored.

3) For the rest of scenes the parser works well and the inner structures are built up completely and correctly. This concerns the node statements with any standard node type except Script, the node instancing (DEF - USE), and ROUTE statements. It concerns also the PROTO statements with an empty body, i.e. only the declarations of attribute scheme.

5 Conclusion and future works

The VRML parser presented here was the first implemented part of a specialized VRML browser. Therefore, it was necessary to create a basic platform, which would connect this part with all those succeeding. VRML API serves as this platform. Regarding the use of Java language, this API is represented by so called package *vrml*, containing classes specific for features and entities of VRML.

The parser uses the VRML 2.0 grammar, modified to LL(1) type in order to make a deterministic syntax analysis of the input file. The result is an inner representation, containing the scene structure and the list of routes. VRML API provides objects, which these structures are built from, and which allow other modules of the system to manipulate with them.

Capabilities of the parser are limited, because not every feature of VRML was implemented in the scope of this work yet. There are many things, that should be worked out in the future. It includes e.g. the complete processing of PROTO and EXTERNPROTO definitions. Very important, but also difficult, is the involving of any scripting language interpreter, in order to allow the *Script* nodes to be processed.

In the next phase of the parser development, there will be created an optimization mechanism, which reduces various unnecessary difficult relations in the scene structure, such as transformation hierarchies of the static parts of the scene. It should be a precaution against the efficiency losses of succeeding modules, such as the topology processor.

There can be surely found other suggestions for the improvement of this part of the browsing system. It is fairly dependent on the requirements of developers of the other parts of the system and of the future users of the system.

References

[1] B. Melichar: *Jazyky a preklady.* Edition center of CTU, Prague, 1996.

[2] *International Standard ISO/IEC 14772-1:1997 – VRML97 Specification.* The VRML Consortium Inc. 1997.

[3] Sun Microsystems, JavaSoft: *JDK 1.1.4 API Documentation.* 1997.

Appendix A: Package *vrml*

- class java.lang.Object
  - class **vrml.Browser**
  - class **vrml.Node**
    - class **vrml.Script**
  - class **vrml.Prototype** (implements java.lang.Cloneable)
  - class **vrml.Route**
  - class **vrml.Event** (implements java.lang.Cloneable)
  - class **vrml.Attribute**
  - class **vrml.Field** (implements java.lang.Cloneable)
    - class **vrml.MField**
      - class **vrml.MFColor**
      - class **vrml.MFFloat**
      - class **vrml.MFInt32**
      - class **vrml.MFNode**
      - class **vrml.MFRotation**
      - class **vrml.MFString**
      - class **vrml.MFTime**
      - class **vrml.MFVec2f**
      - class **vrml.MFVec3f**
    - class **vrml.SFBool**
    - class **vrml.SFColor**
    - class **vrml.SFFloat**
    - class **vrml.SFImage**
    - class **vrml.SFInt32**
    - class **vrml.SFNode**
    - class **vrml.SFRotation**
    - class **vrml.SFString**
    - class **vrml.SFTime**
    - class **vrml.SFVec2f**
    - class **vrml.SFVec3f**
  - class **vrml.LexicalAnalyser**
  - class **vrml.SyntaxAnalyser**
  - class **vrml.DEFNode**
  - class java.lang.Throwable (implements java.io.Serializable)
    - class java.lang.Exception
      - class **vrml.InvalidVRMLSyntaxException**
      - class java.lang.RuntimeException
        - class java.lang.IllegalArgumentException
          - class **vrml.InvalidEventInException**
          - class **vrml.InvalidEventOutException**
          - class **vrml.InvalidExposedFieldException**
          - class **vrml.InvalidFieldException**
          - class **vrml.InvalidProtoNameException**

Appendix B: Browser interface

```
public class Browser
{
    public Browser();

    public String getName();
    public String getVersion();
    public float getCurrentSpeed();
    public float getCurrentFrameRate();
    public String getWorldURL();

    public Node[ ] createVRMLFromURL(String url)
        throws InvalidVRMLSyntaxException, IOException;

    public void addRoute(Node fromNode, String fromEventOut,
        Node toNode, String toEventIn);
    public void deleteRoute(Node fromNode, String fromEventOut,
        Node toNode, String toEventIn);
    public Route[ ] getRouteList();

    public void addEvent(Node node, String eventName,
        double timeStamp, Field eventValue);
    public Event getEvent();
    public Event[ ] getEventQueue();

    public void setDescription(String description);
    public void setStandardProtoPath(String path);
}
```

getName, getVersion:
- return the name and the version of the navigating system.

getCurrentSpeed, getCurrentFrameRate:
- refer about dynamic properties of the browser; not implemented yet

getWorldURL:
- returns a string with the URL of currently loaded world.

createVRMLFromURL:
- gets a string with URL of the scene description, parses it and makes the inner representation. The method returns the list of root nodes of the inner forest structure. Except this, it fills the list of routes.
The method throws InvalidVRMLSyntaxException, if an error occurs during parsing the scene. It can happen not only when there is a trespass against the VRML syntax in the input scene description, but also when there is an unknown node type (i.e. without preliminary PROTO-definition).

addRoute, deleteRoute:
- modify the list of routes (add and delete items). The arguments are attributes determining the route.

getRouteList:
- provides the list of routes.

addEvent:
- adds an event to the event queue. The event is determined by the arguments and is inserted to the queue with the respect to the timestamp.

getEvent:
- returns an event with the lowest timestamp and removes it from the event queue.

getEventQueue:
- provides whole event queue and does not remove anything.

setDescription:
- sets the current description of the browser. There is no usage of this property yet.

setStandardProtoPath:
- sets the path to the PROTO definitions of the standard VRML node types.