Implementation Notes of a VRML 2.0 Browser

Ning WU*, Takami YASUDA** and Shigeki YOKOI**

*Graduate School of Human Informatics
Nagoya University
Furou Chyou, Chikusaku, Nagoya, 464-01 JAPAN
wuning@yokoi.gs.human.nagoya-u.ac.jp

** School of Informatics and Sciences
Nagoya University
Furou Chyou, Chikusaku, Nagoya, 464-01 JAPAN
yasuda@info.human.nagoya-u.ac.jp
yokoi@info.human.nagoya-u.ac.jp

Abstract

In this paper, we present some methods for implementation of VRML 2 browser. Topics involved are implementation of Event model, PROTO and implementation of Java in the Script node.

Key words: Browser Implementation, Virtual Reality, VRML

1. Introduction

Every successful industry must have some kind of standard to exchange information. So do the field of Virtual Reality. A standard to exchange VR information is essential. VRML [1][2] is such a standard. VRML stands for Virtual Reality Modeling Language. It is a description language that is used to publish 3D virtual world on the Internet. VRML 1 was published in May 1995. Its ability is limited to the description of static worlds. VRML 2 was finalized in August 1996. It mainly extends VRML 1 in the area of behavior -- sounds, moving objects, animations and user interactions.

Developing a VRML 2 compliant browser is a difficult job. There are many VRML 2 browsers being developed [3][4][5][6], but none of them is fully compliant to the specification. The most frequent postings on VRML mailing List are VRML 2 files that do not work on current VRML 2 browsers. The problem of incompatibility between different VRML 2 browsers may split the unity of VRML society and seriously damage the standard's future. In order to come out with an optimized fully compliant reference implementation, it is necessary to discuss various aspects of the implementation details of VRML 2 browser.

In this paper, we present some practical implementation methods for the most difficult yet basic aspects of VRML 2 browser. Topics involved are implementation of Event model, PROTO and implementation of Java in Script node. They are causing most compatibility problems. We have not found publicized papers or articles on the implementation details of these areas.

Section 2 deals with implementation of VRML event model. We show an internal data structure that can implement the event model in VRML 2 specification very efficiently. It supports all the features of VRML 2 such as event cascade, fan-in (multiple eventOut routed to one eventIn) and fan-out (one eventOut routed to multiple eventOut) with the least overhead.

Section 3 deals with implementation of PROTO. We propose a method that will solve nested PROTO instantiation problem, default field value initialization problem and DEF/USE pointer recovery problem simultaneously. The method is based on object persistence. It enables a browser developer to manipulate PROTO node types in the same way as a build-in node type.

Section 4 deals with implementation of Java in Script node. We propose a method that requires no Java

Virtual Machine source code to implement the Java Script node interface. The requirement of Java VM source code is a major reason that Java did not become the standard VRML script language.

2. Implementation of VRML event model

The most significant difference between VRML 1.0 and VRML 2.0 is that VRML 2.0 has behavior, which means a VRML 2.0 world may have sound, moving objects, animations and user interactions. Unlike traditional procedure language that defines the detailed flow of a behavior by procedure calling, VRML is generally a descriptive language that only describes what may behave in a Scene. (The VRML Script nodes are based on procedure models. We think that may be a drawback of VRML). The behavior data flow in VRML is indicated by Event object generation and Event cascades through eventOut, ROUTE and eventIn[2].

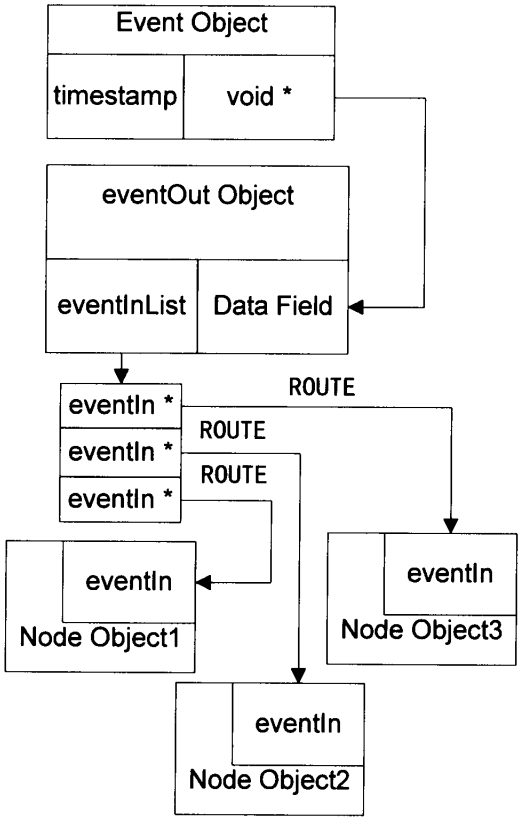

Fig.1: The relations between Event, eventOut, eventIn and ROUTE.

Most Events in a VRML world are generated by Sensor nodes, Collision node or Script nodes. The Events are generated because of time elapse, user interaction or Event cascades. Once an Event is generated in a node (which means the node must have an eventOut field) and there is a ROUTE from the eventOut to another node's eventIn, data exchange happens, which may result in a behavior of the virtual world.

In our implementation, Events and eventOuts are treated as objects; eventIns are considered as object's member methods; ROUTEs are converted into member method pointers during initialization. The relations between Event, eventOut, eventIn and ROUTE are summarized in Fig. 1.

2.1 The structure of Event

An Event Object has two fields: a data field that specifies the specific Event information and a field that contains the timestamp the Event occurs. The Event object may be used in the form of an Event array, it is desirable to have a fix-sized Event object. Because the Event information may vary in size from a SFFloat to a MFString, we put a "void *" pointer in the Event object. The pointer can be casted to a specific field type at runtime. The timestamp may be used to synchronize the Event cascade, when an eventOut is routed to multiple eventIns. Subsequent Event of an Event cascading has the same timestamp of the initial one. It can also be used to stop Event cascade looping as in the situation of Fig. 2:

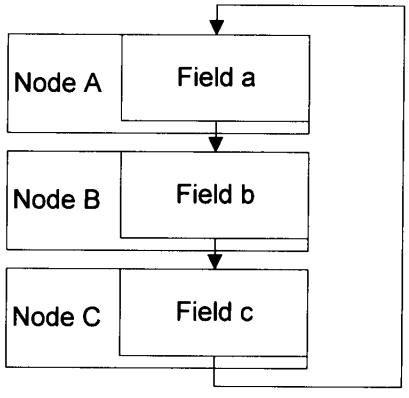

Fig. 2: A legal usage of Event cascade that may cause Event looping

If one wants to keep the value of the exposed fields a, b, c of objects A, B, C respectively always equal, the most straightforward way is to ROUTE the three

fields together (an exposed field has the functionality of both an eventOut and an eventIn). If one of a, b, c is changed, the other two will also be changed by Event cascade, and Event cascade looping can be prevented by stopping the cascade when the second Event with the same timestamp arrives.

2.2 The structure of eventOut, eventIn and the representation of ROUTE

A VRML world consists of VRML node objects.

Some node objects contain data fields that are eventOut objects. An eventOut Object has two fields: a data field that contains a specific Event information and an eventInList. The data field is the field that the "void *" pointer in an Event object points to. The eventInList contains the eventIns that are routed from the eventOut.

An eventIn is a member method of a node object. It takes only one parameter: an Event object.

When a VRML file is processed, the ROUTE statement is converted into items of eventInList in the eventOut object that the ROUTE originated from. If node A's eventOut O is routed to node B's eventIn I, O's eventInList will contain an entry pointing to I.

2.3 How an Event go from eventOut through routes into eventIn

When a node generates an Event and sends the Event to its eventOut, first, an Event object with the current timestamp will be constructed. Then, the Event object's "void *" pointer will be set to the eventOut object's data field. Finally, for every entry in the eventOut's eventInList, the eventIn member method will be called with the Event object. Because the specific Event information type is predefined in VRML 2.0 specification, "void *" pointer of the Event object can be converted into specific data type in the eventIn method, and data field of the eventOut can be accessed through the pointer.

2.4 How Event cascade happens

In the procedure of eventIn, new Event object can be created based on the input Event. Because the eventIn is a member function of a node object, the newly created Event can be sent to the node object's eventOut. If the eventOut has a ROUTE to another node object's eventIn (the eventOut object's eventInList is not empty), Event cascade happens.

2.5 Other notes

From the above description of the internal data structure of VRML event model, the implementation of fan-in (multiple eventOut routed to one eventIn) and fan-out (one eventOut routed to multiple eventOut) is straightforward. Fan-in means eventIn method pointer can exist in more than one eventOut object's eventInList, which is no problem; fan-out means an eventOut object's eventInList may contain more than one eventIn entry, which is why we put a list there.

3. Implementation of PROTO

Introduction of PROTO is an important advantage of VRML 2 over VRML1. New functionality can be introduced by defining a PROTO -- a user defined node type that has the functionality of multiple build-in nodes. A VRML 2 browser recognizes the PROTO definition and treats an instantiated PROTO like an ordinary build-in node. New primitives, components such as NURBS surface can be defined. Many disputes about whether a specific feature should be added to VRML 2 ended because the feature can be implemented by PROTO.

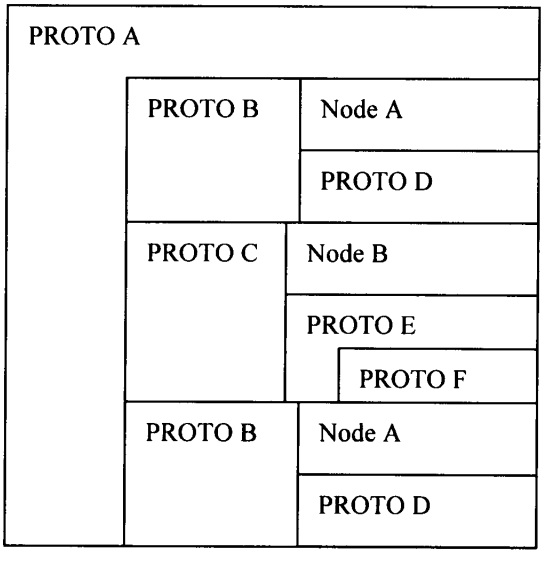

Fig. 3: A nested PROTO may cause Event looping

3.1 Difficulties of implementation

Browser developers must face difficulties of implementing nested PROTO, default initial values of fields and especially the DEF and USE pair within PROTO definition.

Nested PROTO means each PROTO definition may contains multiple nodes, and a previously defined PROTO can be used in a later PROTO definition just like the build-in node, thus create a node hierarchy that has to be reproduced during PROTO instantiation (Fig. 3).

The standard build-in node's default field value is initialized in its default constructor. But if a node becomes a sub node of a PROTO, its field's default initial value may be required to be different from the value in the default constructor. Another way should be found to fill in the default field values.

The DEF and USE pair is a one-object multi-use scheme that is suitable to present objects with the same physical attributes that differ only in space position and/or orientation. One example is the four tires of a car. Because the DEF and USE pair is internally represented by multiple pointers pointing to the same object, changing the appearance of one tire will result in changes in all the four tires, which is desirable. On the other hand, PROTO instantiation is different. Each instance of a PROTO is itself an object. Changing one instance's attribute will not affect the others. If use DEF and USE pair in the PROTO definition, trying to reproduce the pointing structure when PROTO is instantiated will be a headache.

Because a browser has to treat a build-in node instance and a PROTO node instance the same way, it is desirable to have both kinds of instance instantiated in the same way. A build-in node has a corresponding C++ class. Making a new build-in node instance is as simple as making a new C++ object using the constructor. For a PROTO, an ordinary approach might be the C++ copy constructor approach: implementing a general PROTO class as some kind of container; making a template PROTO object when parsing the PROTO definition (so far so good); using C++ copy constructor to reproduce the object (which is very similar to new a build-in node) when there is a request for PROTO instantiation. Unfortunately, this approach simply does not work well.

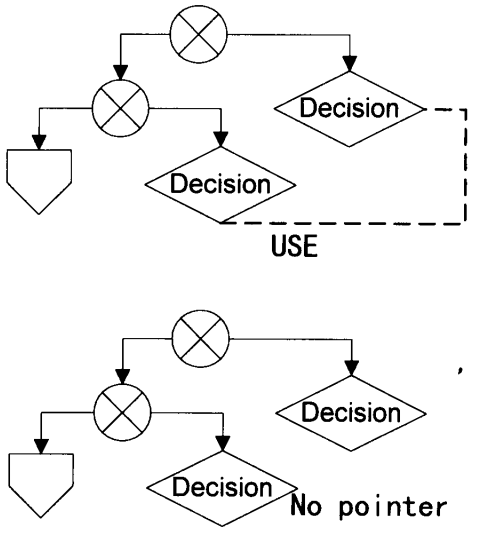

Fig. 4:Incorrect PROTO instance copy with a USE pointer by making a copy of the object the pointer pointed to

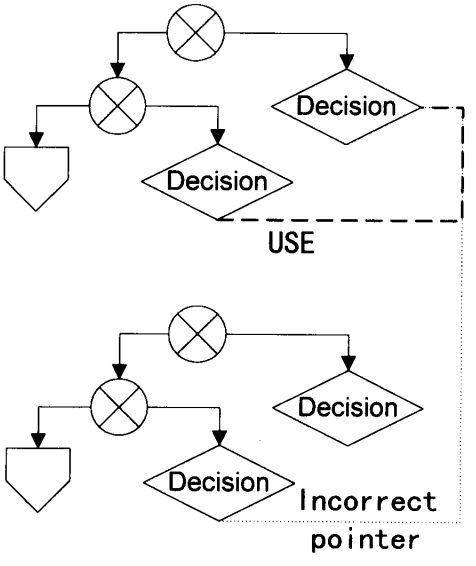

Fig. 5:Incorrect PROTO instance copy with a USE pointer by making a copy of the pointer itself

Let's assume the PROTO to be a container node such as Group. Because C++ class encapsulation, in the situation of a well-designed C++ class library, the container object cannot operate directly on the data fields of the nodes in the container. During the copy construction, it is certainly no problem to make a copy of the container own data structure, but problems arise when making copies of the contents in the container.

There are two sorts of contents in the container. One is a real node object (a DEF), the other is a pointer to the real node object (a USE). For the real object, copying the object is easy. But for the pointer (the USE), correct result cannot be obtained by either making a copy of the object the pointer pointed to (Fig. 4) or making a copy of the pointer itself (Fig. 5). Correct result can only be obtained by tracing all the DEF/USE pairs across container hierarchy, which will break C++ encapsulation. It will become quite annoying in the nested PROTO situation.

The above three problem may be solved by keeping tracks of all the PROTO definition and parse the PROTO definition tree every time when a PROTO is instantiated, but we do not think it is an elegant way.

We solved the problem with a more general approach that is based on object persistence.

3.2 Object persistence method

I read the concept of Object persistence in a book about Object C published in 1986. While an object has all the sorts of advantages such as inheritance, polymorphous over an ordinary data structure, it has a large drawback when the power switch of the computer is turned off and the object is eliminated from the memory: the object states will be lost. The rebuilding of a specific object state from traditional data files is tedious and slow. There is a way to save objects' state between working sessions, which is called Object Persistence. The first C++ object persistence implementation I have seen is Borland C++ 3's OWL library, which introduced TStreamble class. Microsoft MFC combined the same concept in its class library's Serialize method.

The basic idea of Object persistence is to save and restore the object's memory image, which consists of class type (the invisible C++ virtual table) and the object's data members. Ordinary data members can be saved and recovered like a traditional data structure, which one writes and reads sizeof(the data structure) bytes to and from the disk file. The tricky part is data members that hold pointers to other objects. Pointers are memory addresses. It is not likely that an object may have the same memory address in two different working sessions. So it is meaningless to save and restore the pointer's value itself. There must be a scheme to trace what object the pointer pointed to and recover the pointer's value according to the object's actual memory address. This is quite similar to our DEF/USE pair headache in nested PROTO instantiation.

The instantiation of a PROTO is achieved by serializing the template PROTO object into a memory stream and reconstructing the instance from the memory stream through Object persistence interface, which ensure the pointing structure of the PROTO to be persistent.

4. Implementation of Script node

4.1 What is Script node

The ability of describing behavior in VRML2.0 is the most important improvement of VRML2.0 over VRML1.0. However, if one studies the VRML2.0 specification, he can find out that the secret of VRML2.0 behavior relies heavily on the existence of Script node.

A script node is a VRML node containing programs written in languages other than VRML (typically in Java, or JavaScript). It can signify a change of scene (virtual world) or user action, receive events from the other part of scene, make decisions, influence other part of the scene by sending Event out or more directly by modifying field values of other nodes.

Most VRML 2.0 contents contain more than one Script node. Unfortunately, VRML community did not reached agreement on scripting language standardization. There are two scripting language (Java and JavaScript) bindings in the Appendix, but the VRML2.0 specification makes no requirement of scripting language support for a VRML 2.0 browser. This means that even if a browser is VRML 2.0 compatible, it is possible that 90% of the VRML 2.0 worlds published on the Internet may behave incorrectly on the browser. In the real world today, Sony Community Place [3], Worldview [4], Dimension X [5] support Java as scripting language, SGI Cosmoplayer support VRMLScript (a subset of JavaScript)[6], [7]. SGI promised to support Java as scripting language in the future. So Java will probably become the standard scripting language of VRML.

Our browser support Java as its scripting language. There is no publicly available material on how to incorporate Java into VRML Script node. Here, we

present our implementation method.

4.2. Why using Java in Script node is disputable

VRML society did not come out with a general agreement on what should be the standard scripting language. But during the discussion, Java did stand out among the other suggestions. Java is secure, powerful, platform neutral, well defined (have a publicly available specification) and most importantly, most web contents developers have already got experience with Java by embedding Java applet into HTML homepages. The only deficiency of Java (prevent it from becoming the VRML scripting language standard) is that the execution of a Java program requires a Java VM (Java Virtual Machine).

The two most famous and widely available Java VMs are Sun VM (used in Netscape Navigator) and Microsoft VM (used in Microsoft Internet Explorer). Because the security of Java relies highly on Java VM, which requires significant amount of safety verification, an acceptable third party free Java VM implementation may not come into VRML scene in the near future.

The implementation of Java in VRML script node requires that the program runs in native code (VRML browser) and byte codes run in Java VM (script) to communicate with each other. It was assumed that the communication between native code browser and Java VM requires Java VM source code. At the time of VRML 2.0 specification discussion, the majority of VRML society knew nothing about workarounds of the problem. So the VRML 2.0 specification comes out without any requirement for a standard scripting language.

Later, we found out that it is not necessary to have source code access to use the Java VM in a VRML browser. One can use the Java VM through its RNI (Raw Native Interface). We suppose that the current available VRML browsers with Java support used similar approach.

4. 3 Incorporating Java in Script Node

4.3.1 Pre requirement

Our VRML browser was developed in Microsoft Visual C++4.0 (later upgraded to 4.2) and Microsoft SDK for Java 1.5 (It is said to be a superset of Sun JDK 1.02).

The C header file "Native.h" which declaring the RNI, import-library "msjava.lib" which gathers the export-point of Microsoft Java VM and utility "msjavah.exe" are included in the SDK [8].

The details of VRML Script node semantics is shown in [1], [2].

4.3.2Construct a Java object in native code

As stated in [1][2], each Script node has associated programming language code (in our case, written in Java), referenced by the *url* field. How to download a piece of Java byte codes (a Java class file) via a url is beyond this paper's scope (one easy way to download a Java class is through Java programming!) and will not be discussed here.

Once the Java class file is downloaded. An object of the downloaded class can be constructed by calling *execute_java_constructor* [8]. Every Java object in a VRML file is constructed before users see anything, even if the Java object is not at all used in the displaying. It is because that the object's *initialize* method (see below) must be called before the browser presents the world to the user and before any events are processed by any nodes in the same VRML file as the Script node containing this script [2]. It is needless to say that the *initialize* method has to be called *after* the object construction.

The *execute_java_constructor* example is shown in Fig. 6.

4.3.3 Calling Java methods from native code

Basically, after the Java object construction, there are three methods that will be called from native code. The three methods are defined in Appendix B of [2]. They are public methods of class *Script*. The three methods are:

initialize: is called before the browser presents the world to the user and before any events are processed by any nodes in the same VRML file as the Script node containing this script.

shutdown: is called when the corresponding Script node is deleted or the world containing the Script node is unloaded or replaced by another world.

processEvents: is called when the script receives some set of events.

The three methods can be called out by *execute_java_dynamic_method* [8]. How to call the three methods out is shown in Fig.6

4.3.4 Java class packages for VRML

The interfaces of Java class packages for VRML are listed in the Appendix B of [2].

The classes are divided into three Java packages: *vrml*, *vrml.field* and *vrml.node*.

vrml package contains the basic classes like: *Event, Browser, Field, Mfield, ConstField, ConstMField, Base Node,* etc.

vrml.field package contains specific field classes like: *SFColor*, etc.

vrml.node package contains two classes: Node and Script.

Most classes have a counterpart in native code. If the native part of browser is developed in C++, their code even looks very similar. The methods that can be completely programmed in Java are quite simple. The more difficult parts are those methods that calling a native method or a Java method that declared to be native and implemented in native code.

4.3.5 Using native code in Java program

For Microsoft Java VM, there are two methods to use native code in Java program. One is through COM (Component Object Model) interface, a typical Microsoft technique with which a COM object may be imported like an ordinary Java class. It will start an endless dispute to argue about whether using Microsoft limited techniques is right or not. But if one develop Windows application only, it is not bad for him to use COM, because he will have to use COM some where in the program to get a Windows compatible logo.

Our VRML browser used a more simple way of calling native code: declaring a Java method in the Java program as *native*, and implementing the Java method in native code. A simple example is shown in Fig.6.

Most of these Java methods related to native code have the shape of setXXXXX or getXXXXX, which writes or reads values in the data structure of native code. The key of implementing these methods is the data member (*native_address*) of Java class that holds the class's native counterpart's memory address.

The procedure of creating a natively implemented Java method is as follows:

First, define a class in Java, and add *native* keyword into the required method's declaration. Then, use Java compiler (*jvc*) to compile the Java class. Next, use utility *msjavah* to produce a C header file for use by the native code from the compiled class file. Next, implement the method in native code. There are a couple of helper function/Macros in the *native.h* to interpret and access the Java class data members. Last, gather all these native implemented Java methods in a DLL (Dynamic Linking Library). The DLL can be linked into Java program at run time.

4.3.6 Linking the native methods into Java program

The DLL produced in 4.3.4 can be linked into Java program at run time by calling:

```
System.loadLibrary("vrmljavalib") ;
```

vrmljavalib is the DLL's name. *System.loadLibrary* need to and only need to be called once for every Java VM session. There is no problem if Java methods implemented in native code are loaded earlier than its belonged class.

5.Summary

In this paper, we presented the implementation detail of our VRML 2 browser. We discussed the implementation of Event model, PROTO and the implementation of Java in the Script node. VRML Browser development is one of several basic areas in VRML research. We look forward to seeing more publications on this area.

Acknowledgments

This research was supported in part by THE HORI INFORMATION SCIENCE PROMOTION FOUNDATION.

```
Ex 4.3.2, construct a Java object from native code
        HObject *phobj = execute_java_constructor(NULL, "AScript", NULL, "( )");

Ex 4.3.3, calling java methods in native code
        execute_java_dynamic_method(NULL, phAScript, "initialize", "( )V");
        execute_java_dynamic_method(NULL, phAScript, "shutdown", "( )V");
        execute_java_dynamic_method(NULL, phAScript, "processEvents", "(I[Lvrml/Event)V",count, phEvent);

Ex4.3.5, using native code in Java program, a simplified version of class SFFloat
        //The content of SFFloat.java, the field native_address will be set by native program during creation
        public class SFFloat
        {
                int native_address;
                public native void setValue(float f);
        }

        C:>jvc SFFloat.java          (Compile the SFFloat class)

        C:>msjavah SFFloat           (Produce SFFloat.h )

        //The implementation of SFFloat_setValue in SFFloat.c
        #include "varargs.h"
        #include "SFFloat.h"
        __declspec(dllexport) void __cdecl SFFloat_setValue (struct HSFFloat *this, float v)
        {
                *(float *)this->native_address=v;
        }
```

Fig. 6: Examples of Java VM Raw Native Interface

References

[1] *The Virtual Reality Modeling Language Specification*,
http://vrml.sgi.com/moving-worlds/spec/index.html

[2] Rikk Carey and Gavin Bell, *The Annotated VRML97 Reference Manual*, ADDISON-WESLEY DEVELOPERS PRESS 1997

[3] *SONY Community Place*,
http://www.sonypic.com/vs/

[4] *Worldview*,
http://www.intervista.com/products/worldview/index.html

[5] *Dimension X (Microsoft)*,
http://www.microsoft.com/dimensionx/default.htm

[6] *SGI Cosmo Player*,
http://www.sgi.co.jp/TEXT/Products/cosmo/player

[7]*VRMLScript Parser and Implementation*,
http://vrml.sgi.com/moving-worlds/spec/vrmlscript.zip

[8]*Microsoft SDK for Java 1.5*