REIHE INFORMATIK
13/98
Access to and Encoding of VRML State Information
Martin Mauve
Siemens Telekooperations Zentrum, Saarbrücken
Universität Mannheim
Praktische Informatik IV
L15, 16
D-68131 Mannheim

Access to and Encoding of VRML State Information

Martin Mauve
Praktische Informatik IV, University of Mannheim, Germany
Siemens Telecooperation Center, Saarbrücken, Germany
mauve@pi4.informatik.uni-mannheim.de

Abstract. In this paper we propose a concept for transparent access to VRML state information. Our approach enhances VRML browsers to provide additional functionality instead of placing the burden for state access on content developers. The enhanced functionality is realized as an extension to the External Authoring Interface (EAI). Any application which relies on a VRML browser as the 3D presentation engine can use the new EAI functionality to get and set the state of arbitrary VRML content. In order to support diverse applications, the proposed methods allow not only retrieval of the full state of a complete world, but also of the state of single objects and state changes. Since the results of state access should be independent of browser implementations, we also specify an encoding for state information. Data in this form are either produced or consumed during state access. For the encoding of state information we use an efficient, easy-to-parse binary encoding.

Keywords: VRML, Binary Encoding, Persistence

1 Introduction

Within the last two years the Virtual Reality Modeling Language (VRML) [6] has emerged as the prime choice for describing platform-independent, interactive 3D objects and worlds. Currently various work-groups aim at the advancement of VRML and VRML-related technology. However, several areas for improvement remain which have not been adequately addressed so far. Among them is the need for a standardized way to save and restore the state of arbitrary VRML objects and worlds.

The current specifications of the External Authoring Interface (EAI) and the Script node API do not contain methods which provide adequate functionality for state access. Existing proposals for VRML state persistence require that the functionality for getting and setting the state be realized by the VRML content [9] [8]. These approaches are only viable when the content can be specifically developed to support state persistence. Examples where this could be the case are specialized multi-user objects or worlds which are dynamically retrieved from databases. The state of other VRML models is currently not accessible. This is a severe constraint to applications which embed VRML browsers as presentation and execution engines for VRML content. These applications are of prime importance to the general success of VRML and range from presentation tools to 3D telecollaboration applications and multi-user worlds. None of them are currently able to access the state of arbitrary VRML content, although this is necessary to support fundamental functionality such as saving a certain state or transmitting it to a communication peer.

To solve this problem we introduce new methods for the External Authoring Interface. These allow content-independent (transparent) saving and restoring of the VRML state. The new methods produce (save state) and consume (restore state) state information. This information needs to be encoded in a standardized way. We therefore define a binary encoding for VRML state information in addition to methods of accessing it.

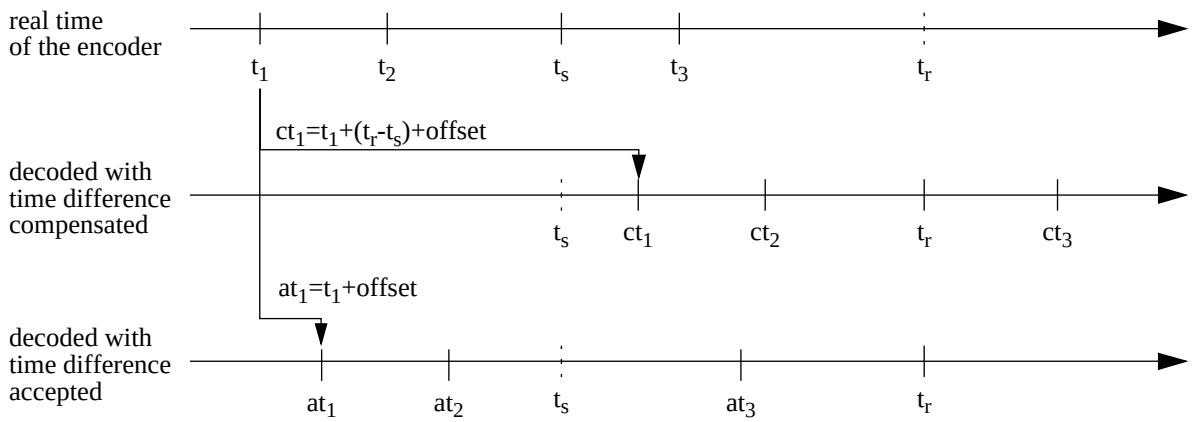
This paper is structured as follows: In Section 2 we describe the requirements for the state access methods and the binary encoding of VRML state. The basic rules and a grammar for the encoding of VRML state are presented in Section 3. Section 4 explains the semantics for encoding prototypes and special nodes such as scripts and sensors. In section 5 we describe the methods for accessing VRML state as an extension to the External Authoring Interface. Section 6 concludes this paper with a summary and an outlook.

2 Requirements

Two aspects have to be addressed in order to realize a service for saving and restoring VRML state. The first one is the definition of access methods. A minimal set of functionality includes methods for saving and methods for restoring state information. The second aspect is the specification of a binary encoding which enables browser-independent storage and exchange of VRML state. A number of requirements pertaining to both aspects can be derived from the areas of application for VRML state access as well as from performance and browser integration issues.

Requirements pertinent to the access methods are:

- **Transparency**. As already mentioned the methods should be generally independent of the content presented. The only exception, where full transparency is not possible, are customized script nodes. As will be shown later, a default can be specified even for those parts of VRML state. The main reason for demanding transparency is that the state of all worlds and objects should be accessible, independently of how they where created.
- **Flexible handling of time**. Time is an integral component of the VRML state. There are two semantically different methods to handle the difference between the point in time when the state of the content is saved and the point in time when the state is restored. Both are shown in Figure 1. The first method compensates for the time that has passed between saving and restoring the state. The content will be presented as if the time between saving and restoring the state never elapsed. This method adjusts all references to time in the encoded state by adding the difference between the point in time the state is restored and the point in time the state was saved. The time values are adjusted when the state is restored. This procedure makes sense if users want to save a state and restore it at a later point in time to continue interaction with the same VRML world.
  However, there are scenarios where this behaviour is not appropriate. If, for example, two users are interacting with a shared VRML world, it might be desirable for one user to update the other about a state change. If the time needed to save, transmit, and restore the state where compensated when the receiver decoded the state, the users would have an inconsistent model. In this model the sender would precede the receiver in time. It is therefore necessary to provide a second method which accepts that time has passed between saving and restoring the state. All references to time are kept as they have been saved in the encoded state, no matter when the state is restored. As is shown, this option is useful in collaborative or streaming applications, where the passage of time continues synchronously for all communication partners (e.g. sender and receiver of state information).
  Both methods can be enhanced by specifying a time offset, to be added to the time values when the state is restored. This provides a limited capability for jumping forward and backward in time.



t_i = encoded reference to a point of time
it_i = decoded time value when the time difference is ignored
kt_i = decoded time value when the time difference is accepted
t_s = point of time the state is saved
t_r = point of time the state is restored

Figure 1: Handling of Time References

- **Access of sub-components**. In the case of large VRML worlds access to the state of sub-components is desirable, so that their state can be saved and restored independently of the remainder of the VRML world. Such a sub-component might be an avatar, a house, a car or any other VRML object. Generally sub-components are encapsulated in grouping nodes. In addition to being able to access the state of whole worlds, it should therefore also be possible to get and set the state of a single node.
- **Delta States**. In cases where the states of worlds and objects are saved and restored frequently, it is important to be able to retrieve only those parts that have changed since the last time the state was saved. We call a state which describes only the parts of an object or world that have changed since the last state access a ***delta state***. A delta state can only be interpreted if the preceding full state and interim delta states are also available (see Figure 2). The main advantage of delta states is their smaller size and that they can be calculated faster than full states.

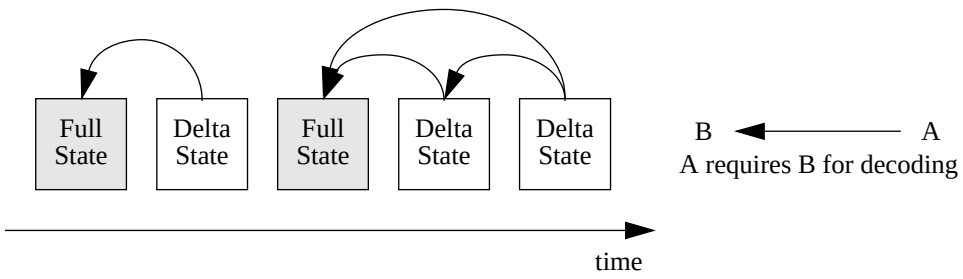


Figure 2: Decoding Delta States

The encoding has to meet the following requirements:

- **Support of access methods**. The encoding must support the access methods, in order for them to fulfil the requirements mentioned above. Specifically it must allow for transparency as well as for the definition of sub-components and delta states.
- **Efficient encoding**. It is important that the encoding be efficient with regard to the amount of data needed to define VRML states. Since state descriptions are generated automatically and are not supposed to be edited by humans, the usage of a human-readable format is not necessary. Instead a binary format should be used.
- **Simple encoding**. A relatively simple persistence format is needed to enable browsers to save and restore VRML state fast and without too much processing. It should be similar to the internal browser representation of the VRML state. As an example, it should be easy to retrieve the state of a node from the persistence format. Sophisticated compression might raise the efficiency of the encoding but also would most likely result in an unacceptable overhead within the browser.

3 Basic Grammar for the Encoding of the VRML State

The specification of our binary state encoding is influenced by the proposal for a VRML Compressed Binary Format (CBF) [7]. Besides compression, the CBF proposal specifies a grammar for the binary description of VRML worlds. This grammar is sematically equivalent to the plain text encoding contained in the VMRL97 international standard. Stripped of the parts which concern compression, and enhanced to support additional state information, the grammar of the CBF proposal can be used to define an encoding for the VRML state. In the following sub-sections we describe the grammar for the encoding of complete VRML worlds, sub-components and delta states. The grammar shown in the main part of this paper is not complete and serves only to demonstrate the basic ideas of the encoding. A full definition of the grammar can be found in Appendix A.

Elements of VRML state

Figure 3 shows the top-level structure of the encoding for the VRML state. Symbols which are printed in **boldface** are terminal symbols in regard to this paper. Lines which are *italic* describe the encoding of sub-components as well as delta states and are explained in the appropriate sub-sections.

The description of a VRML state is a sequence of bytes which starts with a header. This header identifies that the sequence of bytes contains a VRML state and includes the version number of the encoding. In order to be able to decode a state in the persistence format, a decoder needs to know the type of information encoded (a complete world vs. a single node and a full state vs. a delta state) and the encoding methods used. This information is included in a TYPE byte that follows the header.

The main part for the state encoding is divided into the VRML browser state and the state of the scene graph. Information which is needed to define the state of the browser are the user's point of view, the point of time the encoding was done, the URL of the current world and the stack order of bindable nodes. The state of the scene graph contains the description of nodes and routes as well as the declaration and definition of prototypes (PROTO and EXTERNPROTO).

```
VRMLSTATE ::=
  HEADER                        // #VRMLSTATE 1.0\n
  TYPE                          // Information about the encoded content
  BROWSER                       // State of the VRML browser
  SCENEGRAPH;                   // State of the scene graph

TYPE ::=
  BIT isCompleteWorld           // encoding complete world (1) vs. encoding single node (0)
  BIT isFullState               // encoding full state (1) vs. encoding delta state (0)
  BIT isCompleteList            // delta states: lists of nodes are encoded with the
  UINT(5);                      // padding: ignored

BROWSER ::=
  DOUBLE currentTime            // the point of time where the state was recorded
  STRING URL                    // the current URL
  if (isCompleteWorld) {
     NODE pointOfView           // the user's point of view is encoded as a Viewpoint node
     STACKORDER background      // stack order of bindable nodes
     STACKORDER fog
     STACKORDER navigationInfo
     STACKORDER viewPoint
  }

SCENEGRAPH ::=
  UINT(32) nEXTERNPROTO         // number of EXTERNPROTO declarations
  UINT(32) nPROTO               // number of PROTO definitions
  UINT(32) nNODE                // number of top-level nodes
  UINT(32) nROUTE               // number of routes
  EXTERNPROTO[nEXTERNPROTO]     // declaration of EXTERNPROTOs
  PROTO[nPROTO]                 // definition of PROTOs
  NODE[nNODE]                   // state of nodes
  ROUTE[nROUTE];                // state of routes
```

Figure 3: Top-level Structure of VRML State

While the encoding and semantics of the VRML browser state are straightforward, the description of the scene graph state needs more detailed consideration. Its fundamental elements are nodes and routes. In the following sub-sections we will explain how to encode these elements, while the encoding of prototypes is discussed in Section Four.

Encoding of Nodes and Routes

As shown in Figure 4 the encoding of a node state starts with a unique node ID. A node's ID may change with each full-state encoding. However, it is illegal to change the ID for delta states which follow a full or another delta state. Given these restrictions, delta states can reference nodes which have been defined in previous states. At the same time, the number space does not fill up with deleted nodes, since each full-state encoding can reuse the IDs of deleted nodes. Besides their reference purpose in delta states, the IDs are used to define routes and USE nodes.

In order to provide important information to the decoder, the format of a node is encoded as a special byte. This byte tells the decoder whether or not a USE or a DEF node has been encoded and whether the encoding contains fields, exposed fields or events (***node elements***). In addition it includes two bits for the encoding of delta states.

To access a complete world every USE node is encoded by providing the unique ID of the referenced node. All non-USE nodes have a type to identify what kind of node is being encoded. The values 1 to 54 indicate a default node from the VRML97 specification, where the number is derived from the node's alphabetic order, e.g. Anchor is encoded as 1 and WorldInfo as 54. Negative numbers are used to indicate instances of prototypes. The size of the remaining description of a node is given so that it can be skipped by the decoder. For most nodes this description is a list of node elements. Some nodes (Script nodes, Inline nodes and the instances of prototypes) require a special encoding, which is described in Section Four.

```
NODE ::=
  UINT(32) nodeID                                          // unique ID for this node
  NODEFORMAT                                               // what kind of node is encoded?
  if ((!isUNMODIFIED and !isDELETED) or isFULLSTATE) {// node skipped for delta encoding?
    if (isUSE){                                            // is this a USE node?
      UINT(32) referencedNodeID                            // DEFed node for this USE node
    } else {
      if (isDEF) {                                         // is this a DEFed node?
        STRING                                             // name of the DEFed node
      }
      NODETYPE                                             // what type of node is encoded?
      UINT(32) nodeSize                                    // how many bytes does this node have?
      if (NODETYPE==39) {                                  // is it a Script node?
        SCRIPT                                             // special encoding for script nodes
      } else if (NODETYPE==24) {                           // is it an Inline node?
        INLINE                                             // special encoding for inline nodes
      } else is (NODETYPE<0) {                             // is it an instance of a prototype?
        PROTOINSTANCE                                      // special encoding for prototypes
      } else {
        NODEFIELDS                                         // regular encoding of node elements
      }
    }
  };

NODEFORMAT ::=
  BIT isUSE                   // a USE node (1)?
  BIT isDEF                   // a DEF node (1)?
  BIT hasNODEFIELD            // NODEFIELDS contains data (1)?
  BIT isUNMODIFIED            // delta encoding: is this node unmodified (1)?
  BIT isDELETED               // delta encoding: has this node been deleted (1)?
UINT(3)                       // padding: ignored;

NODEFIELDS ::=
  if (hasNODEFIELD) {                  // does this node have node elements?
    NODEFIELD | NODEFIELD NODEFIELDS   // list of fields, exposed fields and events out
  };

NODEFIELD ::=
  FIELDNUMBER                 // this identifies the node element which is encoded
  FIELDVALUE;;                // the value of the node element

ROUTE ::=
  UINT(32) routeID            // unique ID for this route
  if (!isFullState) {         // is this a delta encoding?
    ROUTEFORMAT               // delta encoding: is this a modified or deleted route?
  }
  UINT(32) fromNodeID         // source node
  FIELDNUMBER fromField       // source node element
  UNIT(32) toNodeID           // target node
  FIELDNUMBER toField;        // target node element

ROUTEFORMAT ::=
  BIT bUNMODIFIED             // delta encoding: is this route unmodified?
  BIT bDELETE                 // delta encoding: has this route been deleted?
  UINT(6);                    // padding: ignored
```

Figure 4: Encoding of Nodes and Routes

The basic rule for the encoding of node elements is that any node element which does not contain the default value of its data type must be included in a NODEFIELDS list. Since each list entry consists of two parts, it is possible to skip elements or choose a different order for the elements than is shown in the VRML97 standard. The first part is a number which is defined by the occurrence of the node element in the VRML97 specification and uniquely identifies it. The second part is the value of the element. Each VRML data type has its own encoding which is not shown here. In order to allow for streaming of VRML states, it is required that all node elements of the type SFNode and MFNode be listed after any other node elements, and that children fields be specified as the last

encoded node elements. This supports streaming of the VRML state, since node elements can be decoded and applied as soon as they are received, whereas a different ordering would require the decoder to wait until a node has been completely received [4].

The encoding of a route starts with a unique ID. Similar to the node ID, the route ID is used to reference the route within delta states. Thus they have the same requirements as node IDs. In the case of a full-state encoding, the ID is followed by the description of the route's source. The source is defined by the ID of the source node and the number of the source node element. The target of a route is specified by the ID of the target node and the number of the target node element.

Encoding of Sub-components

The encoding of the state of a single VRML node (sub-component) reuses the grammar presented in Figure 3 and Figure 4. The only part of the grammar that is specifically designed to support sub-components is the state of the browser. When encoding a single node the browser state does not contain the stack order of bindable nodes or the user's viewpoint. This is reasonable since the decoding of a sub-component should affect only the sub-component but not the global behavior of a VRML world.

Besides the different encoding of the browser state some additional restrictions have to be taken into account when encoding the state of a sub-component:

- **Prototypes**. Only the definitions of those prototypes are included which are used by the sub-component.
- **Nodes**. On the top-level there is exactly one node present - the node that represents the sub-component.
- **USE nodes**. The first USE nodes that references a node which is not contained in the sub-component is encoded as a full copy of the referenced DEF node rather then as a USE node. Other USE nodes referencing the same node are encoded as a reference to the fully encoded node.
- **Routes**. Routes are only encoded when source and target of a route are part of the sub-component.

Delta States

A delta state describes the difference of the current state from the last encoded delta or full state. The grammar from Figure 3 and Figure 4 is also used to encode delta states. While the state of the VRML browser is fully encoded as for a full state, the state of the scene graph is reduced for a delta-state encoding.

Since prototype declarations and definitions are not expected to be modified or deleted during the lifetime of a VRML world, the prototype parts of the scene graph state contain only those elements that have been added since the last state encoding. Any previously encoded prototypes are expected to remain unchanged.

For the nodes part of the scene graph only those nodes are completely encoded which have been modified or added since the last delta or full state. Unmodified nodes are included only by giving their ID or they are completely omitted. Whenever a single node element of the type SFNode is encoded there are two possibilities:

- **The node and all of its descendants are unmodified**. In this case the node is encoded by supplying its ID and format with the `isUNMODIFIED` flag set to one. Nothing else is encoded, since the node has not changed.
- **A node element or any descendant of the node has been modified**. In this case the node is completely encoded with `isUNMODIFIED` set to zero. The decision whether or not to do a complete encoding must then be made for each individual descendant of the node.

There are two basic ways to encode a list of nodes such as the nodes on the top level of a scene graph state or MFNode instances. The method for encoding lists of nodes is chosen at the beginning of the encoding and must be the same for all lists:

- **Complete List**. All nodes on the list at the point in time when the delta encoding is done are encoded. Each of the nodes is encoded according to the rules for single nodes described above. E. g. for unmodified nodes only their ID and format need to be given, while modified nodes are completely encoded. The encoder can easily execute this method, since it just needs to traverse the list and take appropriate actions. However, the receiver needs to do some extra work to see whether nodes have been deleted or remained unmodified. If an encoder chooses to use the complete list method it sets the `isCompleteList` flag in the `TYPE` byte to one.
- **Changes Only**. The changes only method encodes only the nodes which have been modified, added to or deleted from the list since the last state encoding. Nodes that have been added or modified are encoded like single nodes as described above, deleted nodes are encoded by giving their ID and format with the `isDELETED` flag set to one. This method requires some overhead in the encoder, since deleted nodes need to be remembered. On the other hand, the decoder can decode faster since its actions can be taken directly from the encoded state. An encoder sets the `isCompleteList` flag in the `TYPE` byte to zero when choosing this method.

While it would be possible to encode delta states on the fine-grained basis of fields, exposedFields and events, this would require the browser to manage a large overhead just for delta-state encoding. We view the proposed level of granularity as a compromise between encoding overhead and encoding efficiency.

The delta encoding of routes is performed in the same way as the encoding of node lists. The encoder must use the same method (complete list vs. changes only) it used for the encoding of node lists.

4 Encoding Semantics of Special Nodes and Prototypes

For most node types the semantics of the encoded nodes are completely defined by the encoding grammar in combination with the VRML97 international standard. For example, the semantics of an encoded Box node is given through a single node element called size. This field is interpreted according to the VRML97 specification as the size of the box. The encoding of prototypes and some special nodes, however, have more complex semantics, which are described in the following sub-sections.

Sensors

Some sensors contain elements that require special handling when included in a state encoding. These elements are the isActive events out of drag sensors as well as the touchTime and the isActive events out of the touchSensor. All of these elements are used to signal that user interaction is in progress or is just finished. It is not likely that a user interaction (like dragging an object) can be simply resumed when the state is restored. For example, a sensor could wait for the mouse button to be released when its state is saved. If the mouse button is not pressed, once the state is restored inconsistencies might occur. In order to avoid these inconsistencies, the user has to restart all ongoing interactions. The isActive events out of drag sensors and TouchSensors are set to false upon restoration of the state, thereby possibly triggering an event cascade. Also, a touchTime event-out is generated if a TouchSensor's isActive event-out is true when restored. This guarantees that all user interactions are finished and that no inconsistencies remain. Environmental sensors, such as the VisibilitySensor, also have an isActive event-out. However, these sensors are evaluated for every simulation tick and therefore do not require any specific handling.

Inline Nodes

Inline nodes have a special semantics, since they import a whole scene graph into a VRML world. They also have their own name spaces for the visibility of DEF nodes and prototypes. In order to capture the state of an Inline node, it is encoded as shown in Figure 5. In addition to regular node elements, the Inline node contains a complete scene graph of the referenced VRML file. Note that it is insufficient to reconstruct the scene graph from a given URL, since the state of the scene graph included might have changed.

```
INLINE ::=
  NODEFIELDS       // standard encoding of node elements
  SCENEGRAPH;      // scene graph of the imported world
```

Figure 5: Encoding of Inline nodes

Prototypes

Two different aspects must be considered when encoding prototypes. The first is the encoding of the prototype definition (PROTO) or declaration (EXTERNPROTO). The second aspect is the encoding of prototype instances, which is a special case of node encoding.

The top-level structure of the encoding for PROTO definitions is shown in Figure 6. Each prototype is assigned a unique ID, which is used by nodes that instantiate this prototype as their node type. Besides the name of the prototype and the declaration of the interface, it is also necessary to encode the scene graph of the prototype definition. The declaration of an EXTERNPROTO is similar to the definition of a PROTO, but the scene graph part is replaced by URL references which describe the location of the definition for the EXTERNPROTO.

Instances of prototypes are encoded as special nodes. As shown in Figure 6, the encoding of prototype instances includes the node elements that have been defined in the prototype's interface. Since the scene graph of a prototype instance can change independently of any other instance of the same prototype, the state of the scene graph is included in each instantiation.

```
PROTO ::=
  UINT(7)                   // unique prototype ID
  BIT                       // padding: ignored
  STRING                    // name of the prototype
  INTERFACEDECLARATION      // interface of the prototype definition
  SCENEGRAPH;               // scene graph of the prototype definition

PROTOINSTANCE ::=
  NODEFIELDS                // the encoded node elements of this instance
  SCENEGRAPH;               // the scene graph of this instance
```

Figure 6: Encoding of Prototypes

Script Nodes

Script nodes are the only parts of VRML in which the state cannot be accessed in a fully transparent way. The main problem is that the state of script nodes is usually kept outside of the VRML browser's scope. In addition the state might contain resources (like network connections), which cannot be transparently saved and restored. It is therefore necessary to extend the script node API by functionality for the management of states. Since it should be possible to save and restore existing VRML content, a predictable default behavior needs to be defined for those script nodes which do not use this API.

As an example of what the extension of the script API looks like, Figure 7 lists the additional methods for the Java Script node API. The first one is the `getState` method, which is called by the VRML browser, when it needs to get the state of the Script node. The authors of scripts implement this method within their extension of the `Script` class, just as they do with the `initialize` or the `shutdown` methods. When the browser calls the `getState` method, it expects that the method writes all state information of the Script node to the given `OutputStream`. Format and content of the written data are defined by the author of the script. When an author supplies a `getState` method, the method is expected to return the constant value `CUSTOMIZED_STATE`. The Script class has a default implementation that simply returns the value `DEFAULT_STATE` without saving anything to the `OutputStream`. This method is called when the author of the script does not supply the `getState` method for his sub-class of `Script`.

The `setState` method is called by a browser when the state of a script node should be restored. This call replaces a call to the `initialize` method. As a parameter the browser supplies an `InputStream` containing the data that was stored by the `getState` method when the script state was saved. The `setState` method is only called when the corresponding `getState` method returned `CUSTOMIZED_STATE` (e.g. was implemented by the script's author), otherwise a standard call to the `initialize` method is executed. This ensures that scripts which do not implement any handling of state will at least be properly initialized once a VRML state is restored.

When encoding delta states it is important that the browser knows whether or not the state of a Script node has changed. The method `stateChanged` of the class `Script` should be called by the script whenever its state changes. It marks the Script node as changed for the encoding of delta states.

```
int getState(OutputStream saveStateTo)
  return values:   DEFAULT_STATE
                   CUSTOMIZED_STATE

void setState(InputStream restoreStateFrom)

void stateChanged()
```

Figure 7: Encoding API for Java Script Nodes

The encoding of Script nodes is shown in Figure 8. Node elements that are defined by the script node are stored like those of regular nodes. If the script does provide handling of state the `isCustomizedState` flag is set to 1. Following the flag is the data returned by the `getState` method, encoded as an array of bytes.

```
SCRIPT ::=
  NODEFIELDS                        // encoding of the node elements
  BIT isCustomizedState             // does the script use the state API
  UINT(7)                           // padding
  UINT(32) length                   // length of the script state
  BYTE[length];                     // script state
```

Figure 8: Encoding of Script nodes

5 Accessing State Information of VRML worlds

We propose to realize access to the state of VRML worlds by extending the External Authoring Interface (EAI). In this section we will introduce an extension to the Java language binding of the EAI that consists of six additional methods for the Browser class. Similar methods can be added to the general EAI or to the Script node API.

The first two methods provide the capability for getting the state of VRML content (see Figure 9). With `getWorldState` it is possible to get the state of a complete world. The caller of the method provides the `OutputStream` to which the encoded state should be saved. If `allowDeltas` is true, the browser will automatically encode a delta state when this is appropriate. If `allowDeltas` is false, the method will produce only full states. The return value of the method indicates whether a full or a delta state was encoded. The method for getting sub-components (nodes) is called `getNodeState` and has one additional parameter: the node that is encoded. In all other respects it acts exactly like `getWorldState`.

```
int getWorldState        (OutputStream os, boolean allowDeltas)
int getNodeState         (OutputStream os, boolean allowDeltas, Node subComponent)
    return values: FULL_STATE
                   DELTA_STATE

short getLevelOfActivity ()
void registerActivityCallback (Callback callMe, short levelOfActivity)

void setWorldState       (InputStream is, boolean keepTimeDifference, double timeOffset)
void setNodeState        (InputStream is, boolean keepTimeDifference, double timeOffset,
                          boolean replace, Node target)
```

Figure 9: Additional Methods for the Browser Class

Getting the state of VRML content is likely to be both time- and CPU-cycle-consuming. In addition, any ongoing activity has to be halted while the state is saved, otherwise inconsistencies might occur. It is therefore important to choose the right time to access VRML state. Specifically, this should be a time when the general activity of the browser is low. It is difficult to judge the level of activity with the methods that are currently offered by the EAI. The only information available on performance is the frame rate. However, the frame rate is not an adequate indication of the amount of activity going on in a VRML world. It might be low because of slow rendering hardware or because other applications are running on the same computer. In addition, the frame rate is calculated for each simulation tick and does not provide information about the development of the performance over a period of time. Furthermore, there is no means of notification when the frame rate reaches a threshold where state access would be acceptable. Because of this we propose two additional methods: `getLevelOfActivity` and `registerActivityCallback`. The first method returns a value between 1 and 10, indicating the current level of activity of the browser. A return value of 1 is generated when there has been no activity (user interaction, animations) for a period of time. On this activity level, it is reasonably safe to access the state of the VRML content without there being an impact on the perception of the user. An activity value of 10 indicates that the activity of the browser has reached a level where it can hardly keep up with processing all user interactions and animations. On this level access to the VRML state is impossible without heavily influencing the quality of the VRML presentation. Values between 1 and 10 will be assigned as soon as we gain more experience with VRML state access. With the described functionality, the `getLevelOfActivity` method is useful to decide whether or not it is acceptable to access the VRML state. A second method, `registerActivityCallback`, allows a customized method to be called when the browser activity reaches a level equal to or below the given threshold. This method allows state access to be deferred until it is acceptable for the user.

The state of VRML content can be restored by using either the `setWorldState` or the `setNodeState` method. The first method accepts an `InputStream`, which should contain the state of a VRML world (full state or delta state). Two parameters are used to control how SFTime node elements are restored. If `keepTimeDifference` is false, the difference between the point in time the state is restored and the point in

time the state was saved is added to the appropriate SFTime fields. If `keepTimeDifference` is true, the difference is not added. In both cases the appropriate SFTime fields are modified by adding the `timeOffset` value. Note that it is only legal to decode a delta state if the preceding full state and all delta states between this full state and the current delta state have already been decoded. If this is not the case, the results are undefined.

Decoding the state of a single node requires additional information about the placement of this node. In the case of a full encoding, the node can overwrite an existing node (`replace` set to true, `target` is the node to replace) or be inserted as a child of an existing node (`replace` set to false, `target` is the parent node). When restoring delta states the new node must replace an existing node. In order to be able to decode the delta state, the preceding full state and all delta states between the full state and the current delta state have to have been already decoded. Otherwise the result of the operation is undefined.

Unlike support for the decoding of full states, the decoding of delta states is likely to require significant additions to the VRML browser. The reason for this is that it cannot be guaranteed that the state of VRML content will remain unchanged after a full or delta state has been restored but before a new delta has state arrived. Since a delta state refers to the state of the VRML content right after the last state has been applied, it could lead to inconsistencies if the browser were to use the current state of the VRML content as a basis for the delta state. The solution to this problem is to keep a separate copy of the of the VRML state after a state has been restored. Once a new delta state has arrived, this copy would be updated and then replace the current VRML content. It is not desirable to force every browser that wants to conform to the proposed API to implement this mechanism. Therefore a browser is allowed to throw an exception when the `setWorldState` or `setNodeState` methods are called for a delta state. This exception indicates that the browser cannot decode delta states.

6 Conclusion and Outlook

In this paper we proposed a concept for transparent access to and encoding of VRML state information. Our approach differs significantly from existing concepts which require that the content handles the saving and restoring of the VRML state. The methods described in this paper enhance the browser so that the state of VRML content can be accessed independently of how the content was developed. In order to be able to exchange and save the state we specified a binary format which allows for the encoding of full worlds as well as of single nodes and state changes.

The next major step is to provide a sample implementation of the access methods described in this paper. We intend to make a prototype publicly available by February 1999. This prototype will be based on the Java3D VRML97 browser and will support the access method defined above. We anticipate that there will be a fair amount of feedback on this work as well as on the sample implementation. This feedback will be evaluated and could possibly lead to a revised version of the state encoding and the access methods.

Parallel to the work described here we are defining a Real Time Transport Protocol (RTP) [5] profile for interactive media [3]. The main purpose of this profile is the ability to develop generalized services such as recording and late joins for conferencing applications which use interactive media. One instantiation of the profile will be a payload type for VRML states and events. This payload will use the VRML state encoding described in this paper as a basis for the specification of the payload type.

The work described here is done in the context of the TeCo3D project [2], which aims at the development of 3D telecollaboration applications for collaboration-unaware VRML content. The capability to transparently access VRML state and to transport it as an RTP payload is expected to significantly enhance the functionality of TeCo3D applications.

7 Acknowledgements

This work is funded by the Siemens Telecooperation Center, Saarbrücken, Germany.

8 References

[1] *Couch, J.*: **Proposal for a VRML97 Spec Addition - External Authoring Interface Reference**, August 1998, on-line: http://www.vrml.org/WorkingGroups/vrml-eai/proposals/justin/proposal.html.

[2] *Mauve, M.*: **TeCo3D - A 3D Telecooperation Application based on VRML and Java**, accepted at Multimedia Computing and Networking 1999 (MMCN99) / SPIE99, San Jose, February 1999.

[3] *Mauve, M.; Hilt, V.; Kuhmünch, C.; Effelsberg, W.*: **A General Framework and Communication Protocol for the Real-Time Transmission of Interactive Media**, Technical Report TR 16-98, University of Mannheim, Germany, October 1998.

[4] *Roehl, B.*: **Draft Proposal for the VRML Streaming Working Group - (DRAFT) Version 0.1**, June 1998, on-line available: http://ece.uwaterloo.ca/~broehl/streams/proposal.html.

[5] *Schulzrinne, H.; Casner, S.; Frederick, R.; Jacobson, V.*: **RTP: A Transport Protocol for Real-Time Applications,** Internet Draft, Audio/Video Transport Working Group, IETF, draft-ietf-avt-rtp-new-01.txt, August 1998.

[6] *VRML Consortium*: **Information technology -- Computer graphics and image processing -- The Virtual Reality Modeling Language (VRML) -- Part 1: Functional specification and UTF-8 encoding**, ISO/IEC 14772-1:1997 International Standard, December 1997, on-line: http://www.vrml.org/Specifications/.

[7] *VRML Consortium - Compressed Binary Format Working Group*: **The Virtual Reality Modeling Language Compressed Binary Format Specification , ISO/IEC 14772-3 Editor's Draft**, October 1997, on-line: http://www.research.ibm.com/vrml/binary/specification/draft5/VRMLB-ED5/spec.DIS/

[8] *VRML Consortium - Database Working Group*: **Database Working Group Charter**, February 1997, on-line: http://www.vrml.org/WorkingGroups/dbwork/charter.html.

[9] *VRML Consortium - Database Working Group*: **VRML Data Repository API - Oracle Proposal**, 1997, on-line: http://www.vrml.org/WorkingGroups/dbwork/oracle/overview.html.

Appendix A - Complete Grammar for VRML State Encoding

As mentioned in the main part of this paper, the grammar is based on the proposal for a VRML Compressed Binary Format (CBF) [7]. The compression parts of the CBF proposal have been removed for VRML state encoding. This was done to enable fast encoding and decoding. There are several additions to the CBF for the capturing of state information. Many examples and explanations in the following grammar are directly taken from the CBF proposal.

```
VRMLSTATE ::=
     HEADER                       // #VRMLSTATE 1.0\n
     TYPE                         // Information about the encoded content
     BROWSER                      // State of the VRML browser
     SCENEGRAPH;                  // State of the scene graph
```

HEADER

The header consists of the following sequence of ASCII bytes: “#VRMLSTATE 1.0 binary\n”

```
TYPE ::=
     BIT isCompleteWorld          // encoding complete world (1) vs. encoding single node (0)
     BIT isFullState              // encoding full state (1) vs. encoding delta state (0)
     BIT isCompleteList           // delta states: lists of nodes are encoded with the
                                  // Complete List (1) or the Changes Only (0) method
     UINT(5);                     // padding: ignored
```

```
BIT ::=
     UINT(1)                      // encoding a boolean value: 1=true 0=false
```

UINT(n)

A binary encoded n bit unsigned integer.

```
BROWSER ::=
     DOUBLE currentTime           // the point of time where the state was recorded
     STRING URL                   // the current URL
     if (isCompleteWorld) {
       NODE pointOfView           // the user’s point of view is encoded as a Viewpoint node
       STACKORDER background      // stack order of bindable nodes
       STACKORDER fog
       STACKORDER navigationInfo
       STACKORDER viewPoint
     }
```

DOUBLE

A double is represented using the IEEE 64bit format

```
STRING ::=
     UINT(32) nUTF8    // Then length of the string
     UTF8[nUTF8];
```

UTF8

A UTF8-encoded character.

```
NODE ::=
     UINT(32) nodeID                                    // unique ID for this node
     NODEFORMAT                                         // what kind of node is encoded?
     if ((!isUNMODIFIED and !isDELETED) or isFULLSTATE) {// node skipped for delta encoding?
       if (isUSE){                                      // is this a USE node?
         UINT(32) referencedNodeID                      // DEFed node for this USE node
```

```
      } else {
        if (isDEF) {                                    // is this a DEFed node?
          STRING                                        // name of the DEFed node
        }
        NODETYPE                                        // what type of node is encoded
        UINT(32) nodeSize                               // how many bytes does this node have?
        if (NODETYPE==39) {                             // is it a Script node?
          SCRIPT                                        // special encoding for script nodes
        } else if (NODETYPE==24) {                      // is it an Inline node?
          INLINE                                        // special encoding for inline nodes
        } else is (NODETYPE<0) {                        // is it an instance of a prototype?
          PROTOINSTANCE                                 // special encoding for prototypes
        } else {
          NODEFIELDS                                    // regular encoding of node elements
        }
      }
    };
```

```
NODEFORMAT ::=
    BIT isUSE               // a USE node?
    BIT isDEF               // a DEF node?
    BIT hasNODEFIELD        // NODEFIELDS contains data?
    BIT hasIS               // if this node is in a PROTO definition, does it have IS statements?
    BIT isUNMODIFIED        // delta-encoding: is this node unmodified?
    BIT isDELETED           // delta-encoding: has this node been deleted?
    UINT(2);                // padding: ignored
```

```
NODETYPE ::=
    SIGNEDINT;          // 1=Anchor, 2=Appearance, ..., negative numbers=prototype instances
```

```
SIGNEDINT ::=
    BIT sign            // 1=negative 0=positive
    UINT(31);           // the absolut value
```

```
SCRIPT ::=
    SCRIPTINTERFACEDELARATION       // interface declarartion of the Script node
    NODEFIELDS                      // encoding of the node elements
    BIT isCustomizedState           // does the script use the state API
    UINT(7)                         // padding
    UINT(32) length                 // length of the script state
    BYTE[length];                   // script state
```

```
SCRIPTINTERFACEDECLARATION ::=
    UINT(32) nInEvent               // number of eventIn EVENTs
    UINT(32) nOutEvent              // number of eventOut EVENTs
    UINT(32) nField                 // number of INTERFACEFIELDs
    FIELD[nInEvent]                 // eventIn
    FIELD[nOutEvent]                // eventOut
    FIELD[nField];                  // fields
```

```
FIELD ::=
    STRING                          // field/event name
    FIELDTYPE;                      // field/event type
```

```
FIELDTYPE ::=
    SIGNEDINT;
```

The value of the SIGNEDINT is calculated by ordering (starting with 1) the field types in the VRML field reference (in current specification Bool=1, Color=2, ...). If the field type is a single-valued field use the positive value of the ordering (SFBool=1), if the field is multiple-value field negate the value (MFColor=-2).

```
NODEFIELDS ::=
     if (hasNODEFIELD) {              // does this node have node elements?
       NODEFIELD | NODEFIELDLIST      // list of fields, exposed fields and events out
     }
     if(isPROTO && hasIS)             // isPROTO is set if NODEFIELDS is contained within a PROTO
    {
      IS | ISLIST                     // up to and including terminating IS
    };
```

```
NODEFIELD ::=
     FIELDNUMBER                 // this identifies the node element which is encoded
     FIELDVALUE;                 // the value of the node element
```

A NODEFIELD is considered to be a terminating NODEFIELD if it contains a terminating FIELDNUMBER. When specifying values for an exposedField use the FIELDNUMBER of the field (as opposed to the set_ and _changed FIELDNUMBERs).

```
FIELDNUMBER ::=
  SIGNEDINT;
```

The absolute value of FIELDNUMBER identifies a field within a node. If the value of FIELDNUMBER is negative then it is considered to be a terminating FIELDNUMBER. The absolute value of the SIGNEDINT is assigned according to context.

For a NODE the value indicates the specification order in VRML ASCII node reference. For a PROTO the value indicates the specification order in the PROTO definition in the PROTO section of the binary file. For a EXTERNPROTO the value indicates the specification order in the EXTERNPROTO definition in the EXTERNPROTO section of the binary file. For a SCRIPT the value for the built-in specifications is calculated as in the case of a NODE. The value for a specification defined in the file is equal to the specification order in SCRIPTINTERFACEDECLARATION section of the binary file plus the number of Script Node built-in specifications. The FieldNumber is incremented once for each field, eventIn or eventOut. For the ASCII data:

```
SCRIPT{
  field SFInt32 _currentState 0
  url “http://foo.com/bar.class”
  eventIn SFString _name
  eventIn _selected
  eventOut SFString _lookto
  field SFBool mustEvaluate TRUE
}
```

So assuming that _selected was defined before _name in the SCRIPTINTERFACEDECLARATION section, the FIELDNUMBERS would be as follows.

field/event	FIELDNUMBER
url	0
mustEvaluate	1
directOutput	2
_selected	3
_name	4
_lookto	5
_currentState	6

In this example the names for user-defined events and fields and have been prefixed with a ‘_’ to emphasize the effect of the numbering scheme.
In all cases, an exposedField corresponds to three consecutive field numbers. The first number for the field, the second for the ‘set_’ eventIn, and the third for the ‘_changed’ eventOut. So for example:

```
Inline {
  exposedField MFString url []
  field        SFVec3f  bboxCenter 0 0 0
  field        SFVec3f  bboxSize   -1 -1 -1
}
```

would have

field/event	FIELDNUMBER
url	0
set_url	1
url_changed	2
bboxCenter	3
bboxSize	4

```
FIELDVALUE ::=                      // fieldType is known by context
     select fieldType
     case SFBool:
       BIT[7]                       // padding, ignored
       BIT                          // 0=False, 1=True
     case SFColor:
       FLOAT[3]
     case MFColor:
       UINT(32) nColor              // number of elements in array
       FLOAT[3*nColor];             // color values
     case SFFloat:
       FLOAT
     case MFFloat:
       UINT(32) nFloat              // number of elements in array
       FLOAT[nFloat];               // float values
     case SFImage:
       UINT(32) width               // width
       UINT(32) height              // height
       UINT(32) nCompon             // number of components in the image
       BYTE[width*height*nCompon]   // image-data
     case SFInt32:
       SIGNEDINT                    // see notes
     case MFInt32:
       UINT(32) nInt                // number of elements in arry
       SIGNEDINT[nInt];             // integer values
     case SFNode:
       NODE
     case MFNode:
       UINT(32) nNode               // number of elements in array
       NODE[nNode]                  // nodes
     case SFRotation:              // see notes
       FLOAT[4]
     case MFRotation:
       UINT(32) nRotation           // number of elements in array
       FLOAT[4*nRotation]           // rotation values
     case SFString :
       STRING
     case MFString:
       UINT(32) nString             // number of elements in array
       STRING[nString]              // string values
     case SFTime:
       DOUBLE
     case MFTime:
       UINT(32) nTime               // number of elements in array
       DOUBLE[nTime]                // time values
     case SFVec2f:
       FLOAT[2]
     case MFVec2f:
       UINT(32) nVec2f              // number of elements in array
       FLOAT[nVec2f]                // vector values
     case SFVec3f:
       FLOAT[3]
     case MFVec3f
       UINT(32) nVec3f              // number of elements in array
       FLOAT[nVec3f]                // vector values
```

FLOAT

A float is represented using the IEEE 32 bit format.

```
BYTE ::=
  BIT[8];
```

```
NODEFIELDLIST ::=
     NODEFIELD | NODEFIELD NODEFIELDLIST   // list of fields, exposed fields and events out
```

```
IS ::=                          // (x IS y)
     FIELDNUMBER                // xFieldNumber (from current NODE)
     FIELDNUMBER;               // yFieldNumber (from current PROTO)
```

An IS is considered to be a terminating IS if xFieldNumber is a terminating FIELDNUMBER.

```
ISLIST ::=
     IS | IS ISLIST           // List of IS definitions
```

```
INLINE ::=
     NODEFIELDS                 // standard encoding of node elements
     SCENEGRAPH;                // scene graph of the imported world
```

```
SCENEGRAPH ::=
     UINT(32) nEXTERNPROTO        // number of EXTERNPROTO declarations
     UINT(32) nPROTO              // number of PROTO definitions
     UINT(32) nNODE               // number of top-level nodes
     UINT(32) nROUTE              // number of routes
     EXTERNPROTO[nEXTERNPROTO]    // declaration of EXTERNPROTOs
     PROTO[nPROTO]                // definition of PROTOs
     NODE[nNODE]                  // state of nodes
     ROUTE[nROUTE];               // state of routes
```

```
EXTERNPROTO ::=
     BIT hasMULTIPLEURLS              // has multiple URLs ?
     UINT(31)                         // unique “NODETYPE” number of the prototype
     STRING                           // prototypename
     EXTERNINTERFACEDECLARATION       // the interface
     if(bHASMULTIPLEURLS) {
       UINT(32) nURL                  // number of URLS
       STRING[nURL]                   // multiple URLs
     } else {
       STRING                         // single URL
     };
```

```
EXTERNINTERFACEDECLARATION ::=
     UINT(32) nEventInt               // number of eventIn EVENTs
     UINT(32) nEventOut               // number of eventOut EVENTs
     UINT(32) nField                  // number of field FIELDSPECs
     UINT(32) nExposedField           // number of exposed field FIELDSPECs
     FIELD[nEventIn]                  // eventIn
     FIELD[nEventOur]                 // eventOut
     FIELD[nField]                    // fields
     FIELD[nExposedField];            // exposed fields
```

```
PROTO ::=
     BIT                              // ignored
     UINT(31)                         // unique “NODETYPE” number
     STRING                           // prototypename
     INTERFACEDECLARATION
```

```
     SCENEGRAPH;
```

```
INTERFACEDECLARATION ::=
     UINT(32) nEventIn                  // number of eventIn EVENTs
     UINT(32) nEventOut                 // number of eventOut EVENTs
     UINT(32) nField                    // number of field FIELDSPECs
     UINT(32) nExposedField             // number of exposed field FIELDSPEC
     FIELD[nEventIn]                    // eventIn
     FIELD[nEventOur]                   // eventOut
     INTERFACEFIELD[nField]             // fields
     INTERFACEFIELD[nExposedField];     // exposed fields
```

```
INTERFACEFIELD ::=
     FIELD                  // definition of the field
     FIELDVALUE;            // specification of a default value
```

```
ROUTE ::=
     UINT(32) routeID            // unique ID for this route
     if (!isFullState) {         // is this a delta-encoding?
       ROUTEFORMAT               // delta-encoding: is this a modified or deleted route?
     }
     UINT(32) fromNodeID         // source node
     FIELDNUMBER fromField       // source node element
     UNIT(32) toNodeID           // target node
     FIELDNUMBER toField;        // target node element
```

```
ROUTEFORMAT ::=
     BIT bUNMODIFIED             // delta-encoding: is this route unmodified?
     BIT bDELETE                 // delta-encoding: has this route been deleted?
     UINT(6);                    // padding: ignored
```

```
PROTOINSTANCE ::=
     NODEFIELDS               // the encoded node elements of this instance
     SCENEGRAPH;              // the scene graph of this instance
```

```
STACKORDER ::=
     nStackSize               // number of elements on the stack
     UINT(32)[nStackSize]     // the stack od node IDs
```
